N° d'ordre: 00000

THÈSE

présentée

devant le CONSERVATOIRE NATIONAL DES ARTS ET MÉTIERS

pour obtenir

le grade de : DOCTEUR DU C.N.A.M.
Mention INFORMATIQUE

par

Cédric DU MOUZA

Équipe d'accueil : Vertigo
École Doctorale : EDITE
Composante universitaire : LABORATOIRE CEDRIC

Titre de la thèse :

Patterns de mobilité

Soutenue le 12 octobre 2005 devant la commission d'examen

Mme. :	Véronique	VIGUIÉ DONZEAU-GOUGE	Présidente
MM. :	Georges	GARDARIN	Rapporteurs
	Michalis	VAZIRGIANNIS	
MM. :	Marie-Christine	FAUVET	Examinateurs
	Witold	LITWIN	
	Philippe	RIGAUX	
	Michel	SCHOLL	

Je dédicace cette thèse à Camelia et Raphaël.

Ceux qui errent ne sont pas toujours perdus.

John Reuel Ronald Tolkien

Homo doctus in se semper divitas habet.
Was mich nicht umbringt, macht mich stärker.
Where there is the will, there is the way.
Dupã furtunã, vine şi vreme bunã.

Enfin ça y est diront les mauvaises langues... Oui je sais, 30 ans et toujours étudiant, mais bon, ce n'est pas que pour bénéficier si longtemps des réductions-étudiant que j'ai choisi cette voie. Certes la gestation pour cette thèse a été longue, sans doute la plus longue du règne animal, mais la joie de la naissance n'en est que plus grande.

Mes premiers remerciements vont au père, Philippe, sans qui rien n'aurait été possible. Philippe, ce bébé est autant le tien que le mien, et c'est bien grâce à ta patience et ton énergie que la grossesse m'a paru si courte et si passionnante. À quand le prochain ?

Si l'accouchement a eu lieu sans douleur, tout le mérite en revient à l'anesthésiste. Michel, tu as été présent depuis le début, pour me conseiller, me soutenir, me botter les fesses, ... Ta bonne humeur, ton franc-parler, ton ouverture d'esprit ont fait de toi non-seulement un mentor mais aussi un véritable ami. Michel, merci.

Merci aux parrains Georges Gardarin et Michalis Vazirgiannis d'avoir accepté de rapporter ma thèse. Merci également aux marraines et parrain, Véronique Viguié Donzeau-Gouge, Marie-Christine Fauvet et Witold Litwin, de bien vouloir participer à mon jury.

Je remercie également toute ma famille *vertigineuse* pour leur support. Je tiens plus particulièrement à remercier les oncles Bernd et Dan avec qui j'ai passé plus de temps ces dernières années que n'importe qui. Nos discussions, tant sur le plan travail que loisir ont été un vrai plaisir et risquent de me manquer. Le tennis perdra également beaucoup avec l'éclatement de notre trio de choc. En tous cas, j'espère que vous serez fiers de votre nièce ! Grand merci également à l'oncle David (j'avais envie de dire Tonton David mais j'aurais dû payer sans doute des royalties au chanteur) pour nos réflexions communes et encore plus pour avoir choisi un jour de mars 2003 de prendre Camelia comme stagiaire. Merci aussi à la plus jeune, tante Valérie (eh oui je suis un vieux dans cette maison !), qui apporte une touche féminine dans ce milieu d'hommes.

Merci à tous les frères et sœurs qui m'ont tenu compagnie lors du développement de cette thèse, l'aînée qui bat de ses propres ailes depuis longtemps, Irini, et les benjamins, Imen, Julien, Nouha, Radu, qui ont décidé de se lancer également dans cette grande aventure. Courage, la thèse ce sont des hauts et des bas, mais un véritable épanouissement qui apporte de nombreuses satisfactions. Et puis on se sent si bien dans cette équipe qu'on a du mal à partir, croyez moi...

Merci aux cousins germains Marie, Éric, Bruno, et Ammar, qui sont venus faire un petit tour dans la famille *Vertigo* et qui ont contribué à cette ambiance détendue mais studieuse. Je n'oublie pas de remercier les amis de la famille, Victor et Luc.

Je remercie également mes parents, les grands-parents en quelque sorte de mon « bébé », qui, comme tout grand-parent qui se respecte, ne comprennent pas grand chose aux nouvelles technologies, mais sans qui je n'aurais jamais pu en arriver là.

Merci enfin, et surtout, à la sage-femme, Camelia, qui a été à mes côtés dans les moments les plus douloureux de l'accouchement et qui a dû supporter tant. J'espère te rendre la pareille bientôt.

Table des matières

Table des figures

Chapitre 1

Introduction

L'avènement de technologies permettant de communiquer en temps réel avec des objets se déplaçant librement a rendu possible – et donc souhaitable – le développement de systèmes d'information gérant des données "mobiles". Cette communication est à double sens. D'une part le système GPS (*Global Positioning Systems*) permet la transmission au système central d'informations précises sur la position, la direction, la vitesse des objets mobiles. D'autre part le développement d'outils de communication sans fil multiplie les échanges d'informations entre unités mobiles. À titre d'exemples, citons le suivi d'une flotte de véhicules, la fourniture aux voyageurs d'informations touristiques en temps réel, et enfin toutes les applications militaires.

Les Systèmes de Gestion de Bases de Données (SGBD) actuels ne permettent pas la représentation d'informations évoluant de manière continue car les valeurs des attributs, entre deux mises à jour, sont constantes, et l'évolution de ces valeurs s'effectue de manière discrète en fonction des modifications explicitement demandées au système. À moins d'effectuer des mises à jour extrêmement fréquentes, on ne peut donc obtenir qu'une représentation approximative de la position d'un objet, et il est encore plus difficile de représenter les positions successives formant une trajectoire complète.

La croissance rapide du volume des informations à traiter et la proximité de la problématique avec celles, plus traditionnelles, des bases de données spatiales [56, 99] et des bases de données temporelles laissent à penser que les SGBD tiendront une place importante à l'avenir dans les systèmes dédiés aux objets mobiles [137]. C'est en tout cas la conviction de nombreux chercheurs, souvent issus de l'une des deux communautés citées ci-dessus, qui sont à l'origine d'un nombre croissant de publications proposant soit l'extension de modèles et techniques initialement conçus pour les données spatiales ou temporelles, soit leur intégration.

Ces propositions visent en premier lieu à élaborer des solutions pour fournir aux applications impliquant des objets mobiles les fonctionnalités principales d'un SGBD, à savoir :

– la capacité à *représenter* l'information et à associer à cette représentation des *opérations* qui permettent d'interroger la base ;

– des *structures* d'accès aux données qui garantissent des temps de réponse acceptables même en présence de volumes très importants.

Ces aspects traditionnels sont bien représentés dans les SGBD classiques par le modèle relationnel (doté du langage SQL) et les structures de hachage ou d'arbre B qui offrent des propriétés optimales en terme d'occupation de l'espace et de performance. Alors que ces questions ont été partiellement résolues au cours des dix dernières années pour les données spatiales ou temporelles, elles restent à explorer pour les données spatio-temporelles. Il faut aussi noter, à côté de ces motivations classiques, que certains aspects particuliers à ce dernier type de données, comme la gestion de l'incertitude concernant la position exacte d'un objet à un instant donné, font l'objet de recherches spécifiques.

1.1 Problématique liée aux données spatio-temporelles

Nous commençons par rappeler (très rapidement) les aspects principaux des SGBD, ou du moins ceux qui peuvent placer dans une perspective générale les recherches sur les BD spatio-temporelles.

1.1.1 Bases de données

Une base de données (BD) est un ensemble d'informations volumineux, le terme "volumineux" étant ici à interpréter relativement à la taille des informations que peut contenir la mémoire centrale d'un ordinateur. L'hypothèse de départ est que cette mémoire est beaucoup plus petite que la BD et que donc seule une faible partie des informations est disponible pour le processeur à un moment donné. Cette hypothèse a de nombreuses conséquences sur la conception du système gérant les accès à la base, le SGBD, et conduit à quelques principes de base :

1. le volume des données à considérer peut être compensé par la simplicité des opérations. En d'autres termes, la conception d'un langage d'interrogation doit rechercher un compromis entre le pouvoir d'expression du langage (i.e., toutes les requêtes que l'on peut exprimer) et le coût potentiel d'évaluation d'une requête. Les limitations apportées au pouvoir d'expression permettent d'assurer que les recherches demandées par l'utilisateur pourront être satisfaites par des techniques peu coûteuses ;

2. le goulot d'étranglement est constitué par l'accès aux disques (environ 100 000 fois plus long que l'accès à la mémoire vive) [48]. Il est donc crucial de disposer de chemins

d'accès ou d'algorithmes permettant de diminuer le nombre de pages du disque à charger en mémoire pour effectuer une opération.

Le modèle relationnel illustre une application réussie des principes ci-dessus. Les langages relationnels, SQL en tête, offrent des possibilités relativement limitées mais, en contrepartie, leur complexité est faible [3]. De plus les tables relationnelles peuvent être indexées par des arbres B qui permettent l'accès à un tuple en temps logarithmique. Ces propriétés sont fondamentales et conditionnent la viabilité d'un modèle du point de vue pratique.

Un autre aspect essentiel est la distinction de plusieurs niveaux d'abstraction dans l'architecture d'un SGBD :

1. *Le niveau physique* est celui du stockage des données. Il est entièrement (ou presque) à la charge du système qui doit assurer la sécurité, la confidentialité, et surtout la performance des accès en utilisant correctement l'espace de stockage et en définissant des chemins d'accès propres à satisfaire les requêtes les plus courantes ;

2. *Le niveau logique* offre à l'utilisateur une représentation simple des données, sous forme de table par exemple dans le modèle relationnel. Cette représentation est indépendante du stockage physique des données (sur un disque ou plusieurs, en centralisé ou en réparti, etc), la transcription étant laissée à la charge du SGBD.

   La conception d'un langage de requêtes est basée sur le même souci de masquer les détails d'implantation. Idéalement, un tel langage est *déclaratif* : il permet à l'utilisateur d'exprimer ce qu'il veut obtenir, laissant au système le soin de déterminer les algorithmes à utiliser et les chemins d'accès à suivre pour récupérer les données.

Les recherches sur les modèles et les langages dans les domaines des données spatiales, temporelles et/ou spatio-temporelles visent, dans la mesure du possible, à respecter les principes rapidement survolés ci-dessus. En résumé, il s'agit de définir une représentation logique aussi simple et naturelle que possible et de s'assurer qu'il existe une transcription de cette représentation logique vers un format physique peu coûteux en espace. Le langage d'interrogation devrait être lui aussi relativement simple et naturel, et préserver un bon équilibre entre pouvoir d'expression et complexité d'évaluation. Enfin le niveau physique devrait pouvoir proposer des chemins d'accès aux données en temps logarithmique.

1.1.2 Données spatio-temporelles

Passons maintenant aux problèmes spécifiques soulevés par les données spatio-temporelles. Si on considère qu'un objet spatial est caractérisé, entre autres, par un attribut géométrique décrivant sa *forme* et sa *position*, on peut définir un objet spatio-temporel comme un objet spatial dont la forme et/ou la position varient au cours du temps. En fait nous nous limitons

(comme la plupart des articles) aux mouvements d'objets dont on ignore la forme (que l'on peut donc assimiler à un point). Cela revient à ignorer les zones à surface variables (incendies, déserts, marées, évolution des paysages).

On peut distinguer deux types d'applications parmi celles qui gèrent des objets mobiles. Les premières s'intéressent principalement à l'analyse des flux d'une population donnée, et considèrent des *trajectoires* décrivant l'historique, pour chaque objet, de ses déplacements successifs au cours d'un intervalle de temps qui peut être arbitrairement grand. Les requêtes envisageables sur ce type de données sont par exemple :

- donner la trajectoire de l'objet o entre t_1 et t_2 (requête temporelle) ;
- donner tous les objets dont la trajectoire passe dans la région R (requête spatiale) ;
- donner tous les objets dont la trajectoire passe dans la région R entre t_1 et t_2 (requête spatio-temporelle).

Ces requêtes requièrent une connaissance de la trajectoire complète des objets, ce qui suppose soit que l'on interroge en fait le passé (les mouvements ayant été enregistrés et stockés au fur et à mesure), soit que l'on fait des hypothèses sur la trajectoire future des objets.

Le second type d'application est beaucoup plus orienté vers le suivi (*tracking* en anglais) des objets en temps réel. On s'intéresse par exemple aux mouvements d'une flotte de taxis, d'avions s'approchant d'un aéroport ou aux engins militaires sur un champ de bataille.

Dans de tels cas on cherche à satisfaire des besoins nouveaux ou à prévoir la situation dans un avenir proche plutôt qu'à faire une analyse *a posteriori* des événements. Les requêtes typiques de ce genre d'application sont [136] :

- Trouver les taxis les plus proches du point p (un client qui attend).
- Trouver les hôtels les plus proches de l'objet o (un voyageur qui voit la nuit approcher !).
- Trouver tous les avions qui vont entrer dans la région R dans moins de 10 minutes.

C'est donc la position *courante* des objets qui est primordiale, et éventuellement la position prévisible des objets dans un futur très proche.

Les deux types d'applications diffèrent sur quelques points importants. Le premier cas semble plus complexe puisqu'on ne peut se contenter de connaître la position à un instant t. Il faut conserver toute la trajectoire, ce qui implique une structure plus complexe et des

données plus volumineuses. En revanche, les données étant connues complètement, on se trouve dans la situation assez confortable où la base peut être considérée comme fixe. Ce n'est pas le cas du deuxième type d'application où la position des objets est sans cesse remise en question par des mises à jour successives. Le problème est alors de s'assurer que l'on est capable de prendre en compte le flux des mises à jour qui peut être important, et de garantir que le résultat d'une requête n'est pas déjà obsolète au moment où on le transmet à l'utilisateur.

Dans ces deux sortes d'applications, on rencontre un ensemble commun de problèmes spécifiques à la gestion de trajectoires.

Représentation. Comment représenter une valeur qui varie constamment ? La solution évidente consistant à effectuer des modifications aussi fréquentes que possible revient à adopter une représentation discrète et est globalement insatisfaisante, aussi bien pour des raisons de précision que de surcharge du système.

Le problème est similaire à celui consistant à représenter des données géographiques linéaires (routes, rivières) : on a affaire à un ensemble infini de points que l'on ne peut représenter finiment qu'à l'aide de structures plus ou moins complexes. Le cas des trajectoires introduit un degré de complexité supplémentaire puisque ces données linéaires sont décrites non pas dans un espace à deux dimensions (x et y correspondant à la longitude et à la latitude), mais dans un espace à trois dimensions où la troisième coordonnée, le temps t, joue un rôle essentiel. Il n'y a pas de support dans les SGBD actuels pour ce type d'information.

Interrogation. Le langage traditionnel d'interrogation est SQL, de plus en plus étendu avec diverses fonctionnalités. De telles extensions ont été proposées pour des données spatiales ou temporelles, mais rarement pour des données intégrant étroitement le temps et l'espace.

Indexation. Les structures traditionnelles comme l'arbre B ne sont pas adaptées aux données spatiales car elles reposent sur une structure d'ordre qui n'existe pas dans un espace multi-dimensionnel. Des solutions permettent de s'approcher d'un temps de recherche logarithmique, mais ces structures (R-tree, quadtree) supposent des données statiques, et ne semblent pas adaptées à des données mobiles.

Incertitude. La notion d'incertitude est très importante à prendre en compte pour les données spatio-temporelles. La position d'un objet n'est connue qu'avec une précision relativement limitée, et surtout, sa trajectoire est soumise à diverses fluctuations qui tiennent au caractère accidenté du réseau parcouru (route) et aux variations de vitesse. Il est important de tenir compte de ces marges de tolérance quand on cherche à évaluer une requête qui spécifie des critères de recherche précis.

1.2 Notre problématique

Plusieurs modèles de données ont été proposés au sein de la communauté des bases de données afin d'offrir de nouveaux moyens d'interrogation de collections d'objets mobiles. Une particularité commune à la plupart de ces modèles est de s'appuyer fortement sur les propriétés géométriques des trajectoires. En effet, dans la plupart des cas, la représentation des données et le langage de requêtes sont considérés comme des extensions de certains modèles de données existants conçus initialement pour (et limité à) la manipulation de données géométriques. Une conséquence est que ces modèles de données reposent généralement sur un ensemble de structures de données fournissant un support pour les opérations géométriques (*e.g.* l'intersection géométrique).

Une hypothèse fréquemment adoptée est de considérer un espace 2D dense et de modéliser les trajectoires par des fonctions continues dans cet espace. Bien que cette propriété permette de réaliser plusieurs calculs utiles (par exemple déterminer la position d'un objet à n'importe quel instant), elle n'est pas très adaptée pour des travaux d'analyse ou de classification. Dans ce document nous étudions une approche alternative, à savoir gérer des requêtes comme un processus traitant des *événements* liés aux déplacements des objets sur une représentation discrète de l'espace sous-jacent, nommé *espace de référence*. Des exemples intuitifs de tels événements sont, par exemple, un objet *entre* dans une zone, un objet *reste* dans une zone et un objet *quitte* une zone. Avec ces hypothèses, une requête se présente comme une séquence d'événements élémentaires qui peut-être spécifiée soit en faisant explicitement référence aux zones qui nous intéressent « Donner tous les objets actuellement en `a` qui sont arrivés il y a 5 minutes en provenance de `b` » ou par des *patterns* de mobilité plus génériques tels que, par exemple, « Donner les objets qui se sont déplacés de `a` à une autre zone et sont revenus ensuite en `a` ».

1.2.1 Contribution

Nous introduisons les *patterns de mobilité* comme des expressions décrivant de telles séquences d'événements. Nous avons étudié essentiellement deux aspects dans ce cadre :

- comparaison et agrégation de trajectoires d'objets mobiles, avec prise en compte éventuellement d'un espace multi-échelle ;
- classification en ligne de trajectoires mises à jour continuellement par des outils GPS.

À noter que par pattern nous désignons alors une représentation simplifiée d'une section de trajectoire permettant de caractériser le comportement d'objets. Nos patterns sont distincts des patterns traditionnels utilisées dans le domaine de la reconnaissance de formes. Ils ne

présentent en effet aucune structure spatiale 2D ou plus, et l'objectif n'est pas de reconnaître une forme dans un ensemble d'images par exemple. Nos patterns se rapprochent en fait des patterns utilisées dans la recherche de chaînes de caractères, comme nous le verrons par la suite.

Nous avons considéré dans un premier temps les données historisées et les opérateurs de post-acquisition qui permettent d'analyser le comportement spatio-temporel d'objets appartenant à une population donnée (*e.g.*, des taxis, des avions, etc.) et de réaliser du regroupement et des analyses et comparaisons de similarité. Les outils d'analyse grâce auxquels on peut créer des « profils » spatio-temporels d'objets trouvent un intérêt dans de nombreuses applications. Par exemple dans le domaine de l'analyse du trafic, cela permet de mieux prédire et comprendre la charge d'un réseau routier lors d'une journée typique. Certains services publics peuvent aussi être rendus plus efficaces quand ils peuvent être proposés en accord avec la disponibilité des utilisateurs. Il en est de même pour les analyses de marché dans les applications commerciales. Pour cela nous avons adopté une représentation simplifiée des trajectoires sous forme de séquences des zones successivement traversées et nous avons défini un langage basé sur les expressions régulières contenant des variables, qui permet de rechercher ces séquences de déplacements. Nous décrivons la syntaxe et la sémantique de notre langage et détaillons la technique d'évaluation de nos patterns en nous appuyant sur des automates. Afin de diminuer l'espace mémoire nécessaire pour l'évaluation, nous proposons une restriction du langage rendant l'instanciation des variables déterministes.

Nous avons ensuite considéré le suivi d'objets à l'aide de *requêtes continues*, *i.e.*, des requêtes dont le résultat doit être maintenu durant un intervalle de temps donné (et éventuellement non borné). Lorsque l'on cherche, par exemple, tous les objets qui appartiennent à un rectangle donné R durant les 3 prochains jours, le résultat initial est sujet à des variations si l'on considère les objets entrant et sortant de R. Traiter de manière incrémentale les évolutions du résultat (*i.e.*, sans recalculer périodiquement la totalité du résultat) est une tâche difficile avec un langage de requêtes s'appuyant sur la géométrie parce que l'hypothèse de densité de l'espace du modèle de données est généralement en contradiction avec la nature discrète des observations. Une trajectoire par exemple est obtenue grâce à un échantillon de points fournis par le système GPS et la représentation continue doit être déduite par un mécanisme d'interpolation entre deux points de l'échantillon, ou par extrapolation depuis la dernière position connue [113]. De plus, suivant les opérations géométriques nécessaires à l'évaluation de la requête, on peut être conduit à consulter la trajectoire passée pour vérifier si un objet appartient ou non au résultat. En fait les rares travaux qui proposent une solution pour ce problème portent sur des classes limitées de requêtes (*e.g.*, requêtes de fenêtrage et k-NN dans [85, 64]). Notre solution consiste à considérer une restriction du langage proposé au paragraphe précédent et de ne garder que les patterns correspondant à des mots sur l'alphabet formé par l'union de l'ensemble des étiquettes des zones et des variables. Pour de tels patterns nous proposons une technique d'évaluation basée sur l'algorithme de recherche de chaîne de caractères KMP [69] permettant de n'évaluer chaque étiquette lue qu'une fois et de

trouver en cas d'échec directement le bon décalage à réaliser. Notre évaluation montre que cette technique diminue drastiquement le nombre de calculs à réaliser par le CPU et permet donc de maintenir le résultat de requêtes continues dans le cadre d'une application temps-réel.

Pour résumer les différentes contributions de cette thèse sont :

- un *état de l'art* présentant les différents modèles de représentation, d'interrogation et d'indexation pour des bases de données d'objets mobiles, ainsi que différents autres aspects liés à la nature particulière de ces données ;

- un *modèle de représentation et d'interrogation* de données original, basé sur la notion d'événements d'entrée/sortie d'une zone de l'espace, dans lequel les trajectoires sont exprimées sous la forme d'une séquence de zones. Les requêtes sont des expressions régulières avec variables, dont nous proposons un ensemble de restrictions afin d'obtenir une évaluation nécessitant un espace mémoire faible ;

- une *technique d'évaluation en temps réel* de requêtes continues exprimées dans le modèle précédent pour lequel on a fait d'autres restrictions. Notre algorithme d'évaluation est linéaire en temps suivant le nombre d'étiquettes de la trajectoire lus ;

- une *représentation multi-échelle* de l'espace avec une technique de classification et d'interrogation à l'aide de patterns, basée sur l'existence de différents niveaux d'échelle ;

- un *prototype* s'appuyant sur un simulateur d'événements GPS nous permettant de tester notre modèle et nos techniques d'optimisation.

1.2.2 Organisation du document

Cette thèse est organisée comme suit.

Chapitre 2 - État de l'art. Dans ce chapitre je présente en détail trois solutions récentes proposant une extension des fonctionnalités actuelles des SGBD pour gérer des données spatio-temporelles. Une large section est ensuite consacrée à l'indexation des objets mobiles. Puis j'introduis succinctement certains autres problèmes liés au traitement de données spatio-temporelles. Enfin je termine ce chapitre par la présentation d'un premier travail réalisé, motivé par cet état de l'art, qui cherche à évaluer un grand nombre de requêtes continues sur un grand nombre d'objets. Ce travail a été le véritable point départ de ma thèse présentée ensuite.

Chapitre 3 - Patterns de mobilité. Ce chapitre présente une nouvelle approche pour interroger et suivre en temps réel des objets se déplaçant sur un espace partitionné, à l'aide

de patterns de mobilité. L'objectif est de retrouver à chaque instant les différents objets dont la trajectoire satisfait une certaine séquence de déplacements. Dans la deuxième partie de ce chapitre, nous identifions un sous-ensemble de requêtes pour lequel nous proposons une technique d'évaluation optimisée en espace nécessaire à l'évaluation.

Chapitre 4 - Optimisation. Dans ce chapitre je reprends le modèle présenté au chapitre précédent en réduisant le pouvoir d'expression de nos patterns de mobilité. Sur ce sous-ensemble de patterns bien identifié, j'applique un algorithme bien connu dans le domaine de la recherche de chaînes de caractères, KMP [69], que nous avons étendu afin d'intégrer les variables de nos patterns. Cette technique nous garantit à la fois un temps CPU et un besoin en mémoire faibles pour l'évaluation, permettant par là-même de réaliser une évaluation en continu des requêtes pour un intervalle de temps donné. Des évaluations viennent confirmer le gain attendu par cet algorithme vis à vis d'un algorithme naïf.

Chapitre 5 - Classification multi-échelle de trajectoires. Ce chapitre décrit une nouvelle approche pour classifier, regrouper et interroger de manière continue des trajectoires. Cette approche repose sur la notion de partition thématique multi-échelle de l'espace. La définition d'un pattern associé à un niveau d'échelle nous fournit un outil souple pour la classification de trajectoire. Nous montrons également comment ces patterns peuvent être utilisés pour l'interrogation des trajectoires.

Chapitre 6 - Prototype. Ce chapitre présente le prototype qui a été réalisé sur la base de nos travaux concernant les patterns de mobilité. Ce prototype se décompose en (i) un générateur de données, afin de simuler une flotte de véhicules équipés de serveur GPS, (ii) un serveur qui gère les connexions/déconnexions des utilisateurs, les enregistrements des requêtes et l'évaluation en continu de leur résultat jusqu'à ce que l'utilisateur se désabonne et (iii) un client graphique qui est en fait un navigateur web affichant un fichier SVG représentant la carte et les objets se déplaçant, ainsi qu'un applet java qui récupère les mises à jour des résultats des requêtes.

Chapitre 7 - Conclusion. Ce dernier chapitre résume le travail réalisé lors de cette thèse et présente certaines directions de recherche intéressantes pour le futur.

Chapitre 2

État de l'art

2.1 Introduction

Nous décrivons dans ce chapitre quelques recherches récentes consacrées aux bases de données spatio-temporelles. Bien que l'exposé qui précède puisse laisser penser que cette recherche se limite aux objets mobiles, la problématique recouvre en fait toute situation où la position et la forme d'un objet varient de manière continue au cours du temps. Cependant les problèmes de déformation tiennent une place mineure - du moins dans la littérature récente - et nous nous limiterons donc à la gestion des objets ponctuels en mouvement. Nous avons choisi de présenter en détail quelques-unes des principales publications récentes consacrées aux problèmes de modélisation et d'indexation, ce qui nous amène à délaisser des aspects que nous considérons (sans doute à tort) comme moins centraux. D'où le plan suivant pour notre présentation :

- la section 2.2 compare trois modèles récents qui constituent chacun une approche possible pour étendre les fonctionnalités actuelles des SGBD ;
- la section 2.3 est consacrée à l'indexation d'objets mobiles ;
- la section 2.4 donne, sous forme de notes bibliographiques, un rapide aperçu sur quelques aspects complémentaires ;
- la section 2.5 présente un premier travail [38], motivé par cet état de l'art, visant à traiter des requêtes continues sur un grand nombre d'objets mobiles. L'idée développée dans cette section est le point de départ pour mon travail de thèse.

Cet état de l'art a fait l'objet de deux publications sous formes de chapitres de livre [37, 1].

2.2 Modélisation

La motivation principale des modèles présentés ci-dessous est de définir, pour un objet qui se déplace de manière continue, une représentation sur laquelle on puisse construire un ensemble d'opérations intégrables à un langage d'interrogation.

2.2.1 Principes généraux

Quelques idées de base sont communes à tous les modèles. En premier lieu on assimile la trajectoire d'un objet à une courbe dans un espace défini par trois variables représentant le temps t, et l'espace x et y avec l'interprétation habituelle. De plus, on ne considère le plus souvent qu'une *approximation linéaire* de la courbe, pour des raisons de simplicité de description, et également d'efficacité algorithmique.

Ensuite on se place dans un espace dense, par exemple $\mathbb{R}^3$, ce qui permet d'obtenir la continuité du mouvement. Dans une telle interprétation, la trajectoire d'un objet mobile est constituée d'un ensemble *infini* de points, ensemble pour lequel on doit définir une représentation *finie*. Le problème est identique dans les bases de données géographiques : un département, une route, considérés dans un espace dense, forment des ensembles infinis que l'on manipule en les représentant finiment par des structures telles que la ligne brisée formant le contour d'un polygone.

Cette approche débouche sur des outils relativement complexes pour l'utilisateur qui doit connaître les structures utilisées, maîtriser la syntaxe des opérations pour chaque structure en particulier, savoir quel est le type du résultat qui lui est fourni, etc. L'extension de cette approche à des données spatio-temporelles semble donc mener à des modèles complexes. Heureusement le caractère très particulier d'une trajectoire d'objet mobile permet de définir deux niveaux de représentation :

- **au niveau abstrait**, on peut définir un objet mobile comme une fonction continue du temps vers l'espace à deux dimensions ;

- **au niveau symbolique**, on utilise des structures qui permettent une représentation compacte de ces fonctions.

La représentation abstraite est simple à appréhender et permet de définir une interface avec l'utilisateur qui est indépendante de la représentation symbolique choisie. Cette approche fondée sur plusieurs niveaux d'abstraction est classique et associe chaque acteur confronté au SGBD (l'utilisateur, le programmeur, l'administrateur) à un mode approprié de présentation de l'information.

Au niveau le plus haut on doit trouver un formalisme qui respecte la continuité de l'espace et du temps. Le modèle doit, conceptuellement, manipuler des ensembles infinis à l'aide d'un langage de requêtes déclaratif proche de SQL. À un niveau plus bas, proche de l'implémentation, le modèle propose une représentation finie des données afin de les stocker et de

les manipuler.

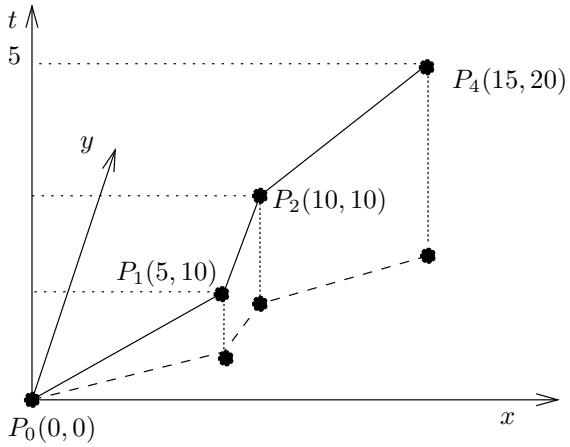


FIG. 2.1 – Une trajectoire

L'exemple de la figure 2.1 représente l'approximation linéaire d'une trajectoire. On peut voir, au niveau abstrait, cette trajectoire comme une fonction par morceaux qui, à chaque valeur de t sur l'intervalle $[0, 5]$, associe une paire $[x, y]$.

Il existe plusieurs représentations symboliques possibles pour ces fonctions. On peut par exemple stocker les points définissant les segments, et reconstruire ensuite les fonctions par interpolation. La forme générale d'une fonction donnant une coordonnée en fonction du temps est :

$$x(t) = x_0 + v(t - t_0)$$

On peut donc la représenter par un triplet $[x_0, t_0, v]$. Noter que la restriction à des données linéaires implique que la vitesse est supposée constante sur un segment. C'est cette forme que nous considérerons dans tout ce qui suit.

Nous présentons maintenant trois propositions qui construisent un modèle de données complet, incluant la spécification d'un langage de requête, sur la base des principes décrits ci-dessus.

2.2.2 Le modèle MOST [113]

Le modèle *Moving Objects Spatio-Temporal* (MOST) a été proposé en 1997 dans [113] et implanté ensuite dans le prototype DOMINO (*Databases fOr MovINg Objects*) [135]. Le modèle gère la position présente et future des objets, mais pas l'historique de leur trajectoire, ce qui le destine principalement aux applications qui font du suivi en temps réel. L'aspect « incertitude des données » a été développé dans [134, 115].

Représentation

Par opposition aux attributs statiques dont la valeur ne change que par modification discrète, les auteurs définissent les *attributs dynamiques* dont la valeur change de façon continue. Un attribut dynamique A est représenté par 3 sous-attributs :

- $A.value$: valeur de A au moment de la mise à jour ;
- $A.updatetime$: moment de la dernière mise à jour ;
- $A.function$: fonction de t qui vaut 0 à $t = 0$.

La valeur de l'attribut est alors une fonction du temps, $A.value + A.function(t)$ au temps $A.updatetime+t$. L'idée est qu'il est nécessaire de faire des mises à jour uniquement quand un objet change de direction. On stocke alors la position au moment du changement, la direction prise et la vitesse, soit les trois informations nécessaires pour représenter une valeur variant linéairement en fonction du temps.

Ce modèle permet donc de représenter de manière implicite les états futurs de la base de données (c'est-à-dire la position future des objets mobiles). Un *état* de la base de données est l'ensemble des valeurs des attributs à un instant donné. Les requêtes peuvent être alors évaluées sur ces états, présents ou futurs. Les auteurs distinguent trois types de requêtes :

Requêtes instantanées. Elles sont évaluées à un instant donné, généralement celui où la requête est effectuée. Exemple : « Donner tous les hôtels à moins de 5 Kms de moi ».

Requêtes continues. Si la requête précédente est faite par un objet mobile, la réponse peut varier d'un instant à l'autre puisqu'elle dépend de la position de l'objet. On peut imaginer un conducteur qui effectue de manière *continue* la requête cherchant les hôtels.

Requêtes persistantes. Ce terme, proposé par [113], correspond à des requêtes qui ne peuvent s'évaluer que sur *l'ensemble* des états de la base. Exemple : quels sont les véhicules dont la vitesse double toutes les 5 minutes ?

Le langage FTL

Afin de tirer parti de la sémantique des attributs dynamiques, une extension de SQL par des prédicats de logique temporelle est proposée. Le nouveau langage, nommé FTL, comprend deux opérateurs temporels de base : **Until** et **Nexttime**. Etant donné deux prédicats f et g, ces opérateurs sont définis ainsi :

- f **Until** g est vérifié si on a g à cet état ou dans le futur, et f vérifié en attendant.

- **Nexttime** f est vrai si f est vérifié au prochain état de l'historique.

Nous avons dès lors une base pour construire d'autres opérateurs utilisés dans FTL :

- **Eventually** $f = true$ **Until** f.
- **Always** $f = \neg$ **Eventually** $\neg f$
- **Eventually_within_**c g : g sera satisfaite dans l'intervalle de temps c.
- **Eventually_after_**c g : g sera vérifié après l'intervalle c.
- **Always_for_**c g : g est vérifié de manière continue pour les c prochaines unités de temps.
- g **until_within_**c h : un futur dans $[0, c]$ où h sera vérifié, en attendant g vérifié.

Ce langage permet de sélectionner les objets qui satisfont une formule construite dans un langage comprenant les opérateurs ci-dessus, les opérateurs arithmétiques binaires ($\leq, >, = ...$), quelques prédicats portant sur la partie spatiale des données, les connecteurs $\wedge$, $\vee$ et $\neg$, mais pas de quantificateurs. Il n'y a pas de définition très précise de ce langage, ce qui rend difficile l'estimation de son pouvoir d'expression, et pas d'étude sur sa complexité.

Voici quelques requêtes FTL.

- *Donner tous les objets qui entrent dans le polygone P dans les 3 prochaines unités de temps, et qui ont l'attribut price $\leq$ 100.*

  retrieve *o*
  where *o.PRICE* $\leq$ *100*
  and **Eventually_within_3** *INSIDE(o, P)*

  Le prédicat *INSIDE(obj, area)* valant vrai si l'objet *obj* se trouve dans l'aire *area*.

- *Donner tous les objets qui entrent dans le polygone P dans les 3 prochaines unités de temps et restent dans P deux unités de temps supplémentaires*

  retrieve *o*
  where *o.PRICE* $\leq$ *100*
  and *(***Eventually_within_3** *INSIDE(o, P)*
  and **Always_for_2** *INSIDE(o, P))*

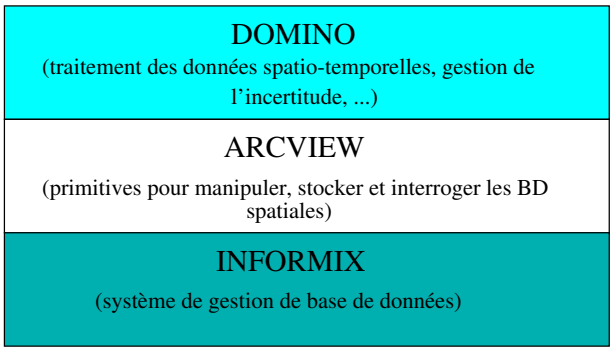


FIG. 2.2 – Architecture du modèle DOMINO

Évaluation

Les auteurs proposent un algorithme d'évaluation des requêtes MOST, restreintes aux formules *conjonctives*, instantanées ou continues mais non persistantes. Lors de l'évaluation d'une requête Q spécifiée par la formule f et les variables libres $x_1, x_2..., x_k$, l'algorithme retourne une relation $Answer(Q)$ avec $k+1$ attributs : les k premiers seront une instanciation des x_i, le dernier un intervalle de validité. Par exemple si $Answer(Q)$ est constituée des deux tuples (2, [10,15]) et (5, [12,14]), alors le système affichera l'objet qui a pour identifiant 2 entre les instants 10 et 15, et l'objet 5 entre les instants 12 et 14.

Le système proposé est conçu pour être implanté *au-dessus* d'un SGBD fournissant les fonctionnalités de base comme un langage d'interrogation de données statiques, la concurrence d'accès, l'indexation, etc. En fait c'est un des prérequis de la plupart des modèles spatio-temporels : au point où en est la technologie des bases de données, toute extension doit pouvoir s'intégrer dans les systèmes existant, et ne pas impliquer une remise à plat complète pour l'introduction de fonctionnalités qui restent somme toute marginales.

L'implantation du modèle DOMINO nécessite un SGBD relationnel ou objet, et des types spatiaux. Les auteurs ont réalisé un prototype [135] selon une architecture basée sur le système Informix pour l'aspect gestion des données, le système ArcView pour les données spatiales et une sur-couche qui constitue le système DOMINO proprement dit. Les principes de l'extension peuvent être résumés ainsi.

- si la requête est du SQL classique, elle est évaluée par Informix ;
- si les attributs dynamiques sont uniquement impliqués dans la clause **select**, Informix recherche les données et DOMINO évalue la valeur à l'instant considéré ;
- en cas de clause **where** portant sur les attributs dynamiques, les auteurs suggèrent de récrire la requête afin d'isoler l'évaluation impliquant les attributs dynamiques.

En conclusion, MOST est une approche pragmatique pour introduire des données spatio-temporelles dans un SGBD. Le modèle laisse beaucoup de questions en suspens, notamment celles relatives au pouvoir d'expression du langage et à la complexité d'évaluation. L'implantation au-dessus d'un SGBD, bien que seulement ébauchée dans l'article, paraît une nécessité. Elle laisse cependant ouverte la question de l'indexation des données spatio-temporelles.

2.2.3 Types abstraits spatio-temporels [46]

L'introduction de types abstraits de données (TAD) pour étendre les fonctionnalités du modèle relationnel à des valeurs complexes est maintenant relativement ancienne. L'utilisation de TAD pour la gestion des données géométriques est décrite par exemple dans [55] pour la partie modélisation et [12] pour la partie concernant le processus d'évaluation et d'optimisation des requêtes. De nombreux systèmes sont maintenant extensibles, comme ORACLE [111] ou PostgreSQL [87], ce dernier proposant un ensemble complet de TAD spatiaux.

Les principes guidant la définition de TAD spatio-temporels ont été initialement présentés dans [43], puis un système de types a été proposé dans [46]. Ce modèle repose sur deux niveaux d'abstractions : le modèle "abstrait" qui permet de raisonner sur des *ensembles infinis*, sans se soucier de savoir si une représentation finie de ces ensembles existe, et le modèle dit "discret", basé sur une *représentation finie*, proche de l'implémentation. Contrairement à [113], on peut représenter l'historique des trajectoires.

Le modèle abstrait

Le modèle se base sur un ensemble de types comprenant les *types atomiques* (*int, real, string, bool*), des *types spatiaux* pour représenter des données dans l'espace 2D (*point*, un point, *points*, un ensemble fini de points, *line*, une suite de courbes continues dans le plan, et *region*, ensemble fini de polygones disjoints) et enfin des *types temporels* (*instant*).

Les instances de ces types sont des valeurs statiques qui ne dépendent pas du temps. Afin de pouvoir instancier des types dont les valeurs sont dynamiques, on utilise le constructeur de type *moving*. Pour un type de données α appartenant aux types de base, il construit un type dont les valeurs sont fonction du temps, *moving*(α). Une instance de ce type est un objet $\alpha(t)$. Un véhicule se déplaçant de manière continue est donc une instance de *moving(point)*.

Le modèle n'est pas limité aux objets mobiles : une région touchée par un incendie, dont la surface évolue en fonction du temps est du type *moving(region)*. On note plus concisément par *mpoint* et *mregion* les types *moving(point)* et *moving(region)*.

Étant donnée une liste de tels types, on étend le modèle relationnel de manière à permettre à certains attributs de prendre leurs valeurs dans le domaine des types "mobiles". Voici par exemple une relation donnant la description des vols d'avion et de leurs trajectoires.

Vols (id : string, compagnie : string, traj : mpoint)

Le principe de la modélisation est de définir des opérations applicables aux instances "fixes", puis de définir un processus (*lifting*) permettant de transposer ces opérations aux instances mobiles. Prenons un exemple simple, avec l'opérateur *distance*.

1. **Instances statiques** : on définit classiquement une opération *distance*, prenant en argument deux points, instances du type *point*, dont le résultat est une instance de *real* ;

2. **Instances "mobiles"** : maintenant si on applique l'opérateur *distance* à des points mobiles de type *mpoint*, on obtient une valeur réelle dépendant du temps, instance de *mreal*.

Autre exemple : le prédicat *inside (point, region)* qui teste si un point est dans une région, renvoie un booléen "mobile" de type *mbool* quand on le transpose à des instances de *mpoint* et *mregion*.

Les auteurs proposent une liste de types et d'opérations, définis de la manière suivante :

- **at** : $mpoint \times time \rightarrow point$, qui permet de connaître la position à un temps t donné.

- **mdistance** : $mpoint \times mpoint \rightarrow mreal$, qui renvoie un réel mobile représentant la distance, variable en fonction du temps, entre 2 points.

La sémantique est alors :

- $f_{at}(r,t) := r(t)$

- $f_{mdistance}(r,s,t) := \begin{cases} d(r(t),s(t)) & si\ r(t) \neq \perp \wedge s(t) \neq \perp \\ \perp & sinon \end{cases}$

Interrogation

On peut dès lors formuler des requêtes en un langage SQL étendu incorporant les opérateurs ci-dessus.

- Donner tous les vols Air France d'une longueur de plus de 5000 Kms.

```
select    *
from      Vols
where     compagnie = 'Air France'
and       length (trajectory (traj)) > 5000
```

L'opérateur *trajectory* donnant la composante spatiale (en 2D) de la trajectoire d'un avion.

– Donner les couples d'avions qui se croisent à moins de 500m.

select	*A.id, B.id*
from	*Vols A, Vols B*
where	*A.id <> B.id*
and	*minvalue (mdistance(A.traj, B.traj)) < 0.5*

Un problème potentiel de cette approche est le nombre très élevé d'opérations à connaître pour l'utilisateur. De plus il faut être très attentif, quand on compose les opérations, à ne pas commettre d'erreur de type.

Le modèle discret

Il s'agit d'un niveau proche de l'implémentation, donc manipulant des ensembles finis, permettant de représenter le modèle abstrait. Tous les types du modèle abstrait ont leur équivalent dans le modèle discret. Le constructeur *moving* est remplacé par un nouveau constructeur de types.

Ainsi pour les types de base *int, real, string, bool*, l'implémentation se fait directement d'après les types correspondant du langage. *Point* est un couple de réel (x, y) et *points* un ensemble fini de couples (x_i, y_i). Pour *line* et *region*, on considère l'approximation linéaire (lignes brisées, polygones...). De même la représentation de *instant* se déduit directement. Le constructeur de types ***mapping*** permet d'obtenir des types pour les objets mobiles. Un objet de type *mapping* est un ensemble $\{(I_\alpha, v)\}$, où I_α est un intervalle de temps et v la représentation de l'objet géométrique se déplaçant dans cet intervalle I_α. Concrètement I_α est l'intervalle de temps maximal pendant lequel les valeurs des attributs de cet objet peuvent être représentées par des fonctions d'interpolation, notées ι_α. Un point mobile est donc un objet de type *mapping(upoint)* et une région mobile du type *mapping(uregion)* où :

upoint : défini sur un intervalle de temps I_α par un quadruplet (x_0, x_1, y_0, y_1), réels représentant la position au début et à la fin de l'intervalle, tel que pour $t \in I_\alpha$ on a

$$\iota((x_0, x_1, y_0, y_1), t) = (x_0 + x_1 \times t, y_0 + y_1 \times t)$$

uregion : basé sur un ensemble de segments mobiles qui ne se chevauchent pas. Les segments gardent la même direction pour tout t ; ils ne peuvent donc pas faire de rotation. Ces régions peuvent avoir des trous ou non. La figure 2.3 montre la représentation discrète d'une région mobile.

Les définitions des fonctions dans ce modèle sont considérablement plus complexes puis-

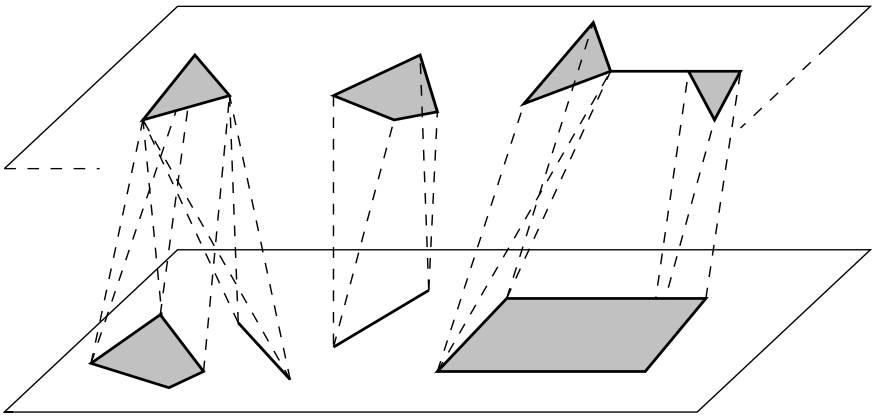


FIG. 2.3 – Représentation discrète d'une région mobile

qu'elles manipulent désormais des ensembles finis et sont par conséquent plus proches de l'implémentation. Ainsi l'opérateur *at* a dans ce modèle la définition suivante : soit $r =< (p_1, t_1, b_1, c_1), ..., (p_m, t_m, b_m, c_m) >$ un *mpoint* et t un instant donné,

$$f_{at}(r,t) \quad := \quad \begin{cases} \bot & si\, m = 0 \vee (m > 0 \wedge (t < t_1 \vee t > t_m)) \\ p_i & si\, m \geq 1 \wedge (\exists i \in \{1, ..., m\} \, : \, t_i = t) \\ lin(p_i, t_i, & \\ \quad p_{i+1}, t_{i+1}, t) & si\, m \geq 2 \wedge (\exists i \in [1, m-1], \, (t_i < t < t_{i+1}) \wedge b_i \wedge \neg c_i) \\ p_i & si\, m \geq 2 \wedge (\exists i \in [1, m-1], \, (t_i < t < t_{i+1}\} \wedge b_i \wedge c_i) \\ \bot & si\, m \geq 2 \wedge (\exists i \in [1, m-1], \, (t_i < t < t_{i+1}\} \wedge \neg b_i \end{cases}$$

où $lin(p_1, t_1, p_2, t_2, t)$ est une fonction qui retourne le point p déduit de l'interpolation linéaire entre les 2 points (p_1, t_1) et (p_2, t_2) au temps t. Le drapeau b_i vaut vrai si le point est défini entre t_i et t_{i+1}. Le drapeau c_i vaut faux si une interpolation linéaire entre p_i et p_{i+1} peut être utilisée.

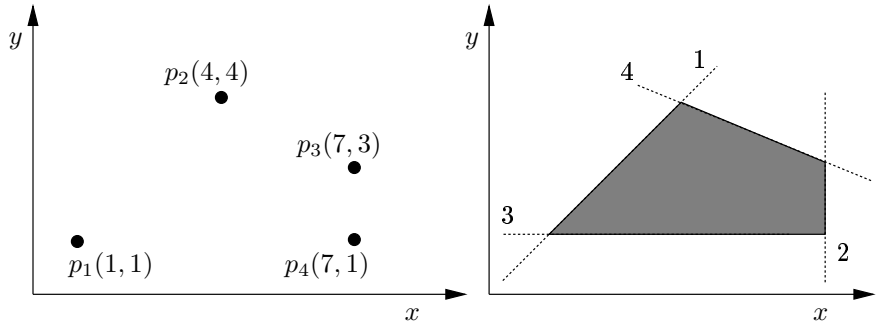
2.2.4 Modèle contraintes [52]

Le troisième modèle que nous présentons s'appuie sur les bases de données contraintes qui proposent un cadre formel permettant de raisonner sur les données géométriques. Nous introduisons brièvement l'intuition avant de développer l'application aux données spatio-temporelles.

Bases de données contraintes

Le modèle de données contraintes a été proposé initialement en 1990 par Kanellakis, Kuper et Revesz [67]. L'idée de base est qu'il est beaucoup plus facile d'interroger une base de données multi-dimensionnelle en considérant que la base est constituée de *l'ensemble* des points, plutôt que de raisonner sur la représentation finie de cet ensemble de points qui peut reposer sur des structures relativement complexes.

Essentiellement, le modèle contraintes propose une extension du modèle relationnel consistant à exprimer des requêtes avec un langage relationnel classique (disons la logique du premier ordre) sur des relations infinies dont chaque tuple est un des points d'un objet géométrique (voir Fig. 2.4). Un polygone par exemple est un ensemble infini de points, considéré comme une relation sur deux attributs, x et y. Voir Fig. 2.5.



Un ensemble fini de points — Un ensemble infini de points

FIG. 2.4 – Idée de base : ensemble de points = relation

Le fait de raisonner sur des ensembles de points (en fait sur des relations) simplifie considérablement la tâche de l'utilisateur. Une opération de *clipping* par exemple, sélectionnant la partie d'un polygone *Spat* située dans un rectangle *Rect*, s'exprime par une simple jointure.

select	*x, y*
from	*Spat, Rect*
where	*Spat.x = Rect.x*
and	*Spat.y = Rect.y*

Bien entendu le raisonnement doit s'appuyer sur une représentation finie, soit en l'occurrence des *contraintes*. Ce niveau, dit *niveau symbolique*, est caché à l'utilisateur mais constitue un support pour l'évaluation des requêtes.

La représentation avec contraintes peut être vue comme une extension de la représentation - finie - relationnelle, au prix d'un léger changement de point de vue. Au lieu de considérer une relation comme un ensemble de tuples, on doit maintenant la voir comme une formule de logique du premier ordre. Par exemple un ensemble fini de quatre points est représenté par la formule :

$$\phi_{Spat} \equiv (x = 1 \wedge y = 1) \vee (x = 7 \wedge y = 3) \vee (x = 4 \wedge y = 4) \vee (x = 7 \wedge y = 1)$$

Le langage logique utilisé dans ce cas ne comprend que le symbole d'égalité, et aucun symbole d'opération arithmétique. En étendant le langage, on obtient la possibilité de représenter un ensemble infini, comme par exemple le polygone de la figure 2.4.

Spat	
x	y
1	1
7	3
4	4
7	1

Une instance finie

Spat	
x	y
1	1
1.001..	1
1.002..	1
1.003..	1

Spat (cont.)	
x	y
...	...
3	2
3.001..	2
...	...

Une instance infinie

FIG. 2.5 – Instances de la relation *Spat*

Les contraintes consistent en équations et inéquations de la forme $\sum_{i=1}^{p} a_i\, x_i \Theta a_0$, où Θ est un prédicat parmi $\{=, \leq\}$, les x_i désignent les variables (interprétées dans le domaine des réels) et les a_i sont des constantes. On peut ainsi représenter un polygone comme celui de la figure 2.4 avec un conjonction d'inéquations correspondant aux demi-plans définissant le polygone.

$$
\begin{array}{ll}
 & y \leq x \quad (1)\\
\wedge & x \leq 7 \quad (2)\\
\wedge & y \geq 1 \quad (3)\\
\wedge & x + 3y - 16 \leq 0 \quad (4)
\end{array}
$$

La sémantique de cette formule est définie naturellement comme l'ensemble des paires de valeurs pour (x, y) dans le domaine des réels qui la rendent vraie. On peut interpréter géométriquement cette représentation comme l'intersection d'un ensemble de demi-plans, chacun étant représenté par une inéquation sur x et y.

Il est possible de représenter, avec des contraintes linéaires utilisant seulement l'addition, toutes les figures géométriques courantes (polygones, lignes, points). De plus *il n'y a a pas de limite sur la dimension des objets représentés.* La dimension de l'espace dans lequel sont décrits les objets géométriques correspond en fait au nombre de variables utilisées : 2 pour un espace 2D, 3 pour un espace 3D, etc. Un polytope (convexe) P en dimension trois est décrit par une conjonction d'équations d'hyper-plans sur des variables x, y et z.

$$
\begin{array}{c}
32x + 5y - 34z \leq 98\\
\wedge\, 40x - 28y + 3z \leq 30 \ldots
\end{array}
$$

L'évaluation de requêtes relationnelles sur des données géométriques consiste alors à déterminer quelle est la formule qui représente le résultat de la requête. Par exemple la sélection relationnelle $\sigma_{2x-3y+8z\leq 3}(P)$ donne le résultat suivant.

$$
\begin{array}{c}
\mathbf{2}\ x\ \mathbf{-}\ \mathbf{3}\ y\ +\ \mathbf{8}\ \mathbf{z} \leq \mathbf{3}\\
\wedge\, 32x + 5y - 34z \leq 98\\
\wedge\, 40x - 28y + 3z \leq 30 \ldots
\end{array}
$$

On a simplement effectué la conjonction de la formule représentant P et de l'argument de la sélection ! Bien entendu il reste à évaluer l'ensemble de points représentés par cette nouvelle formule, ce qui met en œuvre une algorithmique sur les contraintes linéaires. Le livre [74] permet d'en savoir plus sur tous les aspects très rapidement survolés ci-dessus.

Application aux objets mobiles

Ce modèle développé par le CNAM et l'INRIA est présenté dans [52]. L'idée est plus générale que l'application aux objets mobiles : on s'intéresse en fait à des classes d'objets géométriques placés dans un espace de dimension d mais dont la dimension "intrinsèque" est inférieure. Cette propriété est caractérisée dans le modèle par l'existence de *fonctions d'interpolation* permettant d'obtenir une des variables décrivant un objet comme une fonction d'autres variables.

Le modèle contraintes permet une formalisation simple de la classe des objets mobiles. Nous avons vu qu'une trajectoire est approximée par une suite de segments connectés deux à deux dans un espace de dimension 3. La trajectoire de la Fig. 2.1 peut être représenté en mode contraintes par la formule :

$$\begin{array}{ll} x = 10t \ \wedge \ y = 5t \ \wedge \ 0 \leq t \leq 1 & \text{(a)} \\ \vee & \\ y = 5t \ \wedge \ x = 10 \ \wedge \ 5 \leq t \leq 10 & \text{(b)} \\ \vee & \\ 3x = 10t + 10 \ \wedge \ 3y = 5t + 20 \ \wedge 2 \leq t \leq 5 & \text{(c)} \end{array}$$

A l'instant $t = 0$, l'objet mobile est situé au point P_0. Puis il va de P_0 à P_3, où il arrive à $t = 5$, en passant par P_1 ($t = 1$) et P_2 ($t = 2$). Les variables x et y dont des fonctions (linéaires) du temps t, ce qui correspond à la constatation informelle qu'une trajectoire est une ligne (de dimension intrinsèque 1) décrite dans un espace de dimension 3.

En considérant, à un niveau abstrait, ces trajectoires et autres objets spatiaux comme des ensembles de points, on obtient une représentation des données qui est simple à appréhender pour l'utilisateur, et qui est *indépendante* du stockage physique. De plus ces ensembles de points constituent des relations au sens classique du terme et peuvent donc être interrogés avec SQL.

Interrogation

Pour l'utilisateur, il n'y a que des relations, et un langage comme SQL. Voici quelques exemples, basés sur le schéma suivant :

- Une carte $Map(x, y, libellé)$ qui donne l'occupation du sol en chaque point (x, y).
- Un Modèle Numérique de Terrain, $TIN(x, y, z)$ qui donne l'altitude en chaque point.
- La trajectoire d'avions, $Traj(t, x, y, a)$, avec la position et l'altitude à chaque instant.

Les requêtes suivantes illustrent la simplicité de cette approche. Il n'y a pas besoin de penser à la machinerie interne (comme les structures, les opérations, la syntaxe, le type du résultat, etc). Exprimer une requête se limite à considérer les données comme des ensembles de points ou, de manière équivalente, comme des relations. Que ces relations soient finies ou infinies n'a plus d'importance puisque nous avons vu qu'il existe un format fini et des opérations sur ce format qui permettent l'évaluation.

*Donner la position et l'altitude de l'avion au temps t*1.

select x, y, a
from Traj
where t = 't1'

Donner la trajectoire de l'avion quand il survolait une région à plus de 1 000m.

select t, Traj.x, Traj.y
from TIN , Traj
where h $\geq$ 1000
and TIN.x = Traj.x
and TIN.y = Traj.y

Montrer les forêts entre 1 000 et 2 000 mètres.

select t.x, t.y
from TIN t, Map m
where t.x = m.x **and** t.y = m.y
and 1000 $\leq$ h $\leq$ 2000
and name = 'forest'

Montrer les parties de la trajectoire où l'avion était au-dessus de la mer, à une altitude supérieure à 1 000 m.

select l.x, l.y, l.t
from Traj l, Map m
where m.x = l.x **and** m.y = l.y
and name = 'sea'
and a > 1000

Évaluation

Au moment de l'évaluation d'une requête, le nouvel échantillon et les fonctions d'interpolation associées doivent être calculés par le système et non définis par l'utilisateur. Si les requêtes sont exprimées sur des relations abstraites (infinies), l'évaluation doit se faire sur la représentation finie. Les techniques d'évaluation proposées pour ce modèle montrent que l'évaluation peut-être réduite à un petit nombre d'opérateurs appliqués aux attributs qui forment la clé de la relation. Voir [51, 52].

2.3 Indexation

Cette section est consacrée à l'indexation d'objets mobiles. Nous commençons par un rappel des deux techniques les plus couramment utilisées pour indexer des données multi-dimensionnelles, à savoir le R-tree et le quadtree linéaire, avant d'exposer deux adaptations récentes destinées aux objets mobiles.

2.3.1 Indexation de données multi-dimensionnelles

Il existe une multitude de techniques d'indexation d'objets multi-dimensionnels [47]. La plupart sont des variantes de l'une des deux catégories suivantes :

- **partitionnement du jeu de données** en fonction de leur répartition dans l'espace. Essentiellement, on cherche à regrouper dans les mêmes pages du disque les objets proches dans l'espace ;

- **découpage régulier de l'espace**. Dans ce cas on ne tient pas compte de la distribution du jeu de données à indexer. L'espace de référence est découpé *a priori* en cellules, régulières ou non. Les objets sont ensuite affectés aux cellules avec lesquelles ils ont une intersection.

La principale structure de la première catégorie est le R-tree, dont un exemple est donné dans le figure 2.6. Le R-tree est construit sur une hiérarchie de rectangles contenus les uns dans les autres, le niveau le plus bas étant constitué des rectangles minimaux englobant la géométrie des objets spatiaux. À chaque niveau, les rectangles sont groupés en "paquets" pouvant être stockés sur une page du disque.

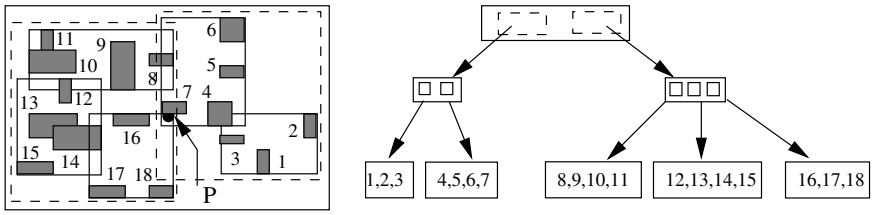


FIG. 2.6 – Indexation par R-tree

Dans l'exemple de la figure 2.6, on suppose qu'une page peut stocker au plus 4 objets, ce qui donne les groupes {1, 2, 3}, {4, 5, 6, 7}, etc. Le regroupement des objets est basé sur leur proximité dans l'espace. On cherche en fait à minimiser le recouvrement des rectangles englobant de chaque groupe (les rectangles dessinés en traits fins sur la figure). Ces rectangles sont à leur tour regroupés pour former le niveau supérieur, et ainsi de suite jusqu'à ce que l'on obtienne un dernier groupe de moins de 4 éléments, stockés sur une page qui forme la racine de l'arbre.

Le R-tree a des propriétés comparables à celles de l'arbre-B : l'arbre est équilibré, sa hauteur est logarithmique dans la taille du jeu de données, et sa complexité en espace est linéaire. La recherche (pointé par exemple) basée sur un R-tree consiste à parcourir, à chaque niveau (en partant de la racine) le sous-arbre dont le rectangle englobant contient le point argument. Malheureusement le R-tree ne garantit pas un temps de recherche logarithmique. La recherche des objets contenant le point P par exemple devra parcourir 5 pages (la racine, les

deux nœuds de niveau 1, et deux feuilles), sans ramener un seul objet. Un autre inconvénient de la structure est le coût de l'algorithme qui divise les objets d'un nœud quand celui-ci est trop plein. La deuxième structure largement utilisée (dans ORACLE par exemple) sous une variante ou une autre est le *quadtree*. Dans sa variante la plus simple, on découpe l'espace en un ensemble de cellules régulières et on associe une page à chaque cellule. Chaque objet est alors inséré dans toutes les cellules qu'il intersecte, ce qui peut mener à référencer plusieurs fois un même objet.

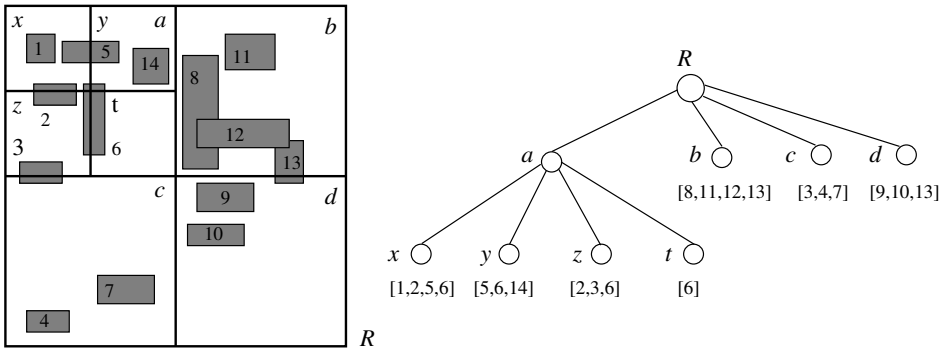


FIG. 2.7 – Indexation par quadtree

Quand une page déborde, on la divise en quatre pages correspondant à quatre sous-cellules égales, et on répartit les objets entre ces quatre pages/cellules. Un exemple est donné dans la figure 2.7, en supposant qu'une page contient 4 objets. Le coin supérieur gauche est plus divisé puisqu'il contient 5 objets.

Cette méthode est très simple, et comprend de nombreuses variantes intéressantes puisqu'elles permettent de réutiliser l'arbre B du SGBD. L'inconvénient est la redondance qui impose un tri pour éliminer les doublons après chaque requête.

En résumé il n'existe pas de structure optimale pour indexer des objets dans un espace de dimension 2. Les solutions ci-dessus apparaissent cependant comme des bons compromis qui se rapprochent du comportement souhaité (temps de recherche logarithmique, occupation de l'espace linéaire) dans le cas de données ayant des propriétés statistiques acceptables (et notamment une répartition dans l'espace à peu près uniforme). Si on considère maintenant que les objets sont mobiles, aucune des deux structures ci-dessus ne semble convenir. Dans l'un et l'autre cas, l'indexation est basée sur l'hypothèse que la position des objets est fixe.

2.3.2 Indexation par Quadtree [122]

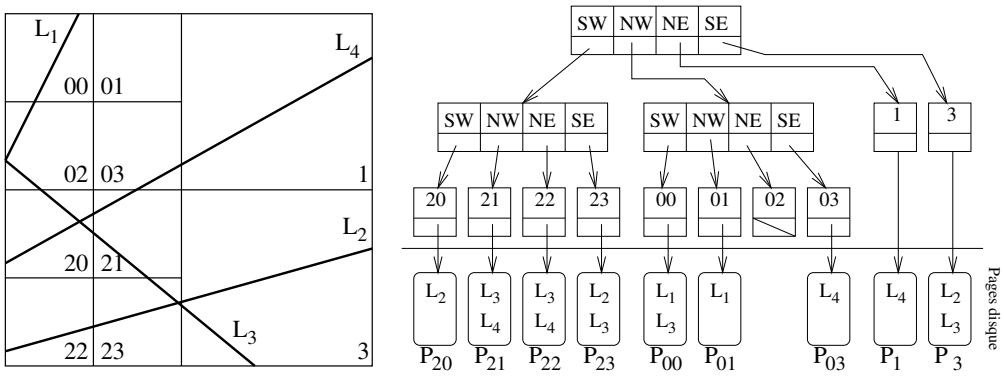
Comme le précédent, cet index vise à indexer les positions futures, et se rapproche donc du modèle [113] auquel il se réfère d'ailleurs explicitement. Chaque trajectoire est donc un segment de droite commençant à t_0 et s'étendant sur un intervalle de temps prédéfini. Elle est représentée par deux fonctions linéaires $x = v_x \times t + x_0$ et $y = v_x \times t + x_0$.

Construction

Les trajectoires sont indexées par une variante du quadtree dite *PMR-quadtree*. Afin de se ramener au cas où on indexe des segments dans un espace 2D, on utilise en fait deux quadtrees : un pour la fonction donnant $x(t)$ et l'autre pour $y(t)$. Lors d'une recherche, il faut utiliser séparément les deux index, puis effectuer l'intersection des résultats obtenus.

La construction de l'index est basée sur le découpage récursif classique du quadtree. Étant donné un segment s décrit par (v_x, x_0), on recherche tous les quadrants qui intersectent s et on y insère l'information (id, v_x, x_0) où id est l'identifiant permettant de rechercher l'objet sur le disque. Comme toutes les structures basées sur un découpage régulier de l'espace, le même objet est référencé plusieurs fois (redondance).

Lorsqu'il y a un dépassement de capacité de la page d'indexation, le quadrant est divisé en 4 régions (qu'on appellera suivant leur position géographique NW, SW, NE et NW). L'enregistrement $< id, v_x, x_0 >$ est ajouté tous les nouveaux quadrants traversés. Les figures 2.8 montre un exemple de l'indexation proposée par les auteurs. Dans cet exemple la capacité de stockage d'une page est fixée à 2. L_1, L_2, L_3 et L_4 sont 4 trajectoires d'objets mobiles traversant l'espace d'indexation. Par exemple le quart 2 en bas à gauche de notre espace est traversé par 3 trajectoires (L_1, L_2 et L_3). Il a donc fallu le découper en quatre quadrants (20, 21, 22 et 23) et associer autant de pages du disque (P_{20}, P_{21}, P_{22} et P_{23}).



a. Les trajectoires b. Le quadtree

FIG. 2.8 – Construction du quadtree

Mise à jour

Il s'agit d'une des principales limites de la structure : que faire quand un objet change de direction ? La solution proposée par les auteurs est assez brutale : un index est construit pour une période de temps déterminée ΔT et devient obsolète à l'issue de cette période. Il faut alors reconstruire complètement un nouvel index pour la période suivante, et ainsi de suite.

Recherche

L'index supporte les fenêtrages spatio-temporels : étant donnés les intervalles $[x_{min}, x_{max}]$, $[y_{min}, y_{max}]$, $[t_{min}, t_{max}]$, trouver les objets qui vont passer dans cette fenêtre (NB : l'intervalle temporel doit être compris dans l'intervalle de validité de l'index). On effectue deux fenêtrages sur les deux quadtrees, le premier avec le rectangle $[x_{min}, x_{max}, t_{min}, t_{max}]$, le second avec $[y_{min}, y_{max}, t_{min}, t_{max}]$, l'algorithme consistant à parcourir récursivement, en partant de la racine, les quadrants intersectant la fenêtre.

Résultats

La première remarque faite par les auteurs concerne la taille du stockage de cette indexation. Le nombre de pages du disque nécessaire croit en fonction du nombre d'objets par paliers. La valeur d'un palier est 4 fois plus grande que celle du palier précédent. Cela est dû au fait que les quadrants en cas de dépassement de capacité sont coupés en 4 et que dans les résultats présentés on a supposé une équi-répartition des données (par conséquent le dépassement de capacité à lieu pour tous les quadrants à peu près au même moment).

Les tests effectués pas les auteurs permettent également d'évaluer le taux de remplissage de l'arbre quadtree. Ce taux est une fonction du nombre d'objets que l'on souhaite indexer. La valeur de celui-ci est comprise, d'après les auteurs entre 50% et 90%. Cette valeur de 90% s'explique par le fait que la $i^{ème}$ "vague" de segmentation des quadrants, correspondant au $i^{ème}$ palier cité au paragraphe précédent, commence avant que la $(i-1)^{ème}$ soit finie.

2.3.3 Indexation par R-tree : le TPR-tree [106]

Le TPR-tree (*Time-Parameterized R-tree*) est une extension du R-tree pour indexer les positions actuelles et futures d'objets mobiles, et non les historiques des positions des objets mobiles. L'index s'applique donc à un modèle de données comme celui de [113], et non à ceux qui permettent d'interroger la trajectoire passée. Chaque objet est décrit classiquement par des triplets $[x_0^i, v_0^i, t_0]$, i variant entre 1 et la dimension de l'espace considéré.

Un index est construit à un instant t_0 définissant les positions de référence des objets indexés. On suppose que l'index est valide pendant un intervalle de temps U. A chaque instant de cet intervalle, les requêtes sont de plus autorisées à interroger une période future W. En résumé on peut effectuer des requêtes sur un intervalle maximal $[t_0, t_0 + U + W]$, sachant que l'index ne garantit pas l'exactitude des résultats pour la partie $[t_0 + U, t_0 + U + W]$ qui déborde de son intervalle de validité.

Construction

Dans un arbre R, chaque rectangle à n'importe quel niveau de l'arbre englobe un ensemble d'objets. Cette propriété est essentielle pour garantir la possibilité d'effectuer correctement une recherche. L'idée de base de l'index TPR est que les rectangles doivent évoluer avec le temps de manière à préserver cette propriété à tout instant.

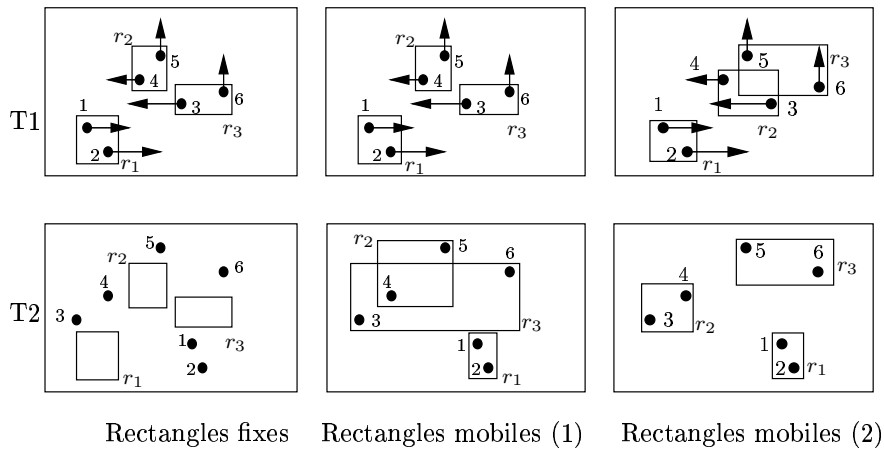


FIG. 2.9 – Rectangles mobiles dans le TPR-tree

La figure 2.9 illustre l'intuition. On considère 6 objets mobiles, et on suppose qu'une page peut contenir au plus trois objets. La partie haute représente les positions à l'instant T1, et la partie basse les mêmes objets à l'instant T2 (T2 > T1).

Clairement (partie gauche de la figure) un rectangle fixe est obsolète à T2 et ne peut plus servir à la recherche. Par exemple r_1 n'englobe plus aucun objet. Si on prend maintenant des rectangles mobiles, le choix des groupes d'objets ne doit plus seulement tenir compte de la proximité dans l'espace, mais aussi de leur position future. Par exemple (partie centrale de la figure 2.9) grouper les objets d'après leur proximité donne un bon résultat pour r_1 qui englobe le groupe $\{1, 2\}$, mais pas pour r_2 et r_3 puisque au sein de ces rectangles les directions des objets sont divergentes, ce qui aboutit à de très larges rectangles au temps T2. En revanche, un regroupement qui tient compte des deux paramètres (partie droite de la figure 2.9) donne de bons résultats.

Pour des raisons de coûts élevés de stockage, on ne peut avoir à tout instant le rectangle englobant minimal. Les auteurs proposent un rectangle englobant qui est minimal pour $t = t_0$ et qui croît suivant les vitesses maximales des objets qu'il contient. Ce qui signifie que la taille d'un rectangle croît toujours, et ne constitue pas le rectangle minimal des objets à tout instant. La figure 2.10 donne un exemple de l'évolution du rectangle englobant entre l'instant t_0 et un instant $t > t_0$. A l'instant de référence t_0, qui correspond donc à l'instant où est chargé l'index, les différents objets a, b, c, d, e, f et g ont une position et une vitesse données.

La vitesse de croissance du rectangle englobant est obtenue quant à elle en cherchant les vitesses minimales et maximales en abscisse et en ordonnée, en valeurs algébriques, des objets qu'il contient. Ainsi par exemple pour les abscisses, la vitesse du bord droit du rectangle se déplacera à la vitesse correspondant à la valeur maximale des différentes composantes horizontales des vitesses des objets contenus, soit dans notre cas $v_a^x(t_0)$, et la vitesse du bord gauche à la valeur minimale, soit ici $v_b^x(t_0)$. L'exemple de la figure 2.10 illustre bien par

ailleurs le caractère non minimal du rectangle englobant pour un instant $t > t_0$.

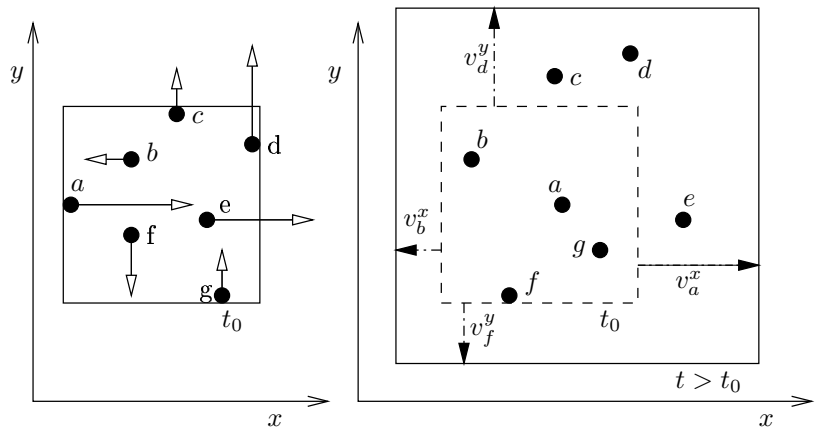


FIG. 2.10 – Croissance d'un rectangle mobile

Les algorithmes d'insertion et suppression d'objets dans un R-tree classique visent à minimiser certains paramètres qui conditionnent la qualité de la structure obtenue. Ces paramètres sont la surface occupée par le rectangle, le recouvrement des rectangles et le périmètre (qui indique si le rectangle est plus ou moins proche d'un carré).

Mise à jour

Une des principales limites de la structure est la mise à jour de l'index. L'index n'est valide que pour un intervalle U, et bien qu'il puisse encore être utilisé ensuite, sa pertinence se détériore avec le temps. Il doit donc être reconstruit périodiquement, ce qui peut être très pénalisant si le nombre d'objets à indexer est élevé.

Recherche

L'index proposé par les auteurs permet de résoudre trois types de requêtes :

1. les requêtes correspondant à un fenêtrage spatial R à un temps t donné ;
2. les requêtes correspondant à un fenêtrage spatial R lors d'un intervalle de temps $[t_{deb}, t_{fin}]$;
3. les requêtes correspondant à un fenêtrage mobile, valant R_{deb} à t_{deb} et R_{fin} à t_{fin}.

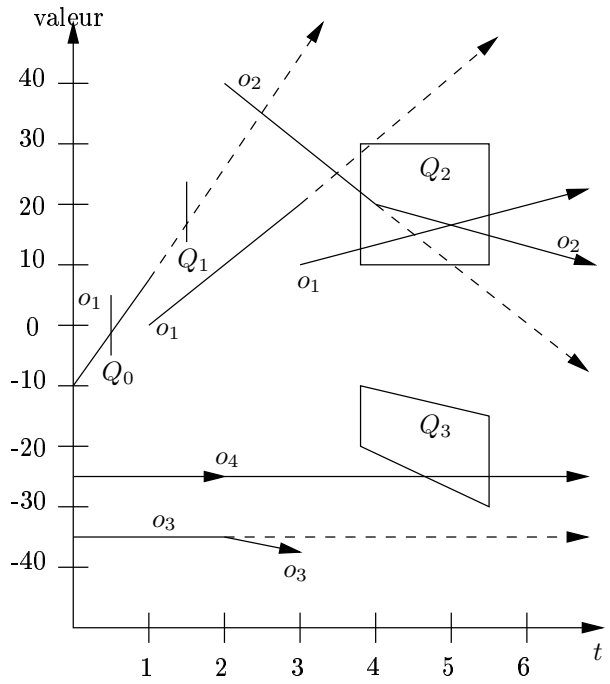


FIG. 2.11 – Exemples de requêtes dans un espace d'une dimension

La figure 2.11 montre des exemples de ces 3 requêtes dans un espace à une dimension. Dans cet exemple, o_x désigne un objet mobile et on suppose que l'on réalise une mise à jour toutes les unités de temps ($U = 1$).

1. $Q0 = ([-5, 5], 0.5)$ est une requête du premier type qui a pour réponse l'objet o_1. $Q_1 = ([15, 25], 1.5)$ est également une requête de type 1, cependant 2 cas se présentent :
   (a) soit la requête a été formulée avant $t = 1$, c'est-à-dire avant la mise à jour, auquel cas la réponse retournée est l'objet o_1 dont on prédit la trajectoire future. Noter que cette requête se situe dans l'intervalle $[t_0 + U, t_0 + U + W]$.
   (b) sinon la réponse est l'ensemble vide puisque la position et la vitesse de l'objet o_1 ont déjà été mises à jour.
2. La requête $Q_2 = ([10, 30], 3.75, 5.5)$ est une requête de type 2. Si la requête a été effectuée à un temps $t < 1$, la réponse est l'ensemble vide puisque la trajectoire prévue pour o_1 ne traverse pas la fenêtre de la requête, et que l'existence de l'objet o_2 n'est pas encore connue. Si la requête est réalisée à $1 < t < 2$ la réponse est o_1 puisqu'après la mise à jour, sa trajectoire prévue intersecte la fenêtre de la requête. L'existence de o_2 n'est toujours pas connue. En revanche si la requête a été réalisée à un instant $t > 2$ alors la réponse retournée est $o1$ et $o2$.

3. Pour Q_4, quel que soit l'instant t où l'on effectue la requête, la réponse retournée est o_4.

Résultats observés

Le premier résultat observé par les auteurs est lié à l'importance de l'intervalle H (cf section précédente). Les meilleurs résultats sont obtenus pour une valeur proche de l'intervalle moyen des mises à jour des objets mobiles.

Ils remarquent également que plus les données contenues dans un rectangle englobant ont un comportement proche, plus l'indexation avec un R-tree ou mieux encore un TPR-tree est performante au niveau du nombre d'entrées-sorties. En effet le rectangle englobant aura une expansion limitée au cours du temps et restera proche du rectangle englobant minimal.

Les performances de l'indexation proposée dépendent également de la taille de l'intervalle temporel qui correspond au futur du moment où est exprimée la requête, que l'on peut interroger. Plus cet intervalle augmente, moins les performances sont bonnes. Néanmoins elles restent supérieures à celle d'un R-tree classique.

Enfin les auteurs notent que le nombre d'opérations d'entrées-sorties ne croît que très lentement en fonction du nombre d'objets représentés. De plus si les performances se dégradent au début avec le temps, elles stagnent ensuite car l'arbre se stabilisera.

2.3.4 Autres techniques d'indexation

Les techniques d'indexation n'ont été étudiées que récemment [125]. Outre les travaux cités, [70] propose des structures optimales en espace et en temps pour l'indexation de la position future des objets mobiles. Les techniques reposent sur la transformation d'une équation $x = v(t-t_0)+x_0$ (v étant la vitesse et x_0 la position initiale) en un point (v, x_0). Ce point peut alors être indexé dans un espace « dual ». La viabilité des structures dépend de plusieurs restrictions qui rendent problématiques leur application pratique. La même remarque vaut pour [5]. La proposition que nous avons détaillée ([106]) nous paraissant la plus convaincante à ce jour. L'indexation des trajectoires (incluant l'historique des positions) a fait également l'objet de quelques travaux, consistant essentiellement à adapter la construction et l'utilisation de l'arbre R pour traiter des polylignes dans un espace 3D [89, 92, 93, 60, 105]. Une tentative - pas très convaincante - d'indexation par quadtree est présentée dans [122]. Dans [119], les auteurs proposent un index, l'arbre MV3R, qui utilise à la fois un arbre R multi-version ([11]) semblable à l'arbre HR et un arbre R 3D construit sur les nœuds-feuilles. L'arbre R multi-version donne de bons résultats pour les requêtes à un instant donné ou pour un intervalle de temps court, alors que l'arbre R 3D est plus adapté aux intervalles longs. Une autre structure intéressante pour l'indexation des trajectoires passées d'objets mobiles est décrite dans [91]. Ce travail exploite également la structure d'arbre R, mais essaye de regrouper les segments d'une même polyligne, ce qui permet de supporter de nouveaux types de requêtes comme par exemple les requêtes dites de « trajectoires » qui prédisent qu'un objet entrera, quittera, traversera, une région.

2.4 Autres axes de recherche des bases d'objets mobiles

La modélisation d'objets mobiles a été jusqu'à récemment fortement influencée par les modèles de données spatiaux existants. La représentation proposée utilisait une modélisation géométrique classique des trajectoires dans l'espace euclidien en étendant les opérateurs et types de données spatiaux pour intégrer la notion du temps. En plus des travaux présentés précédemment, il convient de citer [58, 117, 129, 57, 53, 36, 118, 107]. Dans [53, 36], les auteurs proposent une solution pour le problème bien particulier de la gestion d'objets mobiles se déplaçant sur un réseau contraint (par exemple des véhicules sur un réseau autoroutier). Ainsi, [53] décrit une technique d'évaluation s'appuyant sur les graphes hypercubes pour calculer les plus courts chemins. [118] présente des techniques d'approximation pour évaluer des requêtes sur le présent, le passé et le futur des trajectoires des objets. Les auteurs proposent un histogramme multi-dimensionnel mis à jour de façon incrémentale pour les requêtes portant sur l'instant présent, des historiques pour interroger le passé, et des techniques de prévisions pour les requêtes concernant des positions futures.

L'importance de ce nouvel axe de recherche que sont les bases de données spatio-temporelles a pour conséquence l'émergence de nouvelles problématiques, qui ont pour origine, le plus souvent, certaines applications bien précises. Nous présentons rapidement maintenant cinq de ses problématiques : les requêtes continues, les recherches de plus proches voisins, la détection de comportement, l'incertitude dans les données et la génération de jeux de données.

2.4.1 Les requêtes continues

Une requête *continue* est une requête qui, au lieu d'être évaluée une seule fois à l'instant où elle est soumise au système, s'évalue de manière continue pour un intervalle de temps donné (éventuellement non-borné). Ainsi par exemple lorsque l'on demande tous les objets appartenant à un rectangle R durant les trois prochains jours, le résultat initial (*i.e.*, quand la requête est reçue et évaluée par le système) est sujet à des variations suivant qu'un objet entre ou sort de R.

La notion de « requête continue » est proposée tout d'abord dans [123]. Cette approche ne considère cependant que les bases de données où l'on réalise uniquement des insertions et repose sur une évaluation incrémentale sur les delta relations. La croissance importante du nombre de données disponibles sur Internet a encouragé le développement d'applications de notifications à travers le réseau. Les travaux les plus représentatifs sur la notification sont le système *Active Views* [2], le système NiagaraCQ [27], et les prototypes décrits dans [80, 45].

Dans le domaine des bases de données spatio-temporelles, la notion de requêtes continues apparaît explicitement dans différents travaux [23, 66, 121, 86, 85, 140, 64, 65]. [23] décrit par exemple une architecture web pour réduire le volume et la fréquence des transmissions de données entre le client et le serveur. Une classification suivant leur priorité des différentes

mises à jour est présentée, ainsi que certains critères pour filtrer les mises à jour. [66] propose un système qui indexe les requêtes afin de les recalculer périodiquement le résultat entier de chaque requête. [64] décrit une méthode d'évaluation pour maintenir le résultat d'une requête de plus proches voisins (voir ci-dessous) en trouvant tous les objets dans un rayon donné pouvant affecter le résultat de la requête dans un futur proche. [86, 85, 140] discutent pour leur part d'algorithmes pour évaluer de manière incrémentale un ensemble de requêtes continues, algorithmes se décomposant en trois étapes : (i) jointure avec le cache pour les tuples dernièrement reçus (mises à jour positives), (ii) invalidation de certains résultats (mises à jour négatives) après un certain délai, (iii) jointure avec les données stockées (mises à jour positives et négatives). [65] présente un index basé sur l'arbre B^+, appelé l'arbre B_x, qui permet une évaluation efficace de requêtes continues, fenêtrage ou plus proches voisins. Pour cela ils prennent en compte notamment un agrandissement des régions concernées par les requêtes.

2.4.2 Recherche de plus proches voisins

Un autre type de requêtes, considéré comme très important dans les bases de données d'objets mobiles pour les applications telle que le contrôle de trafic, la gestion d'une flotte, etc, est la recherche du (des k) plus proche(s) voisin(s). La notion de plus proches voisins apparaît d'abord pour les bases de données statiques dans [100], où les auteurs proposent une méthode de calcul des plus proches voisins dans un arbre R. Une alternative s'appuyant sur des cellules de Voronoi a été proposée dans [15]. D'autres travaux [72, 108] proposent de parcourir plusieurs fois l'ensemble des données jusqu'à ce que la distance voulue soit atteinte.

Une première tentative d'appliquer cette notion lorsque les requêtes sont mobiles et non statiques est présentée dans [142], en s'appuyant sur l'arbre R et des diagrammes de Voronoi. Dans [120], les auteurs présentent des requêtes paramétrées par le temps. Le résultat d'une requête est en fait un triplet $< R, T, C >$ où R désigne l'ensemble des objets satisfaisant la requête, T est l'intervalle de temps après lequel le résultat n'est plus valide, et C est l'ensemble des objets qui vont affecter R durant T. Les instants pendant T où le résultat change sont calculés à l'avance, et à chaque fois le résultat est totalement recalculé. [121] propose une technique pour éviter de recalculer de nombreuses fois le résultat. Leur méthode, s'appuyant sur un arbre R, établit en un seul parcours des données les instants où le résultat change dans l'intervalle de temps considéré ainsi que les objets qui entreront dans le résultat.

Dans le domaine des objets mobiles, la recherche de plus proches voisins est une thématique très récente. Les auteurs de [70, 71] proposent une transformation duale afin de trouver l'unique objet qui devient le plus proche d'une requête durant un intervalle de temps défini. D'autres travaux utilisent un arbre TPR présenté dans ce chapitre [13, 97, 63] pour déterminer le résultat. [13], par exemple, propose un algorithme retournant seulement le plus proche voisin pour chaque objet mobile à l'aide d'un arbre TPR, les requêtes envisagées étant assez simples. Récemment, plusieurs travaux [64, 139, 141] présentent des techniques pour évaluer des recherches de plus proches voisins de manière continue. [64] détermine ainsi tous les ob-

jets dans un rayon donné pouvant affecter le résultat de la requête dans un futur proche. [139] décrit un algorithme appelé *SEA-CNN* qui combine une évaluation incrémentale du résultat et une exécution partagée. Chaque requête est affectée à une zone de recherche suivant son dernier résultat. De plus aucune hypothèse sur la trajectoire future n'est considérée. [141] décrit deux approches, une où les objets sont indexés, l'autre où ce sont les requêtes qui sont indexées, basées sur un index « grille » mis à jour dynamiquement. Les résultats sont d'autant meilleurs que le mouvement des objets est localisé. Pour une distribution uniforme, le résultat est obtenu en temps linéaire.

2.4.3 Détection de pattern décrivant un comportement

Le travail présenté dans cette thèse peut être relié à la recherche de « motifs » (ou *patterns*) décrivant un comportement sur un espace de référence éventuellement multi-échelle. Bien que plusieurs structures de représentation multi-échelle aient été proposées récemment pour les bases de données spatiales [98, 116, 143], elles ne considèrent pas les trajectoires d'objets mobiles. [143] implante des schémas d'accès aux données multi-échelle en utilisant des valeurs de priorité aux arêtes des objets géométriques suivant l'échelle. Bien que ce modèle prenne en compte des requêtes portant sur des valeurs d'échelle arbitraire, il ne cherche pas à représenter et interroger les trajectoires. Dans le domaine des bases de données spatio-temporelles, un travail pertinent est l'approche par agrégations et comparaisons proposée par Meratnia et de By dans [84]. Ils présentent une méthode intéressante qui calcule la similarité entre deux trajectoires en s'appuyant sur une représentation raster. Cependant ils ne traitent pas du problème des données multi-échelle, et la partition fixe de l'espace ne peut pas s'adapter à notre partition thématique de l'espace que nous utilisons dans notre modèle.

Dans le domaine des séries temporelles, plusieurs approches [29, 95, 131] pour la reconnaissance de patterns ont été proposées. Elles définissent une distance de similarité pour comparer les séquences de nombres réels, ainsi que des techniques d'indexation et des algorithmes d'évaluation efficaces. Par exemple les auteurs de [49] proposent un stockage externe pour une évaluation en ligne. Cependant leurs travaux s'appuient sur des transformations de données qui n'ont pas de sens dans notre modèle. Dans [131] les auteurs présentent une nouvelle mesure de similarité pour comparer les trajectoires dans l'espace 2D, en utilisant le modèle des plus longues sous-séquences communes (LCSS). Les auteurs axent leurs travaux sur l'étude du bruit qui résulte de mesures de données incorrectes et démontrent, à l'aide d'analyses et d'évaluations expérimentales, la robustesse de leur technique. Bien que l'approche de [131] repose sur des calculs numériques ce qui la rend fortement différente de notre classification thématique de l'espace sous-jacent, l'impact potentiel des erreurs doit être étudié.

D'autres idées intéressantes et assez proches peuvent être trouvées dans d'autres domaines, *e.g.*, les bases de données séquences [6, 110, 82, 96, 103, 102, 114]. Plusieurs langages proposés dans ces travaux étendent les fonctionnalités de SQL avec une certaine variante des expressions régulières à des fins de requêtage, d'agrégation ou de fouille de données.

Dans [110], les auteurs présentent un langage appelé SEQUIN basé sur SQL afin d'interroger des séquences. Dans [96], les séquences sont considérées comme des relations triées où un numéro est affecté à chaque tuple, représentant sa position dans la séquence. Un opérateur de décalage utilisant ce numéro est défini afin de joindre les tuples de la même séquence. Le langage SQL-TS de [103, 102, 101] permet d'exprimer des séquences de prédicats et couvre les patterns répétitifs, avec agrégations ou disjonctions. Le papier décrit une extension des algorithme KMP pour évaluer les requêtes sur de telles séquences de prédicats. D'autres travaux [68, 49, 54, 28, 78] présentent des algorithmes pour interroger et détecter des sous-séquences similaires. Dans [81] les auteurs décrivent des patterns décrivent un algorithme qui permet de trouver rapidement les patterns spatio-temporels périodiques pour des objets se déplaçant sur une carte partitionné.

Les techniques d'évaluation des requêtes reposent souvent sur des algorithmes connus de recherche de patterns. Un problème potentiel associé à toute extension des techniques de détection de patterns est le coût des algorithmes d'évaluation. Plusieurs algorithmes connus ont été proposés pour réaliser efficacement cette recherche [20, 69, 33]. [69], par exemple, garantit une complexité linéaire en temps. D'autres travaux représentatifs sur la recherche de chaînes de caractères sont [7, 31, 112]. Certains papiers traitent également le problème de la recherche de patterns approximatifs [128, 76, 138]. Enfin [9] considère des patterns avec des paramètres et tente de superposer deux chaînes paramétrées en renommant les paramètres.

L'extraction de patterns et leur utilisation pour la modélisation et l'interrogation est un procédé courant dans le domaine des bases de données. On le retrouve par exemple dans les bases de données séquentielles médicales (ADN et protéïnes), les analyses de données économiques et boursières, la recherche d'information dans des textes, etc. Ces patterns, comme n'importe quelles autres données, doivent dès lors être eux-aussi manipulés et gérés. Les caractéristiques spécifiques des patterns rendent les traditionnels SGBD inadaptés à leur gestion, conduisant à l'émergence d'un nouveau domaine de recherche intitulé « gestion de patterns » (*pattern management* en anglais). Par exemple le projet PANDA [50, 16, 10, 17] propose un système de gestion de bases de patterns qui permet d'extraire des patterns à partir de données brutes de manière semi-automatique, patterns mis à jour lors de l'arrivée de nouvelles données, ainsi qu'un langage d'interrogation nommé *PQL* (Pattern Query Language) afin d'exprimer des requêtes à l'aide de patterns. Le projet CINQ [94, 83] propose lui des solutions pour la gestion de certains patterns permettant la fouille de données ainsi qu'un langage basé sur SQL pour l'extraction et la manipulation de ces patterns. Un autre travail pertinent, s'appuyant sur le modèle de PANDA, concernant la gestion de patterns est le système PSYCHO [25]. Les patterns manipulés peuvent être de nature homogène ou hétérogène, mais également définis par l'utilisateur sans s'appuyer sur des résultats obtenus lors d'une fouille des données. Les auteurs décrivent un langage de manipulation des patterns, *PML*, ainsi qu'un langage d'interrogation, *PQL*, qui permet de sélectionner et combiner des patterns et/ou des données.

2.4.4 Incertitude

La position des objets est connue avec une relative incertitude, ce qui soulève des problèmes très spécifiques à ce type d'application. Considérons par exemple le cas d'une autorité portuaire cherchant à retrouver le responsable d'une pollution maritime au pétrole et interrogeant à cette fin un système de surveillance nautique pour connaître tous les bateaux ayant traversé cette zone durant un intervalle donné. Supposons que la trajectoire du bateau responsable de cette pollution est celle de la figure 2.12.

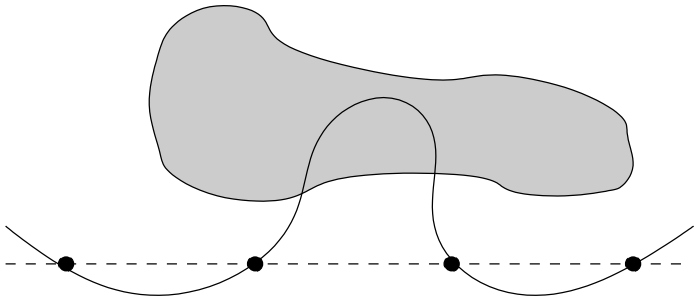


FIG. 2.12 – Une trajectoire indéterminée entre 2 observations.

Sur cette figure, les points représentent les observations et la ligne hachurée, la trajectoire interpolée. On voit qu'avec de telles observations, le crime restera impuni...

Plusieurs modèles ont été proposés pour prendre en compte l'incertitude pour les positions passées, présentes ou futures. Ainsi une adaptation du langage FTL pour prendre en charge les requêtes est décrite dans [134]. Le même article discute des politiques de mise à jour des attributs dynamiques prenant en compte l'incertitude. Les auteurs de [90] considèrent eux le traitement de l'incertitude des positions fournies par le système GPS, ainsi que l'impact sur les techniques de recherche. Le problème a aussi été étudié dans le cadre des bases de données contraintes [73]. Certaines approches proposant un compromis entre le pouvoir de modélisation et l'évaluation efficace des requêtes ont été considérées comme dans [127] où les auteurs introduisent de nouveaux opérateurs spatio-temporels capturant l'incertitude.

2.4.5 Génération de données

La recherche sur les objets mobiles se heurte à la difficulté d'obtenir des jeux de données variés et volumineux. Les études de performances notamment ont besoin d'évaluer les techniques proposées sur des données présentant des profils statistiques bien identifiés, d'où la spécification de générateurs de données dans [104, 126], tous deux disponibles sur le WEB. Dans [21, 22], l'auteur propose un générateur d'objets se déplaçant sur un réseau, déterminant les temps et villes de départ et arrivée, les facteurs extérieur (météorologiques par exemple) influençant le comportement des objets, considérant les vitesses maximales pour

chaque segment, etc. Un dernier travail [124] a été proposé récemment pour les services géo-localisés.

Citons enfin des articles qui reprennent des problématiques issues des bases de données spatiales ou temporelles pour les généraliser au spatio-temporel, comme les langages visuels [18] ou les interfaces graphiques [34]. Inversement le domaine des applications multimédia s'intéresse aux techniques développées en bases de données pour le stockage et la manipulation d'animations vidéos [130, 35].

Suite à cet état de l'art, nous avons eu une réflexion sur un problème bien particulier où les utilisateurs soumettent des requêtes afin de suivre des objets mobiles et veulent être alertés lorsqu'un objet entre dans une zone de requête. Contrairement aux modèles présentés dans cette section, l'aspect continu d'une trajectoire est alors abandonné au profit d'une vision *événementielle* de cette trajectoire, à savoir nous observons les *entrées/sorties* des objets dans les zones de l'espace partitionné. De plus les requêtes considérées sont des requêtes continues, puisque l'utilisateur s'abonne/désabonne à une requête comme on le ferait pour un service de publication sur internet. Dès lors les modélisations et techniques d'indexation précédentes ne sont pas adaptées à de tels types de données. Cette problématique nous a alors conduit à une première réflexion et à une proposition de modélisation et d'interrogation présentées à la section suivante.

2.5 Architecture pour requêtes continues

Le travail décrit ici a été présenté dans [38]. Nous nous plaçons dans le cadre des applications Web visant à :

i) *intégrer* des informations fournies par des serveurs GPS concernant des trajectoires d'un grand nombre d'objets mobiles ;

ii) *notifier* les différents utilisateurs ayant soumis des requêtes afin de suivre ces objets mobiles.

Des requêtes typiques de telles applications sont par exemple :

- Me prévenir chaque fois qu'un camion pénètre dans cette zone durant les deux prochains jours.

- Montrer tous les avions aux environs de l'aéroport durant les deux prochaines heures.

Ces requêtes n'ont pas un résultat fixé dans le temps et impliquent une maintenance pratiquement continue du résultat afin de prendre en compte la position actuelle des objets. Nous renvoyons à la section 2.4 pour un état de l'art sur les *requêtes continues*. Noter que nous nous intéresserons ici uniquement à des requêtes de *fenêtrage*.

2.5.1 Requêtes continues

Nous adoptons pour ce modèle des hypothèses simples et répandues dans le domaine des bases de données spatio-temporelles. Une *trajectoire* est une fonction continue par morceaux qui peut-être représentée par un ensemble de segments dans l'espace 3D connectés deux à deux. Elle peut donc être vue comme un ensemble infini de points dans $\mathbb{R}^3$ dont la représentation finie est obtenue à partir d'un échantillon P de positions $P_i(t, x, y)$ fournies par les serveurs GPS.

La représentation d'une trajectoire que nous utiliserons par la suite est la suivante : une liste

$$< [I_1, f_1^x, f_1^y], \ldots [I_n, f_n^x, f_n^y] >$$

où (i) chaque I_i est un intervalle de temps, (ii) les intervalles $\{I_i, i \in [1, n]\}$ sont distincts deux à deux et (iii) chaque f_i^x (resp. f_i^y) est une fonction linéaire définie seulement sur I_i, du temps t vers l'abscisse x (resp. l'ordonnée y). La position d'un objet à n'importe quel instant t dans I_i est un point $(f_i^x(t), f_i^y(t))$, noté $loc_i(t)$.

Le résultat d'une requête peut être constituée d'intervalles de temps non-adjacent si un objet qui a quitté la région de la requête est ensuite revenu dans cette région. C'est le cas par exemple dans la figure 2.13.

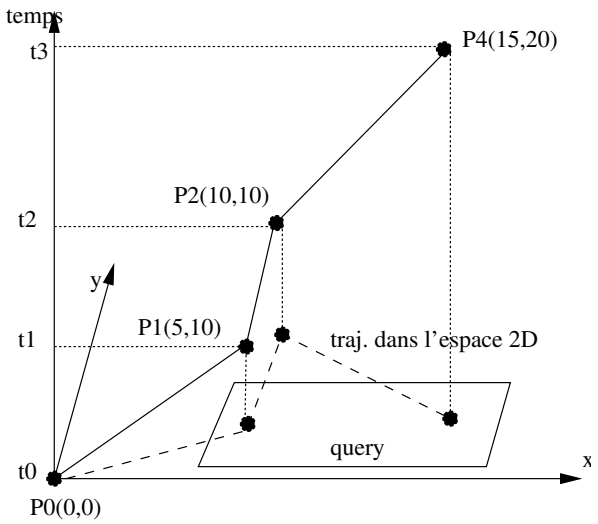


FIG. 2.13 – Une trajectoire

Nous désignons par $\mathcal{O}$ l'ensemble des objets mobiles, et par $\mathcal{T}$ l'ensemble de toutes les trajectoires. Une *relation objet mobile* (ROM) R est une paire $(\mathcal{O}_R, traj_R)$ où $\mathcal{O}_R$ est un sous-ensemble fini de $\mathcal{O}$ et $traj_R$ une application de $\mathcal{O}_R$ vers $\mathcal{T}$.

Dans ce modèle, le définition d'un événement est la suivante :

Définition 1 (Événement). *Un événement est un tuple $< id, t, loc, vitesse, direction >$ où id désigne l'identifiant d'un objet mobile o, t l'instant où o a été localisé, et loc, vitesse et direction caractérisent le mouvement de l'objet à l'instant de l'observation.*

Comme nous l'avons annoncé plus haut, nous nous restreignons à l'étude des requêtes de *fenêtrage*, de la forme $q(rect, I)$ sur l'espace où $rect$ est un rectangle sur la carte 2D considérée, et I est un intervalle de temps débutant à I_{min}, lorsque la requête est soumise au système, et terminant à I_{max}. La sémantique de nos requêtes est la suivante :

Définition 2 (Résultat d'une requête). *Soit $M(\mathcal{O}_M, traj_M)$ une relation objet mobile dans la base de données. L'*ensemble résultat *d'une requête $q(rect, I)$ sur M est une relation objet mobile $RES(\mathcal{O}_q, traj_q)$ telle que (i) $traj_q$ est l'application définie par $traj_q(o) = traj_M(o) \cap [rect \times I]$ et (ii) $\mathcal{O}_q$ est l'ensemble des objets o de $\mathcal{O}_M$ tels que $traj_q(o) \neq \emptyset$*

Ainsi la trajectoire d'un objet peut se traduire au niveau de l'ensemble résultat d'une requête, par un ensemble de segments non-connectés, avec des intervalles de temps disjoints (typiquement lorsqu'un objet sort d'une requête avant d'y entrer à nouveau). Le système doit donc surveiller chaque fois qu'un objet entre ou sort d'un rectangle d'une requête afin de notifier les utilisateurs si le résultat des requêtes qu'ils ont soumises a évolué.

Nous avons pris comme hypothèse que la vitesse des objets est petite en comparaison de la taille de la région de fenêtrage et donc que le résultat de la requête change lentement. Le fait de connaître la position, la vitesse et la direction, nous permet d'anticiper les événements et de les calculer à l'avance, afin de préparer le système pour la notification des utilisateurs. Ces « anticipations » doivent être actualisées chaque fois qu'une mise à jour modifie les caractéristiques du mouvement de l'objet. Ainsi sur la figure 2.14, les caractéristiques du mouvement de o nous permettent de savoir que o va appartenir successivement aux résultats de q_1, q_2 puis q_3, et même de connaître l'instant d'entrée et de sortie dans chaque rectangle. Une mise à jour des caractéristiques du mouvement de o peut rendre obsolètes toutes ces observations et donc les calculs effectués.

Nous adoptons par conséquent une évaluation incrémentale du résultat de la requête, avec les étapes suivantes :

1. quand la requête $q(rect, I)$ est enregistrée (*i.e.*, à l'instant I_{min}), un résultat initial res_0^q est créé avec tous les objets se trouvant dans $rect$ à cet instant ;

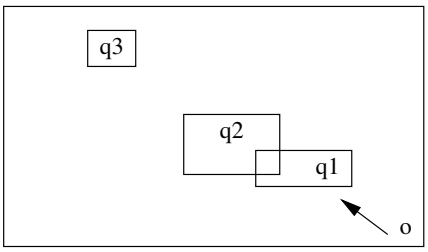


FIG. 2.14 – Une trajectoire qui peut satisfaire trois requêtes

2. le résultat (présent et anticipé) doit être mis à jour chaque fois qu'un nouvel événement indique que les caractéristiques du mouvement d'un objet ont changé, ou chaque fois qu'un objet entre ou sort du rectangle de la requête.

Il faut souligner que puisqu'on a à gérer plusieurs mises à jour de la trajectoire, une telle anticipation des mouvements et donc des résultats futurs d'une requête est difficile. En effet lors d'une mise à jour des caractéristiques du mouvement d'un objet, nous devons retrouver toutes les requêtes pour lesquelles le résultat a été anticipé.

Puisqu'un objet (et par conséquent les événements associés à cet objet) est partagé éventuellement par beaucoup de requêtes, et qu'une requête est concernée probablement par un grand nombre d'événements à chaque instant, il a fallu réfléchir à une association judicieuse entre les requêtes et les événements afin d'obtenir des performances satisfaisantes pour une application de notification web.

2.5.2 Associer un événement à une requête

Afin d'associer les événements et les requêtes, nous proposons :

i) une structure de données spatiales qui permet d'indexer un ensemble de requêtes et permet de déterminer rapidement quelles requêtes sont affectées par un événement ;

ii) un traitement uniforme des deux types d'événements identifiés plus haut (mise à jour des paramètres du mouvement et entrée/sortie d'une zone) ;

Dans une approche traditionnelle, les données sont indexées et l'évaluation d'une requête repose sur l'index pour retourner le résultat. Cependant dans notre modèle nous gérons un grand ensemble de requêtes persistantes sur une base de données spatio-temporelles. Il apparaît donc plus difficile de traiter les objets mobiles, à cause de leurs nombreuses mises à jour et leur mouvement continu, que des requêtes qui sont des rectangles fixes, sans mises à jour. Nous avons eu alors l'idée de définir une structure qui indexe les requêtes et non les objets mobiles, afin de déterminer toutes les requêtes affectées par un événement. La structure

de données est basée sur une *grille fixe* [14], un index spatial simple qui décompose l'espace de recherche en cellules rectangulaires de taille fixe. Chaque cellule couvre un rectangle *rect* et une étiquette l. Nous utilisons cette grille pour notre évaluation de la façon suivante :

- **indexation des requêtes**. L'ensemble des requêtes peut-être vu comme un ensemble de rectangle et peut donc être indexé avec la grille de taille fixe. La technique est classique : pour chaque requête q nous calculons l'ensemble des étiquettes $l_1^q, l_2^q, \ldots l_n^q$ des cellules intersectées par $q.rect$. Nous obtenons un ensemble $\mathcal{Q}$ de paires (q, l) où q est une requête et l une de ces étiquettes ;

- **partition des événements**. Nous assignons à chaque cellule c la relation objet mobile $c.O$ contenant tous les objets qui traversent le rectangle de la cellule, avec le fragment de leur trajectoire qui intersecte $c.rect$. Ce fragment débute à l'instant où l'objet entre dans la cellule et se termine à l'instant t_{quit}, où l'objet est *supposé* sortir de la cellule.

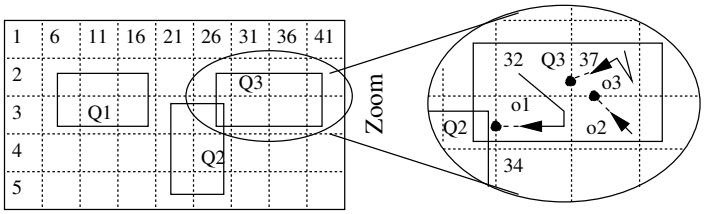


FIG. 2.15 – La grille avec des requêtes et des objets mobiles

Cette structure est illustrée sur la figure 2.15. La grille est un ensemble de $5 \times 9 = 45$ cellules. Trois requêtes sont représentées : $Q1$ associée aux étiquettes $\{7, 8, 12, 13, 17, 18\}$, $Q2$ associée à $\{23, 24, 25, 28, 29, 30\}$ et $Q3$ à $\{27, 28, 32, 33, 37, 38, 42, 43\}$. Toutes les paires (Q, l) peuvent être insérées dans une liste et indexées suivant l'identifiant de la requête.

La partie droite de la figure montre les objets mobiles dans la région contenant les requêtes Q_3 et, partiellement, Q_2. Grâce à leur position, vitesse et direction, on peut facilement connaître l'instant (représenté par un point noir) où ils quitteront leur cellule et enregistrer ces événements dans une relation associée à cette cellule. Considérons par exemple l'objet o_1 de la figure 2.15. Il va quitter prochainement la cellule 33 et entrer dans la cellule 28, devenant par là-même un élément du résultat de Q_2. Pour enregistrer ceci, nous utilisons une liste $\mathcal{E}$ stockant des tuples $< t_{sortie}, o, l_{actuel}, l_{suivant} >$. Chaque tuple précise que l'objet o est actuellement dans la cellule l_{actuel} et qu'il la quittera au temps t_{sortie} pour entrer dans la cellule $l_{suivant}$. Par exemple la liste associée avec l'instance de la figure 2.15 est $< [t_1, o_1, 33, 28], [t_2, o_2, 38, 37], [t_3, o_3, 37, 32] >$, avec $t_1 < t_2 < t_3$.

La liste $\mathcal{E}$ doit supporter des fonctions $insertion(t, o, l_{actuel}, l_{suivant})$, $suppression(t, o)$, $suivant()$ retournant le tuple avec la valeur de temps minimal.

De plus, à chaque fois qu'un événement $evt < id, t, loc, vitesse, direction >$ est reçu d'un serveur GPS, nous trouvons l'unique cellule c telle que le rectangle $c.rect$ contient loc. Soit o l'objet identifié par $evt.id$ et t_{ancien} l'instant où o devait quitter $c.rect$. Nous calculons t_{sortie} quand o devrait quitter $c.rect$ avec les nouvelles caractéristiques reçues de son mouvement, ainsi que la prochaine cellule $l_{suivant}$. On remplace alors l'ancien événement (t_{ancien}, o) dans la liste $\mathcal{E}$ par ce nouvel événement. Si o appartenait déjà à l'instance de la relation objet mobile de la cellule, $c.O$, le nouveau segment est ajouté à sa trajectoire, sinon o est simplement ajouté à $c.O$.

Avec une telle structure, l'évaluation des notifications à transmettre à un utilisateur est triviale : on cherche dans $\mathcal{Q}$ toutes les étiquettes $l_1^q, l_2^q, \ldots l_n^q$ associées à la requête identifiée par id puis on accède à chacune des cellules identifiées par l_i^q. Chacune de ces cellules possède une liste $\mathcal{E}_i$ contenant les événements à transmettre à l'utilisateur.

2.5.3 Réflexion

Ce premier travail nous a permis de réfléchir sur des aspects peu conventionnels dans les bases de données spatio-temporelles, notamment :

- des mises à jour des résultats des requêtes suivant la détection d'une entrée ou sortie d'un objet dans une zone ;
- l'importance d'une indexation des requêtes qui, contrairement aux objets, ne sont pas continuellement mises à jour ;
- l'importance d'une évaluation optimisée en espace et en temps pour des *requêtes continues*.

Dans notre modèle, si l'on ne tient pas compte des événements *anticipés*, le résultat d'une requête n'évolue que lorsqu'un objet entre ou sort du rectangle de la requête. Dans ce cas la position précise de l'objet ne nous est, a priori, d'aucune utilité. De même l'aspect continu de la trajectoire ne semble pas intervenir dans l'évaluation du résultat. Si l'on décide de suivre des objets, on peut alors imaginer qu'après être sorti d'une certaine zone Q_1, on souhaiterait savoir s'il entre dans une autre zone choisie Q_2. En itérant, on obtient une interrogation plus complexe formée d'un ensemble de zones $< Q_1, Q_2, \ldots, Q_n >$. On peut exiger que ces zones soient contiguës, laissant une certaine liberté à l'objet entre deux zones de cette requête, ou non. De plus ces zones peuvent être de taille variable, voire même incluses l'une dans l'autre. D'autre part nous avons considéré ici un découpage en grille de cellules de taille fixe, thématiquement neutre. Il apparaît plus pertinent de découper suivant l'intérêt porté à l'espace de référence, d'autant plus que les zones des requêtes ont le plus souvent une caractéristique thématique bien définie. On verra par la suite que cette idée constitue la base de mon travail réalisé lors de ma thèse.

Plus concrètement, nous avons envisagé :

- le cas d'un espace partitionné avec un seul niveau d'échelle, où les requêtes sont une succession de zones contiguës ou non, voire indéterminées (chapitres 3 et 4) ;

- le cas d'un espace partitionné à plusieurs niveaux d'échelle, où les requêtes sont une succession de zones contiguës (chapitre 5).

Enfin, ce travail nous a permis de réaliser qu'il est utopique de parler de requêtes continues sans proposer une structure de données garantissant une évaluation d'un grand nombre d'objets mobiles et de requêtes en garantissant des besoins en espace mémoire et en temps *réalistes*. Cet aspect a été considéré dans chaque étape de la thèse, principalement dans le travail présenté au chapitre 5.

Chapitre 3

Patterns de mobilité

Nous présentons dans ce chapitre un modèle de données pour le suivi d'objets mobiles et proposons une technique d'évaluation du résultat d'une requête portant sur une base de données gérant de tels objets. Le modèle repose sur une vision *discrète* de l'espace spatio-temporel, où l'espace 2D et la composante temporelle sont partitionnés respectivement en un ensemble fini de zones définies par l'utilisateur et des intervalles de temps constants.

Nous définissons un langage d'interrogation générique pour trouver les objets qui satisfont des *patterns de mobilité* décrivant des séquences de déplacements. Nous identifions également un ensemble de restrictions à ce langage dans le but d'exprimer uniquement des requêtes *déterministes* pour lesquelles nous proposons des techniques d'évaluation afin de maintenir leur résultat.

Ce modèle est conceptuellement simple et constitue une solution pratique et efficace pour le problème du suivi continu d'objets mobiles à l'aide de requêtes de séquences. Il a été publié dans [39, 41].

3.1 Introduction

Ce chapitre propose un modèle de données pour représenter les trajectoires comme des séquences de déplacements dans un espace spatio-temporel discret et étudie les langages pour interroger de telles séquences d'événements. Concrètement, les langages que je considère reposent sur des patterns de mobilité pour exprimer les opérations de recherche. Je me concentre sur la famille de patterns qui satisfont les propriétés suivantes :

- je n'ai pas besoin des déplacements déjà effectués d'un objet o pour déterminer si o satisfait ou non un pattern donné ;

- la quantité de mémoire nécessaire pour maintenir le résultat d'une requête est faible.

Ces propriétés sont essentielles dans le contexte des requêtes continues puisqu'elles garantissent qu'un grand nombre de requêtes peut être évalué efficacement avec des ressources

limitées en considérant simplement le dernier événement associé à l'objet. Je définis une classe de requêtes qui propose un compromis approprié entre le pouvoir d'expression et la satisfaction de ces exigences.

Dans la suite de ce chapitre je développe d'abord une présentation informelle de mon travail (section 3.2) avec des exemples de patterns de mobilité qui illustrent l'intuition derrière le modèle et son intérêt pratique du point de vue de l'utilisateur. Le modèle de données est présenté dans la section 3.3. Je présente ensuite un fragment du langage offrant un pouvoir d'expression moindre mais garantissant un faible besoin d'espace mémoire nécessaire à l'évaluation dans la section 3.4.

3.2 Patterns de mobilité

Dans notre modèle la localisation des objets est projetée dans un ensemble de régions qui partitionnent l'espace auquel nous nous intéressons. La partition considérée est fortement liée à l'application et à l'interprétation thématique de l'espace dans celle-ci, et résulte d'un mécanisme d'analyse spatial classique [77, 99]. Le problème de définir quelle partition est pertinente ou non ne rentre pas dans les objectifs du présent travail. En ce qui nous concerne il suffit de relever pour chaque objet se déplaçant dans $\mathcal{M}$, la séquence des zones distinctes successivement traversées : nos patterns sont construits à l'aide de telles séquences. Il faut souligner que, puisque la partition constitue la base pour la définition de nos patterns, nous obtenons des classifications assez distinctes des trajectoires suivant l'intérêt thématique. Les trajectoires et leur classification sont très sensibles à la partition de l'espace qui dépend de l'intérêt thématique. Une partition « thématiquement neutre » alternative consiste à construire des patterns sur une partition en grille de cellules de taille égale, comme discuté dans [84].

L'espace $\mathcal{M}$ considéré pour ce modèle n'est pas un espace dense sur $\mathbb{R}^2$ mais un espace partitionné en zones identifiées de manière unique par une étiquette. La figure 3.1 montre un tel espace partitionné.

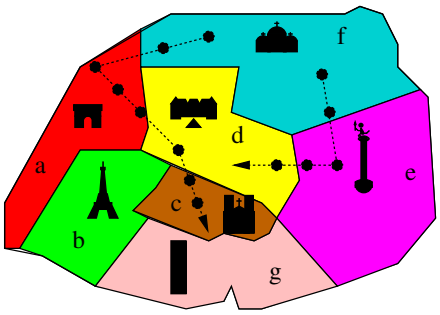


FIG. 3.1 – Objets se déplaçant sur un espace partitionné

L'ensemble de ces étiquettes forme l'alphabet $\Sigma_{\mathcal{M}}$, que nous écrirons pour plus de simplicité Σ. Pour l'exemple de la figure 3.1, Σ vaut ainsi $\{a, b, c, d, e, f, g\}$. Dans cet espace nous suivons un ensemble d'objets $\mathcal{O}$ équipés d'un outil de localisation tel qu'un serveur GPS qui envoie périodiquement leur position ainsi qu'une estampille.

Nous caractérisons la trajectoire d'un objet o par les étiquettes des zones successives traversées par o.

Considérons une application visant à classifier et analyser le trafic à l'aide des requêtes suivantes :

- Trouver tous les objets qui ont voyagé de `a` à `f`, sont restés plus de 10 minutes dans `f` et sont ensuite passés de `f` en `c`.

- Trouver tous les objets voyageant de `f` à `d` ou `c` via une troisième zone différente.

- Trouver tous les objets qui ont quitté une certaine zone, sont allés en `c` et sont ensuite revenus dans leur zone de départ.

Ces exemples ont tous en commun de spécifier la succession de zones traversées par un objet durant sa pérégrination, ainsi que des contraintes temporelles sur le temps passé dans chaque zone. Nous nommons ce type de spécification un *pattern de mobilité*, ou plus concisément *pattern*. L'approche basée sur la géométrie et sur une représentation continue des trajectoires, utilisée dans la majeure partie des modèles de données spatio-temporels jusqu'à présent, n'est pas réellement adaptée à notre problématique, en particulier aux types de requêtes auxquelles nous nous intéressons.

En effet, contrairement à ces modèles nous n'avons plus besoin d'avoir recours à des mécanismes d'interpolation ou d'extrapolation pour déterminer la position précise d'un objet dans une zone donnée à chaque instant, puisque la succession d'événements GPS est suffisante pour évaluer nos patterns. Chaque événement GPS fournit la position d'un objet et cela suffit pour calculer la zone dans lequel se trouve l'objet quand l'événement est reçu. Il est ainsi aisé de construire une représentation discrète de la trajectoire d'un objet comme une séquence de la forme $z_1\{T_1\}.z_2\{T_2\}.\cdots.z_n\{T_n\}$ comprenant la liste $z_1, z_2, \ldots, z_n$ des étiquettes des zones successives traversées ainsi que le temps passé dans chaque zone. Ainsi par exemple, pour la trajectoire de l'objet o_1 dans la figure 3.1, si on suppose que chaque point correspond à un événement GPS et donc que o_1 a passé 2 minutes en `f`, 4 minutes en `a`, 3 minutes en `d` et 6 minutes en `c`, cette trajectoire sera représentée dans notre modèle par la séquence `[f{2}.a{4}.d{3}.c{6}]`. Lorsqu'un nouvel événement est reçu, *i.e.* une nouvelle étiquette de zone, soit la dernière composante temporelle est incrémentée si l'objet reste dans la même zone, soit un nouveau symbole est ajouté à la fin de la séquence s'il change de zone.

Nous définissons pour cela une fonction *Zone* qui pour une localisation l donnée retourne la zone z contenant l. Le partitionnement de l'espace nous garantit l'existence et l'unicité de

z.

Une première utilisation de cette fonction *Zone* consiste en la mise à jour d'une trajectoire quand un nouvel événement GPS est reçu, fournissant une nouvelle localisation (x, y) pour un objet o. Le pseudo-code de la procédure MajTraj résume comment la trajectoire de o de longueur n est actualisée.

MajTraj(o, x, y)
début
  // Calcule la zone actuelle, $\mathcal{Z}$
  $\mathcal{Z}$ = *PointDansPolygone*($\mathcal{M}$, *x, y*)
  // Récupère l'étiquette de $\mathcal{Z}$
  z = *étiquette*($\mathcal{Z}$)
  // Compare z au dernier symbole de la trajectoire $z_n\{T_n\}$
  // si égalité, alors on incrémente le temps passé dans la zone de 1
  si $z = z_n$ **alors** $T_n = T_n + 1$
    // si différent on concatène z à la trajectoire avec un temps de 1
    sinon $o.traj = (o.traj).z\{1\}$
  finsi
fin

3.2.1 Requêtes

Les patterns de mobilité sont utilisés pour interroger cette base de données et retourner les objets qui satisfont ce pattern. Il s'agit essentiellement de séquences d'expressions construites à partir d'un petit ensemble d'opérateurs et comprenant des étiquettes de Σ et/ou des variables appartenant à un ensemble $\mathcal{V}$. Les variables peuvent être instanciées avec n'importe quelle étiquette de la carte et permettent de désigner des classes complexes de trajectoires à l'aide d'un langage compact mais qui offre un grand pouvoir d'expression. La présentation qui suit est essentiellement basée sur des exemples et a pour objectif d'illustrer les principales caractéristiques du langage.

Nous supposons, comme premier exemple, que nous voulons retourner tous les objets qui sont partis de `b`, sont allés en `e` en traversant `c`, `d` ou `f` (voir figure 3.1) et sont finalement allés de `e` en `a` via la *même* zone. On exprime cela dans notre langage comme suit :

```
start_at b, follow @x, follow e repeat, follow @x.a
```

Ce pattern de mobilité est construit à l'aide de deux opérateurs : `start_at` et `follow`, ce dernier étant optionnellement accompagné de l'opérateur `repeat`. Deux autres opérateurs `now_at` et `roam` seront présentés dans d'autres exemples.

Une zone est représentée par son étiquette (ici `a`, `b`, `e`) ou par une variable (ici `@x`) si elle est laissée indéterminée par l'utilisateur. Une variable est ici nécessaire pour représenter la zone où un objet se déplace après avoir quitté `b` et pour exprimer que l'objet doit revenir en `a` via la *même* zone. Les étiquettes et les variables peuvent être concaténées (par exemple

on a ici `@x.a`) pour décrire un chemin et les étiquettes (mais pas les variables) peuvent être regroupées en ensemble (par exemple, `{a, b}`) pour décrire une union de zones.

Les utilisateurs peuvent spécifier un opérateur `start_at` pour indiquer à quel endroit une trajectoire doit commencer. Un autre opérateur, `now_at`, indique où un objet doit se trouver à l'instant présent. Sans ces indications, le pattern de mobilité doit satisfaire n'importe quelle sous-partie de la trajectoire. La définition choisie pour l'opérateur `now_at` implique qu'il ne peut être utilisé que pour les requêtes évaluées en continu, sur un flot de données. Il n'a en effet aucun sens pour les requêtes portant sur des trajectoires historisées.

Un opérateur `follow` décrit un déplacement. La durée de séjour, soit un nombre fixé d'unités de temps, soit un temps non déterminé mais au moins 1 unité de temps, dans la zone apparaissant après un `follow` peut être spécifiée par l'opérateur `repeat`. Dans notre exemple `follow e repeat` signifie que l'objet qui satisfait le pattern s'est déplacé en `e`, est resté en `e` pour une période non définie avant de revenir en `a`.

Intuitivement, un objet *o satisfait* le pattern P ci-dessus si les conditions suivantes sont vérifiées :

1. il existe une instanciation des variables de P telle que le pattern valué soit une sous-chaîne de la trajectoire de l'objet ;

2. si le pattern débute par l'opérateur `start_at` (resp. se termine par l'opérateur `now_at`), le pattern valué est un préfixe (resp. suffixe) de la trajectoire ;

3. le temps passé dans chaque zone satisfait la contrainte temporelle exprimée dans le pattern.

Ainsi, par exemple, un objet dont la trajectoire est représentée par la séquence d'étiquettes de zones `[b.d.e.d.a.c.f]` (nous omettons l'information temporelle dans un souci de simplification) satisfait le pattern ci-dessus où la valeur de la variable `@x` est fixée à l'étiquette `d`. La chaîne de caractères en gras est une valuation de P, préfixe de la représentation discrète de la trajectoire.

Plus généralement, les patterns de mobilité désignent des classes de trajectoires. Un objet o satisfait un pattern P si une sous-chaîne de la représentation discrète de sa trajectoire appartient à la classe désignée par P. Je donnerai par la suite des définitions plus précises de la syntaxe et de la signification des patterns. Étudions d'abord encore quelques exemples intuitifs qui seront utilisés tout au long de ce chapitre pour illustrer le modèle.

– **Q1.** Trouver tous les objets voyageant de `a` en `f` puis ensuite de `f` en `c` en 10 minutes.

```
start_at a, follow f, roam 10, follow c
```

Si la trajectoire d'un objet o satisfait le pattern, alors : `start_at a` signifie que o démarre de `a` ; `follow f` signifie que o quitte `a` pour `f` ; `roam 10` indique que l'objet s'est déplacé dans n'importe quelle zone de la carte pendant 10 minutes. L'opérateur `roam` est neutre et ne restreint jamais en aucune façon la trajectoire d'un objet : en quittant `f`, un objet peut se promener dans différentes zones avant d'atteindre `c` et appartenir au résultat de la requête.

- **Q2.** Trouver tous les objets qui sont restés tout le temps en `a` ou `b` sauf une minute où ils ont été dans une troisième zone distincte de `a` et `b`.

```
start_at {a,b}, follow {a,b} repeat,
          follow @x, follow {a,b} repeat;
@x != a, @x!= b
```

Cet exemple a recours à une variable `@x` qui exprime un déplacement de `a` ou `b` vers n'importe quelle autre zone de la carte. Il est possible d'exprimer des contraintes supplémentaires sur les instanciations possibles des variables en utilisant des égalités ou des inégalités. L'utilisateur demande dans cet exemple que l'objet quitte `a` ou `b` pour une troisième zone.

- **Q3.** Trouver tous les objets qui sont passés par `f` pour aller dans une autre zone, sont allés ensuite en `d` ou `c` et sont revenus en `f` en passant par la même zone qu'à l'aller.

```
follow f.@x, follow {d,c}, follow @x.f;
@x != f
```

On rappelle ici qu'une variable conserve sa valeur une fois instanciée.

Ce langage offre un nouveau moyen d'exprimer des requêtes qui seraient assez difficilement, voire de façon impossible, exprimables ou calculables à l'aide d'un langage d'interrogation spatio-temporel basé sur la géométrie, notamment à cause des nombreuses jointures spatiales nécessaires pour déterminer la zone dans laquelle un objet est situé à un instant donné et à cause de l'aspect séquentiel de nos requêtes. Nous donnons maintenant la grammaire du langage utilisateur. Nous définissons tout d'abord les types suivants :

`ZONE` $\rightarrow x$, `avec` $x \in \Sigma$
`ZONESET` $\rightarrow \{x_1, \ldots, x_k\}$ `avec,` $k \geq 1$, `et` $\forall i, x_i \in \Sigma$
`VAR` $\rightarrow v$, `avec` $v \in \mathcal{V}$
`ZONESTRING` $\rightarrow x_1 \ldots x_k$, `avec` $k \geq 1$, `and` $\forall i, x_i \in \Sigma$
`VARSTRING` $\rightarrow x_1 \ldots x_k$, `avec` $k \geq 1$, `and` $\forall i, x_i \in \Sigma \cup \mathcal{V}$

Un pattern de mobilité est construit avec la grammaire suivante, où `INT` désigne un élément entier positif.

```
START_BLOC  → start_at ZONESET | start_at VAR | ε
NOW_BLOC    → now_at ZONESET | now_at VAR | ε
BLOC         → follow ZONESET REPEAT_BLOC BLOC
                          | follow VARSTRING REPEAT_BLOC BLOC
                          | roam INT | roam | ε
REPEAT_BLOC → repeat INT | repeat | ε

QUERY        → START_BLOC BLOC NOW_BLOC
```

3.2.2 Requêtes continues

Le langage de requête décrit ci-dessus peut être utilisé pour analyser et classer des trajectoires d'objets stockées dans une base de données. Il permet également d'exprimer des *requêtes continues*. Cependant le caractère particulier de ces requêtes nous a conduit à certaines restrictions. En effet un objet peut être ajouté ou supprimé du résultat d'une requête durant la période de validité de la requête, suivant ses déplacements les plus récents. D'après notre langage cela signifie que les patterns de mobilités pertinents pour une évaluation continue sont ceux se terminant par l'opérateur `now_at` : seuls les objets dont le suffixe de la trajectoire satisfait le pattern, à l'instant présent, sont inclus dans le résultat de la requête.

Comme premier exemple supposons que nous souhaitons trouver tous les objets qui sont passés par `a` ou `b`, se sont déplacés en `e` en traversant soit `c`, `d` ou `g` (voir figure 3.1) et sont revenus finalement de `e` en `a` où ils se trouvent actuellement, en traversant la *même* zone. Cette requête s'exprime dans notre langage par :

```
follow {a,b}, follow @x repeat,
            follow e repeat, follow @x repeat, now_at a;
@x != f
```

L'évaluation continue vise à superposer le pattern avec le suffixe de la trajectoire de l'objet. Ce suffixe représente ici la partie la plus récente du flot continu d'événements GPS. Puisque la représentation de la trajectoire évolue lorsque de nouveaux événements sont reçus, la superposition doit être réévaluée périodiquement – presque continuement. Notre objectif consiste à réaliser cette évaluation en consommant un espace et un temps minimaux. Deux critères essentiels ont été considérés pour mesurer la faisabilité et l'efficacité de cette évaluation :

1. avons nous besoin de considérer les déplacements réalisés auparavant par les objets pour évaluer une requête ?

2. quelle est la quantité de mémoire nécessaire pour maintenir le résultat d'une requête ?

Considérons tout d'abord le cas des patterns sans variables. L'évaluation d'un pattern P est alors une opération standard qui nécessite de construire l'automate à états finis (FA) qui reconnaît le langage $\Sigma^*.L_P$ [62], où L_P est le langage régulier défini par P et Σ est l'ensemble des étiquettes de la carte.

Dans le cas général, le FA associé à une expression régulière est non-déterministe. Dans ce cas un objet o doit donc être associé à plusieurs états dans l'automate à un instant donné et cette liste d'états pour o doit être conservée. Cette liste peut-être représentée par un masque de bits, un bit pour chaque état de l'automate. La valeur 1 (resp. 0) pour un bit signifiant que o est (resp. n'est pas) dans l'état associé. Ceci donne une structure assez compacte. Ainsi, par exemple, pour un pattern avec 8 symboles, un masque de 8 bits (un octet) doit être enregistré pour chaque objet. On peut alors suivre une base d'un million d'objets avec seulement un mégabyte en mémoire centrale.

Le pseudo-code de la procédure *TraiteEvt(q, id, x, y)* résume la manière dont le résultat d'une requête q est mis à jour lorsqu'un événement GPS est reçu, fournissant une nouvelle position (x, y) pour un objet o. Rappelons que l'espace de référence est un ensemble de zones appelé $\mathcal{M}$.

TraiteEvt (q, o, x, y)
début
  // Calcule la zone actuelle, z
  $\mathcal{Z}$ = *PointDansPolygone(M, x, y)*
  // Trouve l'étiquette de $\mathcal{Z}$
  z = *étiquette($\mathcal{Z}$)*
  // Pour chaque bit valant 1 dans le statut de o,
  // calcule la transition z
  pour chaque bit i avec une valeur de 1 dans $statut_o$
    Calcule $s_j = \delta(FA_q, s_i, z)$
    Met le bit j à 1 et le bit i à 0 dans $status_o$
  fin pour
fin

L'ensemble résultat de la requête q peut alors être mis à jour en accord avec le nouveau statut de l'objet o. Pour cela, si au moins un des nouveaux états est un état acceptant, o sera dans l'ensemble résultat de la requête, dans le cas contraire il ne fera pas partie du résultat. Dans cette situation très simple, nous obtenons une réponse directe à nos deux interrogations :

1. il n'est pas nécessaire de maintenir des informations sur l'historique de la trajectoire puisqu'il suffit de connaître l'état(s) actuel(s) du FA atteint(s) en prenant en compte les événements reçus jusqu'à présent ;

2. l'espace nécessaire pour maintenir le résultat d'une requête est, dans le pire des cas, l'ensemble de tous les états du FA et est donc par conséquent proportionnel à la taille de la requête.

Si nous considérons maintenant le cas des patterns avec variables, le langage est beaucoup plus expressif, mais certaines précautions sont nécessaires pour exécuter les requêtes. Prenons par exemple la requête Q3 décrite auparavant. Chaque fois qu'un objet quitte la zone `f` pour une autre zone, une nouvelle étiquette est liée à la variable `@x`. On doit alors conserver cette valeur afin de pouvoir vérifier la cohérence de n'importe quelle occurrence future de `@x`.

La prochaine section est consacrée au modèle de données et se concentre sur l'évaluation de requêtes avec variables. Nous montrons qu'il est encore possible ne ne pas s'appuyer sur l'historique des déplacements des objets pour déterminer le résultat des requêtes et nous étudions plus spécifiquement les besoins au niveau de la mémoire pour plusieurs classes de requêtes.

3.3 Le modèle

Nous considérons un espace de référence $\mathcal{M}$ partitionné en un ensemble fini de zones, chaque zone étant identifiée de manière unique par un symbole d'un alphabet fini $\Sigma_{\mathcal{M}}$ que nous écrirons pour plus de simplicité Σ. L'axe temporel est divisé quant à lui en unités de temps égales. Nous choisissons pour la suite une unité de temps d'une minute mais tout autre valeur, choisie en fonction de l'application, peut convenir. Nous supposons également l'existence d'un ensemble $\mathcal{V}$ de variables, tel que $\Sigma \cap \mathcal{V} = \emptyset$ et appelons Γ l'union $\Sigma \cup \mathcal{V}$. Dans la suite, les lettres `a`, `b`, `c`, ... désigneront des symboles de Σ et `@x`, `@y`, `@z`, ... des variables. Nous supposerons le lecteur familier avec les notions de base concernant les expressions régulières et les langages réguliers, tels qu'ils sont présentés par exemple dans [62].

3.3.1 Représentation des données et langage de requêtes

Nous adoptons une représentation assez conventionnelle pour la base de données, avec $\mathcal{O}$ désignant la relation des objets mobiles, et $o.traj$ la trajectoire d'un objet o. La représentation des trajectoires est alors définie comme suit :

Définition 3 (Représentation des trajectoires). *Une trajectoire est représentée par une expression de la forme*

$$z_1\{T_1\}.z_2\{T_2\}.\cdots.z_n\{T_n\}$$

où $z_i, i = 1, \ldots, n$ sont des symboles de Σ et T_i représente le nombre d'unités de temps passées dans la zone z_i.

Par la suite nous emploierons le terme « trajectoire » pour désigner sa représentation. Par souci de simplicité nous omettrons les composants temporels et utiliserons une représentation simplifiée d'une trajectoire comme un mot $[z_1.z_2.\cdots.z_n]$ dans Σ^*.

Exemple 1. *Dans l'exemple représenté figure 3.1 où chaque point représente un événement GPS reçu, les objets o_1 et o_2 ont respectivement les trajectoires* `f{2}.a{3}.d{1}.c{2}` *et* `f{2}.c{2}.d{1}`*. La représentation simplifiée de leur trajectoire est alors respectivement* `f.a.d.c` *et* `f.e.d`*.*

Un choix naturel consiste à construire les patterns de mobilité comme des expressions régulières sur $\Gamma = \Sigma \cup \mathcal{V}$ et de chercher les sous-chaînes de trajectoires qui satisfont l'expression pour certaines instanciations des variables. Considérons par exemple l'expression régulière $E =$`a.@x+.b+.@x`. La trajectoire $t =$`f.d.a.c.b.c.b` satisfait E parce que nous pouvons trouver un mot $w =$ `a.@x.b.@x` dans le langage défini par E (w est appelé un *témoin* par la suite) et une valuation $\nu : \{$`@x` $:= c\}$ telle que $\nu(w)$ est une sous-chaîne de t.

Cependant cette approche conduit à certaines ambiguïtés quant au rôle des variables comme le montrent les deux exemples suivants :

1. soit E l'expression régulière `b.(a|@x)+.c`. La trajectoire `b.a.c` possède alors 2 témoins dans le langage régulier défini par E : `b.@x.c` et `b.a.c`. Dans le premier cas `@x` doit être instanciée à `a` mais dans le second cas n'importe quelle valeur est acceptable pour `@x` ;
2. soit E l'expression régulière `a.(@x|@y).b.(@x|@y)`. Les variables `@x` et `@y` peuvent être utilisées de manière interchangeable, ce qui rend le rôle des variables ambigu. Ainsi le mot `a.c.b.d` possède deux témoins qui sont `a.@x.b.@y` et `a.@y.b.@x`. Pour le premier témoin, `@x` est instanciée à `c` et `@y` à `d` alors qu'avec le deuxième témoin les instanciations sont inversées.

Comme les exemples précédents l'ont montré, l'assignation des variables est non déterministe et parfois même elle n'a aucun sens, lorsque l'on construit des patterns de mobilité avec des expressions régulières sur Γ pour lesquelles on ne pose aucune restriction. Lorsqu'on lit un mot w et que l'on vérifie si w satisfait un pattern de mobilité P, nous exigeons que toutes les variables soient liées explicitement afin de lever toute ambiguïté sur leur utilisation. Nous adoptons alors une définition plus rigoureuse du langage en considérant uniquement les expressions régulières *non-ambiguës* sur Γ telles que chaque variable joue toujours un rôle dans l'évaluation de la requête. Dans un premier temps nous introduisons les expressions régulières *marquées*.

Définition 4 (Expressions marquées [19]). *Soit E une expression régulière sur l'alphabet Γ. Nous définissons le* marquage *de E comme étant l'expression régulière E' où chaque symbole de Γ est marqué par un indice sur $\mathbb{N}$, représentant la position du symbole dans l'expression.*

L'opération inverse du marquage consiste à supprimer tous les indices des symboles pour obtenir une expression régulière dans Γ^*. Elle est notée par le symbole $\natural$.

Le marquage par exemple de l'expression régulière $E =$ `a`*`.@x.((b.a)|(c.b)).c.@x`*`.a` est l'expression $E' =$ `a`$_1^*$`.@x`$_2$`.((b`$_3$`.a`$_4$`)|(c`$_5$`.b`$_6$`)).c`$_7$`.@x`$_8$`.a`$_9$, et bien sûr $(E')^\natural = E$. Nous

pouvons désormais définir les *patterns de mobilité* comme la classe des expressions régulières qui satisfont la propriété suivante :

Définition 5 (Patterns de mobilité). *Un* pattern de mobilité *est une expression régulière P sur Γ telle que chaque variable de P' apparaît dans chaque mot du langage $\mathcal{L}(P')$.*

Cette propriété garantit que chaque variable, dans n'importe quel pattern, est toujours affectée à une étiquette lors de l'évaluation de la requête. L'expression P = `(a|b)`$^+$`.@x.(a|b)`$^+$ est par exemple un pattern de mobilité parce `@x` apparaît dans tous les mots du langage $\mathcal{L}(P)$. Une superposition réussie de P avec une trajectoire t a pour résultat par conséquent une affectation de `@x` à l'un des symboles de t. Il est possible de tester si une expression régulière satisfait cette condition et donc peut être utilisée comme un pattern de mobilité.

Proposition 1. *Il existe un algorithme pour vérifier si une expression régulière est un pattern de mobilité.*

Preuve: Soit E une expression régulière. Alors $\mathcal{L}(E)$ et $\mathcal{L}(E')$ sont des langages réguliers. Nous définissons le langage $\mathcal{L}_m = \{\Gamma^*.\texttt{@x}_1.\Gamma^*.\texttt{@x}_2.\cdots.\texttt{@x}_k.\Gamma^*\}$, où Γ signifie $\Sigma \cup \mathcal{V}$ et $\texttt{@x}_1\texttt{,..,@x}_k$ sont les variables de E'. $\mathcal{L}_m$ est régulier par construction et donc $\overline{\mathcal{L}_m}$ et $\mathcal{L}(E') \cap \overline{\mathcal{L}_m}$ sont également réguliers. Par conséquent le fait que $\mathcal{L}(E') \cap \overline{\mathcal{L}_m}$ est vide est décidable. Or si $\mathcal{L}(E') \cap \overline{\mathcal{L}_m} = \emptyset$ alors toutes les variables marquées apparaissent dans tous les mots de $\mathcal{L}(E')$. □

Exemple 2. *Les expressions régulières suivantes représentent les patterns de mobilité des requêtes **Q1**, **Q2** et **Q3** données en exemple dans la Section 3.2.*

1. P_1 = `a.f{2,}.c`
2. P_2 = `(a|b)`$^+$`.@x.(a|b)`$^+$
3. P_3 = `f.@x`$^+$`.(c|d)`$^+$`.@x`$^+$`.f`

Le langage d'interrogation pour l'utilisateur proposé dans la Section 3.2 permet de construire une expression qui peut être transformée en pattern de mobilité via la fonction d'interprétation [.] suivante :

1. $[x] = x, x \in \Sigma$
   $[v] = v, v \in \mathcal{V}$
   $[x_1 \ldots x_k] = x_1 \ldots x_k, x_i \in \Sigma \cup \mathcal{V}$
   $[\{x_1, \ldots, x_k\}] = (x_1 | \ldots | x_k)$
2. Interprétation d'une requête :
   `[START_BLOCK BLOCK END_BLOCK]=[START_BLOCK].[BLOCK].[END_BLOCK]`
3. Interprétation du `START_BLOCK` :
   `[start_at ZONESET]=[ZONESET]`
   `[start_at VAR]=[VAR]`
   $[\varepsilon]=\Sigma^*$

4. Interprétation du `NOW_BLOCK` :
   [`now_at ZONESET`]=[`ZONESET`]
   [`now_at VAR`]=[`VAR`]
   $[\varepsilon]=\Sigma^*$
5. Interprétation du `BLOCK` :
   [`follow ZONESET repeat INT BLOCK`]=[`ZONESET`]$^{[INT]}$.[`BLOCK`]
   [`follow ZONESET repeat BLOCK`]=[`ZONESET`]$^{+}$.[`BLOCK`]
   [`follow VARSTRING repeat INT BLOCK`]=[`VARSTRING`]$^{[INT]}$.[`BLOCK`]
   [`follow VARSTRING repeat BLOCK`]=[`VARSTRING`]$^{+}$.[`BLOCK`]
   [`follow ZONESET BLOCK`]=[`ZONESET`].[`BLOCK`]
   [`follow VARSTRING BLOCK`]=[`VARSTRING`].[`BLOCK`]
   [`roam INT`]=$\Sigma^{[INT]}$
   [`roam`]=Σ^*

L'interprétation d'une requête est un pattern de mobilité. Cependant le langage utilisateur d'interrogation ne capture pas tous les patterns de mobilité, comme le prouve l'exemple suivant. Soit $E =$`b.(a|c`$^{+}$`).@x`. E est un pattern de mobilité puisque la variable `@x` apparaît dans tous les mots du langage $\mathcal{L}(E)$. Cependant notre langage utilisateur ne permet pas d'utiliser l'opérateur + dans une expression avec un choix (c'est à dire avec un |) et donc ne permet pas d'exprimer ce pattern.

Le pattern `@x.a.b.c.@x` désigne la famille des langages réguliers qui est constituée de mots dans Σ^* avec exactement 5 lettres, la première étant égale à la dernière, séparées par `a.b.c`. Un pattern de mobilité P désigne un langage régulier $\mathcal{L}(P) \subseteq \Gamma^*$. Plus généralement un pattern de mobilité P avec k variables est équivalent à l'union des $|\Sigma|^k$ expressions régulières qui énumèrent les $|\Sigma|^k$ combinaisons possibles des valeurs des variables. Les variables offrent un moyen exponentiellement concis d'exprimer de tels langages.

Dans la suite nous désignerons par $var(P)$ l'ensemble des variables d'un pattern P donné. Le langage d'interrogation et sa sémantique sont maintenant définis comme suit :

Définition 6 (Syntaxe des requêtes). *Une requête est une paire $(\mathcal{P}, \mathcal{C})$ où $\mathcal{P}$ est un pattern de mobilité et $\mathcal{C}$ est un ensemble de contraintes de la forme $s_1 \neq s_2$ avec $s_1, s_2 \in \Sigma \cup var(P)$.*

Soit q une requête de la forme (P, LC) où LC est une liste de contraintes $\{C_1, \ldots C_l\}$. Le résultat de q, noté $q(\mathcal{O})$ est un sous-ensemble de $\mathcal{O}$ défini comme suit :

Définition 7 (Sémantique des requêtes). *Un objet o appartient à l'ensemble résultat $q(\mathcal{O})$ s'il existe un mapping $\nu : \mathcal{V} \rightarrow \Sigma$, appelé une* valuation*, avec les propriétés suivantes :*

1. *ν satisfait toutes les contraintes $C, \forall C \in LC$.*

2. *$o.traj \in \nu(\mathcal{L}(P))$*

Les contraintes dans une requête peuvent être utilisées pour interdire explicitement à une variable de prendre une valeur (e.g., `@x` $\neq$ `a`). Le *domaine* d'une variable `@x` pour une requête donnée q, noté dom_q(`@x`), représente l'ensemble des valeurs possibles pour `@x` d'après les contraintes exprimées dans q.

Exemple 3. *Les requêtes suivantes correspondent aux exemples donnés en section 3.2.*

1. q_1 = `({a.f{2,}.c}, ∅)`
2. q_2 = `({(a|b)`$^+$`.@x.(a|b)`$^+$`}, {@x ≠ a, @x ≠ b })`
3. q_3 = `({f.@x`$^+$`.(c|d)`$^+$`.@x`$^+$`.f}, {@x ≠ f})`

3.3.2 Évaluation des requêtes

Nous décrivons à présent un algorithme pour l'évaluation d'une requête q. Tout d'abord nous montrons comment obtenir un automate qui, pour un pattern de mobilité donné P, accepte les trajectoires qui satisfont P. Cet automate fournit également la valuation des variables apparaissant dans P. Dans un deuxième temps nous expliquons comment l'automate peut être utilisé en temps-réel et nous étudions la taille de la mémoire nécessaire pour stocker l'information pertinente. Pour des raisons de simplicité, nous considérons les automates qui acceptent le langage $\mathcal{L}(P)$: leur extension aux automates qui acceptent $\Sigma^*.\mathcal{L}(P)$ est triviale et peut être trouvée dans des livres spécialisés.

Puisqu'un pattern de mobilité P est une expression régulière sur l'alphabet Γ, nous pouvons construire un automate à états finis non-déterministe (NFA) N_Γ qui accepte le langage de Γ^* désigné par P. A partir de N_Γ nous pouvons construire un nouvel automate, N_Σ, qui vérifie si une trajectoire t de Σ^* appartient à $\nu(\mathcal{L}(P))$ et fournit la valuation ν.

Le principe de base est que N_Σ est similaire à N_Γ avec une gestion des instanciations des variables s'appuyant sur les extensions suivantes :

i) une transition étiquetée avec une variable `@x` que l'on souhaite franchir avec un symbole α, fixe la valeur de `@x` à α si `@x` n'était pas encore liée ;

ii) pour chaque état on maintient les instanciations des variables rencontrées jusqu'à présent.

Les transitions de s_i à s_j, étiquetées avec une variable `@x`, sont alors interprétées comme suit :

1. si `@x` est liée à α dans s_i et que le symbole courant du mot d'entrée est α, alors s_j peut être atteint et l'instanciation de `@x` dans s_j est identique à celle dans s_i ;

2. si `@x` n'est pas liée dans s_i et que le symbole courant du mot d'entrée est α, alors s_j peut être atteint et l'instanciation dans s_j est l'instanciation dans s_i à laquelle on ajoute `@x` $\leftarrow \alpha$;

3. dans les autres cas, on ne peut franchir la transition.

La définition de N_Σ est donnée ci-dessous :

- l'ensemble des états de N_Σ, $states(N_\Sigma)$, est $states(N_\Gamma) \times \Sigma^{|var(P)|}$, i.e., toutes les associations possibles d'un état de N_Γ avec une valuation ν des variables de P. Un état de N_Σ est noté $< S, \nu >$;

- l'ensemble des états acceptants de N_Σ, $accept(N_\Sigma)$ est $accept(N_\Gamma) \times \Sigma^{|var(P)|}$;

- la fonction de transition de N_Σ, δ_Σ, est obtenue de la fonction de transition de N_Γ, δ_Γ, comme suit :

  i) si $\delta_\Gamma(S_i, \alpha) = S_j$ est une transition de N_Γ avec $\alpha \in \Sigma$, alors $\delta_\Sigma(< S_i, \nu >, \alpha) =< S_j, \nu >$. En d'autres mots, la transition n'a pas d'effet sur les instanciations des variables ;

  ii) si $\delta_\Gamma(S_i,$`@x`$) = S_j$ est une transition de N_Σ, alors $\delta_\Sigma(< S_i, \nu >, \alpha) =$

$$\begin{cases} < S_j, \nu + \texttt{@x} := \alpha > \text{ si } \nu(\texttt{@x}) \text{ est non-définie et l'instanciation} \\ \qquad\qquad\qquad\qquad \text{de } \texttt{@x} \text{ par } \alpha \text{ est autorisée par les constraintes} \\ < S_j, \nu > \quad \text{si } \nu(\texttt{@x}) = \alpha \\ \text{est non-définie sinon} \end{cases}$$

Chaque fois qu'un état acceptant $< S, \nu >$ de N_Σ est atteint, la trajectoire en entrée est acceptée et la valuation ν définit l'ensemble des instanciations de toutes les variables (on rappelle que, par définition, chaque mot dans le langage défini par un pattern de mobilité contient toutes les variables).

3.3.3 Évaluation de requêtes continues

Afin de vérifier en temps-réel si un objet o satisfait ou non un pattern de mobilité, nous n'avons pas besoin de construire entièrement l'automate décrit dans la section précédente. Au lieu de cela, nous commençons avec une représentation minimale et construisons progressivement, d'après les symboles concaténés à la trajectoire de o, la valuation des variables qui peut potentiellement conduire à un état acceptant. Cette représentation initiale de N_Σ est

composée uniquement de l'ensemble des états de N_Γ, chacun associé avec la valuation vide. En conservant tous les états courant de N_Σ associés à o, les opérations suivantes peuvent être réalisées chaque fois qu'un nouveau déplacement m est ajouté à $o.traj$ pour tester si o entre, reste, ou sort du résultat de la requête :

1. Si les transactions étiquetées avec m conduisent o à au moins un état acceptant, alors o est ajouté (ou reste) dans le résultat de la requête.
2. Si les transitions étiquetées avec m sont telles que o n'est plus dans au moins un des états acceptants, alors o doit être retiré du résultat de la requête.

Cette caractéristique conduit à une première propriété pratique pour l'évaluation des requêtes continues : le dernier déplacement des objets fournit l'information suffisante pour maintenir le résultat d'une requête. Voici un exemple qui illustre le procédé.

Exemple 4. *Considérons le pattern de mobilité* $P = (a|b)^+.@x.(a|b)^+$. *La figure 3.2 représente un automate NFA* N_Γ *qui reconnaît les mots de* $\mathcal{L}(P)$, S_0 *étant l'état initial et* S_4, S_5 *les états finaux.*

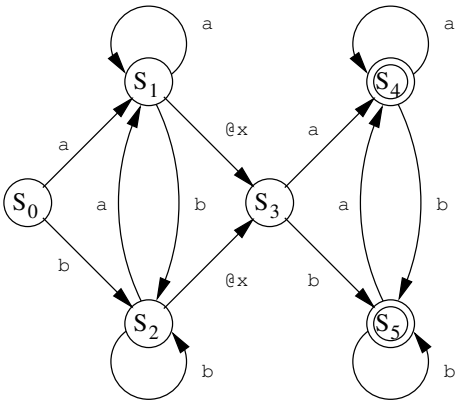


FIG. 3.2 – Un automate pour le pattern de mobilité`(a|b)`$^+$`.@x.(a|b)`$^+$

Supposons que l'on reçoive successivement les événements suivants pour un objet o : **`a`**, **`a`**, **`b`**, **`b`**, **`c`** *et* **`a`**. *Chaque ligne du tableau 3.1 montre les états du NFA* N_Σ *après avoir lu un symbole, ainsi que les valuations possibles de la variable* **`@x`**. *Les états acceptants sont en gras et signifient que la trajectoire appartient à l'ensemble résultat de la requête.*

L'exemple 4 montre que l'on est obligé de maintenir, durant l'analyse d'une trajectoire en entrée, plusieurs valuations associées au même état. Dans le pire des cas, nous devons maintenir $|states(N_\Gamma)| \times |\Sigma^k|$ états simultanés, représentant toutes les valuations possibles des variables qui conduisent à un état acceptant. Considérons maintenant le pattern suivant :

$$\texttt{@x}^+.\texttt{@y}^+.\texttt{@z}^+$$

Entrée	États atteints dans N_Σ
a	$< S_1$,@x=$\perp >$
a[2]	$< S_1$,@x=$\perp >, < S_3$,@x=a$>$
a[2].b	$< S_2$,@x=$\perp >, < S_3$,@x=b$>$, $< \mathbf{S_5}$,@x=a$>$
a[2].b[2]	$< S_2$,@x=$\perp >, < S_3$,@x=b$>, < \mathbf{S_5}$,@x=a$>, < \mathbf{S_5}$,@x=b$>$
a[2].b[2].c	$< S_3$,@x=c$>$
a[2].b[2].c.a	$< \mathbf{S_4}$,@x=c$>$

TAB. 3.1 – Évaluation d'une requête

On trouve alors facilement un mot tel que @x, @y et @z prennent toutes les valuations possibles.

Suivant l'application, la taille de la base de données et le nombre de requêtes, maintenir une grande quantité d'informations pour évaluer de façon continue une requête peut devenir très coûteux. Dans certains cas, on préférera alors restreindre la puissance d'expression du langage pour obtenir de plus faibles besoins en mémoire. Considérons par exemple un serveur web fournissant un mécanisme de souscription/publication sur un ensemble (potentiellement important) d'objets mobiles. Dans un tel système, les utilisateurs web peuvent enregistrer des requêtes et attendre d'être avertis des résultats. Les performances d'un tel système et en particulier sa capacité à traiter un grand nombre de requêtes avec une très grande quantité d'événements entrants, dépendent de l'efficacité de la maintenance du résultat de la requête et par conséquent de la taille des données nécessaires pour réaliser cette maintenance. Nous définissons dans la section suivante un fragment du langage de requêtes qui satisfait les exigences d'un tel type d'application.

3.4 Requêtes déterministes

La classe des requêtes *déterministes* est telle que, à chaque instant, il y a au plus une seule valuation possible pour chacune des variables du pattern de mobilité. Les requêtes déterministes sont définies par la propriété suivante :

Définition 8 (Requêtes déterministes). *Une requête* $q(P, \mathcal{C})$ *est* déterministe *ssi* $\forall u, v \in (\Sigma \cup \mathcal{V})^*, \forall$@x$\in \mathcal{V}, u.$@x$.v \in \mathcal{L}(P) \Rightarrow \nexists \alpha \in dom_q($@x$), \nexists w \in (\Sigma \cup \mathcal{V})^*, u.\alpha.w \in \mathcal{L}(P)$.

L'intuition est qu'il est désormais possible d'instancier une variable durant l'analyse d'une trajectoire si cette transition est le seul choix possible. Cela rend l'instanciation des variables déterministe et assure que, pour un mot donné, il y a une et une seule possibilité d'instancier une variable.

Exemple 5. *Les exemples suivants illustrent des requêtes déterministes.*

- *La requête* $q(\texttt{f.@x.(c|d).@x.f}, \emptyset)$ *est déterministe. A chaque fois qu'un symbole* `f` *a été lu, le seul choix possible est d'instancier* `@x` *au symbole qui suit immédiatement* `f`*.*

- *La requête* $q(\texttt{(a|b)}^+\texttt{.@x.(a|b)}^+, \emptyset)$ *est non-déterministe puisque les mots* `a.@x.a.b` *et* `a.b.@x.b` *appartiennent tous les deux à* $\mathcal{L}(P)$*. Cependant* $q'(\texttt{(a|b)}^+\texttt{.@x.(a|b)}^+, \{\texttt{@x} \neq \texttt{a}, \texttt{@x} \neq \texttt{b}\})$ *est déterministe.*

Proposition 2. *Il existe un algorithme pour vérifier si une requête est déterministe.*

Preuve: Soient q une requête, P le pattern de mobilité de q et N_Γ un automate déterministe qui reconnaît $\mathcal{L}(P)$. Puisque N_Γ est déterministe, pour n'importe quelle chaîne de caractères en entrée nous atteignons au moins un état s de N_Γ. Si l'on peut trouver un état s avec deux transactions : $\delta(s,\texttt{@x}) = s'$ et $\delta(s,\alpha) = s''$, avec $\alpha \in dom_q(\texttt{@x})$, alors il suffit de vérifier s'il existe deux mots $@x.u$ et $\alpha.v$ qui permettent tous deux d'atteindre un état final de s. Si c'est le cas, alors q n'est pas déterministe. □

Proposition 3. *Soit* $q(P,\mathcal{C})$ *une requête déterministe. Alors, pour chaque mot* w *de* Σ^**, il existe au plus un témoin de* w *dans* $\mathcal{L}(P)$*.*

Considérons à nouveau les requêtes de l'exemple 5. Dans le premier exemple un mot accepté peut avoir seulement un unique témoin, soit `f.@x.d.@x.f` soit `f.@x.c.@x.f`. Dans le deuxième exemple, avec les contraintes {`@x`≠`a`,`@x`≠`b`}, n'importe quel témoin sera composé de deux mots de {`a`,`b`}$^+$, séparés par un symbole distinct de `a` ou `b`.

La proposition 3 implique que si $q(P,\mathcal{C})$ est une requête déterministe, alors l'espace mémoire nécessaire pour vérifier si un mot satisfait q est $|P| + |var(P)|$, où $|P|$ représente le nombre de symboles de P. En effet nous avons besoin d'un automate FA pour q ainsi qu'un espace pour stocker la valuation de chaque variable et nous pouvons construire un automate FA avec un nombre d'états égal au nombre de symboles dans l'expression.

Lorsque l'on évalue une requête continue, nous devons maintenir pour chaque objet o l'ensemble des états courants dans l'automate ainsi que les instanciations des variables et cette information est suffisante pour déterminer, à chaque nouvel événement GPS, si o entre ou reste dans le résultat de la requête, ou le quitte.

Les preuves de ces propriétés concernant les patterns de mobilités reposent sur les *automates de Glushkov* d'expressions régulières [19, 24]. Considérons une expression régulière E, nous introduisons tout d'abord les définitions suivantes :

- $first(E) = \{\alpha \mid$ il existe un mot w tel que $\alpha.w \in \mathcal{L}(E)\}$
- $last(E) = \{\alpha \mid$ il existe un mot w tel que $w.\alpha \in \mathcal{L}(E)\}$
- $follow(E, \omega) = \{\alpha \mid$ il existe deux mots v et w tels que $v.\omega.\alpha.w \in \mathcal{L}(E)$ $\}$ pour chaque symbole ω.

L'idée de base est qu'un automate de Glushkov possède autant d'états qu'il y a de symboles marqués dans l'expression régulière qu'il représente. Chaque transition π_i entrante d'un état est étiquetée par le symbole non-marqué π. La définition d'un automate de Glushkov présentée par Book *et al.*[19] est la suivante :

Définition 9 (Automate de Glushkov). *L'automate de Glushkov de l'expression régulière E est l'automate $G_E = (Q', \Sigma, \delta', q_I, F')$ avec :*

1. *q_I l'état initial*
2. *$Q' = sym(E') \cup \{q_I\}$*
3. *Pour $\alpha \in \Gamma, \delta'(q_I, \alpha) = \{x | x \in first(E'), x^\natural = \alpha\}$*
4. *Pour $\sigma_i \in sym(E')$ et $\alpha \in \Gamma, \delta'(\sigma_i, \alpha) = \{x | x \in follow(E', x)$ et $x^\natural = \alpha\}$*
5. *$F' = last(E')$*

On rappelle que $^\natural$ désigne l'opération inverse du marquage d'une expression régulière (cf définition 4).

L'automate de Glushkov G_E reconnaît le langage représenté par E [19]. Le nombre d'états dans G_E est égal au nombre de symboles dans E.

Exemple 6. *La figure 3.3 présente l'automate de Glushkov G_E correspondant à l'expression régulière $E =$* `a.((@x.a)|(b.@x))*.b`*, dont le marquage donne l'expression $E' =$* $\mathtt{a}_1.((@\mathtt{x}_2.\mathtt{a}_3)|(\mathtt{b}_4.@\mathtt{x}_5))^*.\mathtt{b}_6$.

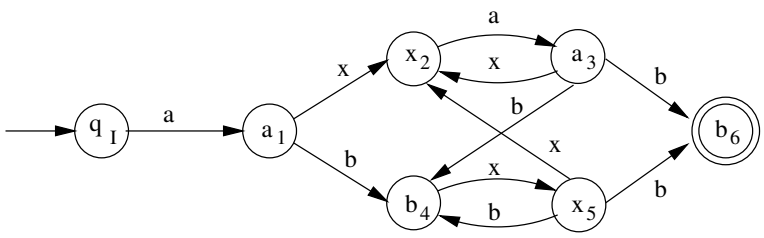


FIG. 3.3 – Automate de Glushkov associé à `a.((@x.a)|(b.@x))*.b`

Entrée	États atteints dans N_Σ	Transitions non-autorisées
a	$< S_1$,@x=$\perp >$	
a[2]	$< S_1$,@x=$\perp >$	$< S_3$,@x=a> car a $\notin dom(@x)$
a[2].b	$< S_2$,@x=$\perp >$	$< S_3$,@x=b> car b $\notin dom(@x)$
a[2].b[2]	$< S_2$,@x=$\perp >$	$< S_3$,@x=b> car b $\notin dom(@x)$
a[2].b[2].c	$< S_3$,@x=c>	
a[2].b[2].c.a	$< \mathbf{S_4}$,@x=c>	

TAB. 3.2 – Évaluation d'une requête déterministe

Les résultats suivants peuvent maintenant être établis à partir de cette définition des automates de Glushkov.

Proposition 4. *Soit P un pattern de mobilité. Alors pour n'importe quel couple de mots w_1,w_2 de $\mathcal{L}(P)$, les variables marquées apparaissent dans le même ordre dans w_1 et w_2.*

Preuve: Supposons l'existence de deux mots acceptés w_1 et w_2 tels que $w_1=p_1.@x_1.q_1.@x_2.r_1$ et
$w_2=p_2.@x_2.q_2.@x_1.r_2$ et $@x_1$ et $@x_2$ n'apparaissent pas dans p_1 et p_2. Si $@x_1$ n'apparaît pas dans r_1, puisque r_1 est un chemin de l'état $@x_2$ de l'automate de Glushkov à un état final, alors $p_2.@x_2.r_1$ est un mot accepté. Ceci entraîne une contradiction puisque ce mot ne contient pas la variable $@x_1$. Le même raisonnement nous conduit à une contradiction similaire en supposant que $@x_1$ apparaît dans r_1. □

Il s'ensuit que, si P est un pattern de mobilité d'une requête déterministe, pour n'importe quelle chaîne de Σ^* en entrée, il existe une seule valuation possible pour chacune des variables de P. En conséquence, l'espace mémoire nécessaire pour vérifier si un mot satisfait P est $|states(N_\Gamma)| + |var(P)|$, où N_Γ est l'automate de Glushkov de P.

En effet, l'automate N_Γ est non-déterministe et donc dans le pire des cas, tous les états peuvent être atteints simultanément. De plus nous devons stocker les instanciations des variables de P. Puisque les variables sont instanciées dans un ordre connu, une liste de $|var(P)|$ unités de mémoire est suffisante.

L'exemple suivant illustre cette propriété. En effet, il montre que pour un mot donné, un seul état peut-être atteint et que nous n'avons besoin de stocker qu'une seule valuation pour @x.

Exemple 7. *Considérons à nouveau la requête $q(P,\mathcal{C})$, avec P =(a/b)$^+$.@x.(a/b)$^+$ et $\mathcal{C}$ = {@x≠a,@y≠b}. L'automate demeure inchangé (voir Figure 3.2) mais l'évaluation de l'entrée a.a.b.b.c.a est maintenant présentée dans le Tableau 3.2.*

Les propriétés des requêtes déterministes garantissent que la quantité de mémoire nécessaire est indépendante de la taille de Σ et donc de la partition sous-jacente de l'espace

utilisée pour décrire les trajectoires des objets mobiles. Cette propriété peut s'avérer particulièrement intéressante dans les situations où l'espace considéré est particulièrement vaste ou si le nombre de requêtes à maintenir est tel que l'utilisation de la mémoire devient source de problèmes.

Nos expressions de requêtes déterministes sont simples et peuvent donc facilement être introduites dans un langage d'interrogation s'appuyant sur SQL. On peut par exemple simplement étendre SQL avec un opérateur booléen `matches`, comme l'illustrent les exemples suivants :

- **Q1.** Trouver tous les objets qui ont voyagé de `a` à `f`, sont restés au moins 2 minutes en `f` et ont ensuite voyagé de `f` en `c`.

```
SELECT *
FROM Mob
WHERE traj matches(a.f{2,}.c);
```

  La fonction `matches` vérifie que le suffixe d'un attribut spatio-temporel *traj* d'un objet *o*, instance d'une relation *Mob*, satisfait le pattern de mobilité `a.f.c`. Une contrainte temporelle additionnelle établit que l'objet doit passer au moins 2 unités de temps (e.g., 2 minutes) en `f`.

- **Q2.** Trouver tous les objets qui sont restés tout le temps en `a` ou `b` sauf une minute où ils ont été dans une troisième zone distincte de `a` et `b`.

```
SELECT *
FROM Mob
WHERE traj matches('(a|b)+.@x.(a|b)+')
AND    @x != 'a' AND @x != 'b';
```

  L'utilisateur se sert ici d'une variable `@x` qui exprime un déplacement de `a` ou `b` vers n'importe quelle autre zone de la carte. Les contraintes sur les instanciations possibles des variables sont exprimées à l'aide d'égalités ou d'inégalités. L'utilisateur demande dans cet exemple que l'objet quitte `a` ou `b` pour une troisième zone.

- **Q3.** Trouver tous les objets qui sont passés par `f` pour aller dans une autre zone, sont allés ensuite en `d` ou `c` et sont revenus en `f` en passant par la même zone qu'à l'aller.

```
SELECT *
FROM   Mob
WHERE  traj matches('f.@x+.(d|c)+.@x+.f')
AND    @x != 'f';
```

Nous avons donc présenté dans ce chapitre une nouvelle approche pour interroger et suivre en temps-réel une base de données d'objets mobiles au moyen de patterns de mobilité. Cette proposition est basée sur un modèle de données qui permet de trouver les objets dont la trajectoire satisfait une certaine séquence paramétrée de déplacements exprimés en fonction d'un ensemble de zones étiquetées. Nous identifions également un fragment de ce langage d'interrogation tel que la quantité d'espace nécessaire pour maintenir le résultat d'une requête est faible. Cependant lorsque l'évaluation d'une trajectoire conduit à une situation d'échec, notre modèle impose de recalculer la nouvelle position dans l'automate associé. Cela peut conduire à des surcoûts importants en mémoire centrale, rendant l'application peu efficace dans le cadre d'un suivi temps-réel d'un grand nombre d'objets à l'aide de nombreuses requêtes. Dans le chapitre suivant nous proposons un algorithme permettant de satisfaire les exigences d'une application temps-réel gérant à la fois un grand nombre de requêtes et d'objets.

Chapitre 4

Optimisation

Dans ce chapitre nous nous plaçons dans le cadre du modèle présenté au chapitre 3 et nous nous concentrons sur un sous-ensemble des requêtes déterministes pour lequel nous proposons une technique d'évaluation optimisée. Le but est de permettre un suivi continu en temps réel d'objets mobiles. Pour cela nous montrons que notre technique garantit un besoin d'espace mémoire et de temps CPU faible lors de l'évaluation et nous fournissons des résultats expérimentaux qui illustrent le gain de cette solution optimisée. Ces travaux ont été acceptés à BDA'05 et à CIKM'05 [42].

4.1 Introduction

Les hypothèses sur le modèle de données sont les mêmes qu'au chapitre 3, à savoir un espace 2D partitionné ainsi qu'une granularité temporelle fixée servant de référence pour décrire la trajectoire des objets mobiles. Cette représentation a des applications dans les systèmes de contrôle de trafic, les analyses post-acquisition, le regroupement et la classification de trajectoires. D'autres études [59, 132] adoptent cette représentation discrète permettant de réduire significativement la complexité des opérations par rapport à une représentation continue de l'espace. Contrairement aux articles précédemment cités, nous ne restreignons pas la partition de l'espace à une grille régulière. N'importe quelle partition dont la granularité et la topologie correspondent aux besoins de l'utilisateur peut être considérée.

Notre approche vise à fournir un langage de requêtes basé sur les patterns à la fois flexible, puissant et efficace. Il existe d'autres types d'applications pour lesquelles le modèle que nous proposons peut être utilisé. Le point commun à ces applications est de stocker sous forme de *séquences* l'évolution des valeurs prises dans un domaine discret pour une caractéristique donnée, et d'interroger et d'analyser ces séquences. Détaillons rapidement quelques exemples de telles applications.

Base de données de protéines

Les applications qui traitent l'ADN ou des protéines reposent sur une base de données qui stocke des millions de séquences [88]. Considérons par exemple une base de données de séquences de protéines. Une protéine est composée d'entre 100 et 200 acides aminés. Il existe 20 acides aminés distincts qui sont communément désignés par une étiquette d'une lettre. Ainsi, par exemple, la lysozyme, composée de 130 acides aminés est représentée par la séquence suivante :

K V F E R C E L A R T L K R L G M D G Y R G I S L A N W M C L A K W E S G Y
N T R A T N Y N A G D R S T D Y G I F Q I N S R Y W C N D G K T P G A V N A
C H L S C S A L L Q D N I A D A V A C A K R V V R D P Q G I R A W V A W R N
R C Q N R D V R Q Y V Q G C G V

où N par exemple est l'étiquette standard pour désigner l'acide aminé appelé Asparagine. En plus de la détection de pattern sur de telles séquences, notre langage vise à permettre des recherches paramétrées avancées. Par exemple le pattern paramétré `Q.@x.L.`ε`.Q.@x.L` capture les séquences où la sous-chaîne `Q.@x.L` est trouvée deux fois, `@x` étant liée à la *même* valeur dans les deux occurrences (ε désigne n'importe quelle sous-séquence). De plus on pourrait envisager d'exprimer des contraintes additionnelles sur les variables, en combinant par exemple le pattern `Y.A.@x` avec la contrainte `Polaire(@x)` où `Polaire` est un prédicat qui vérifie si l'instanciation de `@x` appartient à la classe des acides aminés `Polaire`.

Comportement d'un utilisateur sur un site web

Un autre type d'application pourrait être une application visant à analyser le comportement d'utilisateurs web sur un site afin d'améliorer l'ergonomie de celui-ci ou encore afin de savoir quels sont les meilleurs emplacements pour des publicités. En supposant que chaque page web est référencée de manière unique par une *url*, la base de données peut alors stocker les séquences de pages url – ou *histoires* – accédées successivement par un utilisateur [44, 79]. Une requête simple dans un tel contexte est par exemple la recherche des utilisateurs qui sont revenus à la page `A` après avoir visité une autre page. Ceci peut s'exprimer avec le pattern `A.@x.A`. La valeur de `@x` peut-être récupérée si nécessaire quand le comportement d'un utilisateur a satisfait ce pattern.

Le pattern `A.@x.`ε`.B.@x` capture toutes les séquences pour lesquelles une page `@x` est accédée successivement par deux pages distinctes, respectivement `A` et `B`. Si nous ne désirons pas mentionner explicitement `A` et `B` dans l'exemple précédent, nous pouvons donner plus de liberté au pattern en l'écrivant `@y.@x.`ε`.@z.@x`, avec les contraintes `@x != @y` et `@y != @z`. Ce pattern capture toutes les séquences où la page `@x` est accédée successivement par deux pages distinctes, quelle que soit leur url.

Nous nous restreindrons à l'étude des objets mobiles par soucis de cohérence avec le reste du travail réalisé pendant la thèse, en optant pour les mêmes hypothèses quant au modèle

que dans le chapitre précédent. Noter que nous ne traiterons pas dans la suite le cas des patterns avec des « trous » comme par exemple trouver les objets qui sont allés de la zone `a`, en `d`, puis sont allés en `c`, puis plus tard sont allés de `f` en `d` via une autre zone. En effet puisque les patterns successifs d'une telle requête doivent être satisfait dans l'ordre, la sémantique d'une telle requête est déduite directement de celle d'un pattern une séquence satisfait une telle séquence avec « trous », si elle satisfait successivement chacun des sous-patterns. Par conséquent nous n'étudierons dans la suite que les patterns sans « trous ».

Le modèle présenté maintenant supporte un sous-ensemble des patterns de mobilité présentés au chapitre 3. Nous nous concentrons sur l'évaluation *continue* des requêtes construites à l'aide de patterns de mobilité, *i.e.*, des requêtes qui surveillent une population d'objets mobiles et dont le résultat est mis à jour de manière continue à l'aide d'événements donnant la nouvelle localisation des objets. Nous avons choisi de limiter le pouvoir d'expression de nos patterns afin de développer, pour un sous-ensemble de notre langage d'origine, un algorithme qui minimise à la fois les besoins en espace de stockage et le coût d'évaluation d'une requête. Plus précisément, il satisfait les deux propriétés suivantes :

1. chaque événement est vérifié une seule fois ;

2. le besoin en espace mémoire pour évaluer une requête q est proportionnel aux nombres de variables dans q.

La première propriété garantit que nous n'avons jamais à aller chercher les données du passé concernant la trajectoire d'un objet o, puisque l'information apportée par le dernier événement est toujours suffisante. La deuxième propriété correspond à l'exigence d'un stockage minimal. Il s'en suit que les résultats des requêtes peuvent être mis à jour efficacement et donc que le modèle peut supporter de larges ensembles de données.

Nous avons implanté notre algorithme dans un prototype et avons réalisé des évaluations qui confirment les résultats attendus. Il faut souligner que, contrairement aux analyses de complexité traditionnels en bases de données qui mesurent le coût de l'évaluation d'une requête suivant le nombre d'entrées/sorties, nous considérons le nombre d'opérations en mémoire centrale nécessaires pour maintenir le résultat d'une requête pour un flux d'événements. Avec ces considérations, l'évaluation expérimentale montre que notre algorithme économise une grande quantité de calculs et par conséquent diminue fortement le temps d'évaluation d'une requête continue.

La section 4.2 introduit le modèle de données. La section 4.3 est ensuite dédiée à l'évaluation de requêtes, incluant notre solution optimisée. Nous présentons nos résultats expérimentaux dans la section 4.4. Enfin la section 4.5 conclut ce travail.

4.2 Le modèle

Cette section décrit rapidement les principaux composants du modèle de données, à savoir la représentation des données, les patterns de mobilité et le langage de requêtes. Si l'espace considéré est le même que dans le chapitre 3, la définition des patterns et du langage de requêtes est cependant légèrement différente.

4.2.1 Représentation des données

Comme nous l'avons présenté dans le chapitre précédent, l'espace $\mathcal{M}$ considéré pour ce modèle n'est pas un espace dense sur $\mathbb{R}^2$ mais un espace partitionné en zones identifiées par une étiquette. Dans ce chapitre nous ne prenons pas en considération la composante temporelle des trajectoires des objets. En ce qui nous concerne on relève simplement pour chaque objet se déplaçant dans $\mathcal{M}$, la séquence des zones distinctes successivement traversées et adoptons pour une trajectoire la définition simplifiée suivante :

Définition 10 (Représentation d'un objet.). *Un objet mobile $o \in \mathcal{O}$ est une paire $(oid, traj)$ où oid désigne l'identifiant de l'objet et $traj =< z_1.z_2.\cdots.z_n >$ est un mot dans Σ^* et $z_i \neq z_{i+1}, i \in [1, n-1]$.*

Par la suite nous utiliserons le terme « trajectoire » pour désigner sa représentation. Puisque nous associons chaque localisation à une zone de la carte, nous devons juste supposer que le serveur GPS fournit au moins une localisation pour chaque zone traversée par un objet. Nous soulignons également le fait que nous éliminons d'une trajectoire les étiquettes successives qui sont répétées puisque nous ne pouvons garantir que les événements liés à un objet sont séparés par des intervalles de temps constant. Autrement dit, contrairement au chapitre 3, nous ne prenons pas en compte le temps passé dans chaque zone.

Exemple 8. *La Figure 3.1 représente deux objets mobiles, o_1 et o_2. Chaque point sur la figure durant leur déplacement correspond à la réception d'un événement GPS. Ces objets sont décrits par les paires :*

- $(o_1, \texttt{f.a.d.c})$
- $(o_2, \texttt{f.e.d})$

4.2.2 Pattern

On considère que l'on a un ensemble $\mathcal{V}$ de variables tel que $\Sigma \cap \mathcal{V} = \emptyset$. Dans la suite du chapitre, les lettres `a`, `b`, `c`, ... désignent des étiquettes de Σ, et `@x`, `@y`, `@z`, ... des variables.

Les applications que nous visons en proposant ce modèle sont des applications de suivi de véhicules en temps-réel. Une première différence entre le modèle présenté ici et celui du chapitre 3 est l'absence de la notion de temps passé dans une zone. Par conséquent il n'est

pas possible d'exprimer des contraintes temporelles dans les requêtes. De plus dans un souci de simplification des requêtes, et donc de leur évaluation, nous décidons ne pas autoriser les disjonctions de zones. Nous souhaiterions donc évaluer des requêtes comme :

- Q_1 : quels objets sont passés par `a`, puis ont traversé `d` et sont actuellement en `c` ?
- Q_2 : quels objets sont passés par `b`, ensuite ont traversé `c` et `e` et sont actuellement en `f` ?
- Q_3 : quels objets sont allés de `f` en `d` en traversant une autre zone ?
- Q_4 : quels objets ont quitté une certaine zone pour `a`, ensuite sont revenus à leur zone de départ avant d'aller dans une autre zone ?

Ces requêtes ont toutes en commun de spécifier la succession des zones traversées par un objet durant sa pérégrination et par conséquent représentent une *classe* de trajectoires.

Les requêtes ci-dessus sont des exemples de *patterns de mobilité*, que nous appellerons plus succinctement *pattern* par la suite, définis comme suit :

Définition 11 (Pattern). *Un pattern est un mot* $t_1.t_2 \dots t_n$ *dans* $(\Sigma \cup \mathcal{V})^n$ *tel que* $t_i \neq t_{i+1}, i < n$

Une première utilisation de ces patterns est le balayage d'historiques de trajectoires. Ils peuvent également être utilisés dans des requêtes continues dont l'objectif est de déterminer à la volée les objets satisfaisant la requête à l'instant considéré. On souhaite dans ce cas retourner les objets dont la partie la plus *récente* de la trajectoire peut être classée dans le pattern représentant la requête. Par la suite nous ne nous intéressons qu'au cas des requêtes continues. Dans leur forme la plus simple, les patterns sont des mots (sans répétition de symboles) dans Σ^* tels que, par exemple ceux des deux premières requêtes-exemples, $Q_1 =$ `a.d.c` et $Q_2 =$ `b.c.e.f`. L'interprétation d'un pattern P *sans variables* est naturelle : une trajectoire T satisfait un pattern P si P est un suffixe de T. Le suffixe de T représente ici la partie la plus récente du flot d'événements GPS. Une superposition d'un pattern sur une trajectoire peut être évidemment possible pour un objet, observé à un instant t, et ne plus être possible au prochain instant d'observation.

Exemple 9. *Considérons à nouveau les deux objets de la Figure 3.1 :*

- *puisque* $o_1.traj = \boldsymbol{f}.\mathbf{a.d.c} = \boldsymbol{f}.Q_1$ *alors* o_1 *appartient au résultat de la requête* Q_1.
- *puisque ni* Q_1 *ni* Q_2 *ne sont des suffixes de* $o_2.traj$, o_2 *n'appartient pas à leur résultat.*

Les variables sont utiles pour capturer des séquences plus générales où les déplacements ne sont pas explicitement associés à une étiquette spécifique. Les exemples ci-dessus montrent

que les symboles des zones traversées ne sont pas toujours explicitement formulés dans une requête. Par exemple la requête Q_3 parle d'une « autre zone » et la requête Q_4 d'une « certaine zone de départ ». Pour l'interrogation la notion de pattern doit être par conséquent étendue pour intégrer ces zones « non-spécifiées ». Les patterns pour ces deux requêtes sont alors les suivants :

- $Q_3 =$ `f.@x.d`
- $Q_4 =$ `@x.a.@x.@y`

L'interprétation des patterns (avec variables) est une extension de la sémantique de la superposition des suffixes présentée dans le cadre des patterns sans variables : une trajectoire T satisfait un pattern P si l'on peut substituer chaque variable dans P par un symbole de Σ tel que le pattern résultant soit un suffixe de T. Plus formellement :

Définition 12 (Substitution et valuation). *Une* substitution ν *est un ensemble fini de la forme*

$$\{x_1/t_1, x_2/t_2, \ldots, x_n/t_n\}$$

où $x_i \in \mathcal{V}, i = 1, \ldots, n$*, et chaque* t_i *est soit une variable dans* $\mathcal{V}$ *soit une étiquette dans* Σ*.* ν *est une* valuation *si* $t_i \in \Sigma$*, pour chaque* $i \in [1, n]$

$\nu(P)$ désigne le pattern obtenu à partir de P en remplaçant, pour chaque $x_i/t_i \in \nu$, l'occurrence de x_i dans P par t_i. Chaque élément x_i/t_i est appelé une *instanciation* de x_i et l'ensemble des variables $\{x_1, x_2, \ldots, x_n\}$ est appelé $bound(\nu)$. Si, par exemple, $P =$ `a.b.@x.@y.b.@z.b` et $\nu = \{$`@x/c, @z/@x`$\}$, alors $\nu(P) =$ `a.b.c.@y.b.@x.b`. On désigne l'ensemble des variables d'un pattern P par $var(P)$.

Remarquons que si ν est une valuation et $var(P) \subseteq bound(\nu)$ alors $\nu(P)$ est un mot dans Σ^*. D'où la définition :

Définition 13 (Interprétation d'un pattern). *Une trajectoire* T *satisfait un pattern* P *ssi il existe une valuation* ν *telle que* $\nu(P)$ *soit un suffixe de* T*.*

Chaque fois que le pattern P d'une requête contient des variables, un objet o appartient au résultat de la requête si on peut trouver une valuation ν qui permette de superposer $\nu(P)$ et un suffixe de $o.traj$. Puisque la représentation de la trajectoire évolue lorsque de nouveaux événements sont reçus, la superposition d'un pattern de requête et d'une trajectoire doit être testée périodiquement –pratiquement de manière continue– pour chaque objet. Notre but est de réaliser ce test de superposition avec une consommation en temps et en espace mémoire minimale.

Symbole	**Signification**
P	Un pattern
m	La longueur d'un pattern.
l	Une position à l'intérieur d'un pattern ($0 \leq l \leq m-1$).
$e, e_1, e_2, \cdots$	Des bords.
ν, σ	Resp. : une valuation, une substitution.
$t_1, t_2, \cdots$	Symboles ou variables de $\Sigma \cup \mathcal{V}$
$a, b, c, \cdots$	Symboles de Σ
$@x, @y, @z, \cdots$	Symboles de $\mathcal{V}$

TAB. 4.1 – Tableau des symboles utilisés dans ce chapitre

4.3 Évaluation des requêtes

Nous présentons maintenant deux algorithmes pour une évaluation continue des patterns de mobilité. Le premier suit une approche naïve qui vérifie avec répétitions les événements reçus et revient en arrière dans la trajectoire chaque fois qu'une superposition échoue. Le second algorithme est notre technique optimisée. Tous les symboles utilisés dans ce chapitre sont donnés dans le tableau 4.1.

4.3.1 L'approche naïve

L'expression de requêtes dans notre modèle se fait donc sous la forme de patterns présentés ci-dessus. L'objectif de notre système est de fournir à tout instant la liste des objets dont la trajectoire satisfait le pattern d'une requête. En l'absence de variables, le formalisme de notre modèle nous conduit vers les techniques de *comparaison de chaînes de caractères* standard et notamment les algorithmes permettant de détecter l'occurrence d'un pattern dans une chaîne de caractères comme le montre l'exemple suivant :

Exemple 10. *Considérons le pattern $P_1 = a.c.b.a$ et trois trajectoires T_i de trois objets o_i, à trois instants différents :*

temps	*trajectoires*			*résultat*
t	$T_1 =$ a.b.a.c.b	$T_2 =$ a	$T_3 =$ c.b.a.b	$\emptyset$
$t+1$	$T_1 =$ a.b.**a.c.b.a**	$T_2 =$ a.c	$T_3 =$ c.b.a.c.b	o_1
$t+2$	$T_1 =$ a.b.a.c.b.a.b	$T_2 =$ a.c	$T_3 =$ c.b.**a.c.b.a**	o_3

Le pattern P_1 apparaît dans les trajectoires T_1 (à $t+1$) et T_3 (à $t+2$). Noter que la trajectoire T_2 reste inchangée entre les instants $t+1$ et $t+2$. Cela signifie que l'objet n'a pas changé de zone entre $t+1$ et $t+2$.

Une technique simple pour vérifier si un pattern peut être superposé à une trajectoire consisterait à positionner le pattern sur le premier symbole de la trajectoire et à comparer

symbole à symbole les deux. En cas d'échec on décale le pattern d'une position sur la trajectoire et on recommence. Nous renvoyons le lecteur à [69, 32] pour plus de détails sur cet algorithme.

Comme l'évaluation ne porte pas sur des données statiques mais sur des flux de données, la tentative de superposition porte uniquement sur le suffixe de la trajectoire. Lorsqu'un pattern est enregistré, on tente de superposer les k symboles du pattern avec les k derniers symboles de la trajectoire. En cas d'échec, on essaye de superposer les $k-1$ premiers symboles du pattern avec les $k-1$ derniers de la trajectoire et si la superposition est possible on attend un nouveau symbole de la trajectoire, correspondant à un événement GPS reçu signalant l'entrée dans une nouvelle zone.

Le premier algorithme que nous proposons est donc une simple extension des techniques de comparaison de chaînes de caractères pour permettre de traiter des patterns avec variables et repose sur les opérations suivantes :

1. une *tentative de superposition* entre un pattern P et une trajectoire T à la position i ;
2. un *décalage* de P chaque fois qu'un échec se produit.

Regardons plus précisément comment s'effectuent ces deux opérations que nous nommerons par la suite COMPARE et DÉCALE.

L'opération COMPARE

Une tentative de superposition à la position i de la trajectoire compare, un par un, de la gauche vers la droite, les symboles $P[0], P[1], \ldots, P[m-1]$ du pattern avec les étiquettes $T[i], T[i+1], \ldots, T[i+m-1]$ de la trajectoire. Pendant la tentative de superposition, les variables dans $var(P)$ sont progressivement liées aux étiquettes dans Σ et ces instanciations définissent une valuation ν appelée la *valuation courante*. ν est initialement vide. Si $P[j]$ est une variable, les règles d'instanciation suivantes doivent être appliquées :

1. si `@x` $\notin bound(\nu)$, la comparaison a toujours lieu avec succès et $\nu := \nu \cup \{$`@x/T[i+j]`$\}$ i.e., `@x` est liée à l'étiquette $T[i+j]$. L'instanciation reste valable jusqu'à la fin de la tentative de superposition ;

2. sinon, si `@x` $\in bound(\nu)$ la comparaison est couronnée de succès si et seulement si $T[j]$ est égal à $\nu($`@x`$)$.

En d'autres termes, la superposition de $P[j]$ sur $T[i+j]$ est autorisée quand la variable n'a pas été encore instanciée ou quand elle a été instanciée avec l'étiquette lue actuellement de la trajectoire. Dans le premier cas nous lions la variable à l'étiquette courante de la trajectoire.

Considérons par exemple la tentative de superposition entre $P =$ `a.@x.b.@x` et $T =$ `a.c.b`. Les comparaisons sont réalisées avec succès pour $j = 0, 1, 2$. Quand $P[1] =$ `@x` est comparé à $T[1] =$ `c`, la variable `@x` est déjà liée à l'étiquette `c`. La valuation ν est, à cet instant, $\{$`@x/c`$\}$.

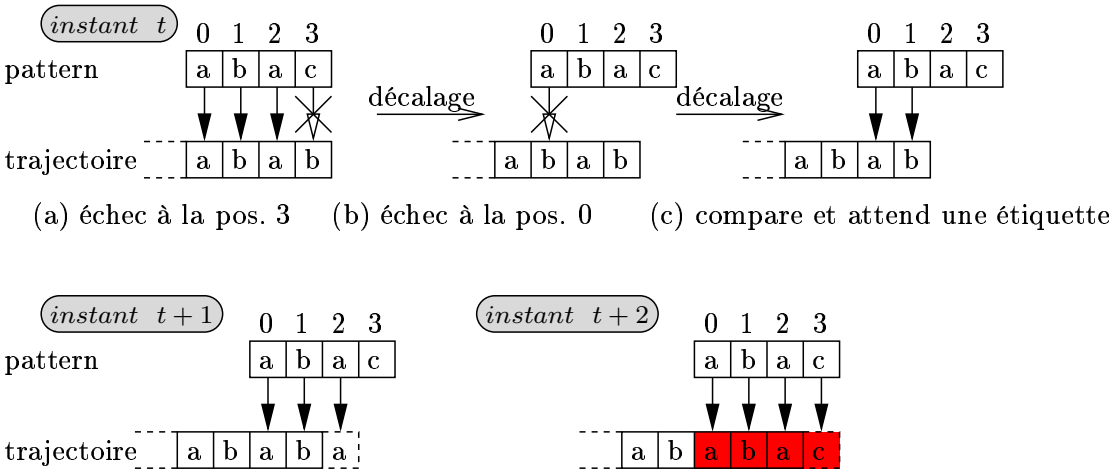
La comparaison suivante entre $P[3]$ et $T[3]$ peut alors réussir si la prochaine étiquette ajoutée en fin de la représentation de la trajectoire est `c`, l'instanciation courante de `@x`. Il s'ensuit que nous devons maintenir, pour chaque objet, la substitution courante, c'est-à-dire, une liste des instanciations courantes des variables du pattern de la requête.

Si toutes les comparaisons ont été réalisées avec succès, alors la tentative de superposition a réussi, sinon il s'agit d'un échec. A noter qu'une superposition réussie implique que toutes les variables du pattern ont été instanciées. Dans les deux cas, l'opération DÉCALE est réalisée.

L'opération DÉCALE

Dans sa forme la plus simple, DÉCALE fait glisser le pattern sur la trajectoire d'un symbole vers la droite et réinitialise la tentative de comparaison depuis le début du pattern. Dans le cas d'un pattern sans variables, cette technique peut être appliquée directement au processus d'évaluation continue.

Une opération COMPARE est réalisée chaque fois qu'un nouvel événement est reçu. Chaque fois qu'un échec se produit (disons à une position l, avec $0 \leq l \leq m-1$), DÉCALE décale le pattern d'une position et une comparaison avec les $l-1$ dernières étiquettes de la trajectoire doit être effectuée. Si la superposition aboutit à un succès, on attend le symbole suivant. En cas de nouvel échec, un nouveau décalage est nécessaire. La figure 4.1 montre un exemple.



(a) échec à la pos. 3 (b) échec à la pos. 0 (c) compare et attend une étiquette

(d) une nouvelle étiquette `a` est ajoutée (e) `c` est ajouté ; superposition réussie

FIG. 4.1 – Tentative de superposition avec un pattern sans variables

Quand le pattern contient des variables, l'algorithme est assez semblable sauf qu'il prend en compte l'instanciation des variables. Chaque fois qu'un échec se produit la substitution courante est effacée : toutes les instanciations sont annulées. Le pattern est décalé d'un symbole vers la droite et on redémarre une nouvelle tentative de superposition. La figure 4.2

illustre cet algorithme.

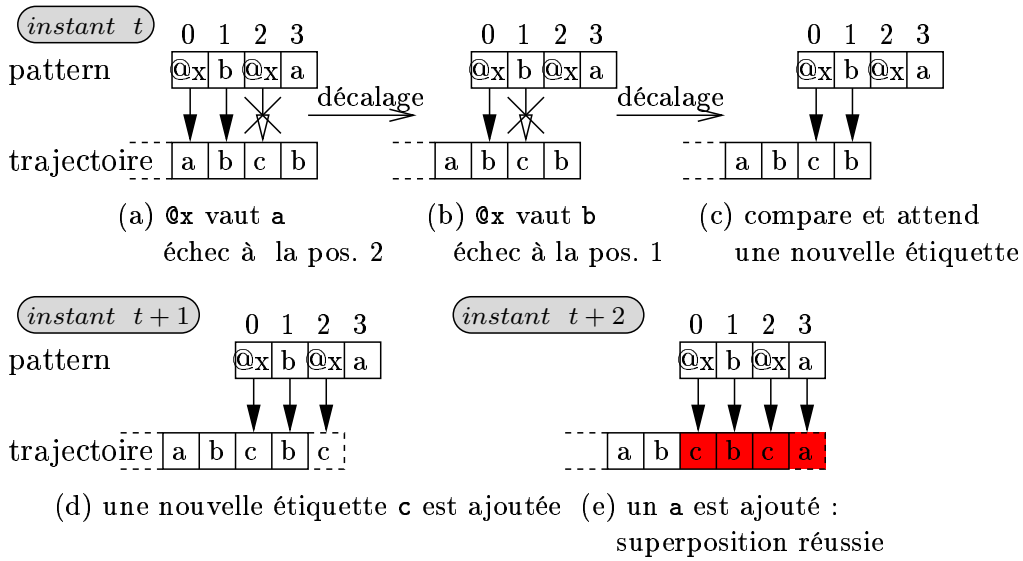


FIG. 4.2 – Tentative de superposition avec un pattern avec variables

Par conséquent l'algorithme d'évaluation d'un pattern P pour une trajectoire t donnée est le suivant :

TESTSUPERPOSITION(t, P)
entrée : trajectoire t, pattern P avec variables
sortie : booléen qui vaut vrai si l'on peut superposer
début
$i := 0$
réussi := faux
tant que réussi = faux **et** $i < t.longueur$ **faire**
// on souhaite tester tous les symboles du pattern
$\nu = \emptyset$ // on initialise la valuation à vide
$j := 0$
tant que $j < P.longueur$ **faire**
si $t[j] \in \Sigma$ **alors**
si $t[i+j] = P[j]$ **alors** $j := j + 1$
sinon
$i := i + 1$
sortir
finsi
finsi
fin
si $j = P.longueur$ **alors** réussi := vrai **finsi**

fin
retourne réussi
fin

Cette technique est simple mais coûteuse puisque l'algorithme s'exécute en $O(m \times |T|)$. Chaque étiquette de la trajectoire est potentiellement comparée plusieurs fois avec un symbole du pattern. On notera toutefois que lorsqu'un échec se produit à la position i, nous ne sommes pas obligés de lire la trajectoire passée. En fait le suffixe de la trajectoire peut-être reconstruit à partir de la position i en remplaçant les variables dans le pattern par leur instanciation courante.

Par exemple dans la figure 4.2(a) nous savons, quand l'échec intervient, que les deux premiers symboles du pattern, `@x.b`, peuvent être superposés sur les deux premiers symboles de la trajectoire. Puisque $\nu = \{$`@x/a`$\}$ à cet instant, le suffixe de la trajectoire est ν`(@x).b=a.b`.

Au niveau de l'implantation la solution naturelle consiste à créer un automate déterministe pour chacun des patterns, chaque état correspondant à un symbole du pattern. On crée donc m états (m longueur du pattern), avec pour toute paire d'états s_i et s_{i+1} une transition étiquetée par le $(i+1)^{ème}$ symbole du pattern. La figure 4.3 représente l'automate correspondant au pattern de l'exemple 2 pour cet algorithme.

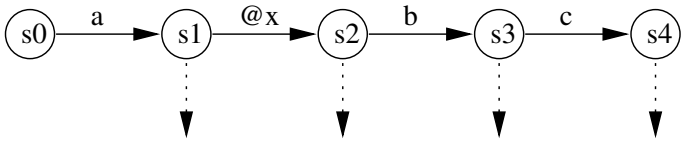


FIG. 4.3 – Automate du pattern *a.@x.b.c*

On garde pour chaque objet pour chaque requête le *statut* défini comme suit :

Définition 14 (Statut). *Le statut d'un objet o pour une requête Q est le couple $(état, \nu)$ où état désigne l'état de l'automate correspondant au pattern de la requête considérée, et ν correspond à une fonction partielle d'instanciation des variables.*

Lorsqu'on a un nouvel événement GPS retournant une position précise, on détermine l'étiquette de la zone dans laquelle est cet objet. Si l'objet a changé de zone, on regarde dans l'automate s'il existe une transition avec cette étiquette permettant de passer de l'état s_i dans lequel était l'objet à l'état s_{i+1}. Si une telle transition n'existe pas, on revient à l'état initial.

4.3.2 Évaluation optimisée

La technique d'évaluation que nous proposons ici s'inspire de travaux sur la comparaison de chaînes de caractères proposés par Knuth, Morris et Pratt (KMP) [69, 33] qui permettent

d'utiliser les informations collectées lors de l'évaluation jusqu'à l'obtention d'une situation d'échec. L'objectif est de trouver directement le décalage à réaliser du pattern pouvant conduire à une superposition sans avoir à relire de symboles de la trajectoire. Nous avons étendu ces techniques en intégrant la notion de variables dans les patterns. KMP fournit une base qui semble particulièrement adaptée pour l'évaluation de patterns contenant des variables sur un ensemble de chaînes de caractères [101]. Nous allons dans un premier temps présenter l'algorithme KMP avant de décrire l'extension que nous en proposons.

L'algorithme KMP [69]

L'objectif de cet algorithme est de détecter toutes les occurrences d'une chaîne de caractères, que l'on appellera ici *pattern*, au sein d'une autre chaîne de caractères (la *trajectoire*) en un temps linéaire en nombre total de symboles des 2 chaînes, là où une technique naïve conduirait à un calcul quadratique.

Intuition de l'algorithme

Considérons l'exemple de la figure 4.4.

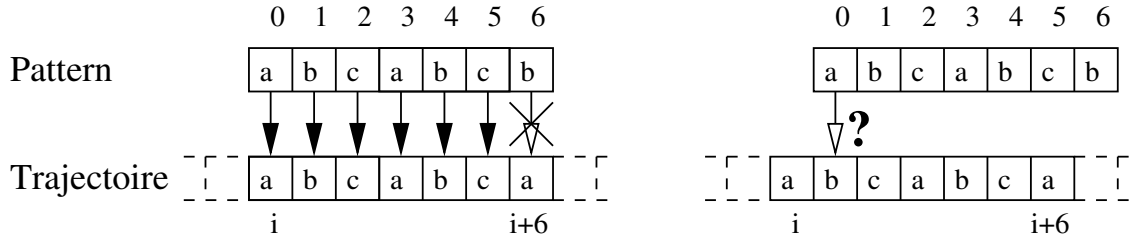


FIG. 4.4 – Algorithme naïf de superposition d'un pattern avec une trajectoire

On représente le pattern que l'on souhaite détecter comme une fenêtre glissant sur la trajectoire. La partie gauche de la figure 4.4 montre une première tentative de superposition qui conduit à un échec sur le 6ème symbole du pattern après avoir comparé tous les symboles précédents un à un. Lors de l'échec de cette tentative, la solution naïve consisterait à décaler le pattern d'une position et à recommencer la tentative de superposition en partant du premier symbole du pattern.

Cette technique est cependant très coûteuse du fait que l'on n'utilise à aucun moment l'information obtenue avant cette situation d'échec, ce qui oblige à faire des comparaisons qui auraient pu être évitées. Revenons à notre exemple de la figure 4.4. La tentative de superposition de la figure 4.4b conduit à un échec immédiat, le symbole `a` du pattern ne

pouvant être superposé au symbole `b` de la trajectoire. Donc décaler de 1 correspond à tenter de superposer le `a` du pattern avec ce que l'on sait *déjà* être un `b` dans la trajectoire. Il en est de même pour un nouveau décalage d'un symbole. Cependant un troisième décalage nous permet de superposer les derniers symboles lus dans la trajectoire avant l'échec, `a.b.c`, avec le début du pattern. Une optimisation consisterait donc à éviter de réaliser les décalages inutiles pour directement trouver ce bon décalage du pattern.

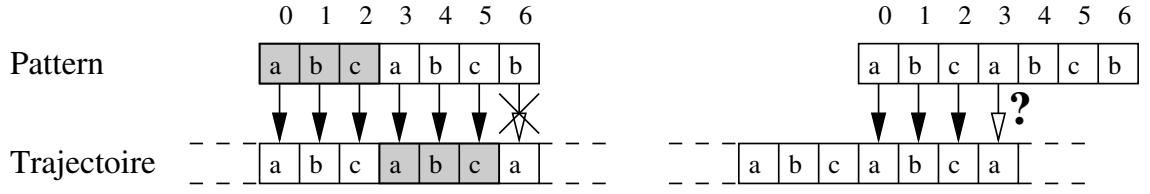
L'algorithme KMP [69] part du constat qu'il n'est pas utile de conserver la trace des caractères de la trajectoire déjà testés, ni même de les relire pour chaque décalage à réaliser. En effet le pattern en lui-même contient toute l'information utile afin de « reconstruire » les derniers symboles lus dans la trajectoire. Les auteurs décrivent une technique de recherche de superposition en 2 étapes :

1. calcul, pour chaque position dans le pattern, du décalage à réaliser en cas d'échec lors d'une tentative de superposition. Ces décalages sont calculés *lors de la compilation* du pattern et sont donc indépendants de la trajectoire considérée ;

2. utilisation de cette information *lors de l'analyse* de la trajectoire afin de déterminer directement le décalage à faire sans avoir à retester des symboles de la trajectoire déjà lus.

Ainsi dans notre exemple nous savons que les 6 derniers symboles lus dans la trajectoire T avant l'échec en position $T[i+6]$ sont les 6 premiers symboles du pattern P, à savoir $T[i \ldots i+5] = P[0 \ldots 5] =$ `a.b.c.a.b.c`. L'examen de cette sous-chaîne nous permet de déduire que le décalage à faire est de 3 et qu'un décalage de 1 ou 2 ne peut pas conduire à une superposition. En effet un décalage de 1 signifie que l'on tente de superposer $P[0] =$ `a` avec $T[i+1]$ or on sait que $T[i+1] = P[1] =$ `b` ; par conséquent la tentative échouera. Au contraire lors d'un décalage de 3 on sait que l'on peut superposer $P[0] =$ `a` à $T[i+3]$ car $T[i+3] = P[3] =$ `a`, mais aussi $P[1] =$ `b` à $T[i+4] = P[4] =$ `b` et $P[2] =$ `c` à $T[i+5] = P[5] =$ `c`. L'algorithme KMP permet de calculer à l'avance le *plus petit décalage* du pattern à réaliser afin de ne manquer aucune solution tout en évitant de relire des symboles de la trajectoire.

Pour illustrer ce propos considérons l'exemple de la figure 4.5, partie (a). La tentative de superposition échoue au niveau du dernier symbole du pattern. On réalise directement le bon décalage de 3, puisqu'on a établi que les trois derniers symboles `a.b.c` du pattern que l'on a superposé correspondent aux trois premiers symboles de la requête. Dans cet exemple on pourrait décaler le pattern non pas d'un symbole, mais le faire glisser pour superposer les derniers symboles de la trajectoire lus avant l'échec avec les premiers symboles du pattern et reprendre l'évaluation en comparant le symbole `a` de la trajectoire sur lequel l'échec a été constaté avec le quatrième symbole du pattern (cf. figure 4.5 partie (b)).



(a) échec de la tentative à la position position 6 du pattern. Un suffixe du pattern est préfixe du pattern.

(b) on décale le pattern afin de superpo le suffixe de la trajectoire avec le prefi du pattern. On repart du 3eme symbole

FIG. 4.5 – Utilisation des symboles validés avant un échec pour un pattern sans variable

Dans la perspective d'une tentative de détection des occurrences d'un pattern au sein d'une trajectoire sans jamais relire un symbole de la trajectoire déjà lu et comparé, les auteurs proposent l'utilisation d'une table stockant tous les décalages correspondant aux différentes positions dans le pattern. Cette table est calculée à l'avance, lors de la compilation. Les décalages, appelés *bords*, sont définis comme suit :

Définition 15 (Bord). *b est un bord d'un pattern P, ssi*

i) il existe $(u, v) \in \Sigma^* \times \Sigma^*$ tels que $P = b.u = v.b$;

ii) s'il existe b' et $(u', v') \in \Sigma^* \times \Sigma^*$ tels que $P = b.u' = v'.b$ alors $|b'| < |b|$.

En d'autres termes, le bord d'un pattern est la plus longue sous-chaîne qui soit à la fois préfixe et suffixe du pattern.

Le calcul de la table des bords présente deux caractéristiques importantes :

- il s'effectue lors d'une phase statique préalable à la tentative de détection du pattern dans la trajectoire. En effet il ne nécessite que la connaissance du pattern que l'on souhaite détecter,

- l'information contenue dans cette table est utilisée lors de la phase de superposition et permet de réaliser directement le bon décalage sans avoir à revenir en arrière, c'est-à-dire sans relire des symboles de la trajectoire, garantissant par là même une évaluation linéaire en nombre de comparaisons.

Noter que ces 2 aspects auront leur importance pour notre application où l'on envisage une détection à la volée des patterns au sein des trajectoires. Nous présentons maintenant comment cette table des bords est construite ainsi que son utilisation.

Construction de la table des bords

Supposons que l'on souhaite enregistrer un nouveau pattern P de longueur m. Lors de l'enregistrement de ce pattern, l'algorithme va établir pour tous les symboles du pattern le décalage à réaliser si on obtient une situation d'échec sur ce symbole. Sur la figure 4.6, on représente les bords pour 2 positions dans le pattern. Pour la position l on a détecté un bord de longueur $i+1$ alors que pour la position k on a un bord de longueur $j+1$. Cela signifie que l'on a $P[0 \ldots i] = P[l-i \ldots l]$ et donc que pour un échec à la position $l+1$ du pattern on peut réaliser un décalage de $l-i$ symboles afin de superposer les $i+1$ derniers symboles de la trajectoire lus (égaux aux $i+1$ derniers du pattern lus) avec les $i+1$ premiers symboles du pattern.

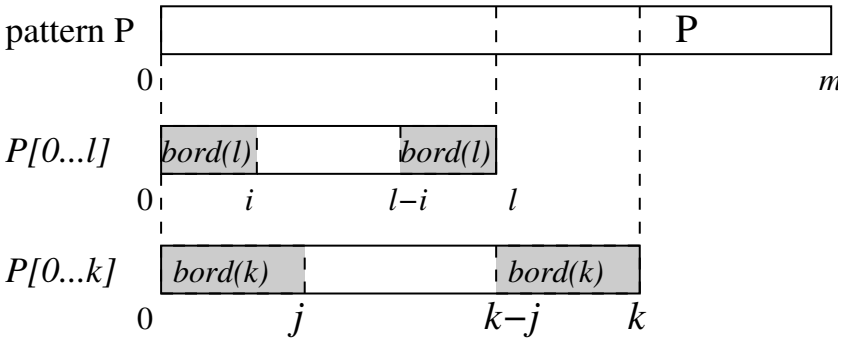


FIG. 4.6 – Exemples de bords pour un pattern P aux positions l et k

On détermine ainsi pour chacun des sous-patterns de P le plus grand bord possible et on l'enregistre dans une table qui sera utilisée lors de la phase d'évaluation. Noter que le fait de prendre le *plus grand* suffixe signifie que le décalage est minimal et garantit que l'algorithme ne manque aucune solution.

Un premier algorithme pour construire cette table consisterait à détecter le bord de chaque sous-pattern indépendamment. Cet algorithme naïf conduirait cependant à une construction de la table quadratique en nombre de comparaisons. Les auteurs de [69] constatent qu'il existe en fait une relation de récursivité liant les bords de deux sous-patterns $P[0 \ldots l]$ et $P[0 \ldots l+1]$ d'un même pattern P. Considérons la figure 4.7.

Supposons qu'un bord $bord(l)$, apparaissant en grisé sur la figure 4.7, de longueur $i+1$, ait été identifié pour le sous-pattern $P[0 \ldots l]$ de P. Cela signifie donc que les $i+1$ premiers symboles de $P[0 \ldots l]$ sont identiques aux $i+1$ derniers. Considérons maintenant le sous-pattern $P[0 \ldots l+1]$. Dans l'hypothèse où le symbole $\beta = P[i+1]$ est égal au symbole $\alpha = P[l+1]$, alors nous avons $P[0 \ldots i+1] = bord(l).\alpha = P[l-i \ldots l+1]$. Cette sous-chaîne de longueur $i+1$ correspond bien au plus long suffixe de $P[0 \ldots l+1]$ qui est également préfixe. Ainsi si $\alpha = \beta$ nous pouvons déduire directement le bord de $P[0 \ldots l+1]$ en fonction de celui de $P[0 \ldots l]$. Envisageons désormais le cas où $\alpha \neq \beta$. Il n'existe donc pas de bord de longueur $i+1$ pour $P[0 \ldots l]$. Supposons qu'il existe le bord $bord(l+1) = \omega.\alpha$ pour $P[0 \ldots l+1]$ avec

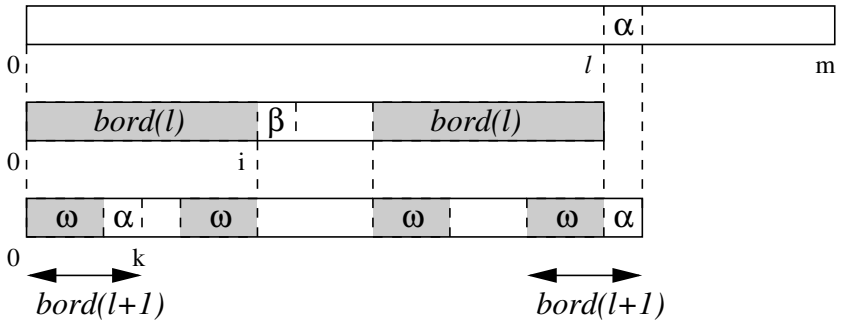


FIG. 4.7 – Illustration de la relation de récursivité entre les bords

$|\omega| < bord(l)$. On a alors ω préfixe de P puisque préfixe de $P[0 \dots l+1]$. De plus ω est un suffixe de $P[0 \dots l]$ puisque $\omega.\alpha$ est un bord de $P[0 \dots l+1]$. Comme $|\omega| < bord(l)$ on en déduit, comme la figure 4.7 l'illustre, que ω est à la fois préfixe et suffixe de $bord(l)$. Comme il est de longueur maximale avec cette propriété, il s'agit du bord de $bord(l)$. Cette situation est illustrée avec la partie basse de la figure 4.7. Si l'on ne peut pas trouver un tel mot ω, on réitère avec le bord de ω.

Cette relation de récurrence entre bords pour un pattern P peut s'exprimer sous la forme suivante :

$bord(P[0 \dots l+1]) =$

$$\begin{cases} bord(P[0 \dots l]).P[l+1] & si \;\; bord(P[0 \dots l]).P[l+1] \;\; préfixe \;\; de \;\; \text{P} \\ bord(bord(P[0 \dots l]).P[l+1]) & sinon \end{cases}$$

Exemple 11. *La figure 4.8 décrit un exemple pour lequel on a identifié un bord de longueur 5 pour le sous-pattern $P[0 \dots 12]$, à savoir $bord(12) =$ `a.b.c.a.b` $= P[0 \dots 4]$. Pour déterminer le bord du sous-pattern $P[0 \dots 13]$ on regarde dans un premier temps si $bord(12).P[13] =$ `a.b.c.a.b.c` est un bord. Pour cela il suffit donc de tester si $P[13] =$ `c` et $P[5] =$ `a` sont égaux, ce qui n'est pas le cas. Dans une étape précédente on avait calculé le bord de $P[0 \dots 4]$: $bord(4) =$ `a.b`. On vérifie ici que $bord(4).P[13] =$ `a.b.c` est un bord. Le bord $bord(13)$ sera donc : $bord(bord(12)).P[13] =$ `a.b.c`.*

Cet algorithme récursif permet d'établir la table des bords en temps $O(n)$, contrairement à l'algorithme naïf qui est en temps quadratique.

Utilisation de la table des bords

Une application directe de cette technique peut être la détection d'un pattern au sein d'un flux. En effet, on peut envisager qu'au lieu de connaître la trajectoire à l'avance, on réalise

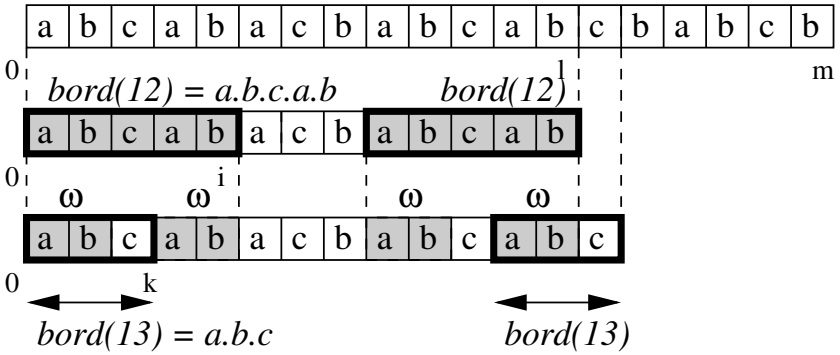


FIG. 4.8 – Exemple de calcul de $bord(13)$ en fonction de $bord(12)$

la tentative de superposition au fur et à mesure que l'on reçoit un nouveau symbole de la trajectoire. Cette détection peut s'effectuer en effet en temps réel grâce à la connaissance des bords permettant de réaliser le décalage du pattern sans avoir à relire ni retester un symbole de la trajectoire. Le nombre de comparaisons est *linéaire* en la taille de la trajectoire en entrée.

Conclusion : application à notre problématique

Dans notre modèle l'algorithme KMP offre des perspectives intéressantes, à savoir :

- puisque les trajectoires sont représentées comme des séquences de zones traversées et les requêtes sont exprimées à l'aide de patterns, ces techniques de superposition nous permettent de déterminer de manière optimisée en nombre de comparaisons (et donc temps de calcul) le résultat des requêtes ;

- il est peu coûteux d'établir une table des bords lors de la compilation d'une requête, surtout si on utilise un algorithme optimisé de construction de la table comme dans KMP ;

- en dehors de la phase statique de construction de la table des bords, l'utilisation de cette table est possible comme on l'a vu précédemment pour détecter une occurrence d'un pattern dans un flux de données, sans avoir à relire de symboles de la trajectoire. Cette propriété est particulièrement intéressante pour nos trajectoires qui sont construites au fur et à mesure, les nouvelles positions reçues formant un flux de données.

La difficulté est d'adapter cet algorithme à nos patterns qui possèdent des variables.

Algorithme de KMP étendu

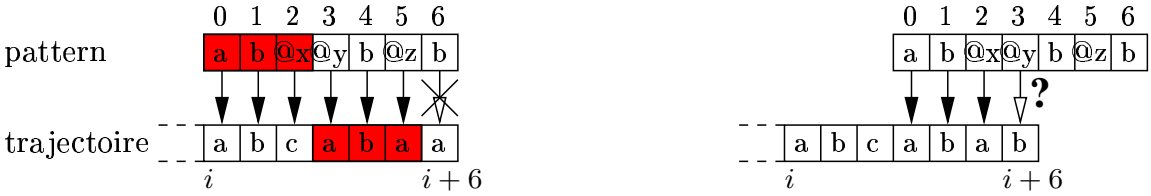
Considérons le pattern P = `a.b.@x.a.b.a.b` avec une unique variable `@x` et l'exemple de la figure 4.9.

(a) un échec se produit à la pos. 6 `@x` est instanciée à `c`

(b) on décale directement de 3 symboles à droite et on instancie `@x` à `a`

FIG. 4.9 – Un décalage pour un pattern avec une variable.

Quand un échec se produit lors de la tentative de comparaison, à la position 6, `@x` est instanciée à `c`. Si nous considérons la chaîne `a.b.c.a.b.a`, le plus long préfixe qui est aussi un suffixe est `a.b`. Cependant ce décalage annule l'instanciation de `@x` et nous pouvons lier cette variable désormais à un autre symbole. En effet il est maintenant possible de superposer les trois premiers symboles de $P[0].\cdots.P[5]$ avec les trois derniers, en établissant que `@x` est liée à `a` *après* le décalage.



(a) échec à la pos. 6, `@x` est instanciée à `c`, `@y` à `a`, et `@z` à `a`

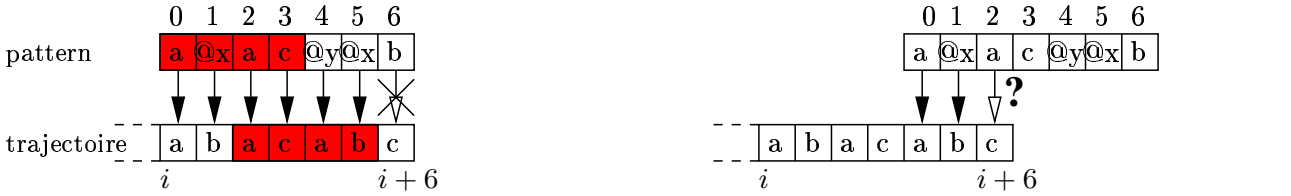
(b) décale de 3 pos. vers la droite, `@x` est instanciée à `a`, `@y` et `@z` n'est pas instanciée

FIG. 4.10 – Un décalage impliquant une substitution

Considérons ensuite un cas plus complexe où les instanciations *après* le décalage dépendent des instanciations *avant* l'échec (figure 4.10). Ici un échec intervient à la position 6, `@x` étant instanciée à `c`, `@y` à `a` et `@z` à `a`. Le meilleur décalage superpose les trois premiers symboles du pattern avec les trois derniers symboles de la trajectoire, avec une nouvelle instanciation de `@x` à `a` tandis que `@y` et `@z` ne sont plus instanciées. Notons que dans le cas présent la nouvelle instanciation de `@x` vaut l'ancienne instanciation de `@z`. Ici encore l'analyse du pattern lors de la compilation fournit toute l'information nécessaire pour réaliser la substitution des valeurs lors de l'évaluation.

Finalement un dernier exemple (figure 4.11) montre que le bord dépend parfois de l'instanciation des variables *avant* le décalage. Dans figure 4.11.a, un échec intervient à la position 6. Si l'on considère un bord de longueur 4, le suffixe `a.c.@y.@x` peut être superposé sur le

préfixe `a.@x.a.c` si `@x` est lié à `c` avant qu'ait lieu le décalage. Ce n'est pas le cas dans la figure 4.11 puisque `@x` est instanciée à `b`. Si nous considérons un bord de longueur 2, le suffixe `@y.@x` doit être superposé au préfixe `a.@x`. Ceci est possible seulement si `@y` est instanciée à `a`. Ici la possibilité d'appliquer un bord dépend de la valuation courante.



(a) échec à la pos. 6, `@x` instanciée à `b`, `@y` à `a` puisque `@x` est instanciée à `b`, le bord de longueur 4 n'est pas possible.

(b) puisque `@y` était instanciée à `a`, appliquer le bord de longueur 2 est possible ; alors `@x` est maintenant instanciée à `b`.

FIG. 4.11 – Un décalage dépendant de l'instanciation courante des variables.

La table des bords

Comme nous l'avons vu dans les exemples précédents, le calcul des bords à utiliser lors d'une tentative d'évaluation est fortement lié aux instanciations des variables. De plus, un décalage peut déterminer une substitution des variables qui dépend, partiellement ou totalement, de la substitution courante lors de l'échec. Nous définissons maintenant la notion de *bord* pour des patterns avec variables.

Définition 16 (Bord d'un pattern). *Soit P un pattern de longueur m. Un bord de P est un triplet $(length, \nu_{min}, \sigma_{shift})$, où ν_{min} est une valuation et σ_{shift} une substitution, qui satisfont les propriétés suivantes :*

- $\nu_{min}(\sigma_{shift}(P[0].\cdots.P[length-1])) = \nu_{min}(P[m-length].\cdots.P[m-1])$
- *il n'existe pas un bord $e' = (length, \nu'_{min}, \sigma'_{shift})$ avec $\nu'_{min} \subseteq \nu_{min}$.*

Un bord $e = (length, \nu_{min}, \sigma_{shift})$ décrit un décalage de taille $m - length - 1$. La valuation ν_{min} exprime la condition nécessaire et suffisante pour appliquer un bord. En effet, si l'on considère une valuation courante ν, le bord e est applicable si et seulement si $\nu_{min} \subseteq \nu$ (on dira parfois que ν est *compatible* avec ν_{min}). Finalement σ_{shift} est la substitution utilisée pour instancier les variables du bord après le décalage.

Supposons que lors d'une tentative de superposition d'un pattern P sur une trajectoire T se produit un échec à la position l de P. Si $(length, \nu_{min}, \sigma_{shift})$ est un bord de $P[0].\cdots.P[l-1]$ et ν_{min} est un sous-ensemble de la valuation courante ν, alors nous pouvons décaler P de $l - length - 1$ symboles vers la droite et recommencer la tentative de

superposition à la position $length+1$ de P (voir figure 4.12). La nouvelle valuation courante est $\nu \circ \sigma_{shift}$. Nous illustrons ces concepts avec l'exemple suivant.

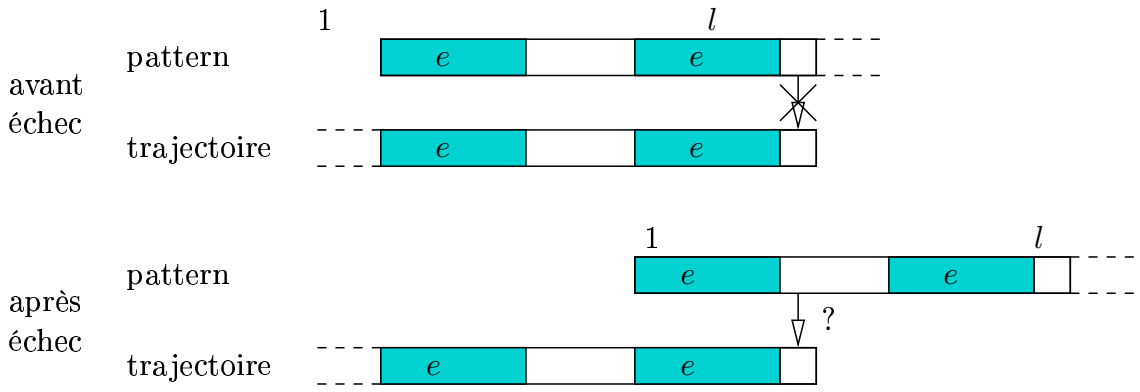


FIG. 4.12 – Exemple de bord.

Exemple 12. *Considérons le sous-pattern* `@x.b.@y.c.@z.@x.a`. *Il existe un bord* $e(3, \nu_{min}, \sigma_{shift})$ *de longueur 3 avec :*

- $\nu_{min} = \{$`@x/b`$\}$
- $\sigma_{shift} = \{$`@x/@z`, `@y/a`$\}$

Lors d'un décalage de taille 3, le sous-pattern `@x.b.@y` *doit être superposé avec* `@z.@x.a`. `@x` *remplace alors* `@z`, `b` *remplace* `@x` *et* `@y` *remplace* `a`. *Cette superposition est possible si et seulement si la valuation courante de* `@x` *est* `b`, *par conséquent la valuation minimale est* $\nu_{min} = \{$`@x/b`$\}$.

Ensuite, puisque `@x` *remplace* `@z`, *elle prend la valeur affectée à* `@z` *par la valuation courante. La variable* `@y` *prend toujours la valeur* `a`. *La substitution est par conséquent* $\sigma_{shift} = \{$`@x/@z`, `@y/a`$\}$. *On vérifie facilement que :*

$$\nu_{min}(\sigma_{shift}(\texttt{@x.b.@y})) = \nu_{min}(\texttt{@z.@x.a}) = \texttt{@z.b.a}$$

Supposons enfin que le suffixe de la trajectoire soit `c.b.a` *quand l'échec intervient. La valuation courante est* $\nu = \{$`@x/b`, `@z/c`$\}$. *Puisque* $\nu_{min} \subseteq \nu$, *ce bord est applicable et le décalage de longueur 3 peut être réalisé. La valuation après le décalage, obtenue à partir de la substitution, est :*

$$\texttt{@x} = \nu(\sigma_{shift}(\texttt{@x})) = \nu(\texttt{@z}) = \texttt{c},\ et\ \texttt{@y} = \nu(\sigma_{shift}(\texttt{@y})) = \texttt{a}.$$

La tentative de superposition est réalisée par l'algorithme SUPERPOSE présenté ci-dessous. SUPERPOSE est invoqué lorsqu'une nouvelle étiquette s est concaténée à une trajectoire T.

Il prend comme paramètres d'entrée le symbole s, la valuation courante ν et la position courante l dans P. SUPERPOSE tente la superposition entre T et le suffixe du pattern P en débutant de la position l. Il retourne la nouvelle valuation ν' et la nouvelle position l' dans P.

SUPERPOSE(s, l, ν)
Entrée : s (étiquette de la trajectoire), l (position courante dans P), ν (valuation courante)
Sortie : ν' (nouvelle valuation pour P), l' (prochaine position à tester dans P)
début
  si ($P[l] \in \mathcal{V}$ **et** $P[l] \notin bound(\nu)$) **alors** // $P[l]$ est une variable pas encore instanciée
    $\nu' := \nu \cup \{P[l]/s\}$
    $l' := l + 1$
  sinon si ($P[l] = s$ **ou** ($\{P[l]/t_l\} \in \nu$ **et** $t_l = s$) **alors** // $P[l]$ est égal à (ou déjà instanciée par) s
    $\nu' := \nu$
    $l' := l + 1$
  sinon // superposition impossible entre $P[l]$ et s : on utilise le bord
    si ($l = 0$) **alors**
      retourne $(\emptyset, 0)$ // Pas de bord applicable : on décale le pattern entier
    sinon
      $(\nu', l') :=$ DÉCALEBORD (ν, l)
      retourne SUPERPOSE (s, l', ν')
    fin
  finsi
  // Si on a atteint le dernier symbole de P sans avoir d'échec, la trajectoire est ajoutée au résultat
  si ($l' = m + 1$) **alors**
    ajouteTrajectoireAuResultat()
    // on décale le pattern pour détecter une nouvelle occurrence possible plus tard
    $(\nu', l') :=$ DÉCALEBORD (ν', l')
  finsi
  retourne (ν', l')
fin

Si l' est égal à la taille de P, alors le pattern a été entièrement reconnu et la trajectoire est ajoutée au résultat de la requête. Sinon SUPERPOSE retourne la nouvelle position $l' < m$ dans P. Lors de la réception d'un nouveau symbole T, une superposition sera à nouveau tentée entre le suffixe du pattern P débutant à la position l' et la trajectoire T.

SUPERPOSE appelle la procédure DÉCALEBORD qui prend le plus long bord e tel que la valuation ν_{min} soit un sous-ensemble de ν. Une fois e trouvée (dans le pire des cas il s'agira du bord par défaut qui décale la totalité du pattern), le décalage est réalisé comme suit :

- P est décalé de $l - length - 1$ symboles vers la droite et la position courante dans P devient alors $e.length + 1$;
- la nouvelle valuation courante ν' est égale à $\nu \circ \sigma_{shift}$.

DÉCALEBORD prend en entrée la valuation courante et la position actuelle dans le pattern P et retourne la nouvelle valuation courante et la nouvelle position dans P.

```
DécaleBord (ν, l)
début
    // Prend les bords associés à la position actuelle dans P, stockés par ordre de longueur décroissante
    ν' = ∅
    pour chaque e dans Bords[l] faire
        si (e.ν_min ⊆ ν) alors
            // Un bord est trouvé, éventuellement le bord par défaut dont la longueur est 0
            // Décalage correct du pattern
            l' := e.longueur
            // Fixe la nouvelle substitution courante
            pour chaque {x_j/t} dans σ_shift faire
                si (t ∈ Σ) alors ν' := ν' ∪ {x_j/t}
                sinon ν' := ν' ∪ {x_j/ν(t)}
            fin
            retourne (ν', l')
        finsi
    fin
fin
```

Considérons par exemple le pattern $P =$ `a.@x.b.a.@x.@y.c.d`. Chaque fois qu'un échec se produit à la position $l = 6$ du pattern, nous avons besoin de considérer les bords suivants pour le sous-pattern $P[0].\cdots.P[5] =$ `a.@x.b.a.@x.@y` :

- $(0, \emptyset, \emptyset)$ le bord par défaut qui correspond au décalage du pattern entier ;
- $(1, \{$`@y/a`$\}, \emptyset)$, parce que si ν est compatible avec $\{$`@y/a`$\}$ (c'est-à-dire, la dernière est un sous-ensemble de la première), alors le suffixe de la trajectoire est de la forme `a.@x.b.a.@x.a`, et un décalage d'un symbole est possible ;
- $(2, \{$`@x/a`$\}, \{$`@x/@y`$\})$, parce que si ν est compatible avec $\{$`@x/a`$\}$, le suffixe de la trajectoire est de la forme **a.a.**`b.a.a.`ν(`@y`). Nous pouvons remplacer `a.@y` par `a.@x`, la nouvelle instanciation de `@x` étant l'ancienne instanciation de `@y` ;
- $(3, \{$`@y/b`$\}, \{$`@x/@x`$\})$ est un bord, pour des raisons similaires.

Nous ne pouvons trouver aucun bord dont la longueur soit 4 ou 5.

Considérons maintenant une tentative de superposition et un échec se produisant à la position 5 du pattern, avec $\nu = \{$`@x/a, @y/c`$\}$. La valuation ν n'est pas compatible avec ν_{min} du bord $(3, \{$`@y/b`$\}, \{$`@x/@x`$\})$. Cependant elle est compatible avec celle du bord $(2, \{$`@x/a`$\}, \{$`@x/@y`$\})$. Par conséquent, en utilisant ce bord, on décale le pattern de deux symboles vers la droite et initialise une nouvelle tentative de superposition démarrant à la troisième position du pattern avec la nouvelle valuation courante.

Construction de la table des bords

Une technique brutale pour construire la table des bords de P consiste à considérer, pour chaque sous-pattern $P[0 \dots m-1]$, tous les bords possibles de longueur $i < m-1$. Cet algorithme s'exécute en $O(m^3)$ (en nombre de comparaisons).

Un meilleur algorithme utilise les bords déjà calculés à la position $l-1$ pour déduire les bords à la position l. Cet algorithme repose sur la propriété suivante (voir Figure 4.13) :

Lemme 1. *Soit $e = (i+1, \nu_{min}, \sigma_{shift})$ un bord pour $P[0 \dots l]$. Il existe alors un bord $e' = (i, \nu'_{min}, \sigma'_{shift})$ pour $P[0 \dots l-1]$ avec*

$$\begin{cases} \nu'_{min} = \nu_{min}|_{var(P[l-i \dots l-1])} \\ \sigma'_{shift} = \sigma_{shift}|_{var(P[0 \dots i-1])} \end{cases}$$

où $\sigma_{|var(q)}$ (resp. $\nu_{min}|_{var(q)}$) désigne la restriction de σ (resp. ν_{min}) aux variables dans q.

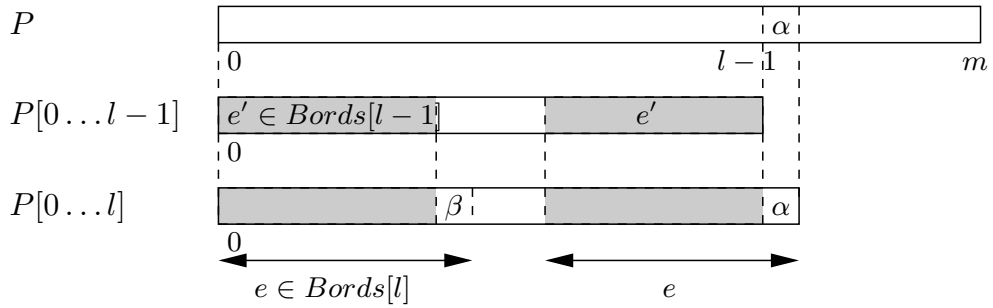


FIG. 4.13 – Calcul de $Bords[l-1]$ à partir de $Bords[l]$

Preuve: Soit $e = (i+1, \nu_{min}, \sigma_{shift})$ un bord de $P[0 \dots l]$. D'après la définition d'un bord, $\nu_{min}(\sigma_{shift}(P[0 \dots i]) = \nu_{min}(P[l-i \dots l]) = \omega.\alpha$, avec $\omega.\alpha \in (\Sigma \cup \mathcal{V})^* \times (\Sigma \cup \mathcal{V})$. Par conséquent nous avons

$$\nu_{min}(\sigma_{shift}|_{var(P[0 \dots i-1])}(P[0 \dots i-1])) = \nu_{min}|_{var(P[l-i \dots l-1])}(P[l-i \dots l-1]) = \omega$$

En d'autres termes, soit $\nu'_{min} = \nu_{min}|_{var(P[l-i \dots k-1])}$ et $\sigma'_{shift} = \sigma_{shift}|_{var(P[0 \dots i-1])}$, alors $(i-1, \nu'_{min}, \sigma'_{shift})$ est un bord e' pour le pattern $P[0 \dots l-1]$. □

Ce lemme conduit à un algorithme optimisé ConstructionBords qui construit de manière itérative la table des bords, déduisant $Bords[l]$ des bords de l'ensemble $Bords[l-1]$. Chaque étape de l'algorithme est de la forme :

ConstructionBords :$Etape[l]$
$Bords[l] := (0, \emptyset, \emptyset)$
pour chaque $(i, \nu_{min}, \sigma_{shift}) \in Bords[l-1]$
  si $P[i] \in \Sigma$ **alors**
  // cas où les deux symboles à comparer sont égaux
  // → un bord avec même ν_{min} et σ_{shift} est trouvé
    si $P[l] \in \Sigma$ **et** $P[l] = P[i]$ **alors** $(i+1, \nu_{min}, \sigma_{shift}) \in Bords[l]$
    sinon si $P[l] \in \mathcal{V}$ **alors**
    // cas où le symbole du suffixe est une variable instanciée par le symbole du préfixe
    // → un bord avec même ν_{min} et σ_{shift} est trouvé
      si $P[l]/P[i] \in \nu_{min}$ **alors** $(i+1, \nu_{min}, \sigma_{shift}) \in Bords[l]$
      // → un bord est trouvé, mais on ajoute une instanciation dans ν_{min}
      sinon $(i+1, \nu_{min} \cup \{P[l]/P[i]\}, \sigma_{shift}) \in Bords[l]$
      finsi
    finsi
  sinon // cas où le symbole du préfixe est une variable
    // cas où les deux symboles forment déjà une substitution
    // → un bord avec même ν_{min} et σ_{shift} est trouvé
    si $P[i]/P[l] \in \sigma_{shift}$ **alors** $(i+1, \nu_{min}, \sigma_{shift}) \in Bords[l]$
      // sinon un bord est trouvé, mais on ajoute une substitution dans σ_{shift}
    sinon $(i+1, \nu_{min}, \sigma_{shift} \cup \{P[i]/P[l]\}) \in Bords[l]$
    finsi
  finsi
fin

Exemple 13. *Considérons le pattern* `a.@x.b.a.@x.@y` *et supposons que nous ayons déjà calculé les bords à la position 4 :*

$$Bords[4] = \{(0, \emptyset, \emptyset), (1, \{\texttt{@x/a}\}, \emptyset), (2, \emptyset, \{\texttt{@x/@x}\})\}$$

Pour chacun de ces bords de longueur respective $0, 1$ *et* 2*, nous calculons les bords possibles de longueur* $1, 2$ *et* 3 *pour* $Bords[5]$ *et ajoutons le bord par défaut* $(0, \emptyset, \emptyset)$*.*

- $(0, \emptyset, \emptyset) \in Bords[4]$ *et* $(P[0], P[5]) \in \Sigma \times \mathcal{V}$
  $\Rightarrow (1, \{\texttt{@y/a}\}, \emptyset) \in Bords[5]$

- $(1, \{\texttt{@x/a}\}, \emptyset) \in Bords[4]$ *et* $(P[1], P[5]) \in \mathcal{V}^2$
  $\Rightarrow (2, \{\texttt{@x/a}\}, \{\texttt{@x/@y}\}) \in Bords[5]$

- $(2, \emptyset, \{\texttt{@x/@x}\}) \in Bords[4]$ *et* $(P[2], P[5]) \in \Sigma \times \mathcal{V}$
  $\Rightarrow (3, \{\texttt{@y/b}\}, \{\texttt{@x/@x}\}) \in Bords[5]$

Pour finir nous ajoutons le bord par défaut $(0, \emptyset, \emptyset)$ *à* $Bords[5]$*.*

Proposition 5. *L'algorithme* ConstructionBords *est dans le pire des cas quadratique en la taille du pattern.*

Preuve:

Remarquons tout d'abord que dans le pire des cas nous avons l bords dans $Bords[l-1]$ d'un pattern P, avec des longueurs allant de 0 à $l-2$. L'algorithme ConstructionBords calcule un bord de $Bords[l]$ de longueur $i+1$ à partir d'un bords de $Bords[l-1]$ de longueur i si la superposition entre $P[l]$ et $P[i]$ est réalisable. Ainsi l'ensemble $Bords[l]$ est calculé à partir de l'ensemble $Bords[l-1]$ en effectuant une comparaison pour chaque bord de $Bords[l-1]$ (nous ajoutons également le bord par défaut à $Bords[l]$). Par conséquent dans le pire des cas, l comparaisons sont nécessaires, ce qui conduit à un total de $\sum_{l=1}^{m-1} l = \frac{(m-1)(m-2)}{2}$ comparaisons pour la table des bords complète. □

Évaluation

Dans le but de réaliser notre évaluation d'une requête en temps-réel sans réévaluer les symboles déjà testés, nous utilisons des automates et la table des bords. A partir d'un état initial nous créons une transition qui correspond au premier symbole du pattern et ainsi de suite jusqu'à ce que nous atteignions le dernier symbole du pattern. De plus nous associons à chaque état de l'automate une fonction de retour h construite à partir de la table des bords et qui nous permet de retourner directement au bon état lorsqu'un échec se produit. Pour franchir une transition étiquetée avec une variable, cette variable doit être instanciée par le même symbole que le symbole de la trajectoire lu, ou alors ne pas être instanciée (dans ce cas la variable est instanciée lorsque l'on franchit la transition). La figure 4.14 illustre un tel automate.

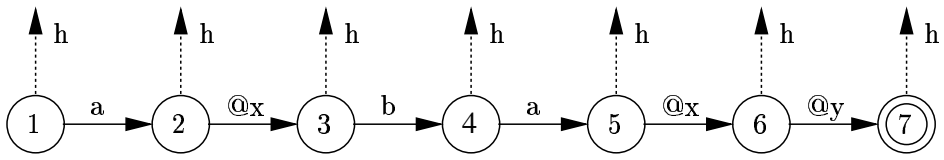


Fig. 4.14 – Exemple d'automate associé au pattern `a.@x.b.a.@x.@y`

Puisque pour une position donnée dans le pattern nous avons plusieurs bords possibles, nous essayons d'abord celui dont la longueur est la plus grande et qui correspond par conséquent au plus petit décalage. Si la valuation actuelle n'est pas compatible avec ν_{min}, alors nous essayons le deuxième bord le plus long, etc. Cette stratégie garantit que nous n'oublions aucune solution en décalant trop le pattern. Noter que le bord par défaut, qui correspond au décalage de toute la longueur du pattern, peut être emprunté quelle que soit la valuation courante.

Une autre solution, plutôt que de tester les bords un par un suivant leur longueur décroissante serait d'associer à chaque état de l'automate un arbre permettant de déterminer directement, suivant les conditions données, le bord à considérer (cf. exemple, l'arbre figure

4.15). Les règles de franchissement de transitions avec variables sont identiques à celles présentées à la section précédente. Pour chaque objet il est toujours suffisant de stocker pour son statut l'état de l'automate dans lequel il se trouve ainsi que la valuation des variables.

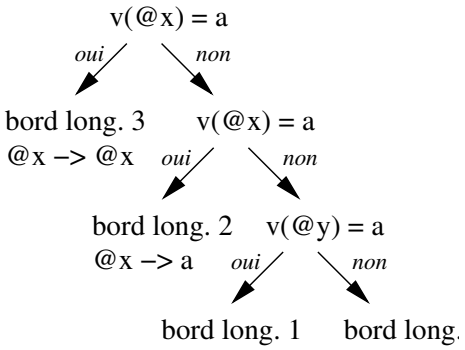


FIG. 4.15 – Arbre de décision du bord associé à l'état 7 de l'automate

Cette méthode garantit donc une évaluation sans réexaminer les symboles du pattern de la trajectoire de l'objet ainsi qu'un faible espace nécessaire en mémoire. Lorsqu'un objet atteint l'état final de l'automate d'une requête, il est ajouté au résultat de cette requête. Lorsqu'il quitte cet état, il est retiré du résultat.

4.4 Expérimentations

Dans le but de valider notre approche, nous avons implanté et comparé deux algorithmes en Java sur un Pentium PIV processor (3000MHz) avec 1024Mo de mémoire vive. Le premier algorithme, NAÏF, est l'algorithme naïf décrit en section 4.3.1, qui décale le pattern d'un symbole vers la droite à chaque fois qu'un échec intervient. L'autre, MATCH, est l'algorithme de KMP étendu qui utilise la table des bords. Nous comparons leurs performances sur une simulation d'application de suivi d'objets mobiles. Les véhicules se déplacent à travers les régions administratives du territoire métropolitain français. Un simulateur génère des trajectoires synthétiques et les patterns de mobilités, et l'évaluation continue des requêtes est réalisée et analysée sur ces données synthétiques. Pour cette évaluation, j'ai développé un prototype simplifié de celui présenté dans le chapitre 6, mais permettant de générer plus d'objets et également un grand nombre de requêtes.

4.4.1 Génération des données

Le territoire choisi pour les trajectoires est une carte de France. Nous considérons plusieurs sous-divisions de la France dont la plus fine est une partition en 21 régions administratives. La trajectoire d'un véhicule donné est simulée comme suit. La région de départ d'un véhicule est choisie au hasard. Pour simuler la réception de positions GPS, le temps est modélisé comme une séquence d'instants. A chaque instant, avec une probabilité p_i, chaque véhicule entre dans la région i ou reste (avec une probabilité $1 - \sum p_i$) dans la région dans

laquelle il se trouvait. S'il quitte une région pour une région contiguë i, alors l'étiquette correspondant à la région i est ajoutée à la représentation de sa trajectoire, et cet événement déclenche un nouveau pas dans les algorithmes de superposition. La probabilité d'entrer ou quitter une région dépend de l'importance de la région (*e.g.*, les grandes agglomérations font que les régions ont beaucoup de trafic entrant et sortant).

Les patterns de mobilité (requêtes) sont générés comme suit. Des séquences de régions de longueur variables sont générées aléatoirement. Avec une probabilité donnée p_v, un symbole de région dans un pattern peut être remplacé par une variable v, choisie aléatoirement dans un ensemble de variables. Il s'agit d'un tirage avec remise, une variable pouvant donc être choisie à nouveau pour un même pattern. Ainsi par exemple si $p_v = 0.25$, un symbole sur 4 dans le pattern sera, en moyenne, remplacé par une variable.

Exemple 14. *Imaginons que nous souhaitons construire un pattern de longueur 4. Tout d'abord nous tirons aléatoirement le premier symbole et nous obtenons celui de la région de Paris, c'est-à-dire l'étiquette* `K`*. Quand un véhicule quitte cette région, les probabilités fixées indiquent qu'il a 40% de chance d'entrer en Bourgogne (cette forte probabilité s'explique par le fait qu'il s'agit d'un axe majeur reliant le nord et le sud du pays), alors que les probabilités pour entrer en Normandie chutent à 8% (en effet peu de grandes villes situées dans cette région et l'axe routier est d'importance moindre). Supposons que l'on tire le chiffre 18 qui désigne une entrée dans la région de Bourgogne, dont l'étiquette est* `N`*, nous concaténons ce symbole au pattern actuel et nous obtenons le pattern* `K.N`*. De manière similaire nous tirons les deux régions suivantes de la trajectoire qui sont les régions Rhône-Alpes et Languedoc-Roussillon, dont les étiquettes respectives sont* `S` *et* `U`*. Notre pattern est donc* `K.N.S.U`*. Supposons maintenant que le taux de variables moyen dans un pattern a été fixé à 25% et que nous ayons un ensemble de 3 variables distinctes,* `@x`*,* `@y`*,* `@z`*. Pour chaque symbole de notre pattern, nous tirons un nombre compris entre 1 et 100 et chaque valeur inférieure à 25 indique que le symbole doit être remplacé par une variable. Ici nous tirons pour notre pattern les 4 nombres suivants : 75, 48, 18 et 29. Ainsi seul le troisième symbole sera remplacé par une variable. Un dernier tirage donnant le nombre 44, nous choisissons la variable* `@y`*. Par conséquent, notre pattern est finalement* `K.N.@y.U`*.*

Durant l'exécution de la simulation, nous construisons aléatoirement les trajectoires des véhicules en ajoutant une nouvelle étiquette de région suivant les probabilités de rester ou quitter une région pour entrer dans une région voisine. Le mécanisme de génération des trajectoires est assez proche de celui des patterns présenté ci-dessus. A chaque unité de temps, pour simuler la réception d'un nouvel événement GPS, nous tirons un nombre qui détermine si le véhicule quitte ou non la région pour une autre, et si oui quelle est cette région. Noter que contrairement à la génération des trajectoires présentée dans le prototype au chapitre 6, nous ne générons pas une position précise sur la carte 2D, mais uniquement l'étiquette de la zone dans laquelle se trouve l'objet. De cette façon nous évitons les coûts importants dus aux opérations de pointé dans un polygone et pouvons traiter plus d'objets

et de requêtes, en revanche nous ne pouvons plus visualiser sur une carte 2D la position précise des objets.

Exemple 15. *Considérons que nous générons aléatoirement le premier symbole d'une trajectoire et que nous obtenions encore la région parisienne, d'étiquette* `K`*. La probabilité qu'un véhicule actuellement dans la région de Paris reste dans cette région est de 60%. Celle d'entrer en région Bourgone est 24%, etc. A chaque nouvelle unité de temps de la simulation, on tire aléatoirement un nombre entre 1 et 100. Si ce nombre est inférieur à 60, le véhicule reste en région parisienne.* 55, 27 *et* 66 *sont tirés successivement aux instants* 2, 3 *et* 4*. Par conséquent ce véhicule reste en région parisienne pour* 3 *unités de temps, et entre à l'instant* 4 *dans une nouvelle région qui est, d'après les probabilités, la région Bourgogne. Ainsi une nouvelle étiquette* `N` *est ajoutée à la représentation de la trajectoire ce qui déclenche des traitements au niveau de chacune des requêtes conduisant à de possibles mises à jour des résultats.*

4.4.2 Expériences

L'évaluation des deux algorithmes est basée sur le nombre total de comparaisons entre un symbole du pattern et un symbole de la trajectoire. Pour nos tests nous avons fixé le nombre de véhicules à 100.000 et le nombre de requêtes continues supportées par le système à 500. Nous observons le coût et la consommation de ressources des deux algorithmes durant 20 unités de temps, et évaluons leurs performances en prenant en considération les paramètres suivants :

- la longueur des patterns ;
- la proportion moyenne de variables dans chaque pattern ;
- le nombre de régions dans la partition de l'espace.

La figure 4.16 illustre l'impact de la proportion de variables sur le nombre total de comparaisons, lorsque le nombre de régions (donc la taille du vocabulaire d'étiquettes) est égal à 21. Nous considérons successivement les patterns de longueur 4, 6, 8 et 10. Les patterns plus courts (resp. plus longs) capturent trop (resp. pas assez) de véhicules et sont par conséquent d'un intérêt limité. Les quatre graphes de la figure 4.16 présentent les résultats pour les patterns de longueur respectives 4, 6, 8 et 10.

Comme attendu, notre algorithme de KMP étendu est plus performant dans tous les cas que l'algorithme naïf. Un aspect intéressant souligné par ces expériences est que la proportion de variables dans les patterns a une influence opposée sur les performances des deux algorithmes. Alors que le nombre de comparaisons diminue pour MATCH lorsque le taux

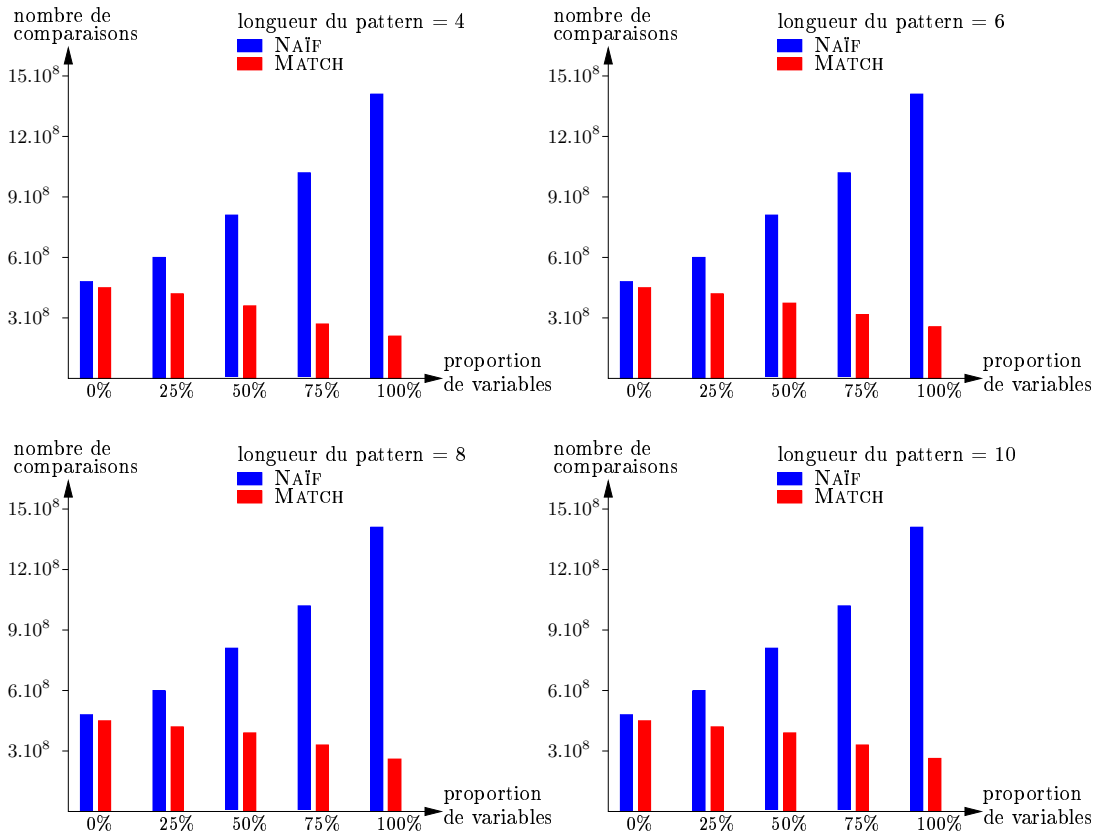


FIG. 4.16 – Nombre de comparaisons suivant la longueur du pattern et le taux de variables

de variables augmente, il augmente pour NAÏF. De plus, plus la proportion de variables est grande et plus la longueur du pattern est petit, plus l'algorithme MATCH permet d'économiser des comparaisons. Par exemple avec 25% de variables et une longueur de 4 pour les patterns, MATCH nécessite 13% de comparaisons en moins. Lorsque le pattern est composé seulement de variables (taux de 100%), le gain passe à 85%.

L'algorithme NAÏF décale aveuglément le pattern d'une position en avant lorsque survient un échec, sans chercher à déterminer si ce décalage peut potentiellement conduire à un succès ou non. Quand le pattern est fortement sélectif (*i.e.*, a peu de chance d'être superposé à une trajectoire), un échec a une grande probabilité de se produire sur le premier ou deuxième symbole, et le comportement de l'algorithme NAÏF reste proche de celui de MATCH parce que dans ce cas l'information donnée par les bords n'apporte pas beaucoup de valeur ajoutée. En fait, la différence entre les deux algorithmes quand le taux de variables est faible correspond au gain de l'algorithme KMP original (non-étendu) sur des chaînes de caractères.

La différence entre les deux algorithmes évolue avec le nombre de variables parce que les variables rendent le pattern plus générique, et donc augmentent sa probabilité d'être super-

posable, au moins partiellement, avec des trajectoires. C'est donc dans ce cas que l'utilisation de la table des bords apporte beaucoup, et que les comportements des deux algorithmes divergent. Dans le cas de NAÏF, le nombre de comparaisons est proportionnel à la taille de la superposition partielle. En effet, quand un échec intervient à la position l, tous les décalages possibles entre 0 et l doivent être successivement étudiés par NAÏF, et chaque décalage répète des comparaisons entre les mêmes symboles de la trajectoire et différents préfixes du pattern.

De son côté, MATCH tire avantage de la table des bords pour limiter le nombre de comparaisons. Lorsqu'un échec se produit à la position l, une forte proportion de variables dans le pattern favorise l'existence d'un ou de plusieurs bords, et donc il est possible grâce à la table des bords de réaliser directement le bon décalage sans faire de comparaisons supplémentaires. La figure 4.17 montre le nombre moyen de bords par requête suivant la proportion de variables.

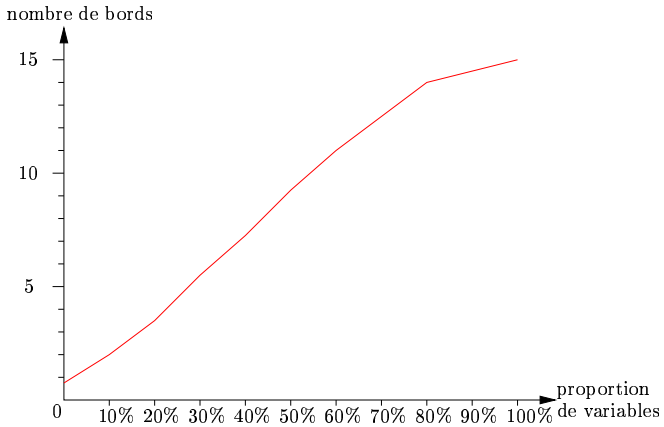


FIG. 4.17 – Nombre moyen de bords pour un pattern de longueur 6

Nous avons également étudié l'influence de la taille du vocabulaire Σ (étiquettes des régions). Clairement, dans le contexte du KMP standard, un vocabulaire de taille réduite augmente la probabilité de trouver des bords pour un pattern. Afin d'évaluer comment cette taille a un impact sur notre algorithme étendu, j'ai agrégé certaines des 21 régions initiales pour obtenir un nombre plus faibles de régions plus grandes. Le résultat obtenu pour une partition en 12 ou 6 régions montre que la taille de la partition a un impact faible sur les performances. Par exemple pour une partition de l'espace en 6 régions et un taux de 25% ou 50% de variables, nous avons constaté une augmentation du nombre de bords de moins de 10% et une diminution du nombre de calculs de moins de 10% également.

Finalement, la figure 4.18 montre que, comme nous nous y attendions, le nombre de comparaisons croît linéairement en fonction de la durée d'exécution des requêtes continues.

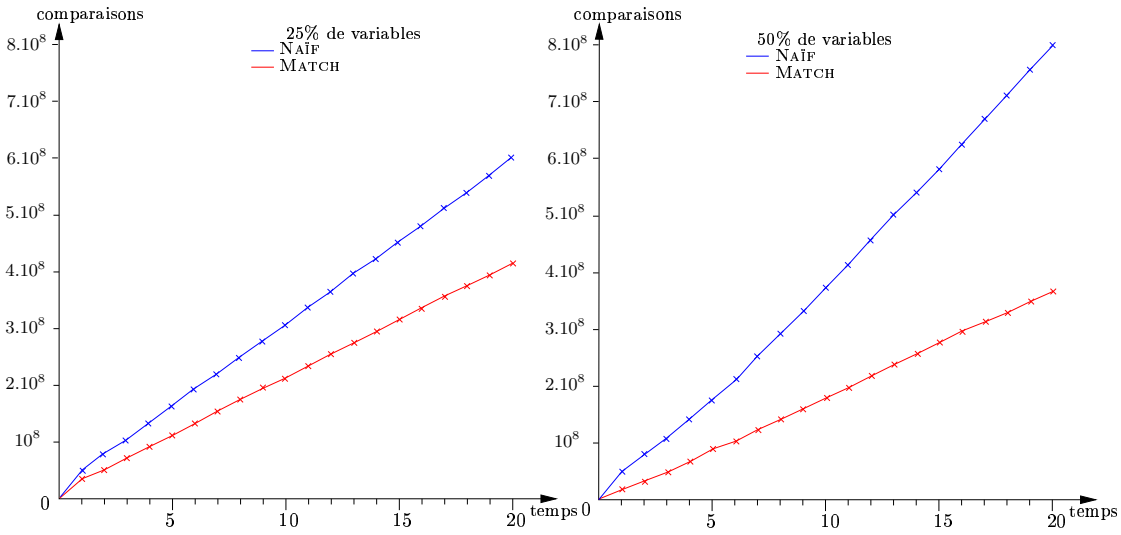


FIG. 4.18 – Nombre de comparaisons suivant le temps pour un pattern de longueur 6

Le nombre de régions est fixé à 21 et la proportion de variables est respectivement 25% et 50%. A chaque unité de temps, le système reçoit en moyenne le même nombre d'événements pour les objets mobiles, avec la même probabilité de rencontrer un échec lors de la tentative de superposition. Ceci justifie la limite en ce qui concerne la durée d'exécution de 20 unités de temps pour nos expériences.

4.5 Conclusion

Dans ce chapitre, j'ai présenté une extension de l'algorithme standard KMP de comparaison de chaînes de caractères [69] afin de l'appliquer à des patterns de mobilité, *i.e.* des séquences paramétrées de localisations. Cet algorithme étendu est adapté à une évaluation continue de requêtes où les requêtes permettent un suivi d'ensembles importants d'objets mobiles. Comme il apparaît dans nos évaluations, notre technique fournit une amélioration significative de l'approche naïve qui décale le pattern d'un unique symbole lors d'un échec. En effet, l'algorithme de KMP étendu évite l'important surcoût lié aux répétitions de comparaisons de mêmes parties de la trajectoire.

D'autres optimisations potentielles restent à explorer. En particulier, nous souhaiterions prendre en compte des relations spatiales plus riches entre les régions de la carte afin d'améliorer la sélectivité de nos requêtes continues. En considérant l'adjacence des régions par exemple, nous pouvons détecter les patterns non-satisfiables, éliminer certains bords incohérents, etc. Finalement nous envisageons d'étudier la transposition de notre technique à certains domaines d'applications traitant de larges séquences (*e.g.* les séquences ADN).

Chapitre 5

Classification multi-échelle de trajectoires

Dans ce chapitre nous proposons un modèle de classification pour les trajectoires d'objets mobiles. Nous supposons que la classification s'appuie sur une carte multi-échelle et nous définissons simplement un pattern de trajectoire comme précédemment, à savoir comme une séquence de zones qu'un objet traverse durant son déplacement. La principale différence vis-à-vis de la modélisation des trajectoires sous forme de patterns présentée dans le chapitre 3 vient de la nécessité de fixer un niveau d'échelle définissant la partition considérée pour établir la représentation d'une trajectoire à l'aide d'un pattern. Nous définissons également un langage de requêtes basé sur les patterns permettant une classification en temps-réel et continue des objets mobiles. Ce modèle a été présenté dans [40].

5.1 Introduction

Dans les chapitres 3 et 4 nous avons présenté comment prendre un ensemble de trajectoires en apparence quelconques et sans lien entre elles et les organiser et les classifier suivant des classes de *patterns* ayant thématiquement une signification. Chaque pattern décrit le comportement d'une classe typique d'objets et le procédé d'analyse s'effectue à l'aide d'opérateurs qui créent des nouveaux patterns, qui comparent les trajectoires aux patterns existants et qui réalisent une classification à la volée. Dans ces deux chapitres nous avons considéré une partition *unique* de l'espace comme base pour nos trajectoires et nos patterns.

Cependant un autre aspect important à prendre en compte, présenté dans ce chapitre, est l'influence de la résolution, ou *échelle*, sur la description des trajectoires. En effet, prendre en compte un niveau d'échelle spécifié par l'utilisateur conduit à des interprétations distinctes. Les différences entre deux trajectoires ne sont plus si apparentes à un faible niveau d'échelle, et les trajectoires auront tendance à être fusionnées ensemble durant le processus de classification. Au contraire, un niveau d'échelle élevé permettra de distinguer deux trajectoires, pourtant très semblables, l'une de l'autre. De plus on pourrait vouloir décrire une trajectoire

avec une résolution très élevée pour certaines régions et avec une résolution plus basse pour les autres régions. Par exemple on aimerait connaître les personnes qui sont partis de Versailles, puis sont passés par Paris, ont traversé la région Bourgogne, puis Rhône-Alpes, avant de traverser le département de l'Allier, pour enfin traverser Ménétrol et arriver à Riom. On voit dans cet exemple que l'on « jongle » entre plusieurs niveaux d'échelle, tantôt villes, tantôt départements, tantôt régions, suivant les portions de trajectoires pour lesquelles on souhaite être plus précis. Ainsi suivant ses besoins, l'utilisateur du système doit pouvoir adapter le niveau d'échelle à la valeur appropriée, rendant donc le système bien plus puissant. Notre modèle prend en compte ce besoin.

Dans la suite de ce chapitre, nous introduisons d'abord (section 5.2) la modélisation multi-échelle de l'espace qui est utilisée comme une base pour la description des patterns. Le modèle de classification est présenté dans la section 5.3 et les requêtes sous forme de patterns dans la section 5.4. Enfin, la section 5.5 conclut le chapitre et présente des perspectives.

5.2 L'espace de référence

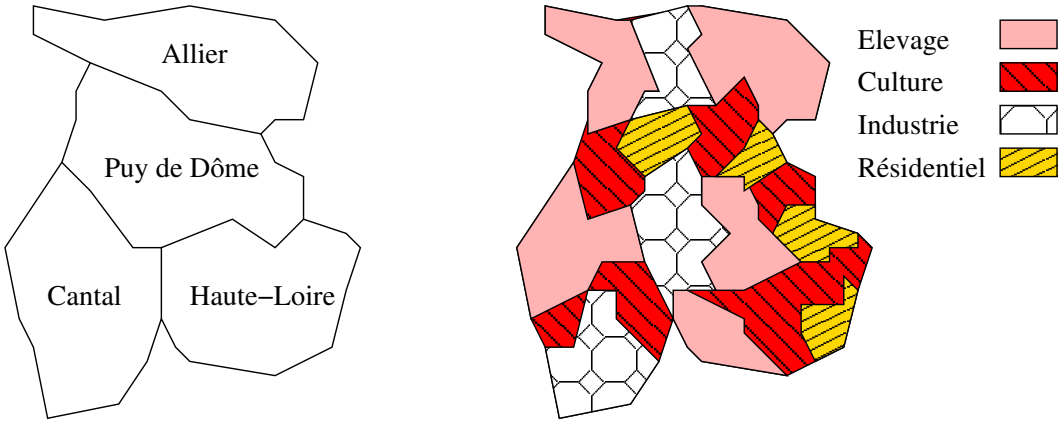
Notre but est de déterminer, pour une population donnée, certains « patterns » représentatifs du comportement qui prennent en compte l'organisation spatiale d'une région spécifique $A \subseteq \mathbb{R}^2$ en zones. Dans la suite de ce chapitre, nous définissons cette organisation spatiale comme étant une *partition multi-échelle* de A et introduisons formellement le concept de niveau d'échelle qui sera utilisé plus tard pour définir les patterns de trajectoire. Cette modélisation de l'espace est partiellement empruntée à [98].

Pour rappel une partition de A est un ensemble fini de zones $\{z_1, z_2, \ldots, z_n\}$ tel que :

$$\bigcup_n z_i = A \text{ et } z_i \cap z_j = \emptyset, i \neq j.$$

La figure 5.1 montre une région (une représentation schématisée de l'Auvergne) avec deux partitions distinctes. La première (partie gauche) correspond à la partition administrative en 4 départements alors que la seconde (celle de gauche) considère l'utilisation de l'espace.

Il apparaît clairement que la partition considérée est fortement liée à l'interprétation thématique spécifique de l'espace qui résulte du choix de l'utilisateur. Nous ne cherchons pas dans ce travail à définir quelle partitions sont pertinentes et lesquelles ne le sont pas. En ce qui nous concerne, il suffit de savoir qu'un objet se déplaçant dans la région partitionnée traversera une séquence de zones distinctes : nous construirons nos patterns à partir de telles séquences. Il faut souligner que, puisque la partition constitue la base pour la définition de nos patterns nous obtenons des classifications de trajectoires fort différentes suivant l'aspect thématique que nous considérons. Une partition thématiquement « neutre » consisterait à construire des patterns sur une partition résultant d'un découpage régulier de l'espace en grille, avec des cellules de taille égale comme présentée dans [84].



a. Partition administrative b. Utilisation de l'espace

FIG. 5.1 – Plusieurs partitions de la même région.

Il est également important de préciser que les zones considérées ne sont pas forcément connexes. On le voit ainsi sur la figure 5.1 où, par exemple, nous n'avons pas une unique zone résidentielle mais quatre, disjointes.

Regardons désormais de plus près le problème de la représentation multi-échelle. Nous devons ici considérer plus spécialement le cas où pour passer d'un niveau d'échelle S_1 à un niveau d'échelle plus grossier S_2 nous *agrégeons* les zones d'une partition p_1, formant de nouvelles zones, plus larges, qui constituent une partition p_2. Il s'agit d'une situation assez naturelle et courante. Ainsi si l'on considère à nouveau la région de la figure 5.1 et ses deux partitions, passer du niveau d'échelle *départements* au niveau d'échelle *régions* conduira en l'agrégation des 4 départements en une zone unique représentant toute la région. On peut imaginer d'un autre côté, que l'on ait un niveau d'échelle *plus fin* donnant une partition de la région en villes.

Les mêmes principes peuvent être appliqués pour la partition de la partie droite de la figure 5.1 : nous pouvons rassembler les zones *Résidentielles* avec les zones *Industrielles* pour former les zones *Construites*, et les zones de *Culture* avec les zones d'*Élevage* pour former les zones *Agricoles*. Dans les deux cas, on obtient des partitions plus *grossières* constituées de plusieurs zones, moins différenciées. Ceci nous conduit ainsi à la relation suivante entre deux partitions :

si p_1, p_2 sont deux partitions de A, on dit que p_1 est *plus fine* que p_2, noté $p_1 \sqsubseteq p_2$, ssi
$$\forall z \in p_1, z' \in p_2, z \cap z' \in \{\emptyset, z\}.$$

En d'autres mots, chaque zone de la partition la plus fine est soit entièrement contenue soit entièrement à l'extérieur d'une zone de la partition la plus grossière. Aucune zone de la partition p_1 ne chevauche partiellement une zone de la partition p_2.

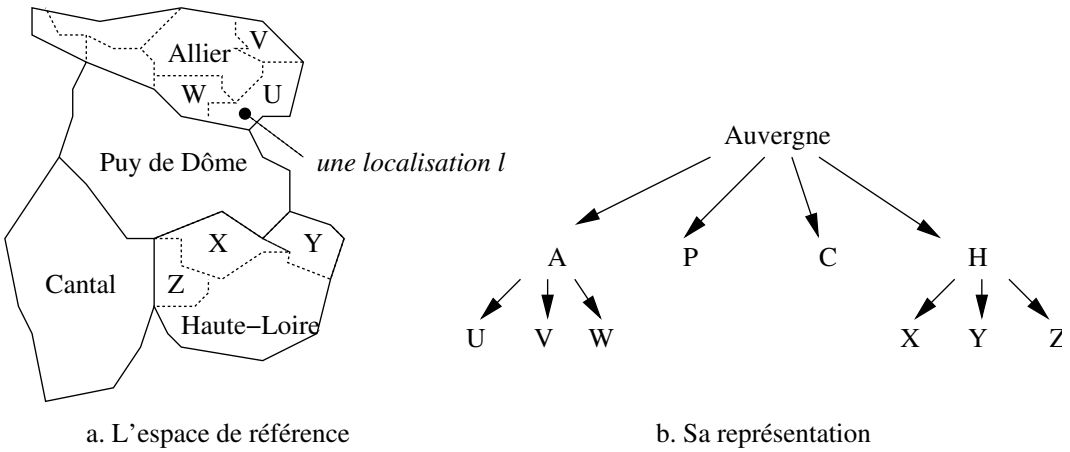
Nous définissons alors une *carte de référence* comme étant l'ensemble des zones d'une partition multi-échelle $p_1 \sqsubseteq p_2 \ldots \sqsubseteq p_n$ de la région à laquelle on s'intéresse.

Définition 17. *Soit $A \in \mathbb{R}^2$ un sous-ensemble du plan, et $\{p_1, \ldots, p_n\}$ un ensemble de partitions de A, avec $p_i \sqsubseteq p_{i+1}$, pour $i < n$. Une* carte de référence *$\mathcal{M}_A$ de A est l'ensemble des zones de $p_1 \cup p_2 \ldots \cup p_n$.*

Rassembler les *villes* en *départements*, les *départements* en *régions*, les zones *industrielles* et *résidentielles* en zones *construites*, peut être vu, à un niveau abstrait, comme un processus uniforme reposant sur la relation d'inclusion sur un ensemble de termes utilisés pour classifier les zones de l'espace de référence. Ceci est illustré dans la figure 5.2(partie gauche), où est représentée une carte de référence $\mathcal{M}$ s'appuyant sur les trois partitions *villes*, *départements* et *régions*. Dans un souci de simplification, seules quelques villes, auxquelles nous donnons des pseudo-noms X, Y, etc., sont représentées. Cet ensemble de zones est structuré par une relation d'inclusion.

Pour des raisons pratiques nous représenterons cette structure en nous appuyant sur un vocabulaire bien défini (partie droite de la figure) qui est constitué d'un ensemble de *symboles* Σ auquel on associe une relation d'*ordre partiel* sur ses symboles. Chaque symbole $\mathtt{a} \in \Sigma$ est utilisé comme identifiant unique d'une zone $z \in \mathcal{M}$. On adopte donc la définition suivante :

Définition 18 (Représentation de la carte de référence). *Une carte de référence $\mathcal{M}$ est représentée par une paire $(\Sigma, \preceq)$ où Σ est un ensemble fini et non-vide de symboles et $\preceq$ est un ordre partiel sur Σ tel que $z_1 \subseteq z_2$ ssi $id(z_1) \preceq id(z_2)$.*



a. L'espace de référence b. Sa représentation

FIG. 5.2 – Une carte de référence multi-échelle et sa représentation

Nous considérons que le graphe de $\preceq$ est un arbre T_Σ, et nous désignons par $racine(\Sigma)$ la racine de T_Σ. Par la suite nous utiliserons de manière indistincte la carte de référence et sa représentation, *i.e.*, nous emploierons les termes « carte de référence » pour signifier sa

représentation. De plus nous dirons qu'une localisation l appartient à $\mathtt{a} \in \Sigma$ pour signifier que l appartient à la zone identifiée par $\mathtt{a}$.

Nous sommes maintenant prêts pour définir formellement le concept de niveau d'échelle.

Définition 19 (Niveau d'échelle). *Soit $(\Sigma, \preceq)$ une carte de référence. L'ensemble des niveaux d'échelle de Σ, noté $\mathcal{S}_\Sigma$, est le sous-ensemble de 2^Σ défini de manière récursive comme suit :*

1. *$racine(\Sigma) \in \mathcal{S}_\Sigma$;*

2. *soit $S \in \mathcal{S}_\Sigma$ et $\mathtt{a} \in S$ tel que $fils(\mathtt{a}) \neq \emptyset$, alors $S - \{\mathtt{a}\} \cup fils(\mathtt{a}) \in \mathcal{S}_\Sigma$ où $fils(a)$ désigne les fils du nœud $\mathtt{a}$ dans T_Σ.*

Cette définition par construction fournit un moyen aisé d'obtenir l'ensemble des niveaux d'échelle, comme on peut le voir avec l'exemple suivant.

Exemple 16. *Les différents niveaux d'échelle de la carte de référence de la Figure 5.2 sont les suivants :*

- S_{racine} = `{Auvergne}`
- S_{dept} = `{A, C, P, H}`
- S_1 = `{U, V, W, C, P, H}`
- S_2 = `{A, C, P, X, Y, Z}`
- $S_{feuilles}$ = `{U, V, W, C, P, X, Y, Z}`

Nous devons toujours considérer la classification des objets en prenant en compte un niveau d'échelle spécifique puisque, intuitivement, un tel niveau fournit l'ensemble des symboles nécessaires et suffisants pour décrire une trajectoire à un niveau de détail spécifique. En effet, on montre facilement que si l est une localisation sur la carte, alors il existe, à n'importe quel niveau d'échelle S, un unique symbole $\mathtt{a}$ tel que $l \in \mathtt{a}$. En d'autres mots, si nous choisissons un niveau d'échelle $S \in \mathcal{S}_\Sigma$ et considérons la localisation d'un objet mobile, il existe une fonction $étiquette(S, l)$ qui associe à l un unique symbole de S.

Exemple 17. *Considérons la localisation l dans la figure 5.2, et les niveaux d'échelle de l'exemple 16. Pour le niveau d'échelle S_{racine}, l est projetée dans la zone $étiquette(S_{racine}, l)$ =* `Auvergne`*, pour S_{dept} et S_2, l est projetée dans* `A`*, finalement pour S_1 et $S_{feuilles}$, l est projetée dans U.*

Un niveau d'échelle S peut être vu comme une « coupe » dans l'arbre T_Σ qui divise Σ en deux sous-ensembles : un en-dessous de S, et l'autre (strictement) au-dessus de S. Par la suite, pour un sous-ensemble $\sigma \subseteq \Sigma$ considéré, nous désignerons par $couverture(\sigma)$ le niveau d'échelle minimal S qui « couvre » σ, i.e., $\forall \mathtt{a} \in \sigma, \exists \mathtt{a'} \in S | \mathtt{a} \preceq \mathtt{a'}$.

5.3 Les patterns de trajectoires

La prise en compte de l'aspect multi-échelle de l'espace de référence a une répercussion sur la représentation des objets mobiles. En effet comme la projection de la position d'un objet ne se fait pas suivant une partition unique mais suivant la partition associée à chaque niveau d'échelle choisi, il convient de garder la trace des localisations précises et successives d'un objet. Par conséquent, dans ce modèle, un objet mobile est représenté par un identifiant et une *trajectoire*. Une trajectoire est simplement une séquence de localisations. Formellement, nous supposons l'existence d'un ensemble fini et dénombrable Obj pour les identifiants. Les localisations sont des points dans l'espace Euclidien $\mathbb{R}^2$.

Définition 20 (Représentation des objets mobiles). *Un objet mobile se compose d'un identifiant o ainsi qu'une liste $o.traj =< l_1, l_2, \ldots, l_n >$ de localisations.*

Il apparaît clairement que la description d'une trajectoire peut être obtenue à partir d'un équipement GPS qui fournit périodiquement la localisation d'un objet. L'aspect continu des trajectoires est à nouveau ignoré puisqu'il n'est pas pertinent pour nos objectifs. Puisque nous projetons chaque localisation dans une zone de la carte, nous devons juste prendre pour hypothèse que l'équipement GPS fournira au moins une localisation pour chaque zone traversée par un objet o.

Nous définissons de manière simple un *pattern de trajectoire* comme une séquence de symboles de zone d'un niveau d'échelle spécifique. Dans la définition suivante, l'opérateur de composition $P.P'$ désigne l'opération $réduire(concat(P, P'))$ où $concat$ est la concaténation de chaînes de caractères et $réduire$ est une fonction qui supprime d'une chaîne de caractères toutes les répétitions consécutives de symboles (i.e., $réduire('\mathtt{aabbcccbbdd}') =' \mathtt{abcbd}'$).

Définition 21 (Patterns de trajectoire). *Soit S un niveau d'échelle dans $(\Sigma, \preceq)$. Alors :*

- *si $\mathtt{a} \in S$, alors $\mathtt{[a]}$ est un pattern de trajectoire ;*
- *si P_1 et P_2 sont des patterns de trajectoire dans S, alors $P_1.P_2$ est un pattern de trajectoire.*

Nous désignerons par $\mathcal{P}_\Sigma$ l'ensemble de tous les patterns de trajectoire dans $(\Sigma, \preceq)$.

La raison qui nous pousse à supprimer les répétitions de symboles d'un pattern de trajectoire est qu'une telle répétition ne fournit aucune information utile puisque nous ne pouvons garantir que les événements liés à un objet sont séparés par des intervalles de temps constants.

Exemple 18. *Les quatre patterns suivants sont des exemples de patterns de trajectoire :*

$$P_1 = \texttt{[Z.X.P.U]} \qquad P_2 = \texttt{[C.Z.P.W]}$$
$$P_3 = \texttt{[H.P.U]} \qquad P_4 = \texttt{[H.P.A]}$$

Nos motivations concernant l'introduction du concept de patterns sont liées à la volonté de regrouper les trajectoires qui présentent un comportement similaire. Un pattern décrit une classe de trajectoires. Par exemple si nous considérons la figure 5.2, le pattern `[H.P.A]` représente toutes les trajectoires qui vont de la `Haute-Loire` en `Allier`, en traversant le `Puy de Dôme`. Pour un niveau d'échelle donné S, une trajectoire est projetée en un pattern dans S suivant :

Définition 22. *Soit $t =< l_1, l_2, \ldots, l_n >$ la trajectoire d'un objet mobile o, et soit S un niveau d'échelle. Le* pattern *de t dans S, désigné par $pattern(t, S)$, est :*

$$[réduire(étiquette(S, l_1).étiquette(S, l_2) \cdots étiquette(S, l_n))]$$

où $étiquette()$ et $réduire()$ sont les fonctions introduites précédemment.

Une trajectoire t *satisfait* un pattern P à l'échelle S si $pattern(t, S) = P$. Suivant le niveau d'échelle, les patterns sont satisfaits par plus ou moins de trajectoires, et par conséquent constituent des outils de classification plus ou moins précis. Prenons un exemple, toujours en nous appuyant sur la Figure 5.2. Le pattern `[U.V.P.X.Y]` du niveau d'échelle $S_{feuilles}$ est satisfait par les trajectoires qui vont des villes `U`, puis `V` en `Allier`, aux villes `X` puis `Y` en `Haute-Loire`, en traversant le `Puy de Dôme`. Il apparaît clairement que n'importe quelle trajectoire qui satisfait ce pattern satisfera aussi le pattern `[A.P.H]` au niveau d'échelle S_{dept} et, inversement, si une trajectoire ne satisfait pas `[A.P.H]`, alors elle ne satisfera pas `[U.V.P.X.Y]`. Par conséquent on peut considérer que certains patterns sont des généralisations d'autres, et ceci nous conduit à la relation suivante sur $\mathcal{P}_\Sigma$:

Définition 23 (Relation de raffinement sur les patterns). *La* relation de raffinement sur les patterns, *$\trianglelefteq$, est définie par induction comme suit.*

1. *si P et P' sont deux patterns de la forme $P =$ `[`$b_1.b_2.\cdots.b_n$`]` et $P' =$ `[a]`, avec $b_i \preceq$ `a` pour $i = 1, \ldots n$, alors $P \trianglelefteq P'$.*

2. *si P et P' sont deux patterns de la forme $P_1.P_2$ et $P'_1.P'_2$, avec $P_1 \trianglelefteq P'_1$ et $P_2 \trianglelefteq P'_2$, alors $P \trianglelefteq P'$.*

Si $P \trianglelefteq P'$, nous dirons que P est plus fin *que P'.*

Exemple 19. *Le pattern `[Z.Y.P.U]` est plus fin que `[H.P.A]`, parce que `[Z.Y]` $\trianglelefteq$ `[H]`, `[P]` $\trianglelefteq$ `[P]` et `[U]` $\trianglelefteq$ `[A]`.*

La proposition suivante se vérifie facilement :

Proposition 6. *La relation $\trianglelefteq$ est un ordre partiel sur $\mathcal{P}_\Sigma$.*

Preuve: La démonstration de la propriété de réflexivité et d'anti-symétrie est triviale. Pour la transitivité, il suffit de remarquer que $P \trianglelefteq P'$ ssi chaque $\mathtt{a} \in P'$ correspond à un sous-pattern $\mathtt{b}_1.\mathtt{b}_2.\cdots.\mathtt{b}_n$ de P avec $\mathtt{b}_i \preceq \mathtt{a}$, $i = 1, \ldots n$.

Soit maintenant P_1, P_2, et P_3 trois patterns avec $P_1 \trianglelefteq P_2$ et $P_2 \trianglelefteq P_3$. Pour chaque $\mathtt{a} \in P_3$ il existe un sous-pattern $B = \mathtt{b}_1.\mathtt{b}_2.\cdots.\mathtt{b}_n$ de P_2 avec $\mathtt{b}_i \preceq \mathtt{a}$, $i = 1, \ldots, n$, et pour chaque $\mathtt{b}_i$, il existe un sous-pattern $C_i = \mathtt{c}^i_1.\mathtt{c}^i_2.\cdots.\mathtt{c}^i_{m_i}$ de P_1 avec $\mathtt{c}^i_j \preceq \mathtt{b}_i$, $j = 1, \ldots, m_i$. Soit C le pattern $C_1.C_2.\cdots.C_n$. Clairement chaque $\mathtt{c} \in C$ est tel que $\mathtt{c} \preceq \mathtt{a}$, et nous avons alors trouvé un sous-pattern de P_1 qui correspond à $\mathtt{a}$ dans P_3. Alors on a $P_1 \trianglelefteq P_3$ et ceci conclut la preuve. □

Les patterns de deux trajectoires peuvent être distincts à un niveau d'échelle et similaires à un autre niveau. Par exemple deux trajectoires dont les patterns au niveau d'échelle $S_{feuilles}$ sont respectivement `[X.P.V]` et `[Y.P.U]`, partagent le même pattern `[H.P.A]` au niveau d'échelle S_{dept}. En fait, on peut montrer que deux patterns de trajectoire ont un unique *plus petit ancêtre commun* d'après la relation $\trianglelefteq$. Cet ancêtre représente le niveau minimal de description des trajectoires auquel ces trajectoires ne peuvent plus être distinguées l'une de l'autre.
Considérons par exemple P_1 =`[X.Y.P.A]` et P_2 =`[H.P.U.V]`. Les deux patterns représentent un déplacement du département `Haute-Loire` au département `Allier` via le département `Puy-de-Dôme`. Il apparaît que P_1 offre une représentation plus détaillée pour la partie de la carte correspondant à la région `Allier`. Nous n'avons ni $P_1 \trianglelefteq P_2$ ni $P_2 \trianglelefteq P_1$, cependant à la fois P_1 et P_2 sont inférieurs à P_3 =`[H.P.A]` d'après la relation $\trianglelefteq$.

Proposition 7. *Soit $\mathcal{D} = \{P_1, .., P_n\}$ un ensemble quelconque de patterns de trajectoire. Soit $\mathcal{U}$ l'ensemble de tous les patterns P tels que $P_i \trianglelefteq P, i = 1, 2, ..., n$. Alors $\mathcal{U}$ a un unique élément minimal, que nous appellerons ppac$(\mathcal{D}, \trianglelefteq)$ (pour* **p***lus* **p***etit* **a***ncêtre* **c***ommun).*

Preuve:

Remarquons d'abord que $\mathcal{U}$ est non vide, puisqu'il contient au moins $racine(T_\Sigma)$. Maintenant soit $P = [\mathtt{a}_1.\mathtt{a}_2.\cdots.\mathtt{a}_p]$ et $P' = [\mathtt{a}'_1.\mathtt{a}'_2.\cdots.\mathtt{a}'_q]$ deux éléments minimaux de $\mathcal{U}$. Nous savons qu'au moins un élément minimal existe parce que $\trianglelefteq$ est un ordre partiel. La preuve construit un élément $P_\perp$ tel que $P_\perp \trianglelefteq P$ et $P_\perp \trianglelefteq P'$. Intuitivement, nous montrons que certains symboles de P (resp. P') sont les plus petits ancêtres communs de sous-chaînes de P' (resp. P). La concaténation de ces sous-chaînes donne $P_\perp$.

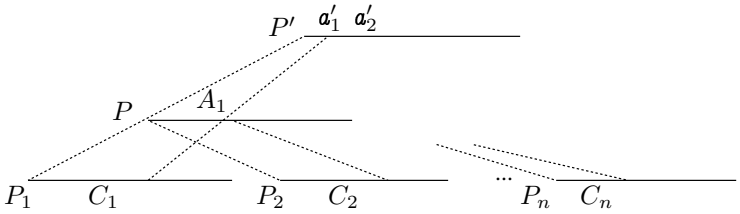


FIG. 5.3 – Calcul du ppac de $\{P_1, P_2, \ldots, P_n\}$

Considérons d'abord $\mathtt{a}_1$ et $\mathtt{a}'_1$. Nous avons $P_i[1] \preceq \mathtt{a}_1$ et $P_i[1] \preceq \mathtt{a}'_1$, pour $i = 1, \cdots, n$. Puisque T_Σ est un arbre, cela signifie que soit $\mathtt{a}_1 \preceq \mathtt{a}'_1$ soit $\mathtt{a}'_1 \preceq \mathtt{a}_1$. Supposons (mais le choix inverse conduirait à une démonstration similaire) que $\mathtt{a}_1 \prec \mathtt{a}'_1$, et soit C_1 le préfixe de P_1 qui est « couvert » par $\mathtt{a}_1$. Puisque $P_1 \trianglelefteq P$, il existe un préfixe $A_1 = \mathtt{a}_1.\mathtt{a}_2.\cdots.\mathtt{a}_i$ de P qui « couvre » C_1 également. De plus (1) chaque symbole $\mathtt{a} \in A_1$ est tel que $\mathtt{a} \prec \mathtt{a}'_1$, puisque sinon P contiendrait des symboles de niveaux d'échelle incompatibles, et (2) il n'existe pas un préfixe C'_1 de P_1 plus grand que C_1 qui est aussi couvert par A_1 puisque le symbole qui suit C_1 dans P_1 est inclus dans $\mathtt{a}'_2$ (voir Figure 5.3). Par un raisonnement similaire on trouve $C_2, \ldots, C_n$, les préfixes de $P_2, \ldots, P_n$, qui sont couverts par A_1.

Il s'ensuit que le pattern $[A_1.\mathtt{a}_{i+1}.\cdots.\mathtt{a}_p]$ appartient à $\mathcal{U}$ et qu'il satisfait $[A_1.\mathtt{a}_{i+1}.\cdots.\mathtt{a}_p] \trianglelefteq P'$, ce qui contredit l'hypothèse de minimalité de P. De plus, la même technique s'applique à l'ensemble de patterns $\{Q_1, Q_2, \ldots, Q_n\}$, avec $P_i = C_i.Q_i$, pour $i = 1, \ldots, n$. Soit $Q = [\mathtt{b}_2.\cdots.\mathtt{b}_q]$ et $Q' = [\mathtt{a}_{i+1}.\cdots.\mathtt{a}_p]$. Alors on trouve A_2 de façon similaire à A_1. La concaténation de $A_1.A_2.\cdots.A_r$ est $P_\perp$. □

L'existence d'un plus petit ancêtre commun (ppac) est importante, à la fois pour nos objectifs de classification et pour l'optimisation lors du traitement en temps-réel de la requête. Du point de vue de la classification, la structure de semi-treillis de nos patterns garantit que nous pouvons toujours obtenir, pour un ensemble de trajectoires, un niveau d'échelle auquel toutes ces trajectoires seront représentées par un unique pattern.

L'aspect optimisation est moins courant, bien qu'aussi important. L'idée est que nous savons que si une trajectoire ne satisfait pas le *ppac* d'un ensemble de patterns, alors elle ne satisfait *aucun* de ces patterns. Cette propriété permet d'éviter de nombreux calculs inutiles dans les systèmes de notification de grande échelle (voir par exemple [123, 27]), en créant des *clusters* de patterns d'interrogation et en filtrant l'évaluation de ces requêtes grâce à leur ppac. Cet aspect sera développé dans la section suivante.

Nous avons besoin d'un algorithme pour calculer le plus petit ancêtre commun d'un ensemble de patterns. Par ailleurs, si nous prenons en compte la motivation précédente (utiliser un ppac comme un filtre pour un ensemble de patterns), il faut que cet algorithme soit *incrémental* parce que nous aimerions pouvoir ajouter un nouveau pattern dans un cluster existant sans avoir à tout recalculer. La technique proposée ci-dessous repose sur l'observation suivante : trouver le ppac d'un ensemble de patterns est équivalent à trouver le niveau d'échelle S pour lequel tous les patterns deviennent similaires. Une fois S trouvé, il reste à déterminer l'ancêtre (unique) des patterns dans S.

L'algorithme PPAC prend en entrée une paire de patterns (P_1, P_2) et retourne le ppac de $\{P_1, P_2\}$. Intuitivement, l'algorithme coupe chaque pattern en n morceaux. Ces morceaux sont comparés deux à deux en s'appuyant sur la fonction $ppp(a, b)$ qui retourne le plus proche parent d'une paire de symboles $(a, b) \in \Sigma^2$ d'après la relation $\preceq$, et la fonction $ancêtre(S, a)$ qui retourne l'ancêtre du symbole a dans le niveau d'échelle S. On rappelle que $couverture(\sigma)$ calcule le niveau d'échelle minimal S qui « couvre » $\sigma \subseteq \Sigma$.

Algorithme PPAC
Entrée :Une paire de patterns P_1, P_2
Sortie : $ppac(\{P_1, P_2\}, \trianglelefteq)$
début
– étape (a) : on calcule le niveau d'échelle S_{ppac} du plus petit ancêtre commun –
// Initialise S_{ppac} pour être le plus fin niveau d'échelle possible, $S_{feuilles}$
$S_{ppac} := S_{feuilles}$
// Parcours de P_1 et P_2, et on détermine à chaque étape si un changement d'échelle est nécessaire
$i := 1, j := 1$ // Positions courantes dans P_1 et P_2.
tant que ($i \leq longueur(P_1)$ et $j \leq longueur(P_2)$) **faire**
// Prend le plus petit ancêtre commun de $P_1[i]$ et $P_2[j]$ dans $(\Sigma, \preceq)$
$l := ppp(P_1[i], P_2[j])$
// Saute la partie « couverte » par ce ppp dans les deux patterns
tant que $P_1[i] \preceq l$ **faire** $i := i + 1$; **tant que** $P_2[j] \preceq l$ **faire** $j := j + 1$
// Calcule le niveau d'échelle minimal qui contient l
$S_{ppac} := couverture(S_{ppac} \cup \{l\})$
fin
// P_1 ou P_2 n'a pas été entièrement parcouru ? Alors S_{ppac} est le niveau d'échelle le plus grossier, S
si ($i \leq longueur(P_1)$ ou $j \leq longueur(P_2)$) **alors** $S_{ppac} := S_{racine}$
– Etape (b) : remplace chaque symbole de P_1 par son ancêtre dans S_{ppac} –
pour chaque ($i \in [1, longueur(P_1)]$) $P_{ppac}[i] := ancêtre(S_{ppac}, P_1[i])$
– Etape (c) : calcule la réduction de P_{ppac} : on obtient le résultat –
retourne $réduire(P_{ppac})$
fin

FIG. 5.4 – L'algorithme PPAC.

L'algorithme PPAC peut être utilisé pour calculer de manière incrémentale le ppac d'un ensemble $\{P_1, .., P_n\}$, avec $n > 2$: d'abord on calcule PPAC(P_1, P_2), ensuite PPAC(PPAC(P_1, P_2), P_3), etc. Nous terminons cette section en illustrant comment fonctionne cet algorithme en l'appliquant à un exemple, en nous référant toujours à la figure 5.2.

Exemple 20. *Considérons les patterns* $P_1 =$`[X.Y.P.A]` *et* $P_2 =$`[X.Z.P.U.V]`. *Afin de calculer*
PPAC$(\{P_1, P_2\})$, *nous initialisons* S_{ppac} *avec* $S_{feuilles} =$ `{U, V, W, C, P, X, Y, Z}`. *Chaque point dans l'énumération suivante représente une étape dans boucle* ***tant que*** .

1. *Tout d'abord* l *est affecté à* $ppp(P_1[1], P_2[1]) = ppp(X, X) = X$. *Nous sautons ensuite la partie couverte par* `X`, *à la fois pour* P_1 *et* P_2 : i *et* j *sont alors fixés à 2.* S_{ppac} *demeure inchangé.*

2. *Ensuite l est affecté à $ppp(P_1[2], P_2[2]) = ppp(Y, Z) = H$. Nous sautons la partie couverte par* `H` *: i et j sont tous deux fixés à 3. Le niveau d'échelle est $S_{ppac} = couverture(S_{ppac} \cup \{H\}) =$* `{U, V, W, C, P, H}`.

3. *$P_1[3]$ et $P_2[3]$ sont comparés. Puisque tous les deux valent* `P` *qui appartient à S_{ppac}, nous avançons simplement de un dans chaque pattern : $i = j = 4$.*

4. *Finalement l est assigné à $ppp(P_1[4], P_2[4]) = ppp(U, A) = A$. Nous sautons la fin des deux patterns, i.e., $i = 5$ et $j = 6$. S_{ppac} est $couverture($* `{U, V, W, C, P, H}` $\cup \{$`A`$\}) =$* `{A, C, P, H}`.

Ensuite l'étape (b) est réalisée : on prend P_1 et on remplace chaque symbole par son ancêtre dans S_{ppac} et on obtient $P_{ppac} =$ `[H.H.P.A]` *qui une fois réduit conduit au résultat final $ppac(P_1, P_2) =$* `[H.P.A]`. *A noter que pour cette étape on aurait pu prendre le pattern P_2 et obtenir le même résultat final : $P_{ppac} = réduire($* `[H.H.P.A.A]`$) =$ `[H.P.A]`.

En résumé, le calcul du plus petit ancêtre commun, obtenu à l'aide de l'algorithme PPAC, fournit le pattern le plus fin commun à toutes les trajectoires auxquelles on s'intéresse. On obtient en même temps le plus fin niveau d'échelle pour lequel les trajectoires seront considérées comme similaires.

5.4 Les patterns de requêtes

Nous nous intéressons maintenant à l'application de notre représentation à base de patterns pour suivre des objets mobiles [113, 121, 66]. Dans la suite de cette section, nous définissons nos requêtes basées sur les patterns et décrivons ensuite un mécanisme simple et efficace pour tester, à chaque nouvel événement, si chaque objet appartient ou non au résultat d'une requête. Ce mécanisme peut être utilisé pour maintenir de manière incrémentale le résultat de la requête en ajoutant ou supprimant des objets.

5.4.1 Langage de requêtes basé sur les patterns

L'application la plus simple de notre approche de classification consiste à définir une requête comme un *pattern de trajectoire*, exprimé à un niveau d'échelle donné, et de déterminer toutes les trajectoires qui satisfont ce pattern. Cependant, dans un contexte de classification en ligne, nous avons besoin d'adapter notre langage s'appuyant sur les patterns afin de détecter les occurrences d'un pattern dans un *flux* de localisations. Le langage d'interrogation et sa sémantique sont définis comme suit :

Définition 24 (Interrogation à l'aide de pattern). *Soit $\mathcal{M}$ une carte de référence $(\Sigma, \preceq)$. Une requête q sur $\mathcal{M}$ est une paire (S, P_q) où S est un niveau d'échelle dans $\mathcal{M}$ et P_q est un pattern de trajectoire dans S.*

La différence essentielle entre cette définition et celle de nos requêtes du chapitre 3 est qu'il est impératif de préciser le niveau d'échelle considéré. En effet, comme nous allons le voir plus loin, le niveau d'échelle intervient lors du processus d'évaluation. La réponse à une requête $q(S, P_q)$ est un ensemble d'objets dont la trajectoire $traj$ est telle que P_q est un suffixe de $pattern(traj, S)$.

Définition 25 (Réponse d'une requête). *La* réponse *à une requête $q(S, P_q)$ sur un ensemble d'objets O, notée $ans_O(q)$, est le sous-ensemble de O défini comme suit : $o \in ans_O(q)$ ssi P_q est un suffixe de $pattern(o.traj, S)$.*

5.4.2 Évaluation de requêtes

L'évaluation de requêtes repose sur la simple application de techniques standard de *pattern matching*. Notre problème revient en effet à trouver les occurrences de P_q (le pattern) dans un texte $t \in \Sigma^*$ (la séquence de localisations). Nous devons trouver tous les préfixes de t qui se terminent par P_q. Ceci peut être réalisé avec un automate $\mathcal{A}(P_q)$ qui reconnaît le langage $\Sigma^* P_q$. La construction de $\mathcal{A}(P_q)$ s'appuie sur la fonction h qui, pour un pattern de trajectoire donné $t \in S^*$, retourne le plus long suffixe de t qui est aussi un préfixe de P_q. Le calcul de h et de la *table des bords* a été présenté au Chapitre 4.

En général, la fonction h conduit à l'état représentant le plus long préfixe de P_q qui est aussi un suffixe du pattern de trajectoire.

Un point important est que l'automate est déterministe. Ceci garantit que l'on peut localiser un pattern d'interrogation dans un flux entrant en gardant simplement l'état courant dans l'automate de l'objet qui génère le flux. L'évaluation de la requête prend comme entrée la définition de la requête, un objet o, sa position actuelle, et retourne `vrai` quand o appartient à la réponse de q ou `faux` dans le cas contraire.

Algorithme EVAL
Entrée :Une requête $q(S, P_q)$, un objet o, une localisation loc
Sortie : Un booléen qui indique si o appartient
ou non à la réponse de la requête
début
// Détermine l'étiquette correspondant à *pos* dans le niveau d'échelle S
$\mathtt{a} = étiquette(S, loc)$
// Vérifie que o est entré dans une nouvelle zone
si (a est distinct de l'étiquette précédente) **alors**
// Applique la fonction de transition sur a,
// et calcule le nouvel état courant de o
$o.s_{cur} = \delta_q(o.s_{cur}, \mathtt{a})$
finsi
// Vérifie si o appartient à la réponse de q
si ($o.s_{cur}$ est l'état acceptant) **retourne** vrai

sinon retourne faux **finsi**
fin

L'algorithme EVAL doit être appelé chaque fois qu'un nouvel événement (*e.g.*, un message GPS) est reçu pour un objet, et son résultat peut être utilisé dans le processus de maintenance du résultat de la requête. Il est intéressant de souligner que nous n'avons pas besoin d'enregistrer l'historique d'un objet o lors de ce processus d'évaluation. En effet, il suffit juste d'enregistrer le statut actuel de o d'après l'automate $\mathcal{A}(P_q)$. Ceci minimise les besoins en mémoire et évite d'avoir à relire des symboles d'un pattern de trajectoire lors de l'évaluation d'une requête.

5.4.3 Optimisation

Notre approche offre plusieurs opportunités d'optimisation. Nous considérons en particulier les systèmes qui visent à classifier en temps réel un ensemble d'objets dont les localisations sont mises à jour périodiquement. Quand le système traite un très grand nombre de patterns d'interrogation sur un très grand nombre d'objets, nous pouvons réduire la charge de travail avec les moyens suivants :

1. *filtrage d'événements* : comme mentionné dans la section précédente, nous pouvons utiliser la structure de treillis d'un ensemble de patterns pour filtrer les calculs inutiles ;

2. *partage de ressources* : nous pouvons grouper ensemble les automates d'un ensemble de requêtes.

Filtrage d'événements

Les deux techniques peuvent être combinées. La première est plutôt spécifique à notre représentation multi-échelle. Elle consiste à grouper les requêtes en *clusters* et à filtrer, pour chaque cluster, les événements GPS qui ne peuvent pas affecter le résultat des requêtes du cluster. Nous savons qu'un ensemble de requêtes $\{q_1, q_2, \ldots, q_n\}$ dans un cluster donné peut être représenté par son ppac q_{ppac} et qu'une trajectoire ne peut pas satisfaire une requête q_i si elle ne satisfait pas aussi q_{ppac}. En quelque sorte, nous pouvons « indexer » l'ensemble de requêtes enregistrées par le système [66, 75] et éviter beaucoup de calculs non pertinents. Il faut souligner que cet index est basé explicitement sur les séquences de localisations et donc se comporte totalement différemment des index spatiaux classiques (*e.g.*, R-tree). De plus il est intéressant de noter que, la requête q_{ppac} étant associée à un niveau d'échelle assez grossier, les opérations de « point dans polygone » sont moins coûteuses que si on devait les effectuer pour un partitionnement plus fin, comme c'est le cas pour les requêtes q_i, puisqu'on a moins de zones à tester.

Regrouper nos patterns en clusters afin de filtrer est un problème d'optimisation classique que l'on trouve dans de nombreuses applications. Nous avons présenté ici nos premiers

résultats de recherche sur cette idée. Actuellement nous réfléchissons sur les algorithmes permettant de créer les différents clusters à partir d'un ensemble de patterns donnés. Une implantation et des comparaisons entre notre approche et les algorithmes bien connus de création de clusters devraient également être menées.

La stratégie que nous proposons pour l'évaluation d'un cluster de requêtes se décompose en deux étapes :

1. évaluation en ligne de la requête associée au pattern ppac du cluster de requêtes ;

2. pour les objets donc la trajectoire satisfait le ppac, on évalue les requêtes du cluster.

Algorithme

La première étape correspond à l'évaluation classique d'une requête. Pour chaque objet nous gardons sa position dans l'automate associé au pattern du ppac du cluster. Lorsque la trajectoire d'un objet satisfait ce pattern et qu'on atteint donc l'état acceptant de l'automate, nous devons tester chacune des requêtes du cluster pour voir si réellement cet objet satisfait une des requêtes posées. Il s'agit d'une évaluation *paresseuse* : on ne garde qu'un seul état courant par objet et par cluster, et on n'évalue les requêtes que lorsque la trajectoire satisfait le filtre. Ceci implique que l'on a sauvegardé les différentes localisations *précises*, dans $\mathbb{R}^2$, de l'objet pour pouvoir connaître l'historique de la trajectoire. Nous avons défini le résultat d'une requête comme l'ensemble des objets dont la partie la *plus récente* de la trajectoire satisfait le pattern, il faut donc vérifier si le suffixe de l'historique de la trajectoire de l'objet qu'on souhaite tester satisfait le pattern de chaque requête. Pour cela nous devons parcourir l'historique depuis le dernier événement reçu et s'arrêter soit lorsque l'on obtient un échec, soit lorsque l'on atteint l'état acceptant de l'automate *inversé* du pattern d'interrogation.

L'algorithme EvalCluster décrit l'évaluation. Il prend en entrée un ensemble de requêtes, le ppac de cet ensemble, un objet o, sa position actuelle dans $\mathbb{R}^2$ retournée par l'équipement GPS. Il retourne l'ensemble des requêtes dont le résultat contient l'objet.

Algorithme EvalCluster
Entrée : Un ensemble de requêtes $q_i(S_i, P_{q_i}), i = 1, \ldots, n$,
son ppac $q(S_{ppac}, P_{ppac})$, un objet o, une localisation loc
Sortie : Un ensemble $\mathcal{Q}$ de requêtes dont o appartient au résultat
début
// teste si on atteint l'état acceptant de P_{ppac}
// détermine l'étiquette correspondant à pos dans le niveau d'échelle S_{ppac}
$\mathsf{a} = étiquette(S_{ppac}, loc)$
// vérifie que o est entré dans une nouvelle zone

si (a est distinct de l'étiquette précédente) **alors**
// applique la fonction de transition sur a,
// et calcule le nouvel état courant de o
$o.s_{cur} = \delta_{ppac}(o.s_{cur}, \mathtt{a})$
finsi
// même si on ne change pas de zone dans S_{ppac} on sauve pos. courante
$o.traj = o.traj + pos$
// vérifie si o appartient à la réponse de q
si ($o.s_{cur}$ n'est pas l'état acceptant) **retourne** $\emptyset$
sinon
$\mathcal{Q} = \emptyset$
// on vérifie pour chaque requête si o appartient à son résultat
pour chaque $q_i(S_i, P_{q_i})$ **faire**
// on inverse le pattern de la requête
$P'_{q_i} = inverse(P_{q_i})$
$o.s_{cur} = init(P'_{q_i})$, échec = faux ;
tant que échec = faux **faire**
// on regarde le dernier symbole de l'historique de la trajectoire
$l = dernier(o.traj)$
// si on atteint l'état acceptant de q'_i, ajoute o à son résultat
si EVAL$(q'_i, o, l) = vrai$ **alors**
$\mathcal{Q} = \mathcal{Q} \cup q_i$
échec = vrai
sinon
// si on est revenu à l'état initial, on a eu un échec
si $o.s_{cur} = init(P'_{q_i})$ **alors** échec = vrai
// sinon on continue le parcours inverse de la trajectoire
sinon $o.traj = o.traj \setminus \{l\}$ **finsi**
finsi
fin
fin
retourne $\mathcal{Q}$
fin

Exemple 21. *Voici un exemple qui illustre cette technique, toujours d'après la Figure 5.2. Supposons que l'on ait un cluster avec les deux patterns d'interrogation suivants* P_1 =`U.V.C.H` *et* P_2 =`W.C.Z.X`*. Le ppac de ces patterns est donc* `A.C.H`*. Supposons que l'on ait un objet* o *dont la trajectoire est* $o.traj =< (x_1, y_1), \ldots, (x_n, y_n) >$ *dont l'état courant dans l'automate associé à* `A.C.H` *est l'état correspondant à la validation de* `A.C`*. On reçoit une nouvelle position GPS* $(x_{n+1}, y_{n+1}) \in \mathbb{R}^2$*, telle que étiquette*$(S_{ppac}, (x_{n+1},$
$y_{n+1})) =$ `H`*. Par conséquent l'objet entre dans l'état acceptant de l'automate du ppca. Il a donc satisfait le* filtre *et est un candidat potentiel pour appartenir au résultat de chacune des requêtes du cluster. On prend la première requête du cluster, disons celle dont le pattern est*

U.V.C.H. On inverse ce pattern, obtenant $P'_1 =$*H.C.V.U, et on construit l'automate associé. On prend alors une par une les localisations enregistrées en partant de la dernière reçue* $((x_{n+1}, y_{n+1}))$*. Si on atteint l'état acceptant de* $P'_1 =$ *U on ajoute o au résultat de* q_1*, et on recommence le même procédé avec* q_2*.*

Estimation du gain

La motivation de cette méthode d'évaluation est qu'on estime que peu d'objets satisfont la *totalité* du pattern du filtre, et donc peut-être une des requêtes du cluster. Ainsi plutôt que de réaliser de nombreuses opérations de « point dans polygone » pour chacune des requêtes à chaque nouvel événement, on en réalise une seule, au niveau de la partition plus grossière correspondant au niveau d'échelle du filtre. Sachant qu'un objet se trouve dans une zone z, qu'une zone a en moyenne d voisins et que l'on considère une équi-probabilité, donc sans tenir compte de la taille de la zone voisine, pour déterminer la zone de sortie de l'objet mobile (donc $prob = 1/d$), on en déduit que la sélectivité γ d'un pattern P de longueur l est :

$$\gamma = prob(P[1] \rightarrow P[2]) \times \ldots \times prob(P[l-1] \rightarrow P[l]) = \frac{1}{d^{l-1}}$$

Ainsi plus le pattern du filtre est long et plus le nombre moyen de voisins d'une zone est grand, plus le filtrage des objets est efficace.

Néanmoins cette technique engendre un grand nombre de calculs lorsque la trajectoire d'un objet satisfait le pattern du filtre, puisqu'on doit évaluer alors l'ensemble des requêtes du cluster pour cet objet. Pour avoir un ordre d'idée du gain obtenu avec cette méthode, nous estimons le nombre d'opérations « point dans polygone » à réaliser. Supposons que l'on ait un cluster contenant p requêtes et que l'on reçoive k positions $pos_j, j = 1, \ldots, k$, alors :

i) traitement séquentiel des requêtes : $\text{coût}_{seq} = p \times k$,
en effet, pour chaque requête $q_i(S_i, P_{q_i})$ considérée séparément, on doit projeter pos_j dans S_i, donc pour chaque événement on a autant d'opérations de pointé que de requêtes ;

ii) utilisation du filtre : $\text{coût}_{filtre} = k + \gamma \times p \times k$,
en effet il faut faire une projection de chaque pos_k dans S_{filtre}, et, avec la probabilité γ estimée plus haut, l'objet satisfait le filtre et donc entraîne l'évaluation des p requêtes du cluster sur l'ensemble des k positions reçues.

Par conséquent le gain $\mathcal{G}$ de cette méthode est :

$$\mathcal{G} = \frac{k+\gamma\times p\times k}{p\times k} = \frac{1+\gamma\times p}{p}$$

Pour conclure cette étude du gain, nous envisageons différentes hypothèses sur γ et p et regardons les conséquences sur le gain. Les résultats sont résumés dans le tableau 5.1.

	filtre très sélectif ($\gamma \ll 1$)	filtre peu sélectif ($\gamma \approx 1$)
beaucoup de patterns	$\mathcal{G} \approx 0$	$\mathcal{G} \approx \gamma$
peu de patterns	$\mathcal{G} \approx \frac{1}{p}$	$\mathcal{G} \approx 1$

TAB. 5.1 – Tableau résumant le gain suivant γ et p.

Noter enfin que le gain correspond également à un gain en espace puisque pour un objet, on ne maintient un état que dans le pattern du filtre et non un état pour chaque automate correspondant aux requêtes du cluster.

Améliorations possibles

Plusieurs améliorations pourraient être apportées à cette technique de filtrage. Tout d'abord, plutôt que de traiter chaque cluster indépendamment, on pourrait considérer le ppac des ppac de chaque cluster, et en étendant, avoir une structure arborescente. Cette structure serait un peu semblable aux structures d'arbres traditionnelles des bases de données spatiales, où un nœud correspond à une zone couvrant une zone plus large que l'union des zones de ses fils. On pourrait également étudier la politique d'insertion d'une requête dans un cluster et notamment la mesure de distance la plus pertinente dans notre cadre bien particulier d'un espace multi-échelle. Enfin l'introduction de la notion de topologie pourrait permettre d'éviter des opérations de point dans polygone en testant directement les zones *voisines*.

Partage de ressources

L'autre optimisation consiste, dans chaque cluster, à « fusionner » ensemble les automates associés aux requêtes du cluster en un *unique* automate global partagé par toutes les requêtes, et par là-même économiser les ressources pour le calcul. La technique est très connue dans le domaine du pattern-matching (« automate dictionnaire »), et rejoint la recherche en optimisation conduite depuis longtemps dans la communauté des bases de données [109, 27], qui est aujourd'hui étudiée intensivement dans le contexte des systèmes de gestion des flux de données [8, 61].

Concrètement un automate dictionnaire est l'automate qui reconnaît un *ensemble* de mots et dont :

- l'ensemble des états est l'ensemble des préfixes de l'ensemble de mots ;
- l'état initial est le mot vide ϵ ;
- l'ensemble des états terminaux est l'ensemble des mots ;
- les transitions sont de la forme (u, a, ua).

Il est aisé de voir qu'un tel automate est déterministe.

Exemple 22. *Supposons que l'on ait les trois patterns d'interrogation, toujours sur l'espace de référence représenté figure 5.2,* `X.Y`, `X.Z.X.Z` *et* `X.Z.X.Y.X`. *L'automate dictionnaire associé au dictionnaire* {`X.Y`, `X.Z.X.Z`, `X.Z.X.Y.X`} *est représenté sur la Figure 5.5.*

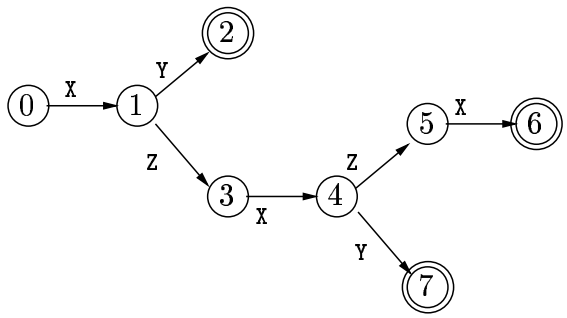


FIG. 5.5 – Automate du dictionnaire {`X.Y`, `X.Z.X.Z`, `X.Z.X.Y.X`}

Si l'on souhaite construire un automate dictionnaire permettant de reconnaître un ensemble de mots sur un *flux* de données, il faut avoir un automate nous permettant de ne jamais revenir en arrière lors d'un échec, et donc de toujours avancer suivant les symboles lus en entrée. Pour cela on introduit la fonction h qui pour un mot w donné retourne le plus long préfixe de cet ensemble de mots. La définition de l'automate est alors la suivante :

- l'ensemble des états est l'ensemble des préfixes de l'ensemble de mots ;
- l'état initial est le mot vide ϵ ;
- l'ensemble des états terminaux est l'ensemble des mots ;
- les transitions sont de la forme $(u, a, h(ua))$;

Exemple 23. *La figure 5.6 montre l'automate dictionnaire correspondant à celui de la figure 5.5 permettant de reconnaître l'ensemble des mots* {`X.Y`, `X.Z.X.Z`, `X.Z.X.Y.X`} *sur un*

flux de données. Les transitions « d'échec », ramenant à l'état initial, n'ont pas été représentées. On remarque aussi que l'état 5 est devenu terminal car il correspond au mot `X.Y`.

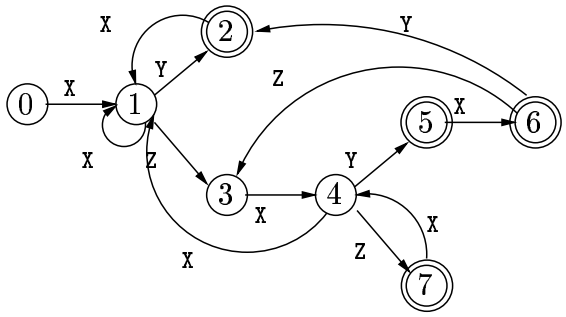


FIG. 5.6 – Automate du dictionnaire {`X.Y`, `X.Z.X.Z`, `X.Z.X.Y.X`} pour analyser un flux

A noter qu'une telle optimisation est d'autant plus intéressante que les requêtes visent à classer des trajectoires relativement proches, ayant de nombreux symboles en commun (ce qui se produit lorsque l'on souhaite par exemple une analyse *locale* de trafic).

5.5 Conclusion

Nous présentons dans ce chapitre une nouvelle approche pour classifier, regrouper et interroger de manière continue des objets mobiles. Les calculs reposent sur une partition thématique multi-échelle de l'espace sous-jacent et peuvent être réalisés en appliquant des techniques simples et efficaces (*i.e.*, « pattern-matching » sur des chaînes de caractères). Il s'agit d'un travail original qui constitue une façon pratique de gérer des requêtes continues sur un ensemble d'objets ayant un mouvement continu dans l'espace 2D. À notre connaissance cette approche n'a pas été explorée jusqu'à présent dans la littérature des bases de données spatio-temporelles.

Chapitre 6

Prototype

Nous avons implanté notre architecture avec à l'esprit les objectifs suivants :

i) montrer l'applicabilité de notre modèle ;

ii) disposer d'une plate-forme pour tester nos techniques d'optimisation ;

iii) expérimenter des techniques d'animation et de visualisation couplées avec les fonctionnalités du serveur.

6.1 Introduction

La figure 6.1 présente l'architecture client/serveur globale du prototype.

Sur cette figure, on a associé une couleur à chacun des composants du prototype. Les principales caractéristiques sont les suivantes :

- les serveurs GPS sont représentés par un simulateur qui génère incrémentalement les trajectoires d'un ensemble d'objets mobiles sur un réseau contraint ;
- le serveur reçoit les événements du simulateur et les traite afin de mettre à jour le résultat des requêtes. Les mises à jour sont transmises au client ;
- les clients sont des navigateurs web équipés du navigateur *SVG* d'*Adobe*. Quand une requête est enregistrée, chaque client reçoit une petite applet qui se connecte au serveur et envoie des demandes de mises à jour ;
- le serveur envoie initialement au client un document SVG (voir plus bas) représentant les données géographiques (carte et réseau routier) ainsi que les objets mobiles. Ce document est représenté du côté du client par un arbre *DOM* qui est rafraîchi

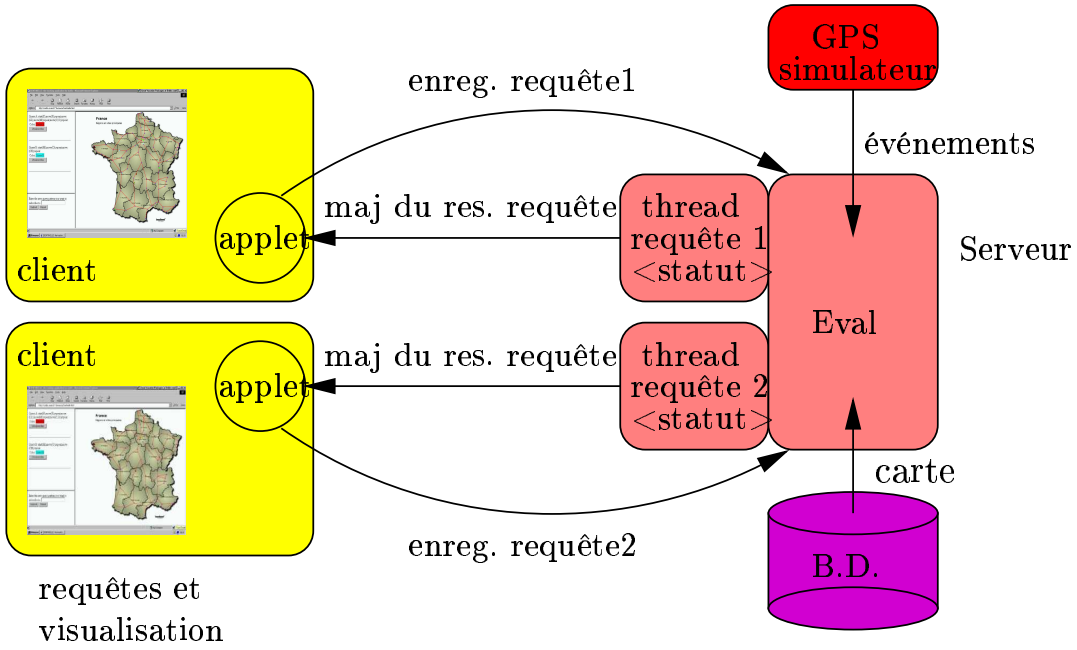


FIG. 6.1 – Architecture du système

par l'applet à chaque fois qu'une mise à jour est reçue. Le navigateur SVG modifie automatiquement l'affichage présenté à l'utilisateur.

Dans l'état actuel de l'implantation du prototype, le client reçoit, lorsqu'il s'enregistre, une représentation graphique de la carte entière. Il peut ensuite enregistrer des requêtes et visualiser leur résultat à l'instant de soumission des requêtes, mais également suivre l'évolution de ce résultat au cours du temps.

Dans la suite de ce chapitre nous allons d'abord introduire le langage de visualisation SVG (section 6.2), puis regarder en détail comment a été implanté le simulateur d'événements GPS (section 6.3). La section 6.4 présente les différents composants du serveur alors que la section 6.5 décrit les choix d'implantation pour le client. La section 6.6 conclut le chapitre en décrivant les résultats obtenus à l'aide de ce prototype.

6.2 Le langage SVG

XML est un méta-langage qui permet de définir des langages spécialisés pour un large panel d'applications. Un de ces langages spécialisés est *SVG* (Scalable Vector Graphics) qui permet de décrire des objets graphiques avec des couleurs, des textures et des animations [133]. SVG peut représenter et afficher les principales primitives utilisées dans la représentation cartographique, dans un format vectoriel adapté pour l'affichage à différents

niveaux d'échelle.

6.2.1 Représentation d'objets géométriques

SVG admet trois types d'objets graphiques : des contours graphiques vectoriels (par exemple, des tracés consistant en lignes droites et courbes), des images et du texte. Les objets graphiques peuvent être regroupés, stylés, transformés et composés dans d'autres objets. Du point de vue géométrique, le type qui nous intéresse particulièrement est le type « tracé », représentant le contour d'une forme, qui peut avoir un remplissage ou non. Les tracés représentent la géométrie du contour d'un objet, définis selon les instructions des éléments *moveto* (établit un nouveau point courant), *lineto* (dessine une droite), *curveto* (dessine une courbe à l'aide d'une courbe de Bézier cubique), *arc* (un arc circulaire ou elliptique) et *closepath* (clôt la forme courante en dessinant une ligne jusqu'au dernier *moveto*). Il est possible de composer plusieurs tracés. On définit un tracé avec un élément « path » ayant un attribut $d =$ *"<mon_ chemin>"*, où d contient les commandes moveto (noté M), line (L, H ou V), curve (C, S, Q ou T) (avec des courbes de Bézier cubique ou quadratique), arc (A) et closepath (Z).

L'exemple suivant montre la description SVG d'un tracé avec cinq points.

```
<path d="M0,0 L 100,0 100,100 100,200 150,250"
      style="fill:none;stroke:#00FF00;
      stroke-width:5"
/>
```

d permet de décrire la géométrie de ce tracé, M permettant de se placer à la coordonnée $(0, 0)$, le L précisant que les points dont on donne ensuite les coordonnées sont reliés par des segments. D'autres attributs décrivent les caractéristiques de ce tracé, comme la couleur, l'épaisseur de la ligne, etc. À l'aide de ce type, nous avons créé la plupart des objets géométriques de la carte utilisée dans le prototype, notamment les *régions* et le réseau autoroutier dans l'exemple que nous avons choisi pour illustrer ce chapitre. Noter qu'il existe également des types standard géométriques pour les *rectangles, cercles, lignes et polygones.*

6.2.2 Les animations

De plus, les dessins SVG peuvent être interactifs et dynamiques. On peut définir et déclencher des animations, soit en incorporant les éléments d'animation SVG directement dans un contenu SVG, soit via un script. Les animations dans SVG s'appuient sur le temps, à savoir qu'il est nécessaire d'indiquer le départ et la durée d'une animation. Le départ peut être lié à un événement ou un délai écoulé après le chargement de la page. Regardons plus en détail comment réaliser une animation avec SVG.

City		
idcity	integer	identifiant de la ville
name	character varying(40)	nom de la ville
x	integer	position (abscisse) de la ville
y	integer	position (ordonnée) de la ville
weight	integer	importance de la ville (entre 1 et 5)

TAB. 6.1 – Table stockant les *villes*

Il existe 5 balises XML s'intégrant dans la définition d'un objet pour déclarer une animation : *animate, set, animateMotion, animateColor* et *animateTransform*. La plus utilisée dans notre prototype est *animateMotion*, qui entraîne le déplacement continu et à vitesse constante d'un élément le long d'un tracé de mouvement. Le code suivant permet ainsi de faire déplacer un cercle le long du tracé précédemment décrit.

```
<circle x="0" y="0" r="20" style="fill:red;">
    <animateMotion dur="30s" repeatCount="indefinite"
                   path="M0,0 L 100,0 100,100 100,200 150,250" />
</circle>
```

Ici l'absence d'attribut *begin* spécifiant le début de l'animation signifie qu'elle s'exécute dès le chargement de la page. L'attribut *path* de l'élément *animateMotion* signifie que l'objet doit se déplacer sur ce tracé (le *L* ici indique que l'objet suit une trajectoire linéaire entre deux points étapes), et la durée totale du déplacement dure le temps fixé par l'attribut *dur*.

Afin de représenter nos objets mobiles nous utilisons donc cet élément *animateMotion* pour décrire leur trajectoire. Comme la trajectoire n'est pas connue à l'avance mais est générée au fur et à mesure, nous allons modifier les valeurs prises par cet attribut à l'aide de scripts extérieurs, ici du *javascript*. Nous décrirons plus en détail le code javascript dans la section consacrée au client.

6.2.3 Chargement de la carte

Nous avons voulu que ce prototype soit le plus évolutif possible. Pour cela nous avons voulu qu'il ne repose pas sur une carte unique et qu'il soit possible d'ajouter autant de cartes que l'on souhaite. Nous avons fait le choix de stocker ces différentes cartes dans une base de données (une base *PostgreSQL* [87]) et de charger la carte désirée par un utilisateur. Dans la suite nous décrirons en détail notre exemple du réseau autoroutier français sur une carte du territoire métropolitain divisé en régions administratives. Les tableaux 6.1 6.2 et 6.3 décrivent le schéma de la base permettant de stocker la carte.

Region		
idregion	character varying(1)	identifiant de la région
name	character varying(40)	nom de la région
border	polygon	polygone décrivant la géométrie
label_x	integer	position (abscisse) pour affichage du nom
label_y	integer	position (ordonnée) pour affichage du nom

TAB. 6.2 – Table stockant les *régions*

Road		
idroad	integer	id. du segment de route
iddep	integer	id. de la ville de départ du segment
idarr	integer	id. de la ville de d'arrivée du segment
cost	integer	temps de parcours de ce segment
succ	character varying(3)	étiquettes des régions traversées

TAB. 6.3 – Table stockant les *routes*

Dans le programme java permettant de construire la carte de référence, nous établissons une connexion à la base de données afin de récupérer l'information décrivant les objets géographiques de l'espace. Ainsi pour consulter chacune des trois tables présentées dans 6.1, 6.2 et 6.3, nous utilisons les trois requêtes suivantes :

- ```
  select idregion,name,border,label_x,label_y
  from region
  order by idregion ;
  ```

  qui retourne l'identifiant, le nom, le contour (polygone) et la position de l'étiquette contenant le nom de la région ;

- ```
  select idcity, name, x, y
  from city
  order by idcity ;
  ```

  qui retourne l'identifiant, le nom et les coordonnées des différents villes ;

- ```
  select r.idroad, c1.x,c1.y,c2.x,c2.y
  from road r, city c1, city c2
  where c1.idcity = r.iddep and c2.idcity = r.idarr order by idroad ;
  ```

  qui retourne l'identifiant du segment de route, ainsi que les coordonnées de sa ville
  ```

de départ et d'arrivée.

Le résultat de ces requêtes est ensuite transformé en code SVG. Ce code est ensuite envoyé au client.

6.3 Le simulateur

Le simulateur fournit une génération en ligne de trajectoires pour un ensemble d'objets mobiles sur un réseau contraint. Par « en ligne » on entend ici que la trajectoire est construite pas à pas, plutôt que générée entièrement à l'avance. Le principe du générateur est le suivant : à un instant donné, le simulateur sait qu'un objet o se déplace sur un segment de route du réseau, dans une direction et à une vitesse données. Lorsque que o atteint un nouveau nœud (ville) du réseau, le simulateur décide si o s'arrête à ce nœud ou s'il continue. Dans ce cas il calcule le nouveau segment affecté à o ainsi que sa nouvelle vitesse. Noter que pour notre simulateur nous avons considéré que la vitesse est constante sur un segment de route.

Les choix pris par notre simulateur sont guidés par un ensemble de propriétés associées au réseau considéré, propriétés permettant de décrire la distribution du trafic. Plus particulièrement, nous prenons en compte l'information suivante :

- un *poids* affecté à chaque nœud qui représente l'« importance » relative de ce nœud comme point de départ d'une trajectoire. Dans le contexte de notre réseau autoroutier par exemple, une métropole aura une importance de 5 alors qu'une petite ville aura une importance de 1 ou 2. Si on imagine une application suivant des déplacements au sein d'une ville, un nœud localisé dans une zone résidentielle aura une importance de 4 ou 5, alors qu'un carrefour dans une zone déserte aura une importance de 0 ;

- un *taux d'absorption* affecté également à chaque nœud. Il s'agit du pendant du *poids*. En effet il représente le pouvoir d'attraction d'un nœud pour un objet mobile. Intuitivement, chaque fois qu'un objet atteint un nœud avec un fort taux d'absorption, il a une grande probabilité de s'arrêter et rester là. Par exemple dans le cas de l'application « ville », une zone de bureau présente un fort taux d'absorption, alors qu'un carrefour présente lui un faible taux d'absorption ;

- une *importance* affectée à chaque segment de route *orienté* partant d'un nœud n, traduisant la probabilité qu'un objet arrivant en n puisse emprunter ce segment. Par exemple, une autoroute sera plus fréquemment empruntée qu'une route communale. Noter que nous avons choisi de considérer les segments comme orientés, et donc nous devons préciser deux segments entre les nœuds du réseau, éventuellement avec des valeurs différentes. Nous permettons ainsi de modéliser par exemple une voie à sens unique (un des deux segments ayant une importance de 0), ou une autoroute en travaux dont l'une des directions sera alors moins empruntée ;

– une *longueur* affectée à chaque segment de route. Il représente le temps mis pour parcourir ce segment de route.

Ces paramètres, ainsi que d'autres que nous omettons par soucis de simplicité, sont relativement simples. Notre objectif ici est limité à simuler des informations fournis par des équipements GPS avec un semblant de réalisme. Nous renvoyons à la section 2 pour la présentation de modèles plus sophistiqués.

En réalité notre simulateur procède en deux étapes :

1. une génération des trajectoires lors du démarrage du serveur pour tous les objets mobiles qui interviendront dans la simulation. Une trajectoire est concrètement une liste de nœuds traversés avec le temps de passage. Ces données sont sauvegardées dans la base de données (ou éventuellement, si on a peu de données, dans un tableau en mémoire centrale) ;

2. à chaque instant de la simulation, on détermine la position exacte (par interpolation entre deux positions connues et stockées dans la base) de l'objet à cet instant et on transmet cette position au serveur traitant les événements GPS. Pour ce dernier, nous n'avons donc aucune connaissance *a priori* des trajectoires, uniquement des positions précises transmises telles des événements GPS.

Plus précisément, une simulation est créée à l'aide des classes suivantes :

– `MapCreator`. Cette classe accède à la base de données PostgreSQL (via un driver jdbc) et afin de créer le fichier SVG correspondant aux données demandées par l'utilisateur elle évalue un ensemble de requêtes sur la base. Chaque nuplet (régions, villes, routes, etc.) retourné est transformé en élément SVG afin d'être affiché. Le document SVG généré contient à la fois l'information permettant d'afficher la carte de référence mais également le code représentant la couche thématique supplémentaire choisie, comme par exemple ici le réseau autoroutier. De plus cette classe incorpore dans le code SVG le code Javascript permettant de manipuler l'arbre DOM du document.

– `ReadBase`. Cette classe est similaire à la précédente, même si son objectif est tout autre. Elle transforme les données de la base nécessaire à la génération des trajectoires et surtout à l'évaluation des requêtes, en objets qui seront gardés en mémoire centrale. L'intérêt est ici d'optimiser le temps de mise à jour d'un résultat d'une requête en réduisant les accès à la base de données et en effectuant tous les calculs (position exacte, pointé, évaluation d'une requête, etc.) en mémoire centrale.

– `City2D`. Classe représentant une ville avec ses paramètres utiles pour la génération des

données et l'évaluation des requêtes.

- `Stage`. Classe associant une ville (*City2D*) à un temps de passage. Il s'agit d'une des villes-étapes appartenant à une trajectoire.

- `Region2D`. Classe représentant une région avec ses paramètres utiles pour la génération des données et l'évaluation des requêtes (notamment le contour de la région pour les opérations de pointé).

- `Trajectory`. Cette classe contient la liste des villes-étapes traversées successivement par un véhicule. Elle offre les fonctions pour ajouter une ville étape à une trajectoire, pour réaliser une opération de pointé afin de trouver dans quelle région se trouve un véhicule, ainsi qu'une fonction permettant de faire boucler indéfiniment un véhicule sur une trajectoire donnée.

- `Dijkstra`. Cette classe est une implantation de l'algorithme de Dijkstra. Elle permet de trouver le plus court chemin entre deux villes pour un véhicule donné, en fonction du poids (ici défini par le paramètre *longueur*) de chacun des segments de route.
- `CreateData`. Cette classe permet de créer les différentes trajectoires en prenant en compte les différents paramètres présentés plus haut. Nous la présentons en détail, avec des exemples ci-dessous.

La classe `CreateData` est le cœur du simulateur. Elle prend comme paramètre d'appel le nombre total d'objets ainsi que le temps de la simulation. Elle fixe la distance (en nombre de villes-étapes) minimale que doit comprendre une trajectoire. On a choisi pour l'application de référence que le *poids* d'un nœud et son *taux d'absorption* sont égaux (une ville génère autant de trafic qu'elle en absorbe). Nous avons choisi de générer les trajectoires en appliquant les principes suivants :

- les villes de départ et d'arrivée sont tirées aléatoirement, mais en prenant en compte la pondération liée à leur importance respective ;

- l'instant auquel démarre l'objet est tiré aléatoirement, en choisissant une loi de distribution uniforme sur toute la durée de la simulation ;

- le chemin choisi pour relier deux poids correspond à celui retourné par l'algorithme de Dijkstra ;

- une fois que l'objet a démarré, il se déplacera jusqu'à la fin de la simulation. Pour cela au lieu d'une seule ville destination, deux villes sont tirées aléatoirement et le véhicule une fois arrivé à la première ville destination, se rendra à la seconde, puis retournera à sa ville de départ pour recommencer ce parcours.

Pour chacun des objets de la simulation, elle génère une trajectoire suivant le pseudo-code ci-dessous.

GenereTrajectoires
début
poids_total=Σ poids_nœud
pour chaque $o \in \mathcal{O}$ **faire**
tirer aléatoirement un nœud de départ n_d (probabilité de chacun : $\frac{poids\_noeud}{poids\_total}$)
tirer aléatoirement un nœud d'arrivée n_{a_1} (probabilité de chacun : $\frac{poids\_noeud}{poids\_total}$)
appliquer dijkstra(n_d,n_{a_1}) pour obtenir la suite des villes traversées
si distance(n_d,n_{a_1}) $<$ MIN_DIST
alors recommencer avec un autre nœud d'arrivée
sinon
générer aléatoirement l'instant de départ t_d de l'objet
créer un objet *Stage* pour le départ (n_d, t_d)
pour chaque nœud de la trajectoire trouvée **faire**
calculer le temps de passage de l'objet
créer un objet *Stage* pour chaque étape
fin
tirer aléatoirement un nœud d'arrivée n_{a_2} (probabilité de chacun : $\frac{poids\_noeud}{poids\_total}$)
appliquer dijkstra(n_{a_1},n_{a_2}) pour obtenir la suite des villes traversées
si distance(n_{a_1},n_{a_2}) $<$ MIN_DIST **ou** distance(n_d,n_{a_2}) $<$ MIN_DIST
alors recommencer avec un autre nœud d'arrivée
sinon
pour chaque nœud de la trajectoire trouvée **faire**
calculer le temps de passage de l'objet
créer un objet *Stage* pour chaque étape
fin
appliquer dijkstra(n_{a_2},n_d) pour obtenir la suite des villes traversées
pour chaque nœud de la trajectoire trouvée **faire**
calculer le temps de passage de l'objet
créer un objet *Stage* pour chaque étape
fin
finsi
finsi
fin
fin

6.4 Le serveur

Le serveur joue le rôle central dans l'architecture (figure 6.1). Nous avons choisi de l'implanter comme un programme java connecté à une base PostgreSQL [87]. Les différentes fonctions assurées par le serveur sont :

1. lancement de la simulation ;
2. gestion des connexions/déconnexions des utilisateurs ;
3. enregistrement de nouvelles requêtes et désabonnement ;
4. récupération des événements GPS et calcul des zones correspondantes ;
5. mises à jour des résultats des différents requêtes ;
6. enregistrements des données dans la base.

La classe `LaunchSimul` permet de lancer la simulation. Un objet de type ServerSocket permet d'attendre sur le port 1805 les connexions des clients sur le réseau. L'administrateur donne le nombre de véhicules qui sont utilisés dans la simulation. Il donne aussi le temps total de la simulation, au bout duquel on réinitialise le compteur et recommence éventuellement une nouvelle simulation. Le serveur crée d'abord les données utilisées dans la simulation (les trajectoires pour chaque véhicule, la liste des régions traversées pas chaque route qui se trouve dans l'objet de type *Trajectory*,...). Elle crée pour cela une classe `CreateData` (cf section 6.3) permettant de générer à l'avance les trajectoires des différents véhicules. Ces trajectoires (suites des villes traversées avec temps de passage) sont gardées en mémoire centrale mais ne sont connues que par la classe simulant les événements GPS, `SimulGPS`, lancée lors de l'initialisation de la simulation également par `LaunchSimul`. et non par la partie du serveur évaluant les requêtes.

Pour chaque client qui se connecte, le serveur crée un nouveau *thread* qui s'occupera de la communication avec le client. Le serveur continue simultanément à écouter sur le port pour détecter de nouvelles connexions.

Lorsqu'un client enregistre une nouvelle requête, celle-ci est ajoutée à la liste des requêtes à traiter par le serveur. Ce dernier crée un automate à états finis en analysant la requête. Les transitions de cet automate sont étiquetées par des symboles représentant les régions, ou par des variables. On associe ensuite à chaque objet de la simulation un *statut* qui indique la position courante de l'objet dans l'automate, ainsi que l'instanciation actuelle des variables pour cet objet. Lors qu'une nouvelle position fournie par le simulateur est reçue par le serveur,

celui détermine par une opération de point-dans-polygone la zone dans laquelle est localisé l'objet. Il récupère l'étiquette correspondant et la donne en entrée à tous les automates. Le serveur détermine à l'aide des statuts la position de l'objet dans chaque automate, et teste la transition avec ce nouveau symbole. Si l'objet parvient à atteindre l'état final, il est ajouté au résultat de la requête et cette information est transmise au client qui a enregistré cette requête. Si l'objet était dans le résultat et le quitte, alors le serveur envoie également cette information au client. En cas d'échec, suivant la nature de la requête, une des techniques de retour présentées dans les chapitres précédents est appliquée.

Nous résumons les différents appels de méthodes et verrouillages de données sur la figure 6.2, dont les explications sont ensuite fournies.

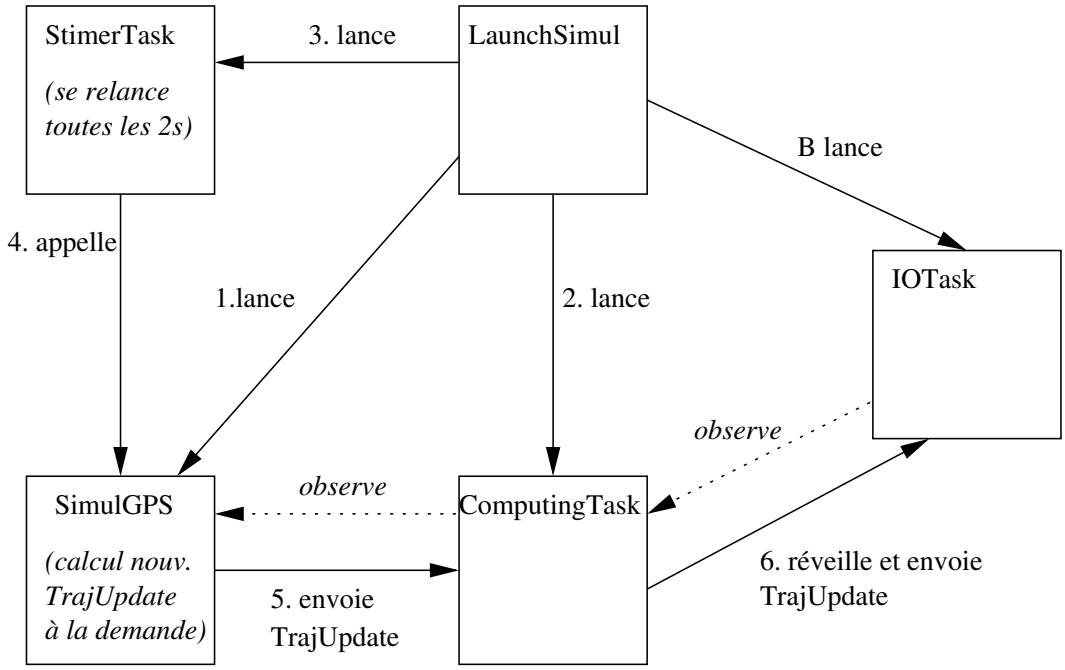


FIG. 6.2 – Architecture de l'implémentation

ComputingTask

1. se lance en tant que *thread* : *ComputingTask.run()*
2. obtient le verrou sur *ComputingTask.trajUpdate*
3. dort tant que *ComputingTask.trajUpdate* est vide
4. prend la donnée dans *ComputingTask.trajUpdate*
5. relâche le verrou sur *ComputingTask.trajUpdate*
6. prend le verrou sur *GlobalLock*
7. envoie *trajUpdate* aux différents *IOTask* en exécutant un par un l'appel *IOTask.update()*, qui met à jour les automates de chaque client suivant les étapes suivantes :

- obtient le verrou sur *myAutomata*
- met à jour l'automate associé à chaque requête du client
- prépare la réponse pour le client
- ajoute la chaîne de caractères de réponse à *sentMessage* (*sentMessage* est la chaîne des chaînes de réponses, afin de stocker les messages au cas où on n'a pas le temps de tout envoyer)

SimulGPS

1. obtient le verrou sur *ComputingTask.trajUpdate*
2. modifie *ComputingTask.trajUpdate*
3. si *ComputingTask.trajUpdate* vaut "arrêt", réveille le *thread ComputingTask*

IOTask

1. se lance en tant que *thread* : *IOTask.run()*
2. obtient le verrou sur *GlobalLock*
3. dort tant qu'il n'a pas de messages à envoyer au client
4. se réveille lorsque tous *les IOTask* ont fini leur mises à jour, et qu'il y a eu le *notifyAll* du verrou sur *GlobalLock*
5. libère le verrou *GlobalLock* dès son réveil
6. envoie ses données et fait l'affichage dans la fenêtre
7. regarde si le client veut ajouter/supprimer une requête
8. en cas d'ajout de requête, synchronisation sur *myAutomata*, ajoute la requête et crée un automate pour *IOTask*

Fin de la Simulation

- *LaunchSimul* appelle la méthode *finish* de *SimulGPS* qui envoie *null* au lieu de *TrajUpdate*, à *ComputingTask*
- *ComputingTask* envoie *null* à tous les *IOTask* et se finit
- les *IOTask* en recevant *null* se finissent

6.5 Le client

Le client est un navigateur web fournissant une interface graphique qui peut se décomposer en trois parties : la page HTML (incluant du code Javascript), le document SVG et l'applet. Afin de pouvoir afficher un document SVG dans un document HTML, un *plug-in* est nécessaire (nous utilisons le visualiseur SVG d'Adobe [4]). Quand le client charge la page HTML envoyée initialement par le serveur, il charge et affiche aussi le document SVG. Il faut ensuite mettre à jour ce document (et la représentation graphique) quand les événements

sont retournés par le serveur.

Un document SVG, comme n'importe quel document s'appuyant sur XML, correspond à un arbre DOM (Document Object Model) où chaque nœud est un élément [30]. La spécification DOM décrit une interface orientée-objet pour représenter et manipuler l'arbre. Grâce à cette interface nous pouvons insérer, effacer, accéder et modifier un nœud particulier. Par exemple la fonction *GetNodeById(myID)* est utilisée pour obtenir l'élément dont l'identifiant est *myID*. Nous utilisons une implantation Javascript de l'interface du DOM pour modifier le document SVG du côté du client. Lorsque le code SVG est modifié, le navigateur transmet immédiatement la modification à l'affichage graphique.

Deux sortes de modifications sont nécessaires. Tout d'abord, un événement GPS contient des informations sur la localisation ainsi que sur la vitesse et la direction. Ces informations doivent être utilisées pour actualiser les caractéristiques du mouvement de l'objet affiché sur la carte, indépendamment du fait que l'objet appartienne ou non au résultat de la requête. Le fait qu'un objet entre ou sorte du résultat d'une requête est une autre sorte d'information qui doit être représentée graphiquement. Pour cela nous avons choisi d'utiliser une palette de couleurs. Une couleur c_q caractérise une requête q, et un objet o satisfaisant q portera un marqueur de couleur c_q. Une couleur est choisie comme couleur par défaut pour représenter tous les objets affichés chez l'utilisateur et n'appartenant à aucun résultat des requêtes de cet utilisateur.

Imaginons par exemple que le client reçoive l'événement suivant :

Le véhicule 18 quitte le nœud 8 à l'instant 12 et atteindra le nœud 13 à l'instant 75.

Alors le code (Javascript/DOM) ci-dessous modifie l'objet animé représentant le véhicule o (plus précisément l'élément *animateMotion* imbriqué dans l'élément représentant l'objet) :

```
   CurrentNode=SVGDocument.getElementById(“node8”) ;
Vehicle_X=SVGDocument.getElementById(“node8_X”) ;
Vehicle_X.setAttribute('from',CurrentNode.getAttribute(“cx”)) ;
Vehicle_Y=SVGDocument.getElementById(“node8_Y”) ;
Vehicle_Y.setAttribute('from',CurrentNode.getAttribute(“cy”)) ;
CurrentNode=SVGDocument.getElementById(“node13”) ;
Vehicle_X.setAttribute('to',CurrentNode.getAttribute(“cx”)) ;
Vehicle_Y.setAttribute('to',CurrentNode.getAttribute(“cy”)) ;
Vehicle_X.setAttribute('begin','12s') ;
Vehicle_Y.setAttribute('begin','12s') ;
Vehicle_X.setAttribute('dur','63s') ;
Vehicle_Y.setAttribute('dur','63s') ; V.getStyle().setProperty('fill','red') ;
```

Dans ce code nous utilisons la fonction Javascript *getElementById* pour récupérer l'élément correspondant au nœud 8 que nous stockons dans la variable *CurrentNode*. Nous fixons

ensuite les attributs x et y de départ de l'animation comme étant les valeurs des attributs correspondant de l'élément *CurrentNode*. On procède de même avec le nœud d'arrivée. Il faut ensuite préciser que l'animation débute au temps 12 et finit au temps 63. Comme l'affichage traite l'abscisse et l'ordonnée comme deux attributs distincts et indépendants, il faut préciser les caractéristiques de l'animation à la fois pour x et pour y. Enfin la dernière ligne permet de changer la couleur et signaler que l'objet o appartient au résultat de la requête q identifiée par la couleur rouge.

En résumé, le cycle de mise à jour d'un client est le suivant :

1. l'applet envoie régulièrement des requêtes *pull()* au serveur pour demander d'éventuelles mises à jour ;

2. le serveur envoie en retour les (éventuels) événements intéressants l'utilisateur ;

3. l'applet et le Javascript appliquent ces mises à jour en modifiant l'arbre DOM du document SVG.

La figure 6.3 montre l'interface graphique du prototype.

Elle se décompose en un ensemble de cadres affichés dans un navigateur web. La fenêtre principale affiche les différents objets géographiques, à savoir la carte de référence (ici le territoire français divisé en régions administratives) ainsi que n'importe quelle autre couche géographique utile. Pour cet exemple particulier, la carte de référence est combinée avec un réseau routier joignant les principales villes françaises.

La fenêtre en bas à gauche est utilisée pour enregistrer des requêtes auprès du système en utilisant le langage présenté au chapitre 3. Chaque fois qu'une requête est saisie, on lui affecte un ensemble d'attributs graphiques spécifiques pour permettre de distinguer les objets appartenant à son résultat et ceux appartenant aux résultats d'autres requêtes. Dans l'implantation actuelle, un résultat de requête est simplement associé à une couleur spécifique, mais on pourrait envisager une caractérisation graphique plus « sophistiquée » si de nombreuses requêtes sont enregistrées simultanément.

On voit ici sur la figure 6.3, par exemple, deux requêtes enregistrées simultanément par ce client :

Requête A : `start(A) ; move(P) ; repeat ; move(LL) ; move(M) ; repeat ; move(C,0) ;repeat`

Cette requête permet de désigner les véhicules qui sont partis de la région `A` pour aller en `P` à l'instant suivant, sont restés en `P` un temps indéterminé (`repeat`), puis sont allés

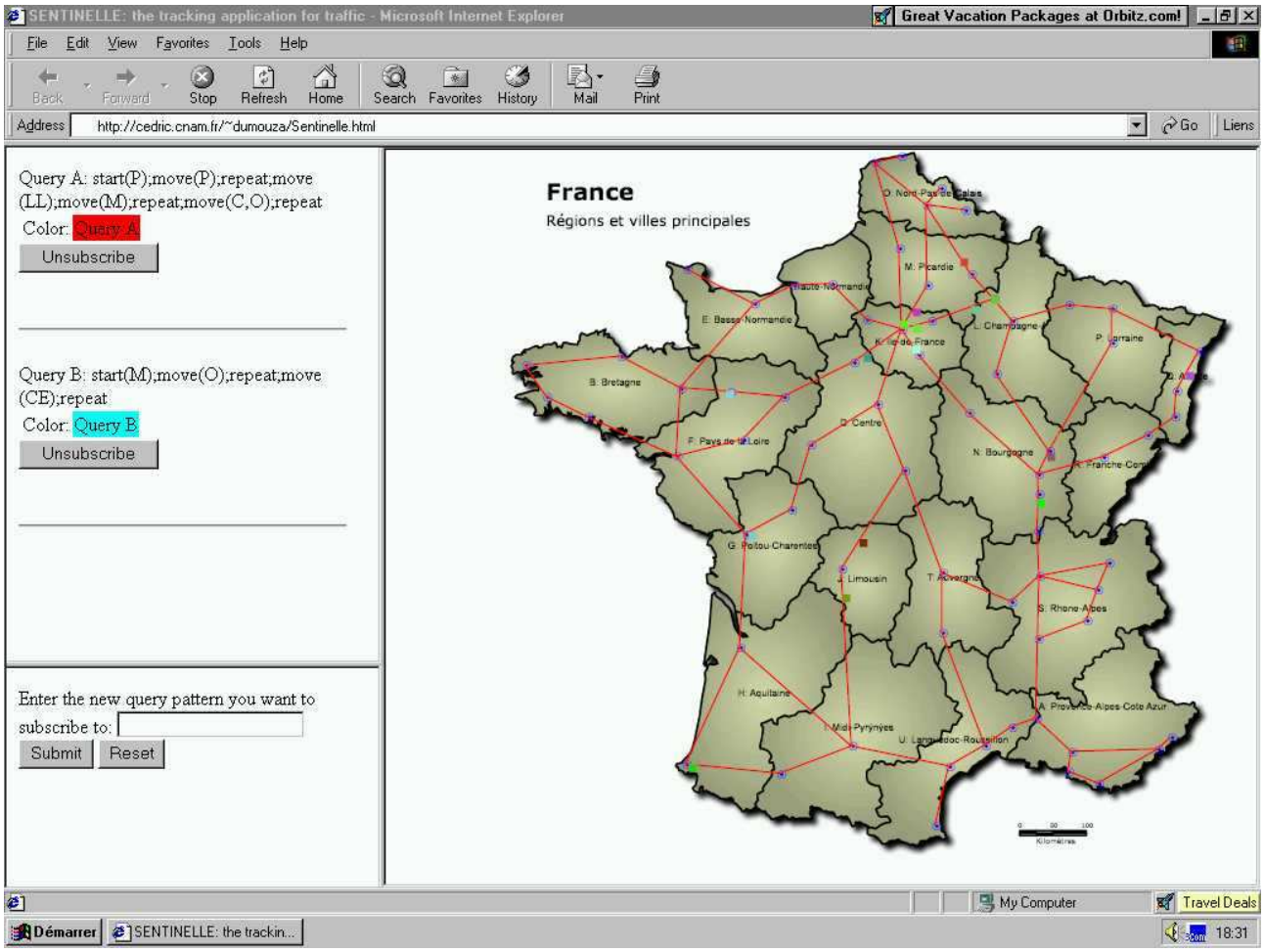


FIG. 6.3 – L'interface graphique du système

en `L` qu'ils ont quittés après deux unités de temps pour la région `M`. Après un temps indéterminé en `M`, ils sont allés en `C` ou `O`.

Requête B : `start(M) ; move(O) ; repeat ; move(CE) ; repeat`

Cette requête trouve les véhicules partis de `M` pour aller à l'instant suivant en `O`, qui ont ensuite quitté, après un temps indéterminé, `O` pour aller en `C` puis `E`.

Ces requêtes sont listées dans la fenêtre en haut à gauche. Pour chacune sont affichés un bouton permettant de désenregistrer la requête et une indication sur la couleur affectée aux objets appartenant au résultat de cette requête (rouge pour la requête A et indigo pour la requête B).

Dans la fenêtre principale sont affichés les véhicules représentés par des points dont la

couleur précise la requête dont ils font partie du résultat. Chacun de ces objets se déplace de manière continue sur le réseau routier affiché.

6.6 Résultats obtenus

Notre prototype nous a permis de montrer l'applicabilité de notre modèle. Nous avons en effet pu implanter notre représentation discrète des trajectoires à partir de la succession de localisations précises d'un objet mobile fournie par notre simulateur de données GPS. Pour l'implantation du simulateur nous avons opté pour une modélisation simple du trafic intégrant seulement certains paramètres. Le serveur reçoit ses données simulées et pour chacune réalise en mémoire centrale une opération de point dans polygone afin de déterminer la région dans laquelle se trouve l'objet. Il en déduit l'étiquette de la région qu'il utilise pour déterminer les mises à jour des résultats de la requête. Pour les requêtes nous avons choisi de créer un automate pour chacune des requêtes et chaque objet possède un *statut* pour cette requête, permettant de conserver les instanciations des variables de la requête. La requête a été saisie par l'utilisateur à l'aide d'un langage d'interrogation basé sur celui présenté au chapitre 3.

Tous ces choix sont détaillés dans les sections précédentes. Sur cette base nous avons pu réaliser des évaluations pour le modèle présenté au chapitre 4. Nous avons pu ainsi montrer que notre technique d'évaluation est plus performante dans toutes les situations à une technique d'évaluation naïve. Le gain est d'autant plus important que le pattern de la requête contient une grande proportion de variables, ce qui corrobore nos suppositions, vu qu'un grand nombre de variables entraîne de nombreuses superpositions partielles aboutissant à des situations d'échec.

Nous avons effectué également la conception d'une interface web pour le client. Pour cela nous avons souhaité tester des technologies nouvelles de représentation graphique. Le client affiche un document SVG qui est modifié via du code javascript appelé lorsque l'applet du client connecté au serveur reçoit des mises à jour. Le rendu graphique nous a satisfait, ainsi que l'affichage instantané des modifications (changements de direction et vitesse, mise à jour des résultats des requêtes) transmises par le serveur. Cependant ces choix d'affichage ne permettent pas le passage à l'échelle. En effet lorsque le nombre de véhicules devient trop important, le client web ne parvient plus à afficher toutes les animations. Il faudrait donc réserver cette interface à des applications de suivi de véhicules « locales » afin que chaque client n'ait à visualiser qu'un faible nombre d'objets.

Chapitre 7

Conclusion

Cette thèse a présenté essentiellement trois contributions correspondant à trois modélisations complémentaires de trajectoires d'objets mobiles ainsi qu'à l'élaboration d'un prototype à des fins à la fois d'étude de faisabilité du modèle et d'évaluation de performances. Ce chapitre résume les différentes contributions présentées dans cette thèse et s'achève sur diverses perspectives ouvertes.

Modèle des patterns de mobilité

Ma première contribution consiste en un *modèle de représentation et d'interrogation* de données original, basé sur une représentation discrète de l'espace et du temps. Ces dernières années, un grand nombre de modèles de données spatio-temporelles ont vu le jour, témoignant de l'importance croissante de cet axe de recherche. Ces nouveaux modèles sont nés du constat que les modèles traditionnels de bases de données spatiales ou temporelles ne peuvent généralement pas être étendus de manière simple pour prendre en compte les deux aspects simultanément.

Lors de ma thèse je me suis particulièrement intéressé à la représentation des objets (ponctuels) se déplaçant dans l'espace et par conséquent à la notion de trajectoire. Lors d'un premier travail, je me suis intéressé au problème des requêtes *continues* de fenêtrage lorsque le nombre d'objets et de requêtes est important. Il a été alors nécessaire d'associer au mieux les objets et les trajectoires. Comme les requêtes sont fixes dans l'espace, il est apparu pertinent d'indexer les requêtes plutôt que les trajectoires afin de trouver, pour chaque nouvel événement, les requêtes dont le résultat peut être affecté. Ce travail m'a permis de constater que, puisque la mise à jour des requêtes se produit lorsqu'un objet quitte ou entre dans une zone de requête, on peut maintenir le résultat (i) en partitionnant la carte d'après une grille, (ii) en détectant si un objet entre/quitte une zone et (iii) en recherchant les requêtes couvrant cette zone pour réaliser les mises à jour des résultats. L'idée d'un espace partitionné, souvent suivant une thématique choisie, m'a semblé séduisante puisque pour bon nombre d'applications, on ne s'intéresse pas à la localisation précise mais plutôt à la zone

dans laquelle se trouve l'objet. Des exemples de telles applications sont la gestion du trafic aérien ou l'espace est divisé en zones de contrôle couverte par un aéroport, la classification à la volée de trajectoires à des fins d'analyse statistique, etc. Nous avons en particulier étudié l'exemple concret de l'analyse des mobilités dans la région péri-urbaine grenobloise, dans le cadre du projet MOTIF [26]. Dans ce contexte, et pour ce type d'applications, une trajectoire serait alors représentée par une succession de zones traversées, traduisant un comportement de l'objet, et les requêtes porteraient sur cette représentation.

Une des caractéristiques de la majorité des modèles que j'ai étudiés lors de mon étude bibliographique est l'aspect continu des trajectoires considérées. Dans le cadre de ma problématique, la continuité ne permet non seulement pas de traduire un comportement de l'objet mais apparaît de plus comme pénalisante quant aux opérations géométriques coûteuses qu'elle engendre (calcul d'intersection par exemple).

Cela m'a conduit à présenter une nouvelle approche pour représenter les trajectoires des objets liée à la partition thématique de l'espace sous-jacent. Ainsi à chaque nouvel événement GPS reçu pour un objet, nous récupérons l'étiquette de la zone correspondant à cette position et l'ajoutons à notre séquence de déplacements, obtenant ainsi un mot dans l'alphabet des étiquettes de zones. Afin d'interroger et de suivre en temps réel des objets se déplaçant sur notre espace partitionné, nous utilisons des patterns de mobilité qui sont des séquences de déplacements pour lequel nous offrons certaines flexibilités. Il s'agit en fait d'un sous-ensemble des expressions régulières pour lequel nous introduisons des variables. L'objectif d'une requête est de retourner à chaque instant les différents objets dont la trajectoire satisfait cette séquence suivant une instanciation des variables déterminée. Cette représentation nous a conduit à étudier les techniques traditionnelles de recherche de chaînes de caractères dans un texte.

Nous identifions également un fragment de ce langage d'interrogation pour lequel l'espace nécessaire à la maintenance du résultat d'une requête est faible. Nous montrons les restrictions à apporter à notre premier langage afin d'exprimer des patterns pour lesquels nous pouvons réaliser une évaluation optimisée. Ces patterns permettent notamment de réaliser une évaluation en ne conservant au plus qu'une valuation possible des variables pour chaque état de l'automate associé.

Évaluation optimisée de patterns de mobilité

La technique précédente donne de bons résultats pour l'évaluation sur l'aspect espace mémoire, mais lors d'un échec dans la comparaison, elle implique la relecture de symboles de la trajectoire. Ce surcoût est d'autant plus important que le pattern contient des variables, et donc admettra plus fréquemment des superpositions partielles. Dans la continuité du travail précédent nous avons alors proposé une restriction du langage de requêtes pour laquelle nous avons pu développer un algorithme d'évaluation garantissant une évaluation linéaire en

nombre de comparaisons suivant la longueur de la trajectoire analysée.

Notre technique s'appuie sur un algorithme bien connu de recherche de chaînes de caractères appelé KMP [69], que nous avons étendu en ajoutant des variables dans la formulation des chaînes recherchées. Dans KMP, le décalage à réaliser, dépendant de la position de l'échec dans le pattern, est calculé sans avoir connaissance des données. En effet un simple parcours du pattern recherché permet d'établir le décalage, puisque celui-ci consiste à superposer le préfixe du pattern avec les derniers symboles lus, et comme ceux-ci ont été superposés au suffixe du pattern, trouver le bon décalage revient alors à trouver le plus long préfixe qui est également suffixe, appelé *bord*. Pour nos patterns avec variables, nous avons appliqué une technique similaire, sachant que le décalage à réaliser en cas d'échec dépend désormais de l'instanciation qui a été faite des variables avant l'échec. La conséquence du décalage sera une instanciation des variables du préfixe que l'on aura réussie à superposer *sans avoir eu à retester* la superposition. L'évaluation est alors linéaire en la taille de la trajectoire testée. L'ensemble des bords est stocké dans une table, dont nous donnons un algorithme de construction quadratique.

Pour conclure ce travail de recherche de patterns avec variable et valider notre technique, nous avons implanté un prototype s'appuyant sur un générateur d'événements GPS, permettant de suivre des véhicules se déplaçant sur le territoire français partitionné en 21 régions administratives. Nos différentes évaluations confirment les résultats attendus sur le gain apporté par notre algorithme quant au nombre total de comparaisons à réaliser.

Classification multi-échelle de trajectoires

Un autre aspect orthogonal que nous avons abordé est le problème du multi-échelle, toujours dans le cadre d'une application visant à suivre une flotte d'objets mobiles. Un comportement naturel consiste en effet à exprimer une requête plus précise dans les zones de l'espace qui intéresse l'utilisateur et exiger moins de précision dans des zones d'intérêt moindre. Pour cela on considère qu'on s'appuie sur un découpage multi-échelle de l'espace, par exemple le découpage *régions-départements-villes*. L'objectif de notre modèle est de définir un outil de classification et de regroupement et d'interrogation continue de trajectoires à l'aide de patterns que l'on spécifie pour un niveau d'échelle choisi.

Un pattern représente dans ce modèle un comportement typique d'une classe d'objets, suivant un thème choisi. Une fois le niveau d'échelle pour le pattern fixé par l'utilisateur, et donc la partition sous-jacente utilisée, les trajectoires sont projetées sur cette partition. Cette projection nous donne une séquence des zones du niveau d'échelle choisi, successivement traversées. Passer d'un niveau d'échelle plus fin à un niveau plus grossier permet de faire disparaître les différences entre deux trajectoires, alors que se placer à un niveau d'échelle fin, au contraire, permet de mieux distinguer les différences entre deux trajectoires. Un pattern permet alors de classer toutes les trajectoires dont la projection est égale au

pattern. Si l'on se place dans le cadre d'une application d'interrogation continue de trajectoires, on peut utiliser les patterns comme outil d'interrogation, cherchant les requêtes dont uniquement le suffixe, c'est-à-dire les événements les plus récents, satisfont le pattern.

Après avoir décrit notre modèle multi-échelle, nous avons formalisé nos patterns et décrit notre algorithme de classification et notre technique incrémentale d'évaluation de nos requêtes continues.

Perspectives

Cette thèse a ouvert plusieurs perspectives dont le cadre dépasse souvent les seules applications de suivis d'objets mobiles.

Regroupement de requêtes

Le contexte « web » permet d'envisager des systèmes gérant de grands nombres de requêtes continues. Cette problématique, assez récente, est motivée par les systèmes de « notification » où des serveurs permettent à des utilisateurs de « s'abonner » pour être prévenus quand un certain type d'événement survient (par exemple l'évolution de titres boursiers, le lancement d'offres promotionnelles sur des produits choisis, etc). Dans ces systèmes de notification, ce n'est plus la taille de la base qui devient un problème mais bien le nombre de requêtes à évaluer en continu. Dans ce cas, on peut envisager de s'appuyer sur une piste originale et prometteuse : nous avons montré dans [40] que l'on peut définir une relation d'ordre partiel sur l'ensemble des requêtes, et que cet ordre doit permettre de « filtrer » l'évaluation de certaines requêtes en évaluant au préalable celles dont elles dépendent. L'évaluation effective d'une requête n'est réalisée que si un objet franchit ce filtre. Cette évaluation « paresseuse » permet de réduire le nombre d'opérations puisque seuls les objets appartenant potentiellement à la solution d'une requête sont testés. Un modèle de coût ainsi que des tests sont en cours de réalisation afin de confirmer l'apport de ce regroupement de requêtes.

Implémentation

Une première implantation de mon architecture a été réalisée pour présenter mes premiers travaux. Ce premier prototype se compose d'un générateur d'objets mobiles fournissant un flot de positions, un serveur enregistrant les différentes requêtes et les évaluant à partir des données transmises par le simulateur et en conservant le résultat des requêtes à l'aide d'une structure compacte tenant en mémoire. Le client est un client web et dispose d'une interface graphique en *SVG*, langage fondé sur *XML*. Les requêtes évaluées actuellement sont sous forme de patterns avec variables, et la carte présente un découpage à un seul niveau. Sur la base de ce prototype j'ai également pu effectuer plusieurs évaluation de performances afin de montrer l'apport de ma technique d'évaluation optimisée pour un certain type de patterns.

Je suis actuellement en train d'intégrer dans le prototype déjà réalisé la notion de multi-échelle et les optimisations envisagées à ce jour. Ce prototype permet de tester mon modèle dans le cadre d'une application concrète et de réaliser différents tests de performance, étape indispensable de la validation d'une technique d'optimisation.

Améliorations du modèle

L'introduction de variables dans les patterns d'interrogation présentés au chapitre 3 a conduit au développement d'un modèle théorique et d'un algorithme d'évaluation. Tous deux sont déjà rédigés et sont corrects. Il me semble maintenant intéressant de se pencher sur le problème de l'évaluation simultanée de plusieurs patterns avec variables, ceci dans le cadre où un grand nombre d'utilisateurs soumettraient un grand nombre de requêtes.

Différentes améliorations peuvent être envisagées pour ce modèle spatio-temporel s'appuyant sur des patterns. Tout d'abord la prise en compte de l'*information topologique* de l'espace considéré devrait me conduire à des optimisations intéressantes pour l'évaluation des requêtes, en évitant par exemple de tester des zones dans lesquelles un objet n'a aucune chance de se trouver.
La *granularité temporelle* pourrait également être adaptée à la taille des zones. En considérant qu'un objet passe en moyenne plus de temps dans une zone de taille importante qu'une zone de taille moindre, on peut limiter des tests de localisation inutiles en espaçant dans le temps les tentatives de localisation pour la zone plus grande.
Il me semble également important de pouvoir intégrer des fonctionnalités de classification et d'interrogation axées sur le *temps*. Ainsi on peut envisager de s'intéresser non seulement aux zones traversées par un objet, mais également le temps passé dans chaque zone. On pourrait dès lors envisager des requêtes du type « donner moi les objets arrivant en `V`, après avoir passé *10mns* en `G` et entre *7 et 9mns* en `A` ». On peut également imaginer l'utilisation de variables temporelles. L'ajout de ces fonctionnalités nécessite un important travail de recherche, tant du point de vue de la modélisation que de l'évaluation.

Le traitement des jointures est également un aspect intéressant à considérer. La gestion des instanciations des variables et des bords, si l'on envisage une jointure "spatiale" n'est pas triviale. Ce type d'applications spatio-temporelles permet également de réaliser des jointures portant sur le temps. Il faudrait alors pouvoir introduire un mécanisme de synchronisation des trajectoires.

Un autre objectif à envisager est la fusion entre les deux modèles sur lesquels j'ai travaillé, à savoir le modèle multi-échelle et le modèle intégrant les variables. En effet ces deux notions semblent complémentaires, mais leur juxtaposition dans un même modèle a révélé plusieurs problèmes nécessitant une recherche plus poussée.

Bibliographie

[1] T. Abdessalem, J. Moreira, C. du Mouza, and P. Rigaux. *Spatial Databases : Technologies, Techniques and Trends*, chapter Management of Large Moving Objects Data Sets : Indexing, Benchmarking and Uncertainty in Movement Representation. Idea Group, Inc., 2005.

[2] S. Abiteboul, B. Amann, S. Cluet, A. Eyal, L. Mignet, and T. Milo. Active Views for Electronic Commerce. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 1999.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Adobe. The SVG Viewer, 2001. URL : http ://www.adobe.com/svg.

[5] P. Agarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *Proc. ACM Symp. on Principles of Database Systems*, pages 175–186, 2000.

[6] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 3–14, 1995.

[7] A. Apostolico and R. Giancarlo. The boyer-moore-galil string searching strategies revisited. *SIAM J. Comput.*, 15(1) :98–105, 1986.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *Proc. ACM Symp. on Principles of Database Systems*, pages 1–16, 2002.

[9] B. S. Baker. Parameterized Pattern Matching by Boyer-Moore Type Algorithms. In *Proc. of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 541–550, 1995.

[10] I. Bartolini, E. Bertino, B. Catania, P. Ciaccia, M. Golfarelli, M. Patella, and S. Rizzi. PAtterns for Next-generation DAtabase systems : preliminary results of the PANDA project. In *SEBD*, pages 293–300, 2003.

[11] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal*, 5(4) :264–275, 1996.

[12] L. Becker and R. Güting. Rule-Based Optimization and Query Processing in an Extensible Geometric Database System. *ACM Transactions on Database Systems*, 17(2) :247–303, 1992.

[13] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *Proc. Intl. Database Engineering and Applications Symposium (IDEAS)*, pages 44–53, 2002.

[14] J. L. Bentley and J. H. Friedman. Data Structures for Range Searching. *ACM Computing Surveys*, 11(4), 1979.

[15] S. Berchtold, B. Ertl, D. A. Keim, H.-P. Kriegel, and T. Seidl. Fast Nearest Neighbor Search in High-Dimensional Space. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 209–218. IEEE Computer Society, 1998.

[16] E. Bertino, B. Catania, M. Golfarelli, M. Halkidi, A. Maddalena, S. Rizzi, S. Skiadopoulos, M. Terrovitis, P. Vassiliadis, and M. Vazirgiannis. The Logical Model for Patterns. Technical Report TR-2003-02, PANDA, 2003.

[17] E. Bertino, B. Catania, and A. Maddalena. Towards a Language for Pattern Manipulation and Querying. In *PaRMa*, 2004.

[18] C. Bonhomme, C. Trépied, M.-A. Aufaure-Portier, and R. Laurini. A Visual Language for Querying Spatio-Temporal Databases. In *Proc. Intl. Symp. on Geographic Information Systems*, pages 34–39, 1999.

[19] R. Book, S. Even, S. Greibach, and G. Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2) :149–153, 1971.

[20] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10) :762–772, 1977.

[21] T. Brinkhoff. Generating network-based moving objects. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, pages 253–255. ACM, 2000.

[22] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2) :153–180, 2002.

[23] T. Brinkhoff and J. Weitkämper. Continuous Queries within an Architecture for Querying XML-Represented Moving Objects. In *Proc. Intl. Conf. on Large Spatial Databases (SSD)*, 2001.

[24] A. Bruggemann-Klein and D. Wood. One-unambiguous regular languages, 1998.

[25] B. Catania, A. Maddalena, and M. Mazza. PSYCHO : A Prototype System for Pattern Management. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 1346–1349, 2005.

[26] S. Chardonnel, C. du Mouza, M.-C. Fauvet, D. Josselin, and P. Rigaux. Patrons de mobilité : Proposition de définition, de méthode de représentation et d'interrogation. In *Conférence CASSINI-SIGMA, Géomatique et Analyse Spatiale*, 2004.

[27] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ : A Scalable Continuous Query System for Internet Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2000.

[28] L. Chen and R. T. Ng. On The Marriage of Lp-norms and Edit Distance. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 792–803, 2004.

[29] K. Chu and M. Wong. Fast Time-Series Searching with Scaling and Shifting. In *Proc. ACM Symp. on Principles of Database Systems*, 1999.

[30] W. Consortium. The Document Object Model, 2000. URL :http ://www.w3.org/DOM.

[31] M. Crochemore, A. Czumaj, L. Gasieniec, T. Lecroq, W. Plandowski, and W. Rytter. Fast practical multi-pattern matching. *Inf. Process. Lett.*, 71(3-4) :107–113, 1999.

[32] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du texte.* Vuibert, 2001.

[33] M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, 1994.

[34] C. Daassi, M. Dumas, M. Fauvet, L. Nigay, and P. Scholl. Visual Exploration of Temporal Object Databases. In *Actes des Journées Bases de Données Avancées (BDA)*, 2000.

[35] C. de Souza Baptista and Z. Kemp. Querying Multimedia Spatiotemporal Databases. In *Proc. Inl. Symp. on Database, Web and Cooperative Systems*, pages 145–150, 1999.

[36] Z. Ding and R. H. Güting. Managing Moving Objects on Dynamic Transportation Networks. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, pages 287–296, 2004.

[37] C. du Mouza and P. Rigaux. *Documents et Evolution*, chapter Bases de Données Spatio-Temporelles. ed. Cépaduès, publié dans le cadre de l'école thématique du GDR I3, 2000.

[38] C. du Mouza and P. Rigaux. Web architectures for scalable moving object servers. In *Proc. Intl. Symp. on Geographic Information Systems*, pages 17–22, 2002.

[39] C. du Mouza and P. Rigaux. Mobility Patterns. In *Proc. Intl. Workshop on Spatio-temporal Databases (STDBM)*, 2004. Toronto, Canada.

[40] C. du Mouza and P. Rigaux. Multi-scale Classification of Moving Objects Trajectories. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, 2004. Santorin, Greece.

[41] C. du Mouza and P. Rigaux. Mobility Patterns. *GeoInformatica*, 2005. To appear.

[42] C. du Mouza, P. Rigaux, and M. Scholl. Efficient Evaluation of Parameterized Pattern Queries. In *Proc. Intl. Conf. on Information and Knowledge Management*, 2005. To appear.

[43] M. Erwig, R. Güting, M. Schneider, and M. Vazirgiannis. Spatio-Temporal Data Types : An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica*, 3(3) :269–296, 1999.

[44] C. I. Ezeife and M. Chen. Mining Web Sequential Patterns Incrementally with Revised PLWAP Tree. In *WAIM*, pages 539–548, 2004.

[45] F. Fabret, H. Jacobsen, F. Llirba, K. Ross, and D. Shasha. Filtering Algorithms and Implementations for Very Fast Publish/Subscrib Systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2001.

[46] L. Forlizzi, R. Güting, E. Nardelli, and M. Schneider. A Data Model and Data Structures for Moving Objects Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2000.

[47] V. Gaede and O. Guenther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

[48] H. Garcia-Molina, J. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, 2000.

[49] D. Q. Goldin and P. C. Kanellakis. On similarity queries for time-series data : Constraint specification and implementation. In *Proc. Intl. Conf. on Principles and Practice of Constraint Programming (CP'95)*, 1995.

[50] I. W. Group. PANDA project. URL :http ://dke.cti.gr/panda.

[51] S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE System for Complex Spatial Queries. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 213–224, 1998.

[52] S. Grumbach, P. Rigaux, and L. Segoufin. Manipulating Interpolated Data is Easier than you Thought. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2000.

[53] S. Gupta, S. Kopparty, and C. V. Ravishankar. Roads, Codes and Spatiotemporal Queries. In *Proc. ACM Symp. on Principles of Database Systems*, pages 115–124, 2004.

[54] V. Guralnik and J. Srivastava. Event detection from time series data. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 33–42, 1999.

[55] R. Güting. Gral : An Extensible Relational Database System for Geometric Applications. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 1989.

[56] R. Güting. An Introduction to Spatial Database Systems. *The VLDB Journal*, 3(4) :357–399, 1994.

[57] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, E. Nardelli, M. Schneider, and J. R. R. Viqueira, editors. *Spatio-temporal Models and Languages : An Approach Based on Data Types. Spatio-Temporal Databases.* The CHOROCHRONOS Approach, 2003.

[58] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Quering Moving Objects. *ACM Trans. on Database Systems*, 25(1) :1–42, 2000.

[59] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras. On-Line Discovery of Dense Areas in Spatio-temporal Databases. In *Proc. Intl. Symp. on Spatial and Temporal Databases (SSTD)*, pages 306–324, 2003.

[60] M. Hadjieleftheriou, G. Kollios, V. Tsotras, and D. Gunopoulos. Efficient Indexing of Spatio-temporal Objects. In *Proc. Intl. Conf. on Extending Data Base Technology*, pages 251–268, 2002.

[61] M. Hammad, W. Aref, and A. Elmagarmid. Stream Window Join : Tracking Moving Objects in Sensor-Network Databases. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, 2003.

[62] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[63] G. S. Iwerks, H. Samet, and K. Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 512–523, 2003.

[64] G. S. Iwerks, H. Samet, and K. Smith. Maintenance of Spatial Semijoin Queries on Moving Points. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 828–839, 2004.

[65] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B^+-Tree Based Indexing of Moving Objects. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 768–779, 2004.

[66] D. Kalashnikov, S. Prabhakar, W. Aref, and S. Hambrusch. Efficient Evaluation of Continuous Range Queries on Moving Objects. In *Proc. Intl. Conf. on Databases and Expert System Applications (DEXA)*, pages 731–740, 2002.

[67] P. Kanellakis, G. Kuper, and P. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, 51(1) :26–52, 1995. A shorter version appeared in PODS'90.

[68] S.-W. Kim, J. Yoon, S. Park, and T.-H. Kim. Shape-based Retrieval of Similar Subsequences in Time-Series Databases. In *Proc. ACM Symposium on Applied Computing*, pages 438–445, 2002.

[69] D. Knuth, J. Morris, and V. Pratt. Fast Pattern Matching in Strings. *SIAM J. Computing*, 6(2) :323–350, 1977.

[70] G. Kollios, D. Gunopolos, and V. Tsotras. Nearest Neighbor Queries in a Mobile Environment. In *Intl. Workshop on Spatio-Temporal Database Management (STDBM'99)*, LNCS 1678, 1999.

[71] G. Kollios, D. Gunopolos, and V. Tsotras. On Indexing Mobile Objects. In *Proc. ACM Symp. on Principles of Database Systems*, pages 261–272, 1999.

[72] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast Nearest Neighbor Search in Medical Image Databases. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 215–226, 1996.

[73] M. Koubarakis. The complexity of query evaluation in indefinite temporal constraint databases. *Theoretical Computer Science*, 171(1–2) :25–60, 1997.

[74] G. Kuper, L. Libkin, and J. Paradaens, editors. *Constraint Databases.* Springer Verlag, 2000.

[75] L. Lakshmanan and P. Sailaja. On Efficient Matching of Streaming XML Documents and Queries. In *Proc. Intl. Conf. on Extending Data Base Technology*, 2002.

[76] G. M. Landau and U. Vishkin. Fast string matching with k differences. *J. Comput. Syst. Sci.*, 37(1) :63–78, 1988.

[77] R. Laurini and D. Thompson. *Fundamentals of Spatial Information Systems.* Number 37 in The A.P.I.C. Series. Academic Press, New York, 1992.

[78] Y.-N. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 492–503, 2004.

[79] H.-F. Li, S.-Y. Lee, and M.-K. Shan. On mining webclick streams for path traversal patterns. In *WWW (Alternate Track Papers & Posters)*, pages 404–405, 2004.

[80] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering*, 11(4) :610–628, 1999.

[81] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. In *ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'04)*, pages 236–245, 2004.

[82] G. Mecca and A. J. Bonner. Finite Query Languages for Sequence Databases. In *Proc. Intl. Workshop on Database Programming Languages*, 1995.

[83] R. Meo, P. L. Lanzi, and M. Klemettinen, editors. *Database Support for Data Mining Applications : Discovering Knowledge with Inductive Queries*, volume 2682 of *Lecture Notes in Computer Science*. Springer, 2004.

[84] N. Meratnia and R. de By. Aggregation and Comparison of Trajectories. In *Proc. Intl. Symp. on Geographic Information Systems*, 2002.

[85] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA : Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2004.

[86] M. F. Mokbel, X. Xiong, W. G. Aref, S. E. Hambrusch, S. Prabhakar, and M. A. Hammad. PLACE : A Query Processor for Handling Real-time Spatio-temporal Data Streams. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 1377–1380, 2004.

[87] B. Momjian. *PostgreSQL, Introduction and Concepts.* Addison Wesley, 2000. To appear. See `http ://postgresql.org`.

[88] D. W. Mount. *Bioinformatics Sequence and Genome Analysis, Second Edition.* CSHL Press, 2004.

[89] M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis. Evaluation for Access Structures for Discretely Moving Points. In *Intl. Workshop on Spatio-Temporal Database Management (STDBM'99)*, LNCS 1678, 1999.

[90] D. Pfoser and C. Jensen. Capturing the Uncertainty of Moving-Object Representations. In *Proc. Intl. Conf. on Large Spatial Databases (SSD)*, pages 111–132, 1999.

[91] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Objects. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2000.

[92] D. Pfoser, Y. Theodoridis, and C. S. Jensen. Indexing Trajectories of Moving Point Objects. Technical report, Chorochronos Network, 1999.

[93] K. Porkaew, I. Lasaridis, and S. Mehrotta. Querying Mobile Objects in Spatio-Temporal Databases. In *Proc. Intl. Symp. on Spatial and Temporal Databases (SSTD)*, 2001.

[94] E. U. Project. CINQ project. URL :http ://www.cinq-project.org.

[95] D. Rafiei and A. Mendelzon. Querying Time Series Data Based on Similarity. *IEEE Transactions on Knowledge and Data Engineering*, 12(5), 2000.

[96] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad. Srql : Sorted relational query language. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, pages 84–95. IEEE Computer Society, 1998.

[97] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast Nearest-Neighbor Query Processing in Moving-Object Databases. *GeoInformatica*, 7(2) :113–137, 2003.

[98] P. Rigaux and M. Scholl. Multi-scale partitions : Application to Spatial and Statistical Databases. In *Proc. Intl. Conf. on Large Spatial Databases (SSD)*, 1995.

[99] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Databases*. Morgan Kaufmann, 2001.

[100] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1995.

[101] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi. Expressing and optimizing sequence queries in database systems. *ACM Trans. on Database Systems*, 29(2), 2004.

[102] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. A Sequential Pattern Query Language for Supporting Instant Data Mining for e-Services. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2001.

[103] R. Sadri, C. Zaniolo, A. M. Zarkesh, and J. Adibi. Optimization of sequence queries in database systems. In *Proc. ACM Symp. on Principles of Database Systems*, 2001.

[104] J.-M. Saglio and J. Moreira. Oporto : A Realistic Scenario Generator for Moving Objects. In *Proc. Intl. Conf. on Databases and Expert System Applications (DEXA)*, pages 426–432, 1999. Version étendue à paraître dans GeoInformatica.

[105] S. Saltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, 2002.

[106] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 2000.

[107] M. Schneider. Evaluation of Spatio-Temporal Predicates on Moving Objects. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, 2005.

[108] T. Seidl and H.-P. Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 154–165, 1998.

[109] T. Sellis. Multiple-Query Optimization. *ACM Trans. on Database Systems*, 13(1) :23–52, 1988.

[110] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ : A Model for Sequence Databases. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 232–239, 1995.

[111] J. Sharma. Oracle8i Spatial : Experiences with Extensible Databases. An Oracle Technical White Paper, May 1999.

[112] I. Simon. String matching algorithms and automata. In *Results and Trends in Theoretical Computer Science*, pages 386–395, 1994.

[113] A. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 422–433, 1997.

[114] A. P. Sistla, T. Hu, and V. Chowdhry. Similarity based retrieval from sequence databases using automata as queries. In *Proc. Intl. Conf. on Information and Knowledge Management*, pages 237–244, 2002.

[115] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Querying the Uncertain Position of Moving Objects. In *Temporal Databases : Research and Practice*, volume 1399, pages 310–337. Springer-Verlag, 1998.

[116] J. G. Stell and M. F. Worboys. Generalizing Graphs Using Amalgamation and Selection. In *Proc. Intl. Conf. on Large Spatial Databases (SSD)*, volume 1651 of *LNCS*. Springer, 1999.

[117] J. Su, H. Xu, and O. Ibarra. Moving Objects : Logical Relationships and Queries. In *Proc. Intl. Conf. on Large Spatial Databases (SSD)*, pages 3–19, 2001.

[118] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the Past, the Present, and the Future in Spatio-Temporal. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 202–213, 2004.

[119] Y. Tao and D. Papadias. The MV3R-Tree : a Spatial-temporal Access Method for Timestamp and Interval Queries. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, 2001.

[120] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 334–345, 2002.

[121] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 287–298, 2002.

[122] J. Tayeb, O. Ulusoy, and O. Wolfson. A Quadtree Based Dynamic Attribute Indexing Method. *Computer Journal*, 41 :185–200, 1998.

[123] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous Queries over Append-Only Databases. In *Proc. ACM SIGMOD Symp. on the Management of Data*, 1992.

[124] Y. Theodoridis. Ten Benchmark Database Queries for Location-based Services. *Computer Journal*, 46(6) :713–725, 2003.

[125] Y. Theodoridis, T. K. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for Efficient Indexing in Spatiotemporal Databases. In *Intl. Conf. on Scientific and Statistical Database Management*, 1998.

[126] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Intl. Conf. on Large Spatial Databases (SSD'99)*, 1999.

[127] G. Trajcevski, O. Wolfson, F. Zhang, and S. Chamberlain. The Geometry of Uncertainty in Moving Objects Databases. In *Proc. Intl. Conf. on Extending Data Base Technology*, pages 233–250, 2002.

[128] E. Ukkonen. Finding approximate patterns in strings. *J. Algorithms*, 6(1) :132–137, 1985.

[129] M. Vazirgiannis and O. Wolfson. A Spatiotemporal Model and Language for Moving Objects on Road Networks. In *Proc. Intl. Conf. on Large Spatial Databases (SSD)*, volume 2121 of *LNCS*, pages 20–35, 2001.

[130] M. Vazirgiannis, Y.Theodoridis, and T. Sellis. Spatio-Temporal Composition and Indexing for Large Multimedia Applications. *Multimedia Systems*, 6(4) :284–298, 1998.

[131] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering Similar Multidimensional Trajectories. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, 2002.

[132] W. Wang, J. Yang, and R. R. Muntz. STING : A Statistical Information Grid Approach to Spatial Data Mining. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 186–195, 1997.

[133] A. Watt. *Designing SVG web graphics*. New Riders, 2002.

[134] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and Imprecision in Modeling the Position of Moving Objects. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 588–596, 1998.

[135] O. Wolfson, A. P. Sistla, B. Xu, J. Zhou, and S. Chamberlain. DOMINO : Databases fOr MovINg Objects tracking. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 547–549, 1999. (Demo sessions).

[136] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases : Issues and Solutions. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, pages 111–122, 1998.

[137] M. Worboys. A Unified Model for Spatial and Temporal Information. *Computer Journal*, 31(1) :36–34, 1994.

[138] S. Wu and U. Manber. Fast text searching allowing errors. *Commun. ACM*, 35(10) :83–91, 1992.

[139] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN : Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, 2005.

[140] X. Xiong, M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Scalable Spatio-temporal Continuous Query Processing for Location-aware Services. In *Proc. Intl. Conf. on Scientific and Statistical Databases (SSDBM)*, pages 317–326, 2004.

[141] X. Yu, K. Q. Pu, and N. Koudas. Monitoring K-Nearest Neighbor Queries Over Moving Objects. In *Proc. IEEE Intl. Conf. on Data Engineering (ICDE)*, pages 631–642, 2005.

[142] B. Zheng and D. L. Lee. Semantic Caching in Location-Dependent Query Processing. In *Proc. Intl. Symp. on Spatial and Temporal Databases (SSTD)*, pages 97–116, 2001.

[143] S. Zhou and C. Jones. Design and Implementation of Multi-Scale Databases. In *Proc. Intl. Symp. on Spatial and Temporal Databases (SSTD)*, 2001.

Résumé

Dans cette thèse j'envisage une approche originale de gestion des requêtes comme un processus traitant des *événements* (par exemple l'entrée dans une zone) liés aux déplacements des objets sur une représentation discrète de l'espace. Une requête se présente alors comme une *séquence* d'événements élémentaires donnés explicitement ou non.

Nous introduisons les *patterns de mobilité* comme des expressions décrivant de telles séquences d'événements. Nous avons étudié essentiellement deux aspects dans ce cadre :

- comparaison et agrégation de trajectoires d'objets mobiles, avec prise en compte éventuellement d'un espace multi-échelle ;
- classification en ligne de trajectoires mises à jour continuellement par des outils GPS.

Pour chacun des aspects abordés, nous proposons un modèle et une technique d'évaluation s'appuyant sur des algorithmes de recherche de patterns. Un prototype valide nos optimisations.

Title

Mobility patterns.

Abstract

In this thesis I investigate an original approach, namely the management of queries as a process relying on *events* (for instance, an object enters a zone) related to the moves of objects over a discrete representation of the underlying space. A query is thus a sequence of primitive events. We introduce *mobility patterns* as expressions describing such sequences of events. In the present paper we examine specifically the following aspects of this framework :

- comparison and aggregation of moving objects trajectories, with respect to, eventually, a multi-scale map ;
- on-line classification of trajectories continuously provided by GPS-like devices.

For each aspect, we propose a model and an evaluation technique based on pattern-matching algorithms. A prototype validates our optimizations.

Mots-clés

Objets mobiles, patterns, classification, optimisation, multi-échelle, requêtes continues

Keywords

Moving objects, patterns, classification, optimization, multi-scale, continuous queries.

Intitulé et adresse du laboratoire

Équipe Vertigo, laboratoire CEDRIC
Conservatoire National des Arts et Métiers
292 rue Saint-Martin
75141 Paris Cedex 03 - France