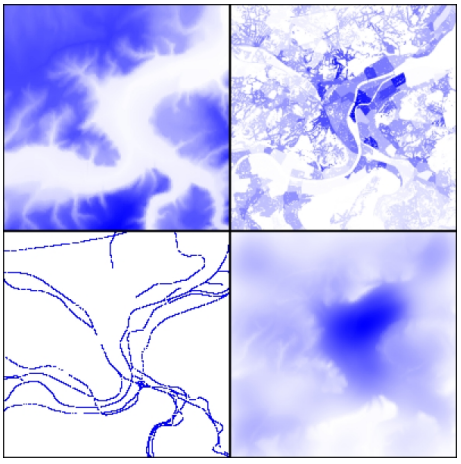
Algorithmes de contournement de barrières



Travail de fin d'études

Diplôme d'Etudes Approfondies en Informatique

Université de Liège

Par

Cyril Briquet

Juin 2003

Sous la supervision du Professeur P.-A. de Marneffe

Remerciements

Je tiens à remercier ...

Le Professeur Pierre-Arnoul de Marneffe, Département Montefiore, Université de Liège, pour m'avoir permis de réaliser ce travail et pour ses conseils judicieux.

Delphine Daxhelet et Jean-Marc Lambotte, Laboratoire d'Etude en Planification Urbaine et Rurale, Université de Liège, pour leur collaboration fructueuse et pour leur patience à m'expliquer certains concepts de la modélisation géographique.

Yves Cornet, Département Géomatique, Université de Liège, pour avoir fait se rencontrer l'offre et la demande, me permettant de collaborer avec l'équipe du LEPUR.

Sylvain Martin, Département Montefiore, Université de Liège, pour nos discussions toujours intéressantes permettant d'élargir mes perspectives.

Les relecteurs assidus qui ont donné la chasse aux coquilles, omissions et erreurs rédactionnelles de ce document.

Mes collègues présents et passés du Laboratoire d'Algorithmique pour leur soutien et l'excellent cadre de travail qu'ils contribuent à développer quotidiennement.

Table des matières

Table des algorithmes

Table des figures

Chapitre 1

Introduction

1.1 Emergence des Systèmes d'Information Géographique dans l'aide à la décision

Le Système d'Information Géographique est un système d'information qualifié de géographique dans la mesure où les données qu'il mémorise, gère et communique sont de nature géographique, c'est-à-dire localisées dans l'espace, par exemple par l'intermédiaire de coordonnées géographiques [Don02b].

Les solutions qu'apportent les Systèmes d'Information Géographique logiciels - souvent désignés par GIS[1] - **sont de plus en plus pertinentes mais requièrent davantage de recherche et de développements algorithmiques** à mesure que les problèmes traités se complexifient et que l'avènement de bases de données spatiales rend accessibles de grandes quantités de données géographiques.

Ainsi, au travers de la modélisation d'un problème et de la génération de données nouvelles à partir de données géographiques existantes, un GIS peut mettre à la disposition d'un décideur une vaste gamme d'informations visuelles éclairant considérablement sa perception de la situation à laquelle il doit faire face.

1.2 Contexte

Le LEPUR (Laboratoire d'Etude en Planification Urbaine et Rurale, ULg) **est un laboratoire de l'Université de Liège** basé au Sart-Tilman. Il participe à de nombreux programmes de recherches multidisciplinaires.

[1] Il s'agit de l'acronyme de la traduction en langue anglaise.

Le **calcul de profils d'accessibilité des modes lents**[2] en Région wallonne s'insère dans un vaste programme de recherches interuniversitaires fédérées par la CPDT (Conférence Permanente du Développement Territorial) et auquel participe le LEPUR.

L'objet de ce programme est *notamment* de sélectionner la localisation d'activités en Région wallonne en maximisant les possibilités d'utilisation de modes de transport alternatifs à la voiture (modes lents, transports en commun), sous contrainte de respecter la quiétude et le mode de vie des habitants des endroits sélectionnés.

Dans ce contexte, pour chaque élément du territoire wallon discrétisé, **un indice d'accessibilité des modes lents est calculé dans une perspective d'aide à la décision**.

En effet, cet indice d'accessibilité, stocké sous forme d'une couche de données, est superposable aux données géographiques correspondantes (couches images de type topographique ou attributaire) pour former une image composite utile.

Cet indice d'accessibilité *"rend compte de la possibilité de générer de nombreux déplacements cyclistes ou piétons"* [LEP02] à partir d'un point donné. Cela peut être vu comme la mesure en ce point du potentiel de déplacements avec des modes lents. L'indice d'accessibilité est exprimé dans une unité synthétique nommée *Equivalent-Usager*.

Les recherches en cours au LEPUR ont amené à considérer qu'une valeur élevée de l'indice d'accessibilité d'un élément signifiera que le relief vers les éléments de son voisinage est peu marqué et que son voisinage exhibe une forte densité de population.

Le LEPUR a développé un modèle de calcul de l'indice d'accessibilité et réalisé une implémentation pilote de celui-ci. Dans une perspective d'extension et d'amélioration de l'exactitude du modèle développé, la **prise en considération des obstacles non franchissables par les modes lents s'est imposée pour mieux rendre compte de la distance effective entre deux points** (cf. figure 1.1). De tels obstacles, qu'ils soient d'origine naturelle ou issus de l'activité humaine, sont appelés **barrières**.

C'est à ce moment que nous avons été contacté, de manière tout à fait informelle. Après plusieurs discussions sur l'extension du modèle et l'impact sur les algorithmes à développer et implémenter, il est rapidement apparu que cette extension - en apparence relativement simple mais dont l'impact sur les algorithmes utilisés est en réalité très important - du modèle de calcul offrait de nombreuses possibilités de recherche algorithmique.

[2]Modes de transport ayant comme caractéristique principale une faible vitesse de déplacement. Par exemple, la marche à pied et le vélo.

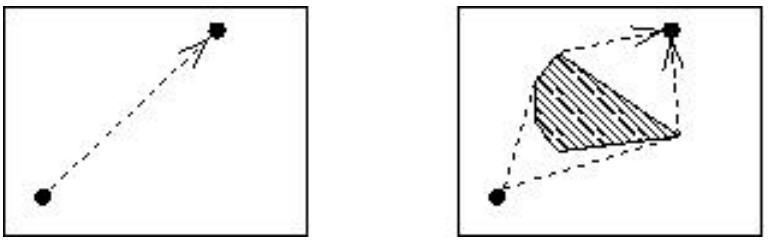


FIG. 1.1 – Contournement de barrière

En effet, l'impact sur tout profil d'accessibilité est très fort puisque **les barrières doivent être nécessairement contournées dès lors qu'elles sont prises en considération**. La compréhension intuitive du problème est relativement abordable mais le résoudre complètement constitue le fondement de ce travail.

1.3 Solutions

Les données utilisées sont finement spatialisées sur la densité de population : chaque élément des matrices issues des bases de données géographiques rassemblées pour ce projet correspond à une zone de 50 mètres de côté.

Cela implique que le volume de données à traiter pour l'ensemble du projet est considérable. Il est dès lors nécessaire d'effectuer les traitements requis de manière performante.

Les **objectifs** fixés sont les suivants :

- **étendre le modèle** développé par le LEPUR **en intégrant la prise en considération des barrières** ;
- **développer les algorithmes** décrivant les processus de calcul que sous-tend le modèle ;
- **réaliser une implémentation de ces algorithmes** performante du point de vue du temps d'exécution, avec et sans prise en considération des barrières.

Nous avons développé **deux composants logiciels résolvant les deux variantes du problème de calcul d'indice d'accessibilité qui sont définies par le modèle** :

- `DLOSP` (*Discrete Line-Of-Sight Paths*), sans prise en considération des barrières ;
- `DGSP` (*Discrete Geodesic Shortest Paths*), gérant le contournement des barrières.

Ces composants présentent les caractéristiques suivantes :

- **efficacité** : les algorithmes intervenant dans le processus de résolution sont analysés et les aspects d'implémentation sont considérés en vue de diminuer les temps de calcul ;
- **portabilité :** le code source est portable (↪ implémentation en C ANSI) ;
- **réutilisabilité :** les composants peuvent être exécutés séparément (données et résultats sous forme de fichiers immédiatement exploitables) ou intégrés dans un système de plus grande envergure.

Les **bénéfices escomptés** sont :

- **augmenter la connaissance** relative aux chaînes d'algorithmes adaptées au traitement de grands volumes de données de nature essentiellement géométrique typiques des GIS en général et des problèmes de contournement de barrières en particulier ;
- **améliorer de manière significative la production (génération, intégration, diffusion) d'images composites de qualité** d'un GIS logiciel dans un contexte réel à finalité décisionnelle.

1.4 Organisation de ce document

Le reste de ce document est composé comme suit :

Le chapitre 2 modélise le problème posé.

Le chapitre 3 présente le composant traitant le problème de base `DLOSP`.

Le chapitre 4 présente le composant traitant le problème `DGSP` avec contournement des barrières.

Le chapitre 5 poursuit le développement entamé au chapitre précédent, en présentant un modèle de visibilité puis en étudiant les requêtes de visibilité et de calcul de chemin le plus court.

Le chapitre 6 propose plusieurs optimisations pour accélérer les requêtes étudiées au chapitre précédent.

Le chapitre 7 discute de la distribution sur un réseau de stations de travail des calculs des composants logiciels développés.

Le chapitre 8 conclut et offre un résumé des développements réalisés.

Chapitre 2

Modèle

Quel résultat doit être finalement obtenu ? Un indice d'accessibilité, sous forme matricielle. De manière plus précise, cet indice rend compte de la capacité des usagers des modes lents à accéder au lieu pour lequel le calcul est effectué.

Les résultats attendus sont donc les valeurs de la matrice d'accessibilité, calculées d'après les valeurs d'autres matrices décrivant l'environnement géographique traité et en respectant un certain nombre de contraintes.

Nous allons nous attacher dans ce chapitre à définir **quelles valeurs et quelles contraintes** doivent intervenir.

2.1 Données

A quels éléments va s'appliquer le modèle qui va être développé ? Quelles sont les données qui doivent être traitées ?

Un échantillonnage des données géographiques doit être réalisé de manière à obtenir des données discrétisées sous forme de plusieurs couches de pixels. Il est réalisé par le LEPUR, laissant donc les problèmes de discrétisation (précision, apparition d'artefacts, ...) hors de portée de ce travail.

Etant donné la qualité des données et la considération du fait que résolution plus grande et temps de calcul plus longs sont intrinsèquement liés, la **résolution** sélectionnée par le LEPUR est **de 50 mètres par pixel** ($1\ pixel = 50\ m$).

Le LEPUR a divisé arbitrairement la surface à couvrir (la Région wallonne) en 13 zones de dimensions variant entre 250×250 et 1500×750 pixels. Une matrice carrée de 600 pixels de côté représente donc une surface de 900 kilomètres carrés.

Quelle est la **nature des éléments à traiter** ? Pour chaque zone, un ensemble de 3 couches de pixels constitue les données du problème. Les valeurs associées à chaque pixel sont stockées dans les éléments correspondants de ces **3 matrices** (1 matrice par couche de pixels) :

- B (figures 2.1 & 2.2) : couche de données topographiques indiquant la **présence d'une barrière** pour chaque pixel (sous forme de booléens) ;
- Pop (figures 2.3 & 2.4) : couche de données attributaires estimant la **densité de population moyenne** de chaque pixel urbanisé[1] (sous forme de nombres en virgule flottante signés et codés sur 32 bits) ;
- MNT (figures 2.5 & 2.6) : couche de données topographiques estimant l'**élévation moyenne** (exprimée en mètres arrondis à l'unité la plus proche) de chaque pixel : il s'agit du modèle numérique de terrain (sous forme de nombres entiers non signés et codés sur 16 bits).

Quelle est la **représentation des données produites** ? C'est **1 matrice** unique :

- Acc : couche de données calculées mesurant selon le modèle établi l'**accessibilité** de chaque pixel en Equivalent-Usager (sous forme de nombres en virgule flottante et codés sur 32 bits).

Les matrices de test utilisées dans le cadre de ce projet (figures 2.1-2.6) concernent 2 zones : une zone rurale de dimensions 600×600 pixels ainsi qu'une zone urbaine bien connue (région liégeoise) de dimensions 200×200 pixels.

2.2 Référentiel

On fera très souvent référence aux éléments des matrices et il est évident que les 3 matrices de données à traiter ainsi que la matrice de données produites auront exactement le même référentiel, leurs éléments pouvant être mis exactement en correspondance.

Décrivons un **référentiel pour une matrice abstraite qui pourra figurer n'importe laquelle des matrices de données ou de résultats**.

[1] La densité de population moyenne d'un secteur statistique est distribuée uniquement sur les pixels urbanisés parmi l'ensemble des pixels qui le composent.

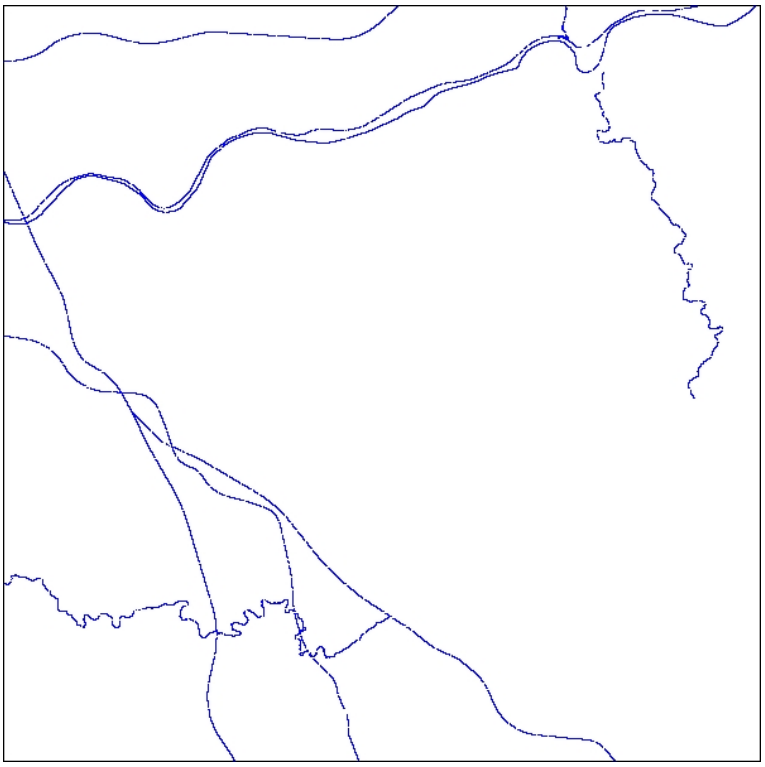


FIG. 2.1 – Barrières (zone rurale)

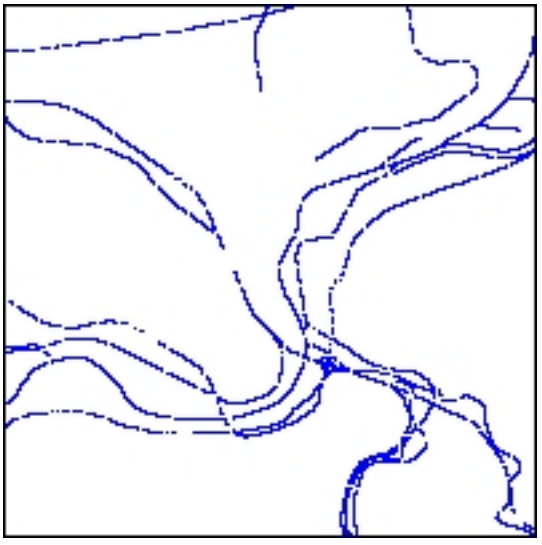


FIG. 2.2 – Barrières (zone urbaine)

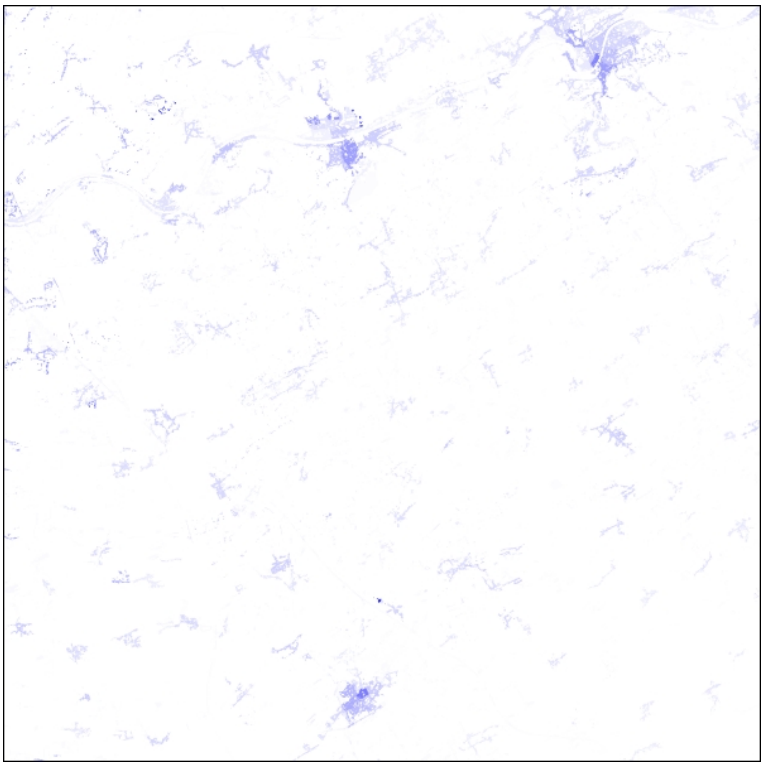


FIG. 2.3 – Densité de population (zone rurale)

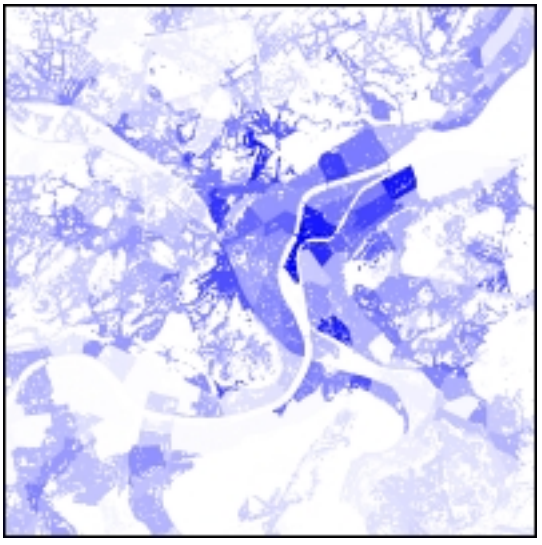


FIG. 2.4 – Densité de population (zone urbaine)

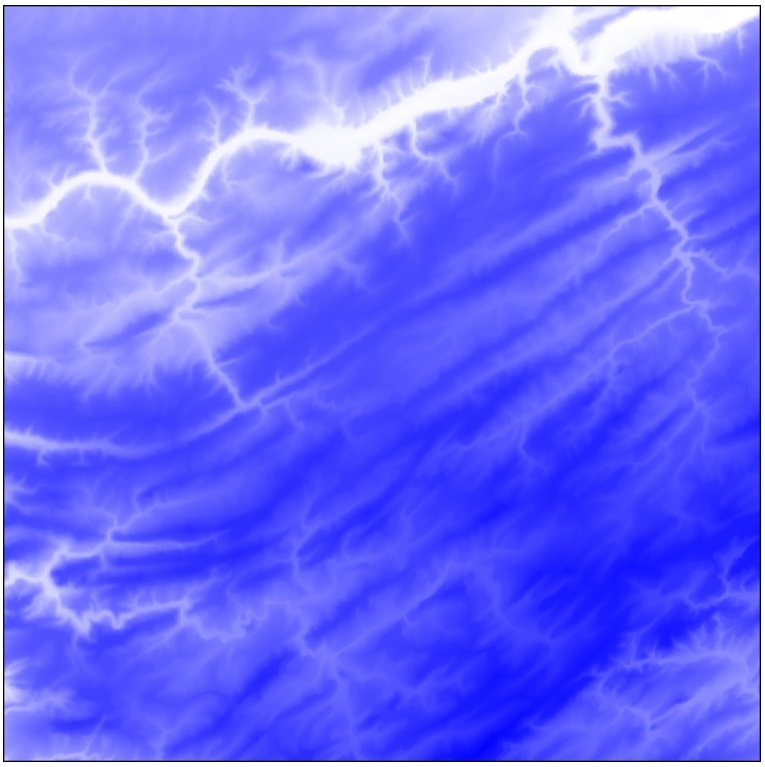


FIG. 2.5 – Modèle numérique de terrain (zone rurale)

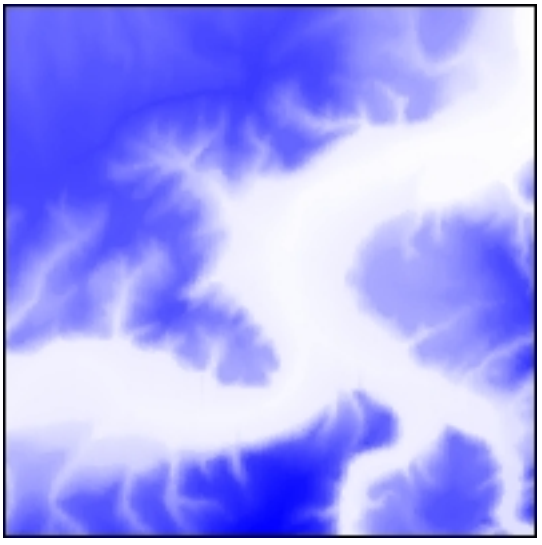


FIG. 2.6 – Modèle numérique de terrain (zone urbaine)

Classiquement, on nommera M le nombre de lignes des matrices et N leur nombre de colonnes. L'origine, l'orientation des deux axes orthonormés et la graduation discrète de ceux-ci sont choisies comme suit :

	0	1	...	$N-1$
0	$\ddots$			
1		$\ddots$		
$\vdots$			$\ddots$	
$M-1$				$\ddots$

FIG. 2.7 – Origine et axes

Deux conventions seront utilisées pour référencer un élément matriciel :

- étiqueter i et j les coordonnées verticale et horizontale (représentation algorithmique classique) ;
- étiqueter y et x les coordonnées verticale et horizontale (représentation classique en visualisation ou en traitement d'images).

2.3 Position du problème

Soit B - la matrice indiquant la présence de barrières - un tableau de booléens de M lignes et N colonnes, avec $M \in \mathbb{N}_0$, $N \in \mathbb{N}_0$. On a "$\underline{A}\ i, j : 0 \leq i < M, 0 \leq j < N$: $B[i][j] = false$ **signifie que l'élément** $B[i][j]$ **est *accessible* et** $B[i][j] = true$ **signifie que l'élément n'est pas *accessible***".

Une **barrière** (ou **obstacle**) est un groupement connexe de plusieurs éléments qui ne sont pas accessibles (cf. figure 2.8). Plus précisément, une barrière est définie comme étant une **région connexe d'éléments de** B **de valeur** $true$ **et jointifs en 8-connexité**.

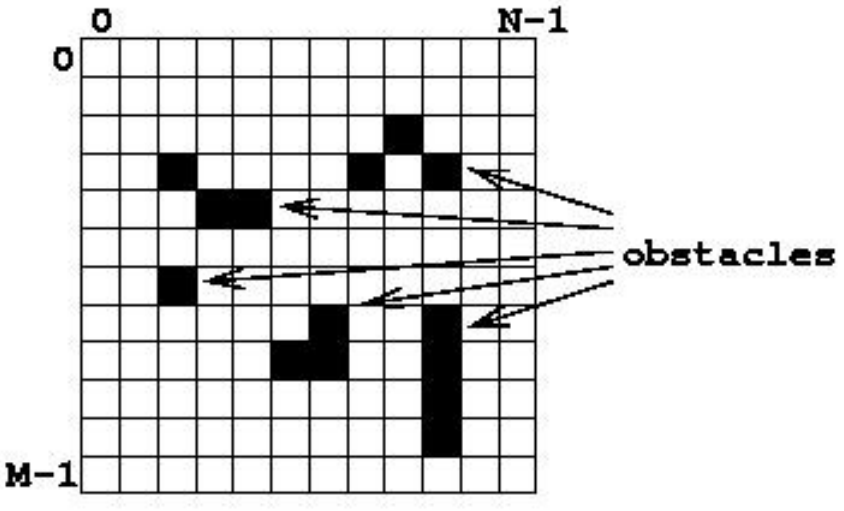


FIG. 2.8 – Obstacles

2.3.1 Distance

On définit la ***distance*** d entre deux éléments de B, p et q, de coordonnées (i, j) et (k, l) comme[2]

$$d((i,j),(k,l)) = \sqrt{(i-k)^2 + (j-l)^2}$$

Il s'agit bien de la distance euclidienne.

Il est important de remarquer que la distance entre toute paire d'éléments **est mesurée de centre de pixel à centre de pixel** (cf. figure 2.9). La zone réelle (carrée) couverte par un pixel est donc représentée par son centre géométrique lorsque l'on souhaite mesurer la distance la séparant d'une autre zone.

2.3.2 Rayon d'accessibilité et voisinage

On introduit une constante R telle que $R \in \mathbb{N}_0$. On appelle cette quantité ***rayon d'accessibilité*** ou, plus simplement, *rayon*.

On définit le ***voisinage*** V d'un élément $B[i][j]$ comme l'ensemble des éléments inclus dans un disque centré en $B[i][j]$ et de rayon R, y compris l'élément $B[i][j]$.

[2]La notation $d(p, q)$ portant sur les éléments plutôt que sur les indices de ceux-ci sera considérée comme équivalente.

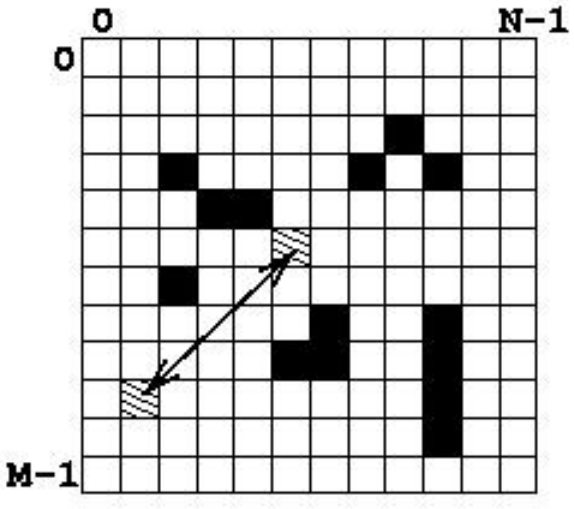


FIG. 2.9 – Notion de distance (d = 5.66)

Autrement dit, le voisinage de l'élément de coordonnées (i, j) est :

$$V(i,j) = \{(x,y) \in \mathbb{N} \times \mathbf{N} \mid (0 \leq d((x,y),(i,j)) \leq R) \; and \; (x < M) \; and \; (y < N)\}$$

La taille N_V d'un voisinage est le nombre (nécessairement non nul) d'éléments du voisinage. La figure 2.10 représente un voisinage de taille $N_V = 13$.

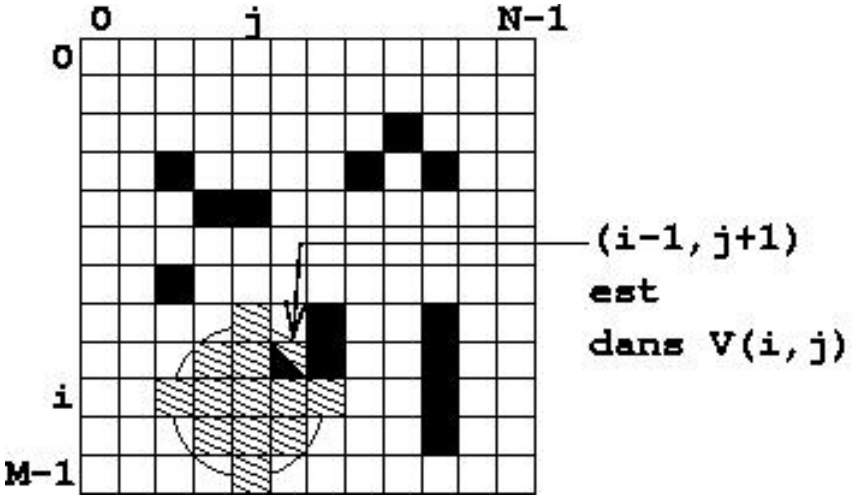


FIG. 2.10 – Notion de voisinage ($R = 2$) : V(i,j) est hachuré et $N_V = 13$

Un rayon R est associé à chaque ensemble de matrices de données : **la valeur de l'indice d'accessibilité d'un élément n'est influencée que par les éléments**

se trouvant dans son voisinage.

Dans le contexte de ce document, le rayon R sera généralement petit ($R \sim 30$). De ce fait, la taille de tout voisinage sera petite par rapport au nombre d'éléments de la matrice. Le développement d'algorithmes performants pour toute taille de voisinage n'est donc pas un objectif prioritaire de ce travail. Nous prendrons plutôt en considération et montrerons comment tirer parti du fait que la taille d'un voisinage est petite par rapport à la taille de la matrice.

2.3.3 Chemin géodésique discret

Un ***chemin géodésique discret*** C, d'un élément accessible appelé *source* vers un autre élément accessible appelé *cible*, est défini comme étant une suite ordonnée, finie et non redondante[3] d'éléments accessibles de coordonnées discrètes

- dont le premier élément est la source,
- dont les éléments intermédiaires marquent les changements de direction éventuels dus à la présence d'obstacles,
- dont le dernier élément est la cible (ou destination),
- pour laquelle, pour toute paire d'éléments consécutifs, chaque segment de droite déterminé par deux éléments consécutifs n'intersecte pas un obstacle mais est localement optimal (c'est-à-dire effectuant un contournement aussi serré que possible de l'obstacle rencontré).

On note N_C la taille de ce chemin, c'est-à-dire le nombre d'éléments constituant ce chemin (ces derniers étant notés $C_0, C_1, \ldots, C_{N_C-1}$).

On peut voir à la figure 2.11 les éléments constitutifs de deux chemins, un premier d'orientation verticale ainsi qu'un second d'orientation horizontale, respectivement de taille 3 et 2. On remarque que le chemin d'orientation verticale n'est pas direct car le segment de droite reliant ses deux extrémités intersecte un obstacle. Le contournement de celui-ci impose donc un détour par un élément intermédiaire.

La figure 2.11 représente un seul chemin pour chacune des deux paires d'éléments mises en évidence. Il est toutefois évident qu'il existe un très grand nombre de chemins entre toute paire d'éléments donnés.

---

[3]Cela signifie que tout élément de la matrice B apparaît au plus une seule fois dans chaque chemin auquel il est intégré.

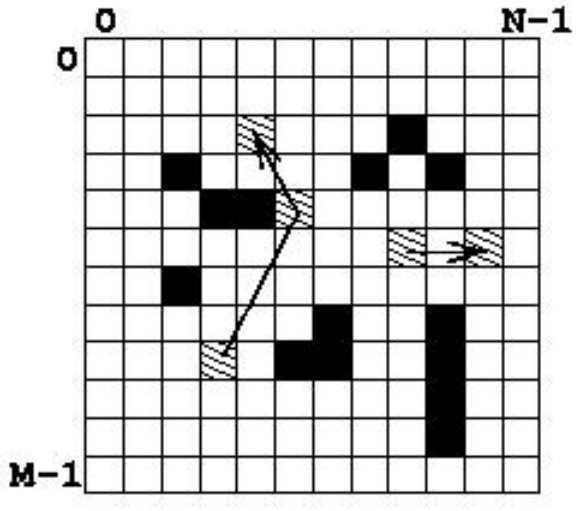


FIG. 2.11 – Eléments constitutifs de deux chemins

On définit la *longueur L_C d'un chemin C* comme étant la somme[4] des distances entre toute paire d'éléments consécutifs :

$$L_C = \sum_{i=0}^{N_C-2} d(C_i, C_{i+1})$$

Le **chemin *optimal* (le chemin géodésique discret le plus court)** entre deux éléments est celui de longueur minimale parmi tous les chemins les reliant.

La figure 2.11 représente, pour chacune des deux paires d'éléments mises en évidence, le chemin optimal les reliant.

Notons que **le plus court chemin géodésique discret entre deux éléments n'est pas nécessairement unique** : il peut y avoir plusieurs chemins optimaux équivalents, c'est-à-dire de même longueur. Si on doit choisir un chemin parmi plusieurs chemins équivalents, on privilégiera toujours celui de plus petite taille, c'est-à-dire celui comportant le moins d'éléments et consommant donc le moins d'espace mémoire.

Notons de plus que le plus court chemin géodésique discret entre deux éléments peut être de longueur nulle : cela se produit lorsque les éléments source et cible sont confondus.

[4]Dans le cas particulier où tous les éléments de C sont géométriquement alignés, $L_C = d(C_0, C_{N_C-1})$.

On notera $C^*((i,j),(k,l))$ le chemin optimal (géodésique discret le plus court) entre (i,j) et (k,l) et $L_{C^*((i,j),(k,l))}$ sa longueur.

2.3.4 Ensemble des destinations possibles

L'***ensemble D des destinations possibles*** d'un élément accessible $B[i][j]$ est l'ensemble des éléments accessibles parmi ceux constituant le voisinage de cet élément et pour lesquels on peut trouver un chemin de longueur inférieure ou égale à R vers cet élément $B[i][j]$.

Il vient dès lors que :

$$D(i,j) = \{(x,y) \in V(i,j) \mid L_{C^*((i,j),(x,y))} \leq R\}$$

On déduit immédiatement que : $D(i,j) \subseteq V(i,j)$. On note N_D la taille de cet ensemble D et donc $N_D \leq N_V$ pour un élément donné.

En effet, un élément du voisinage doit être rejeté de l'ensemble des destinations :

- soit parce qu'il n'est pas accessible (figure 2.12 (a)),
- soit parce que, bien que licite du point de vue de son accessibilité, il est trop éloigné de l'élément source (figure 2.12 (b)), lorsque $L_{C^*} > R$.

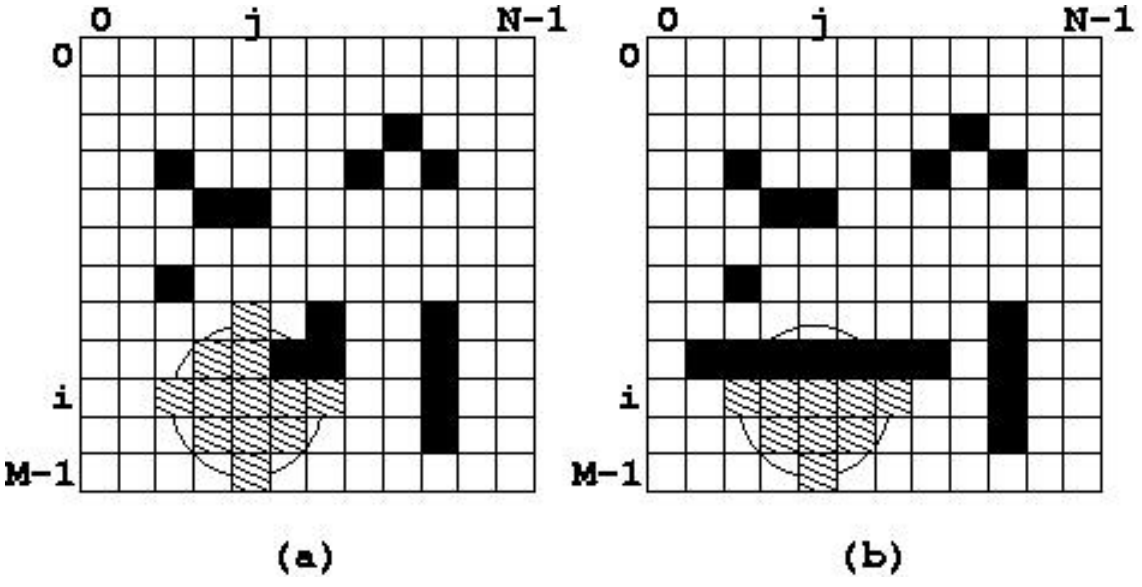


FIG. 2.12 – Notion de destinations ($R = 2$) : D(i,j) est hachuré (a) $N_V = 13$ et $N_D = 12$ (b) $N_V = 13$ et $N_D = 9$

Malgré ces restrictions, $N_D = O(R^2)$ (c'est-à-dire $O(N_V)$, la taille du voisinage).

2.4 Définition du problème

Le problème qui est posé est de calculer une matrice stockant les valeurs d'accessibilité relatives aux données (les 3 matrices stockant les couches de pixels précédemment décrites). On distinguera deux variantes du problème : sans considération des barrières (autrement dit, on ne tient pas compte de la matrice de données B) et avec considération des barrières.

2.4.1 Problème DLOSP

Commençons par définir le problème **sans prendre en considération les barrières** et nommons-le ***problème*** `DLOSP` (*Discrete Line-Of-Sight Paths*).

Ici, tout chemin entre deux éléments donnés est réduit au segment de droite reliant ceux-ci (soit le chemin le plus direct, en ligne de visée).

Résoudre `DLOSP`, pour un rayon R donné, consiste à calculer les coefficients de la matrice α des indices d'accessibilité selon l'équation suivante, proposée par le LEPUR [LEP02] :

$$\alpha[i][j] = \sum_{k=0}^{N_V-1} (FACT_1 \times FACT_2 \times FACT_3)$$

L'indice d'accessibilité d'un élément source est la somme des termes d'accessibilité (chacun résultant du produit de 3 facteurs) de chaque élément de son voisinage, défini pour un rayon d'accessibilité R donné pour l'instance du problème `DLOSP` à résoudre.

On introduit les notations suivantes :

- les coordonnées des éléments du voisinage $V(i,j)$ sont notées $\gamma_0(i,j), \gamma_1(i,j), \ldots, \gamma_{N_V-1}(i,j)$ et forment un ensemble noté $\Gamma(i,j)$;
- les longueurs des chemins les plus courts correspondant aux destinations $\gamma_k(i,j)$ sont notées $d_0((i,j),\gamma_0(i,j)), d_1((i,j),\gamma_1(i,j)), \ldots, d_{N_V-1}((i,j),\gamma_{N_V-1}(i,j))$.

Le premier facteur, $FACT_1$, modélise l'influence de la densité de population moyenne pour le $k-$ème élément du voisinage :

$$FACT_1 = Pop[\gamma_k(i,j)]$$

Signification : si la population est nombreuse dans la zone recouverte par l'élément cible, l'accessibilité est grande.

Le deuxième facteur, $FACT_2$, modélise l'influence de la distance entre l'élément source et le $k-$ème élément du voisinage, en-deça du seuil R :

$$FACT_2 = \frac{2 \times R}{R + \min(R, d_k((i,j), \gamma_k(i,j)))} - 1$$

ou encore

$$FACT_2 = \frac{R - \min(R, d_k((i,j), \gamma_k(i,j)))}{R + \min(R, d_k((i,j), \gamma_k(i,j)))}$$

Signification : si la distance entre source et cible est grande, l'accessibilité est faible. L'influence d'un élément destination sur l'accessibilité d'un élément source décroît avec la distance et devient nulle au-delà du rayon d'accessibilité R. Contrairement au facteur de population, l'influence du facteur de distance n'est pas considérée comme linéaire :

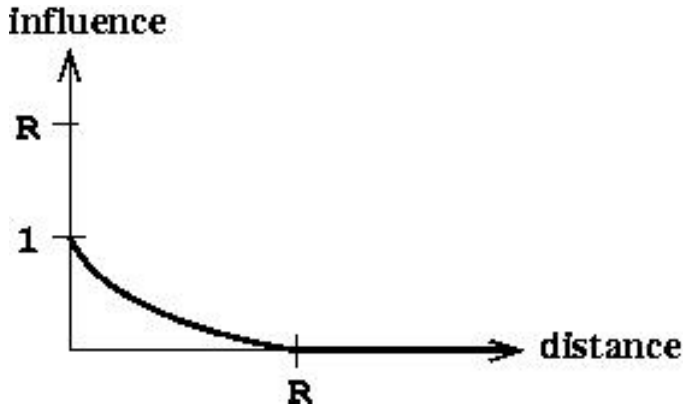


FIG. 2.13 – Influence de la distance dans l'indice d'accessibilité

En pratique, les chercheurs du LEPUR utilisent souvent un rayon d'accessibilité R de 1600 mètres, soit 32 pixels étant donné la résolution de 50 mètres par pixel.

Le troisième facteur, $FACT_3$, modélise l'influence de la différence d'élévation moyenne entre l'élément source et le $k-$ème élément du voisinage, en-deça du seuil $\bar{Z}$:

$$FACT_3 = \frac{\bar{Z} - \min(\bar{Z}, |MNT[i][j] - MNT[\gamma_k(i,j)]|)}{\bar{Z}}$$

Signification : si la différence d'élévation moyenne (en valeur absolue) entre source et cible est grande, l'accessibilité est faible. L'influence d'un élément destination sur l'accessibilité d'un élément source décroît avec la différence d'élévation et devient nulle au-delà du seuil d'élévation $\bar{Z}$. L'influence du facteur d'élévation moyenne est considérée comme linéaire tant que le seuil d'élévation n'est pas atteint :

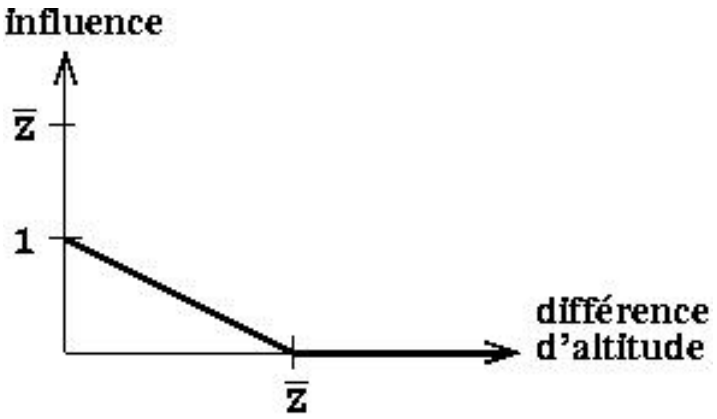


FIG. 2.14 – Influence de l'élévation moyenne dans l'indice d'accessibilité

En pratique, les chercheurs du LEPUR utilisent un seuil d'élévation $\bar{Z}$ de 80 mètres.

En composant l'équation à partir des facteurs décrits, on obtient :

$$\alpha[i][j] =$$

$$\sum_{k=0}^{N_V-1} (Pop[\gamma_k(i,j)] \times (\frac{2 \times R}{R + \min(R, d_k(i,j))} - 1) \times (\frac{\bar{Z} - \min(\bar{Z}, |MNT[i][j] - MNT[\gamma_k(i,j)]|)}{\bar{Z}}))$$

à calculer pour chaque élément (i, j).

2.4.2 Problème DGSP

Comment **prendre en considération les barrières** et définir le ***problème*** `DGSP` (*Discrete Geodesic Shortest Paths*) ?

Par définition du voisinage D d'un élément source, on associe un chemin issu de l'élément source (i, j) à chaque élément de l'ensemble $D(i, j)$ des destinations possibles de celui-ci. En tenant compte de la matrice B localisant les barrières, on sélectionnera chaque chemin le plus court parmi tous le très grand nombre de chemins envisageables entre l'élément source et l'élément destination déterminant l'association.

Le contexte implique la présence de nombreux obstacles, ce qui augmente la complexité du problème de calcul des chemins les plus courts : de nombreux contournements de barrières seront inévitables.

Résoudre `DGSP`, pour un rayon R donné, consiste à calculer les coefficients de la matrice α des indices d'accessibilité selon l'équation suivante :

$$\alpha[i][j] = \sum_{k=0}^{N_D-1} (FACT_1 \times FACT_2 \times FACT_3)$$

L'indice d'accessibilité d'un élément source est la somme des termes d'accessibilité (chacun résultant du produit de 3 facteurs) de chaque élément de son ensemble de destinations, défini pour un rayon d'accessibilité R donné pour l'instance du problème `DGSP` à résoudre.

Deux modifications ont été introduites par rapport à l'équation proposée pour résoudre `DLOSP` :

- les termes d'accessibilité considérés concernent uniquement les éléments de l'ensemble des destinations de l'élément source (pour rappel, $D(i, j) \subseteq V(i, j)$),
- le deuxième facteur est évidemment différent puisque la prise en considération des barrières modifie la notion de distance.

On introduit les notations suivantes :

- les coordonnées des éléments de l'ensemble $D(i, j)$ des destinations des chemins les plus courts issus de (i, j) sont notées $\gamma_0(i, j), \gamma_1(i, j), \ldots, \gamma_{N_D-1}(i, j)$ et forment un ensemble noté $\Gamma(i, j)$;
- les longueurs des chemins les plus courts correspondant aux destinations $\gamma_k(i, j)$ sont notées $\delta_0((i, j), \gamma_0(i, j)), \delta_1((i, j), \gamma_1(i, j)), \ldots, \delta_{N_D-1}((i, j), \gamma_{N_D-1}(i, j))$.

Ainsi, le facteur $FACT_2$ devient :

$$FACT_2 = \frac{2 \times R}{R + \min(R, \delta_k(i,j))} - 1$$

En composant l'équation à partir des facteurs $FACT_1$ et $FACT_3$ décrits pour `DLOSP` et du facteur $FACT_2$ décrit pour `DGSP`, on obtient :

$$\alpha[i][j] =$$

$$\sum_{k=0}^{N_D-1} (Pop[\gamma_k(i,j)] \times (\frac{2 \times R}{R + \min(R, \delta_k(i,j))} - 1) \times (\frac{\bar{Z} - \min(\bar{Z}, |MNT[i][j] - MNT[\gamma_k(i,j)]|)}{\bar{Z}}))$$

à calculer pour chaque élément (i, j).

Etant donné le contournement d'une ou plusieurs barrière(s), la longueur $\delta_k(i, j)$ du chemin le plus court entre l'élément source et un élément accessible de son voisinage peut être plus grande que R. Par définition de D, cet élément accessible du voisinage ne devra pas être considéré comme une destination possible et est donc exclu de l'ensemble D des destinations possibles : la raison est que l'influence de cet élément est considérée comme négligeable puisque la distance *réellement* parcourue est plus grande que R.

2.5 Influence de l'élévation moyenne

Une remarque au sujet de l'influence de l'élévation moyenne mérite d'être soulevée. Etant donné que la troisième dimension spatiale n'est pas prise en considération, on intègre un facteur dans le calcul de l'indice d'accessibilité dont le but est d'exclure les chemins présentant une trop grande dénivellation.

Toutefois, une situation telle que présentée sur la figure 2.15 échappe au contrôle de ce facteur. En effet, la différence d'altitude observée est inférieure au seuil $\bar{Z}$ défini, mais la plus grande différence d'altitude existante le long du chemin lui est supérieure.

Une solution pour éviter cette situation est de prendre un ou plusieurs points de contrôle entre la source et la destination et vérifier que les différences d'altitude intermédiaires sont toutes inférieures au seuil $\bar{Z}$.

Combien de points de contrôle faut-il placer et selon quelle répartition entre source et destination est une question ouverte. Vu que les données à notre disposition couvrent

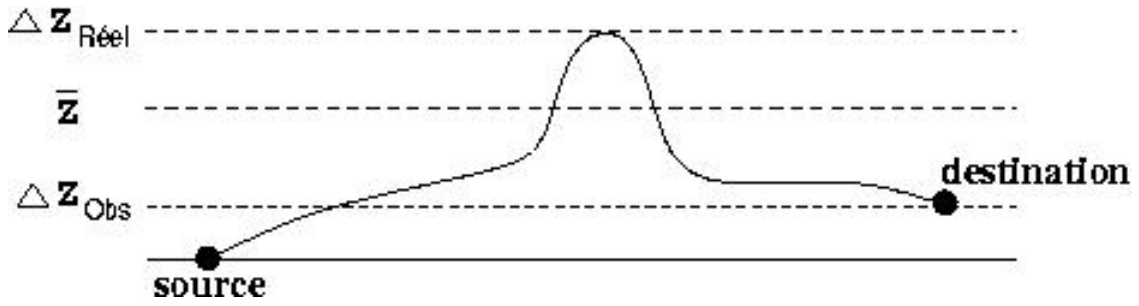


FIG. 2.15 – Situation de forte dénivellation non contrôlée (vue en coupe)

seulement deux zones, il est difficile de poursuivre la modélisation et d'évaluer l'impact des choix du nombre et de la répartition des points de contrôle.

Si de nombreux points de contrôle sont placés, on se retrouve presque dans une situation où la troisième dimension est prise en compte. Il y a clairement dans cette optique un équilibre à établir et une réflexion à mener.

2.6 Conclusion

Les données, le contexte de traitement, les définitions et les problèmes posés (`DLOSP`, `DGSP`) constituent le modèle de calcul de l'indice d'accessibilité.

De nombreux chemins les plus courts devront être calculés. Il sera nécessaire d'identifier des patterns de calcul répétitifs lors de la construction de chemins les plus courts, de manière à exploiter au mieux l'itération.

Par ailleurs, on peut effectuer cette observation fondamentale dans le cadre de `DGSP` : **la valeur de l'indice calculée pour un élément ne peut être déduite de la valeur de l'indice calculée pour l'un de ses voisins**. En effet, le fait que la notion de distance soit euclidienne rend non local le calcul d'un chemin optimal : il faut considérer bien plus qu'un petit voisinage de l'élément source du calcul courant de chemin optimal si l'on veut exploiter les résultats du calcul précédent de chemin optimal.

On peut dès lors déduire ces deux conséquences :

- **Le calcul des valeurs de l'indice d'accessibilité pourra se faire a priori dans un ordre quelconque :** cela permettra une distribution massive des calculs puisque la matrice de l'indice d'accessibilité peut être construite à partir de la concaténation de sous-matrices calculées sur des processeurs différents.
- **Les techniques d'exploitation de l'itération devront porter sur des questions globales.** Par exemple, on pourra envisager l'introduction de structures de données répondant de manière performante à cette interrogation : y a-t-il

des obstacles *proches* d'un élément donné ?

Pour conclure, remarquons que l'on peut distinguer **4 niveaux de calcul** en considérant qu'une distribution des calculs peut être réalisée :

1. Distribution du calcul des sous-matrices de la matrice de l'indice d'accessibilité sur différents noeuds de calcul.
2. Calcul de la valeur de l'indice d'accessibilité pour chaque élément d'une sous-matrice.
3. Calcul de la valeur de l'indice d'accessibilité pour un élément source donné (c'est-à-dire appliquer `DLOSP` ou `DGSP`).
4. Calcul de l'influence d'un élément de l'ensemble des destinations d'un élément source donné sur la valeur de l'indice d'accessibilité pout cet élément source donné.

Les figures 2.16 et 2.17 représentent les indices d'accessibilité calculés pour les deux zones de test.

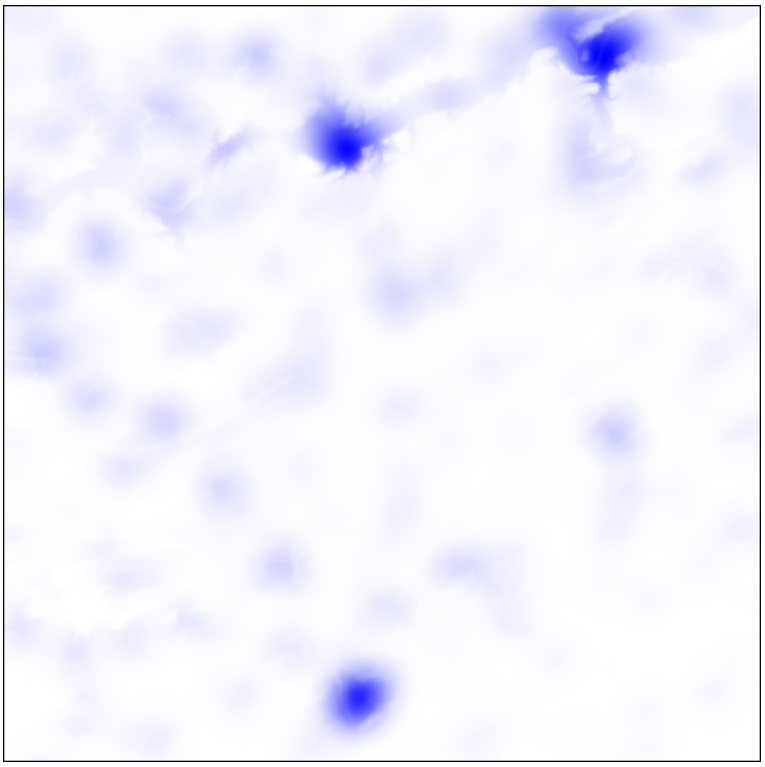


FIG. 2.16 – Indice d'accessibilité (zone rurale)

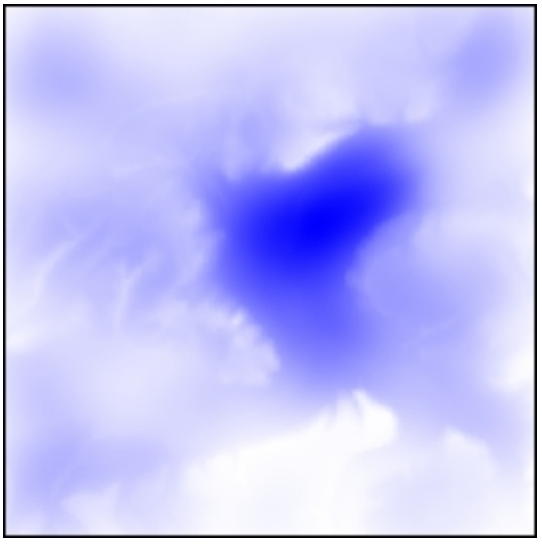


FIG. 2.17 – Indice d'accessibilité (zone urbaine)

Chapitre 3

DLOSP

3.1 Un algorithme simple de calcul d'indice d'accessibilité

Comment résoudre le problème `DLOSP` ? Nous allons proposer un algorithme simple pour répondre à cette interrogation de manière performante puis nous montrerons comment réduire le nombre de calculs réalisés.

Tous les concepts présentés dans ce chapitre, à l'exception de celui de plus court chemin, constitueront une bonne base pour résoudre le problème `DGSP`.

3.1.1 Parcours général

La matrice α contenant l'indice d'accesssibilité calculé peut être remplie en suivant un parcours choisi arbitrairement (classiquement, ligne par ligne, colonne par colonne). Pour chaque élément de α, il faut calculer `DLOSP`.

Comme souligné précédemment, le calcul de `DLOSP` pour un élément donné ne peut totalement se déduire, même indirectement, du calcul de `DLOSP` pour un élément proche.

Les données requises par le calcul de `DLOSP` pour un élément donné sont comprises dans une zone limitée par le voisinage de cet élément. Un découpage de la matrice que l'on veut calculer est donc envisageable : il suit qu'un processus de distribution des calculs pourra être opéré.

3.1.2 Calcul de l'indice d'accessibilité pour des coordonnées fixées

La première question à envisager est : quel est le voisinage V de l'élément donné (dont les coordonnées sont fixées) ?

Fournissons deux éléments de réponse (figure 3.1) :

- dans le cas général, il s'agit des éléments délimités par le cercle dont le rayon est le rayon d'accessibilité R ;
- lorsque les coordonnées fixées sont proches des bords de la matrice α (la distance qui les sépare est inférieure à R), il faut s'interroger sur la manière de traiter les éléments situés (virtuellement ...) hors des limites de la matrice.

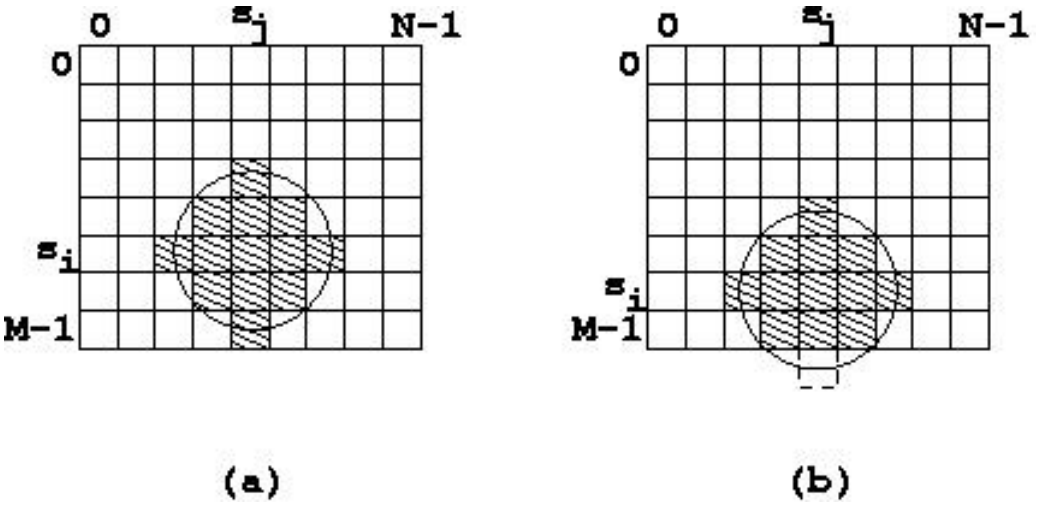


FIG. 3.1 – Voisinage (a) cas général (b) cas limite

Une fois le voisinage déterminé, il convient de calculer puis sommer tous les termes d'accessibilité, c'est-à-dire prendre en compte tous les éléments du voisinage au travers des valeurs associées des matrices Pop (densité de population) et MNT (modèle numérique de terrain).

3.2 Détermination du voisinage d'un élément

Dans le cas général, le voisinage d'un élément donné est l'ensemble des éléments éloignés de celui-ci d'une distance inférieure ou égale au rayon d'accessibilité R.

Remarquons que si l'on considère un élément - résultant du processus initial de discrétisation - comme étant un objet géométrique présentant en réalité une surface plus que ponctuelle, il faut sélectionner des coordonnées de référence de cet élément

pour lui appliquer la fonction de calcul de distance par rapport à un autre élément. Cela ne pose aucun problème puisque, par convention, les coordonnées d'un élément sont les coordonnées du centre de celui-ci.

3.2.1 Information contenue dans la notion de voisinage

Travaillant dans un contexte discret et le rayon d'accessibilité étant fixé, on peut précalculer quels éléments composent un voisinage.

L'information que l'on peut extraire de la notion de voisinage est la suivante (figure 3.2) :

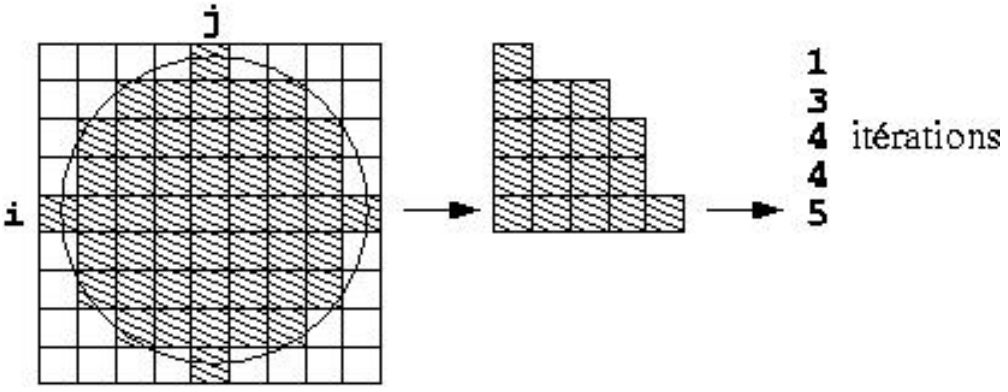


FIG. 3.2 – Résumé de l'information contenue dans la notion de voisinage (exemple pour $R = 4$)

On peut donc précalculer une table (un vecteur de $R + 1$ composantes). La valeur de chaque composante représente le nombre d'itérations à effectuer pour couvrir une partie de voisinage (gauche ou droite) de la ligne correspondante dans la matrice en partant de l'élément central (j-ème colonne).

Le schéma de la figure 3.2 peut être étendu par deux symétries (gauche/droite et haut/bas) pour tout le voisinage.

La consultation de cette table évitera des calculs d'indices un grand nombre de fois.

3.2.2 Voisinage hors limites

Quelle décision prendre lorsque les éléments d'un voisinage sont (virtuellement) situés en dehors des limites de la matrice ?

Cette situation courante dans le domaine du traitement numérique d'images peut être gérée notamment comme suit [Vdb00] :

- **prolongement** de la matrice **par miroirisation** des valeurs des bords existantes ;
- **prolongement** de la matrice **par sélection d'une valeur arbitraire** ou moyenne ;
- **prise en compte uniquement des éléments existants** et donc réduction du nombre total d'éléments considérés par rapport à la valeur nominale attendue.

En accord avec les chercheurs du LEPUR, nous avons sélectionné la dernière option car c'est celle qui minimise le biais introduit si l'on considère l'objectif du modèle développé (rendre compte de l'accessibilité d'un lieu d'après son voisinage).

3.3 Calcul d'un terme d'accessibilité

Le **calcul d'un terme de l'indice d'accessibilité** pour un élément source et un élément destination donnés est très simple :

`computeAccessibilityTerm`

```
computeAccessibilityTerm(s_i : int ; s_j : int ; i : int ; j : int ;
                         var sum : float )

var
  dz : int ;
  dst : float ;

|[ dz := abs(MNT[s_i][s_j]−MNT[i][j]);
  if (dz < ZBAR) →
    dst := distance( s_j , s_i , j , i );
    if (dst < R) →
      sum := sum +
             ( Pop[i][j] ∗
               (((2.0 ∗ R / (R + dst)) − 1.0) ∗
               ((ZBAR−dz) / ZBAR)));
    [] dst = R → skip
    fi
  [] (dz ≥ ZBAR) → skip
  fi
]|
```

Algorithme 3.1 – `computeAccessibilityTerm`

La variable *sum* contiendra, après application du code ci-dessus à tous les éléments (i, j) du voisinage de (s_i, s_j), la réponse au problème `DLOSP` pour (s_i, s_j).

Les calculs réalisés en virgule flottante introduisent des erreurs d'arrondi. Celles-ci ne sont pas gérées car la précision demandée est de l'ordre de 10^{-1} alors que la précision machine du type de données utilisé dans l'implémentation est de 10^{-6}.

De plus, les données utilisées présentent également des biais. Toutefois, la méthode de génération des données actuellement[1] mise en oeuvre ne permet pas de quantifier les biais existants.

Dès lors, nous ne conduirons pas de raisonnement sur la propagation des erreurs.

3.4 Calculer une fois, stocker deux fois

En observant que le calcul du terme d'accessibilité en (s_i, s_j) vers (i, j) diffère peu du calcul d'accessibilité en (i, j) vers (s_i, s_j) (seule la densité de population est différente, la différence d'altitudes et la distance étant égales dans les deux cas), une économie de calcul (a priori de l'ordre de 50 %) peut être réalisée.

En effet, lors du calcul d'un terme d'un indice d'accessibilité, on peut calculer une seule fois la différence d'altitudes ainsi que la distance entre les deux éléments impliqués, puis stocker le résultat (le terme d'accessibilité) deux fois, en ayant tenu compte de la densité de population appropriée les deux fois.

Cela a deux conséquences :

- le **calcul des termes d'accessibilité** formant, lorsqu'on les somme, la valeur d'un indice d'accessibilité, n'est plus effectué en bloc, mais est **fractionné** lors du calcul de nombreux indices d'accessibilité ;
- le **parcours général de la matrice influence désormais la séquence de calcul des termes d'accessibilité** pour un élément donné : il peut toujours être choisi arbitrairement[2], mais les instructions exécutées pour chaque élément parcouru doivent être adaptées en conséquence.

[1] C'est-à-dire début 2003. Le processus de génération des données par le LEPUR ne sera d'ailleurs pas terminé avant la fin de notre travail.

[2] L'implémentation réalisée a conservé un parcours classique ligne par ligne, colonne par colonne.

La figure 3.3 met en évidence graphiquement les calculs à réaliser pour résoudre DLOSP pour (s_i, s_j) :

- les éléments du parcours général déjà traités et hors du voisinage de (s_i, s_j) sont représentés en traits hachurés ;
- les éléments du voisinage de (s_i, s_j) pour lesquels le terme d'accessibilité a déjà été calculé et stocké sont représentés en noir ;
- les éléments du voisinage de (s_i, s_j) pour lesquels le terme d'accessibilité doit être calculé sont représentés par un X.

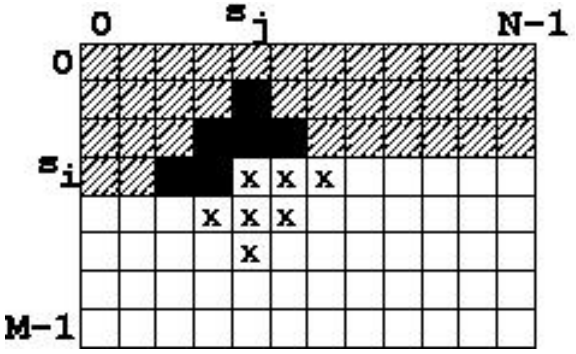


FIG. 3.3 – DLOSP : calculer une fois, stocker deux fois ($R = 2$)

Les temps de calcul pour résoudre DLOSP pour les deux zones de données reflètent un gain de temps de 40%.

Données	Zone urbaine	Zone rurale
DLOSP normal	5.7 s	58 s
DLOSP avec double stockage	3.4 s	34.3 s

FIG. 3.4 – Temps de calcul de DLOSP

3.5 Calcul de distances

Envisageons pour terminer un concept a priori très simple et ne méritant que peu d'attention ...

La notion de distance telle que définie est simplement la distance euclidienne calculée comme suit :

$$d(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

avec $P_1 = (x_1, y_1)$ et $P_2 = (x_2, y_2)$ deux points de coordonnées entières. La distance entre deux éléments est donc, comme déjà mentionné, calculée de centre à centre.

Les aspects relatifs à l'implémentation d'un calcul aussi trivial ne sont toutefois pas à négliger. En effet, le nombre de calculs de distances intervenant dans les algorithmes proposés dans ce document est vraiment très important, comme le montre la relation présentée à la figure 3.5.

Taille de la matrice	Distances calculées (`DLOSP`)
10^2	97 320
3.6×10^3	5 136 900
10^4	14 963 370
3.6×10^5	536 701 188

FIG. 3.5 – Nombre de calculs de distances ($R = 32$)

Dans le problème de base, `DLOSP`, même lorsque le rayon d'accessibilité est petit, un très grand nombre de distances doivent être calculées. Une borne supérieure que l'on peut calculer en première analyse est $\pi \frac{R^2}{2} MN$ lorsqu'on stocke deux fois chaque terme d'accessibilité calculé. Tout gain de performances dans l'algorithme réalisant le calcul d'une distance est donc appréciable.

Initialement, la définition d'une distance a été utilisée de manière directe pour développer l'algorithme correspondant, basé sur une fonction de calcul de racine carrée.

Ensuite, nous avons observé que le cas du calcul d'une distance $\leq 2R$ et celui d'une distance $> 2R$ peuvent être distingués. Lors du calcul de `DLOSP`, les distances restent évidemment bornées par R et, lors du calcul de `DGSP`, dans certains cas, les morceaux de chemins ne seront jamais plus longs que $2R$ (plus grand contournement direct possible dans un cercle de rayon R).

Dès lors, précalculer une table des distances $\leq 2R$ lors de l'initialisation du logiciel peut s'avérer judicieux dans le premier cas, tandis que l'on conserverait l'algorithme issu directement de la définition d'une distance dans le second cas.

En observant que tous les couples de points présentant respectivement les mêmes deltas en x et en y sont éloignés de la même distance, on peut calculer les valeurs des distances recherchées comme étant les valeurs de la fonction à valeurs réelles et à deux variables discrètes Δx, Δy :

$$f(\Delta x, \Delta y) = \sqrt{(\Delta x)^2 + (\Delta y)^2}$$

avec $\Delta x = |x_1 - x_2|$ et $\Delta y = |y_1 - y_2|$ définis sur $[0..2R]$.

Chaque valeur $f(\Delta x, \Delta y)$ correspond aux coordonnées de 4 éléments étant donné que l'on exploite les propriétés de symétrie des éléments équidistants par rapport à l'élément origine.

La taille de la table stockant les valeurs de cette fonction est donc égale à $4R^2$, ce qui est tout à fait raisonnable vu que $O(R^2) \sim O(M)$ et $O(R^2) \sim O(N)$.

Si la consommation d'espace mémoire devait être davantage réduite, on pourrait aller encore plus loin en observant que tous les éléments d'une ligne de la table ne doivent pas être définis (cf. figure 3.6). En stockant uniquement les portions de la matrice contenant de l'information réellement exploitée et en calculant un vecteur de décalage, on diminuerait l'espace de stockage nécessaire.

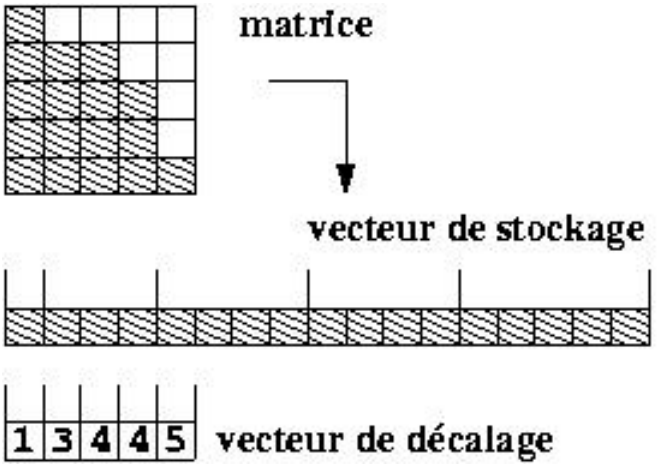


FIG. 3.6 – Stockage minimum de la table des distances

En pratique, la lecture d'une variable en mémoire est évidemment bien plus rapide que l'appel à une routine de calcul de la FPU[3] :

Distances calculées	10^7	10^8	10^9
sqrt (fonction)	1.92 s	19.24 s	191.61 s
sqrt (macro)	1.86 s	19.10 s	185.77 s
table (fonction)	1.48 s	14.76 s	148.46 s
table (macro)	0.02 s	0.24 s	0.34 s

FIG. 3.7 – Temps de calcul d'un grand nombre de distances

[3]Tant que l'algorithme de calcul de racine carrée implémenté dans une FPU ne fait pas lui-même appel à une telle table et que le compilateur utilisé n'injecte pas directement dans le code assembleur généré le code correspondant à la fonction appelée plutôt qu'une instruction de branchement vers celle-ci ...

Il est à noter que les temps mesurés concernent des calculs effectués en rafale.

3.6 Conclusion

Etant donné le travail de modélisation préalablement effectué, résoudre `DLOSP` est immédiat.

Un algorithme simple de calcul d'indice d'accessibilité et plusieurs optimisations ont été proposés.

Chapitre 4

DGSP

4.1 Un algorithme de contournement de barrières

Les perspectives et approches envisagées pour `DLOSP` restent valides pour `DGSP`. Dès lors, seule la notion de chemin le plus court, réduite pour `DLSOP` à un simple calcul de distance, nécessite une recherche approfondie pour développer un algorithme de calcul d'indice d'accessibilité prenant en compte les barrières.

La compréhension intuitive du problème de contournement de barrières est relativement abordable mais beaucoup de développements algorithmiques seront nécessaires pour résoudre le problème de manière satisfaisante.

La recherche d'un chemin le plus court tel que défini par le modèle établi au chapitre 2 peut être envisagée selon plusieurs plans de réflexion.

Une première distinction concerne le degré de vectorisation à appliquer aux données [Don02a]. Va-t-on travailler avec des données

- en mode image (*raster*) : pas de vectorisation des matrices données
- en mode vectoriel

Une deuxième distinction concerne la stratégie d'optimisation. La méthode choisie sera-t-elle

- optimale
- heuristique

Une troisième distinction concerne la quantité de prétraitements effectués. Quels résultats intermédiaires conserver ? Comment reconditionner les données pour permettre des traitements plus performants ?

La stratégie que nous choisissons d'explorer dans le reste de ce document est une **approche totalement vectorielle, permettant beaucoup de prétraitements**

ainsi que l'utilisation d'un algorithme optimal de calcul de plus court chemin dans un graphe.

Le raisonnement qui a conduit au choix de cette option est le suivant.

Tout d'abord, la notion de chemin telle que définie dans le modèle établi - étant donné les attentes des chercheurs du LEPUR - se prête naturellement bien à une vectorisation importante des données, d'autant plus que la prise en compte de la troisième dimension spatiale n'est que partielle.

Ensuite, on peut remarquer qu'un certain nombre de requêtes spatiales (positionnement d'un élément par rapport à un obstacle, énumération des obstacles présents dans le voisinage d'un élément, ...) devront certainement être calculées de manière massive. Un reconditionnement des données permettant des requêtes performantes est certainement souhaitable. De telles techniques sont souvent regroupées dans le domaine des GIS sous l'appellation d'indexation spatiale.

Par ailleurs, un autre point à considérer est le suivant :

- une heuristique sera efficace si la distance entre les points source et destination est courte ($\hookrightarrow$ faible nombre de points dans la zone à traverser, ce qui évitera de trop grosses pénalités temporelles lorsque l'heuristique dégénère) mais est par ailleurs relativement insensible à la quantité d'obstacles à contourner ;
- un algorithme optimal de recherche appliqué à des données vectorisées sera performant indépendamment de la distance mais sera sensible au nombre d'obstacles à contourner.

Toutefois, des algorithmes heuristiques de recherche de chemin, comme par exemple le célèbre A^*, existent et peuvent donner de bons résultats en pratique. L'étude très complète d'algorithmes de propagation menée dans [Cui99] mentionne même, sous certaines conditions, des algorithmes optimaux (en fonction du nombre de points de la zone de propagation).

De plus, si la résolution des données était telle que l'on puisse distinguer clairement le détail d'artères urbaines, une stratégie multirésolutions pourrait utiliser à la place des méthodes que nous décrirons dans ce document des méthodes heuristiques lorsque des zones urbaines exhibant une géométrie compliquée devront être traitées. Cette approche présente un intérêt évident mais sa portée dépasse le cadre de ce document.

Enfin, il sera montré comment reconditionner (simplifier) l'espace de recherche de manière à pouvoir utiliser des algorithmes opérant sur des données vectorielles. En particulier, il sera montré comment un graphe peut abstraire les zones à traiter, permettant l'utilisation d'un algorithme de recherche optimal.

Présentons maintenant une **décomposition du processus de calcul** en ses opérations essentielles :

- **calculer** `DGSP` pour chaque élément de la matrice d'accessibilité
- **détecter les barrières**
- **indexer les barrières**
- **reconditionner l'espace de recherche**
- **calculer les chemins les plus courts contournant les barrières**

La première opération peut être réalisée en suivant simplement le schéma introduit pour `DLOSP`. Les deux opérations suivantes seront étudiées dans le reste de ce chapitre. Les deux dernières opérations feront l'objet du chapitre suivant.

4.2 Détection des obstacles et collecte des régions résultantes

La première étape à réaliser est de détecter les obstacles et collecter les régions résultantes à partir de la matrice des barrières B.

On appellera région la structure de données contenant un obstacle ainsi que le résultat de plusieurs traitements relatifs à celui-ci. Dans la suite du document, on emploiera indistinctement les termes obstacle et région.

4.2.1 Détection de régions : `Sweep and Merge`

Comment détecter les obstacles ? **En première analyse, nous avons pensé effectuer un parcours classique** ligne par ligne, colonne par colonne, de B et intégrer les éléments non accessibles dans la structure de données de la région correspondante. Nous appellerons cette méthode *Sweep and Merge*.

Une telle opération a un coût linéaire en fonction du nombre d'éléments dans la matrice, ce qui est optimal. **Cependant, cette approche pose le problème de la mise à jour dynamique des structures de données stockant les obstacles :** outre un coût d'insertion d'un élément dans une structure, il faut tenir compte des opérations de *fusion* de régions et de gestion de mémoire dynamique.

La figure 4.1 présente en effet la situation où la découverte d'un nouvel élément non accessible oblige à fusionner les deux régions précédemment découvertes en une région unique.

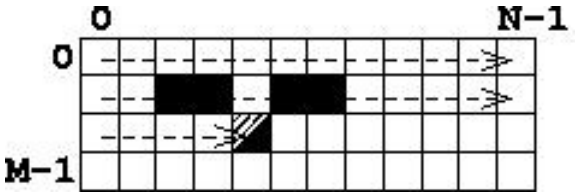


FIG. 4.1 – Fusion de deux régions

Guidons notre raisonnement à l'aide d'un invariant :

$$\mathcal{D} = \bigcup_k \mathcal{R}_k \mid \{ \mathcal{R}_k \text{ est complètement détectée } \}$$

$$\mathcal{O} = \bigcup_k \mathcal{R}_k \mid \{ \mathcal{R}_k \text{ est ouverte} \}$$

L'ensemble $\mathcal{D}$ contient toutes les régions $\mathcal{R}_k$ complètement détectées pour la zone déjà traitée (cf. fig. 4.1 pour le sens de parcours). Aucune région de $\mathcal{D}$ ne peut se trouver dans la zone frontière (c'est-à-dire les éléments le plus récemment traités) illustrée en noir sur la figure 4.2. En effet, il ne peut être garanti qu'une région dont un élément est dans la zone frontière n'est plus ouverte.

L'ensemble $\mathcal{O}$ contient toutes les régions $\mathcal{R}_k$ détectées dans la zone traitée dont au moins un élément est localisé dans la zone frontière. Au fur et à mesure du parcours, les régions de O dont plus aucun élément n'est dans la zone frontière sont transférées vers $\mathcal{D}$.

La figure 4.2 met en évidence les zones couvertes par $\mathcal{O}$ et $\mathcal{D}$.

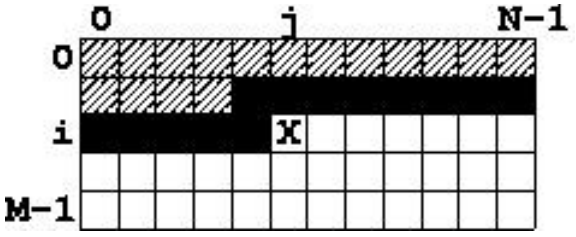


FIG. 4.2 – Zones couvertes par $\mathcal{D}$ (zone hachurée) et par $\mathcal{O}$ (zone pleine), élément courant (X)

En fin d'exécution de l'algorithme, la taille finale de $\mathcal{O}$ est le nombre de régions présentes dans B, c'est-à-dire le nombre de régions à détecter. L'ensemble O est donc une structure de données dont la taille est influencée par la taille des résultats obtenus (structure de données *output-sensitive*).

Par contre, la taille de $\mathcal{D}$ est à tout moment bornée par N, le nombre de colonnes de la matrice.

L'implémentation de $\mathcal{D}$ peut être quelconque (tout du moins en ce qui concerne l'étape de détection des régions). Celle de $\mathcal{O}$, quant à elle, doit permettre de répondre rapidement à la question suivante : quelles sont les régions ouvertes voisines de l'élément courant du parcours ?

De plus, si des relations de voisinage sont découvertes, il convient de fusionner les régions ouvertes voisines du point courant si celui-ci n'est pas accessible. Il faut aussi transférer de $\mathcal{O}$ vers $\mathcal{D}$ l'éventuelle région s'étendant en $(i-1, j-1)$ si le point courant est accessible et que cette région ne s'étend par ailleurs pas à un autre élément de la zone de $\mathcal{O}$.

Les concepts présentés sont valides à condition d'être attentif près des bords de la matrice (références à $i-1$, $j-1$, ...).

La complexité dans le pire cas de l'algorithme est $O((4+U)(MN))$ où U représente la complexité dans le pire cas de l'opération de fusion de régions ouvertes. Le facteur 4 met en évidence le fait que 4 éléments voisins de l'élément courant doivent être consultés lors de chaque itération.

En employant une structure de données adéquate pour la construction des régions ouvertes (par exemple, pour chaque région ouverte, chaînage de tous ses éléments + comptage du nombre et référencement des éléments dans la zone frontière), la complexité dans le pire cas de U peut certainement rester linéaire, voire constante.

Le point délicat de cet algorithme est donc la mise au point d'une structure de données dynamique permettant d'effectuer de manière efficace l'opération de fusion de régions ouvertes.

4.2.2 Détection de régions : `Mark and Explore`

Une autre voie peut être envisagée pour effectuer l'opération de détection de régions.

Envisageons une autre méthode de coût linéaire : dès qu'un élément non accessible est détecté, on effectue la détection de toute la région à laquelle il appartient avant de poursuivre le parcours de la matrice. Nous appellerons cette méthode *Mark and Explore*.

On marque tous les éléments constitutifs d'une région au fur et à mesure qu'ils sont détectés. Marquer un élément revient, d'une part, à écrire dans une matrice auxiliaire (de mêmes dimensions que la matrice de données) un numéro d'identification de la région à laquelle il appartient et, d'autre part, à l'insérer dans la file d'attente (structure de données auxiliaire également).

Explorer est un processus de nature récursive : il s'agit de retirer un élément de la file d'attente, puis d'inspecter et traiter chacun de ses voisins en 8-connexité. Pour chaque voisin, on vérifie s'il est non accessible (il fait donc partie d'un obstacle) et s'il n'a pas encore été marqué : si ces deux conditions sont remplies, il est alors marqué.

Quand lancer l'exploration ? Lorsque, lors du parcours général, l'élément courant n'a pas encore été marqué. Lancer le processus d'exploration permet alors de détecter, de proche en proche, toute la région à laquelle appartient l'élément courant.

Quand l'exploration s'arrête-t-elle ? Une fois que la file d'attente est vide, cela signifie que tous les voisins (en 8-connexité) des éléments déjà marqués ne sont pas des obstacles.

La figure 4.3 met en évidence le marquage total de l'obstacle de gauche avant que ne soit détecté l'obstacle de droite.

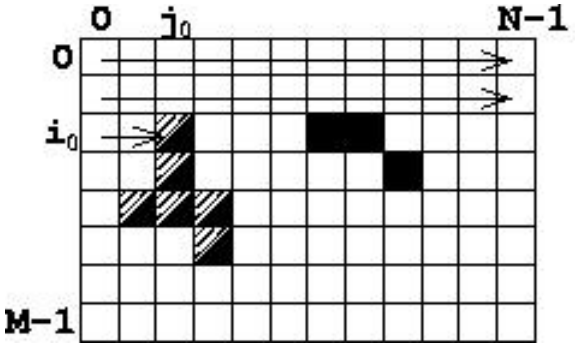


FIG. 4.3 – `Mark and Explore`

Comment être certain qu'on n'effectue pas d'inspections inutiles lors du processus d'exploration ? Etant donné les deux conditions requises pour le marquage d'un élément (non accessible et pas encore marqué), cela veut dire qu'on ne peut insérer dans la file d'attente que des éléments qui font effectivement partie de la région en cours de détection, pas des éléments d'une autre région ni des éléments accessibles.

Comment définir la file d'attente mentionnée ? La politique de retrait de la file n'est pas importante (LIFO, FIFO ou autre ...) en soi. Nous choisissons donc une politique LIFO : la file d'attente est donc une pile. Cela permet de gérer implicitement la récursion plutôt que d'effectuer des appels récursifs.

La taille de la file d'attente est à tout moment bornée par le nombre d'éléments de la matrice, puisque le marquage reste confiné aux éléments existants et non encore marqués.

Quelle est la différence de cet algorithme par rapport au `Sweep and Merge` vu précédemment ? `Sweep and Merge` construit plusieurs régions simultanément, tandis que `Mark and Explore` construit une région à la fois, complètement.

Explicitons notre raisonnement à l'aide d'un invariant :

Notons $\mathcal{R}_k$ une région considérée et complètement détectée. Notons *conn* la relation de connexité entre deux éléments. Elle prend des valeurs booléennes et est vraie si les deux éléments auxquels on l'applique sont connexes.

B est, pour rappel, la matrice de données contenant les barrières. Notons RM la matrice auxiliaire de marquage : pour un élément donné, une valeur nulle signifie une absence de marquage et une valeur $k > 0$ signifie que l'élément de B de mêmes coordonnées est marqué comme appartenant à la région k.

$\mathcal{R}_k(s)$ désigne une partie de la région $\mathcal{R}_k$ en cours de détection et dont s éléments ont déjà été détectés :

$\mathcal{R}_k(s) = \mathcal{R}_k(s-1) \cup (\{ (i,j) \} \mid ((B[i][j] = true) \underline{and} (RM[i][j] > 0) \underline{and} (\underline{E}(k,l) \in \mathcal{R}_k(s-1) \mid conn((i,j),(k,l)))))$ lorsque $s > 1$

$\mathcal{R}_k(1) = \{(i_0, j_0)\} \mid ((B[i_0][j_0] = true) \underline{and} (RM[i_0][j_0] > 0))$

(i_0, j_o) est l'élément à partir duquel on lance le processus d'exploration pour la région k.

Au cours de la détection de la région k, tous les éléments dans la file d'attente $\in \mathcal{R}_k(s)$. De plus, tous les éléments $\in \mathcal{R}_k(s)$ sont dans la file d'attente ou en ont déjà été retirés.

La terminaison de la boucle d'exploration est assurée car on peut retirer à chaque itération un élément de la file d'attente et que tout élément de la matrice n'est inséré qu'au plus une seule fois dans la file d'attente.

Il suit que la complexité temporelle est linéaire en fonction du nombre d'éléments de la matrice. La complexité dans le pire cas est $O(9MN)$ (1 parcours de tous les éléments de la matrice + 1 évaluation de chaque élément du voisinage en 8-connexité extrait de la file d'attente). La complexité attendue varie entre $O(MN)$ et $O(9MN)$. La complexité spatiale est également **linéaire** et vaut, dans le pire cas et dans le cas attendu $O(3MN)$.

`Mark and Explore`

```
type QVertex = record
  begin
  i , j : int ;
  next : ↑QVertex;
  end;

markAndExplore(var qbase : ↑QVertex ; var size : int) : int

var
  qv : ↑QVertex;
  i , j , k , l : int ;

|[ size := 0
 ;do qbase ≠ nil →
      qpop(qbase,qv)
      ; size , i , j := size +1,qv↑.i,qv↑.j

      { Parcours des 8 voisins et marquage si nécessaire }
      ;k := −1
      ;do k <= 1 →
          l := −1
          ;do l <= 1 →
              if ((k ≠ 0) or (l ≠ 0)) →
                 if (((i+k ≥ 0) and (i+k < M) and (j+l ≥ 0) and (j+l < N))
                     cand isUnmarkedRegion(i+k,j+l)) →

                    markRegion(qbase,i+k,j+l)

                 [] not ... → skip
                 fi
              [] ((k = 0) and (l = 0)) → skip
              fi

              ;l := l+1
           od
          ;k := k+1
       od

      ;dispose(qv)
 od ]|
```

Algorithme 4.1 – `Mark and Explore`

Détection des régions

```
var { B, M, N sont accessibles }
  qbase : ↑QVertex; { Pointeur d'accès à la file d'attente }
  rtag : int;
  RM : array[0..M−1][0..N−1] of int;
  i,j : int;

{ Insertion dans la file d'attente repérée par qbase }
qpush(var qbase : ↑QVertex ; qv : ↑QVertex)
|[ qv↑.next := qbase
 ;qbase := qv ]|

{ Retrait de la file d'attente }
{ Précondition : qbase ≠ nil }
qpop(var qbase : ↑QVertex ; var qv : ↑QVertex)
|[ qv := qbase
 ;qbase := qbase↑.next ]|

{ Test de marquage }
isUnmarkedRegion(i : int ; j : int ) : boolean
|[ isUnmarkedRegion := (B[i][j] = true) and (RM[i][j] = 0) ]|

{ Marquage }
markRegion(var qbase : ↑QVertex ; i : int ; j : int)
var
  dummy : ↑QVertex;
|[ RM[i][j] := rtag
 ;new(dummy)
 ;dummy↑.i,dummy↑.j,dummy↑.next:=i,j,nil
 ;qpush(qbase,dummy) ]|

{ Algorithme de détection des régions }
|[ rtag,qbase := 1,nil
 ;“ Initialiser les composantes de RM avec la valeur 0 ”

 ;i := 0
 ;do i < M →
     j := 0
     ;do j < N →
         if isUnmarkedRegion(i,j) →
             markRegion(i,j) { Initialisation de qbase}
             ;size := markAndExplore(qbase)
             ;rtag := rtag+1
         [] not isUnmarkedRegion(i,j) → skip
         fi
         ;j := j+1
      od
     ;i := i+1
  od ]|
```

Algorithme 4.2 – Détection des régions

4.2.3 Collecte de régions

Une fois les régions détectées, l'étape suivante est de stocker une représentation de celles-ci dans une structure de données en permettant une exploitation aisée.

La suite de l'exposé indiquera que les éléments d'une région donnée doivent être stockés sous forme de tableaux, dans des zones mémoire contiguës. Effectuons cette hypothèse et poursuivons.

Pour `Sweep and Merge`, parvenir à un stockage dans des zones contiguës revient à parcourir l'ensemble $\mathcal{D}$ et, pour chaque région, allouer puis remplir la zone de mémoire contiguë correspondante.

Pour `Mark and Explore`, une allocation de toutes les zones de mémoire contiguës - le nombre d'éléments de chaque région étant connu suite à l'exécution de `Mark and Explore` - suivie d'un parcours de la matrice auxiliaire RM contenant les informations de marquage des régions permet d'effectuer l'opération de stockage requise.

En surface, il est peu évident d'effectuer une distinction entre les deux algorithmes à ce sujet. Pourtant, une différence fondamentale existe.

Il sera montré dans la suite de l'exposé (consacrée à la convexification des régions) que `Mark and Explore` a un avantage essentiel : la collecte des éléments constituant une région peut se faire suivant un ordre donné sans aucun coût supplémentaire.

Cet ordre est l'ordre lexicographique sur les coordonnées à deux dimensions des éléments et collecter les éléments suivant cet ordre s'avérera très utile dans la suite des opérations.

Par contre, `Sweep and Merge` ne permettra pas, sans opérations de tri intermédiaires, de conserver cette relation d'ordre entre les éléments de la région. De telles opérations pénaliseraient en effet la complexité générale de l'algorithme, augmentant le coût U de la très importante opération de fusion.

En introduisant la double hypothèse que les éléments de chaque région doivent être stockés de manière contiguë et suivant un ordre lexicographique donné, on préférera donc `Mark and Explore` à `Sweep and Merge`.

En dernière analyse, nous sélectionnerons donc l'algorithme `Mark and Explore` pour effectuer la détection de régions :

- sa complexité attendue est égale à sa complexité optimale ;
- il ne requiert pas de structure de données dynamique complexe ;
- il permet, a posteriori, d'optimiser toute une chaîne d'algorithmes (détection + collecte + opérations ultérieures).

4.3 Approximation des obstacles

Après collecte des éléments constituant les régions, on dispose pour chacune d'entre elles de l'ensemble des éléments qui la composent.

Comment **organiser ces informations de manière à les rendre exploitables**, c'est-à-dire répondre de manière performante à ces interrogations :

- est-ce qu'un obstacle intersecte le segment de droite reliant deux éléments donnés ?
- quels obstacles sont proches d'un élément donné ?
- comment contourner au plus près un obstacle ?

Effectuer une approximation de chaque obstacle peut se révéler intéressant : en effet, on peut s'attendre à une réduction du nombre d'éléments composant chaque obstacle ainsi qu'une structuration de chaque obstacle qui en permettrait une perception se prêtant bien à des requêtes de positionnement.

Aucune contrainte n'étant fixée à ce sujet, une réflexion doit être menée de manière à effectuer un choix raisonnable.

Les approximations suivantes sont envisageables :

- cercle couvrant minimum,
- boîte couvrante minimum (*bounding box*),
- enveloppe générale (*nonconvex hull*),
- enveloppe convexe (*convex hull*).

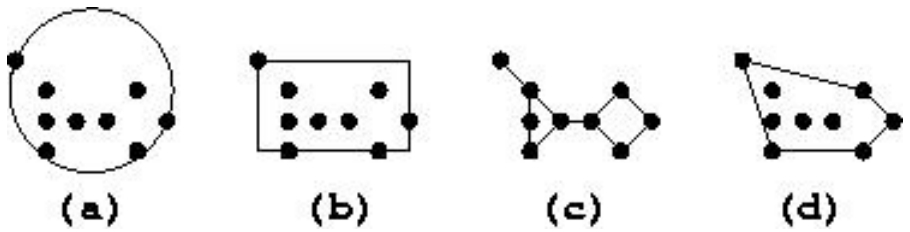


FIG. 4.4 – Approximations d'un obstacle : (a) cercle couvrant minimum (b) boîte couvrante minimum (c) enveloppe générale (d) enveloppe convexe

Le cercle couvrant minimum comme la boîte couvrante minimum, dont l'avantage évident est le faible nombre d'éléments requis pour les définir, dégradent beaucoup la précision de l'approximation. Cela rendrait vraiment très imprécis tout contournement de l'obstacle.

De même, les utiliser en vue de savoir si l'obstacle approximé intersecte le segment de droite reliant deux éléments n'est pas intéressant non plus, sauf peut-être dans le

cadre d'une organisation des données hiérarchique, permettant une approximation sur plusieurs niveaux.

Par contre, ils peuvent être utiles pour déterminer la proximité d'un obstacle.

Les enveloppes sont, quant à elles, excellentes pour approximer l'obstacle au plus près. L'enveloppe convexe constitue en effet le chemin le plus court pour contourner l'obstacle.

Les chemins partant de, et/ou arrivant vers, l'intérieur d'un obstacle (cf. figure 4.5) sont quant à eux calculés avec plus de précision si l'on approxime par son enveloppe générale, mais celle-ci comporte certainement plus d'éléments que l'enveloppe convexe, d'autant plus que les éléments formant les obstacles sont connexes. C'est pourquoi on préférera l'enveloppe convexe.

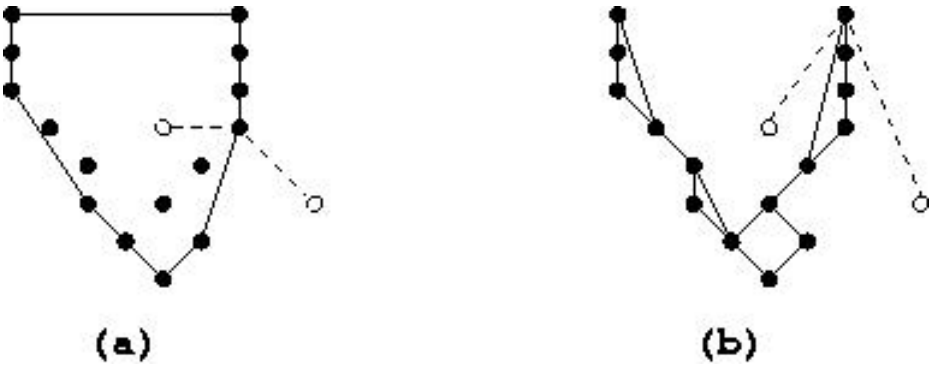


FIG. 4.5 – Approximation : (a) par enveloppe convexe (b) par enveloppe générale

4.3.1 Boîte couvrante minimum

Le calcul de la boîte couvrante minimum (*bounding box*) d'un ensemble de points est très simple. Un algorithme de recherche de l'élément maximum dans un vecteur convient et permet de réaliser cette opération avec une complexité dans le pire cas linéaire en fonction du nombre d'éléments à approximer.

On peut conserver soit les coordonnées de 2 points cardinaux opposés, soit les coordonnées des 4 segments de droite formant la boîte.

4.3.2 Enveloppe convexe

La convexification des obstacles permet de conditionner le problème en vue d'une exploitation efficace de l'information extraite de la détection des régions.

En effet, comme précédemment mis en évidence, contourner un obstacle en vue de calculer le chemin le plus court entre deux éléments séparés par un obstacle revient à contourner l'enveloppe convexe de cet obstacle, tel qu'illustré sur la figure 1.1.

Dans une perspective de convexification des régions, il est nécessaire d'introduire une **définition plus opérationnelle d'un *chemin***. En effet, la définition donnée au chapitre 2 implique le positionnement d'un chemin par rapport à un élément non accessible tel qu'illustré sur la figure 4.6 (a).

FIG. 4.6 – Chemin : 2 définitions

Puisque la résolution des données est de 50 mètres par pixel, cela signifie qu'un chemin passe toujours à plus de 25 mètres du centre des éléments de l'enveloppe convexe de l'obstacle contourné.

Or, il semble plus correct de considérer qu'un chemin contournant un obstacle doit passer par les centres des éléments de l'enveloppe convexe de celui-ci. Dès lors, redéfinissons la notion de chemin.

Chemin géodésique discret C d'un élément accessible appelé *source* vers un autre élément accessible appelé *cible* : suite ordonnée, finie et non redondante[1] d'éléments accessibles à coordonnées discrètes

- dont le premier élément est la source,
- dont les éléments intermédiaires marquent les changements de direction éventuels dus à la présence d'obstacles,
- dont le dernier élément est la cible (ou destination),

[1]Tout élément de la matrice B apparaît au plus une seule fois dans chaque chemin auquel il est intégré.

- pour laquelle, pour toute paire d'éléments consécutifs, chaque segment de droite déterminé par deux éléments consécutifs n'intersecte pas un obstacle *à l'exception éventuelle des éléments de son contour polygonal convexe* mais est localement optimal (c'est-à-dire effectuant un contournement aussi serré que possible de l'obstacle rencontré).

Cette nouvelle définition, illustrée sur la figure 4.6 (b), implique qu'un chemin contournant un obstacle passe à l'intérieur de celui-ci, par son centre, à une distance inchangée (par rapport à la définition originale) de 25 à 35 mètres (vu que 35 mètres = demi résolution $\times\sqrt{2}$).

Le problème est donc déplacé ... Sa complexité demeure toutefois identique.

Quels avantages sont obtenus par rapport à la définition originale d'un chemin ?

- un chemin, selon la nouvelle définition, sera légèrement plus court qu'un même chemin suivant la définition originale, ce qui modélise un contournement plus serré d'un obstacle ;
- le positionnement d'un chemin induit par la nouvelle définition semble conceptuellement plus acceptable.

La nouvelle définition sera adoptée car elle apparaît plus réaliste tout en restant neutre du point de vue de la complexité du problème.

La convexification des régions signifie que de nombreux calculs d'enveloppe convexe doivent être réalisés.

Ces calculs seront réalisés *in situ*, c'est-à-dire en place, dans la structure de données à traiter et sans utiliser de structures de données auxiliaires. Cela éclaire nos propos antérieurs soulignant qu'il était nécessaire que les résultats de la collecte des régions soient stockés dans des zones de mémoire contiguës.

En première analyse, la littérature sur le sujet désigne le QuickHull comme algorithme de calcul d'enveloppe convexe le plus performant. Nous l'avons donc initialement sélectionné.

Un autre algorithme de calcul d'enveloppe convexe, le SortHull, présente en toute généralité **la même complexité attendue que le QuickHull**[2]. En pratique, le QuickHull présente souvent des performances meilleures que le SortHull. De plus, **contrairement au QuickHull, le SortHull opère sur des données triées** par ordre lexicographique sur les coordonnées à deux dimensions des éléments. Cela implique qu'un prétraitement des données est indispensable. Ces arguments ne plaident a priori pas en faveur du SortHull.

Dans la section consacrée à la collecte des régions, il a été mis largement en évidence que l'ordre de collecte des éléments constitutifs d'une région donnée peut être choisi si l'on utilise l'algorithme `Mark and Explore`.

En suivant la numérotation des éléments des régions représentées sur la figure 4.7, on peut observer qu'une collecte des éléments des régions qui se ferait colonne par colonne (de la gauche vers la droite et de haut en bas) offrirait implicitement la version triée des éléments de chaque région.

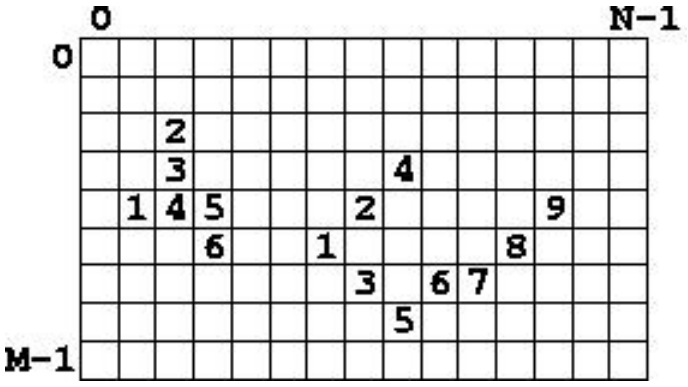


FIG. 4.7 – Collecte des éléments des obstacles par ordre lexicographique

Sans modifier la complexité de l'opération de collecte des régions, **le tri requis par le SortHull est obtenu sans effort de calcul supplémentaire**.

De plus, les éléments intéressants peuvent être filtrés en effectuant ce parcours : étant donné le caractère discret des coordonnées des éléments, parmi les éléments d'abscisse fixée d'un obstacle, seuls les éléments d'ordonnée extrémale intégreront potentiellement l'enveloppe convexe.

Ceci constitue une première motivation pour utiliser le SortHull plutôt que le QuickHull.

[2]Etant donné la relation entre tri et calcul d'enveloppe convexe : $O(N \log N)$, où N représente le nombre de points à convexifier.

Considérons en outre l'observation suivante, proposée dans un document [Ber98] consacré aux calculs d'enveloppes convexes :

[...] *pour la base de données utilisée, l'algorithme le plus rapide semble être le QuickHull, excepté dans le cas où le nombre de sites traité est faible et/ou lorsque ces derniers sont confinés à l'intérieur d'une bande étroite proche de l'enveloppe convexe.*

La base de données en question est relativement exhaustive et est conçue en vue de mettre en évidence le comportement d'algorithmes de calcul d'enveloppe convexe dans des contextes difficiles. Il s'agit donc de données non quelconques, choisies avec soin de manière à étudier le comportement de ces algorithmes.

Introduisons une mesure de forme d'un objet qui pourra aider à poursuivre le raisonnement. L'indice de forme de Morton [Don02a] est une mesure variant entre 0 (ligne) et 1 (cercle). Il est défini par

$$\mathcal{M} = \frac{4 \times S}{\pi \times L^2}$$

où S est la surface de l'objet et L la dimension de son plus grand axe.

Des mesures ont été effectuées sur les 2 jeux de données (zone rurale et zone urbaine, présentées au chapitre 2) à notre disposition : en particulier, la moyenne du nombre d'éléments de chaque obstacle (avant convexification) ainsi que la moyenne des indices $\mathcal{M}$ de forme de Morton (après convexification) ont été calculées.

Ces mesures indiquent que l'enveloppe convexe typique est plutôt rectiligne (moyennes des indices de Morton : 0.13 et 0.15) et résulte du traitement d'un petit nombre d'éléments (moyennes du nombre d'éléments avant convexification : 27.1 et 18.7).

La plupart des obstacles rencontrés dans le contexte qui nous occupe comportent en fait peu d'éléments ou présentent un aspect de bande étroite (cours d'eau, voies de communication, bâtiments, ...). Ainsi, **le contexte dans lequel nous travaillons ne serait pas favorable au QuickHull si l'on se réfère à l'observation de [Ber98].**

Ceci constitue une seconde motivation pour utiliser le SortHull plutôt que le QuickHull.

Ensuite, des mesures ont été effectuées pour un très grand nombre d'obstacles générés aléatoirement de manière à vérifier si le SortHull est effectivement plus rapide que le QuickHull.

Ces mesures sont réalisées après collecte des régions (dont le QuickHull ne tire pas de bénéfice). Les temps de calcul du SortHull exhibent évidemment le même ordre de

Données	Résultats	QuickHull	SortHull
619505 points	24064 régions, 69040 noeuds, 53100 arcs	0.24 s	0.11 s
990684 points	38502 régions, 110562 noeuds, 84968 arcs	0.36 s	0.16 s

FIG. 4.8 – Temps de calcul d'un grand nombre d'enveloppes convexes

grandeur que ceux du QuickHull, mais cette réduction du temps de calcul n'est pas négligeable puisque, dans la plupart des autres situations que celle mise en évidence dans ce travail, c'est le QuickHull qui est plus rapide que le SortHull selon un facteur constant similaire.

En dernière analyse, nous utiliserons donc le SortHull.

Le lecteur désirant compléter sa connaissance des algorithmes de calcul d'enveloppe convexe mentionnés pourra consulter une étude de ceux-ci dans les annexes, ainsi qu'un résumé visuel du processus de convexification des régions.

Il est important de noter que les opérations de convexification auront pour effet de reclasser les éléments du tableau donné de la manière suivante : les éléments formant l'enveloppe convexe seront placés en début de tableau suivant l'ordre trigonométrique à partir de celui qui est géométriquement le plus à l'Est. Les éléments qui auront été classés comme étant internes seront placés en fin de tableau, après ceux constituant l'enveloppe convexe.

En outre, il va de soi, d'une part, que l'ordre relatif de stockage des éléments internes les uns par rapport aux autres n'a pas d'importance et que, d'autre part, les opérations de convexification devront fournir l'indice du tableau où s'arrête le stockage des éléments constituant l'enveloppe convexe et où débute celui des éléments internes.

4.3.3 Tests d'inclusion

Pour compléter l'étude du problème d'approximation d'obstacles, on peut s'intéresser à une opération annexe : le **test d'inclusion d'un élément donné dans une région donnée**.

Deux algorithmes [Oro98] vont être présentés. Leurs complexités attendue et dans le pire cas sont proportionnelles au nombre d'éléments de l'enveloppe convexe.

La première approche consiste à **vérifier si l'élément donné est situé dans l'union de tous les demi-plans extérieurs à l'enveloppe convexe délimités par les segments de celle-ci** (figure 4.9).

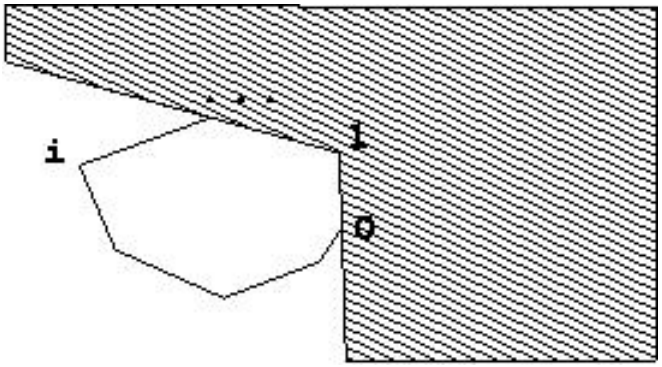


FIG. 4.9 – Test d'inclusion (enveloppe convexe), première approche

L'invariant de cet algorithme est : la variable flag est vraie si $P \in \bigcup_{k=0}^{i-1} \mathcal{P}_k$, où $\mathcal{P}_k$ dénote le demi-plan délimité par le segment k de l'enveloppe convexe. La terminaison de la boucle est évidente.

La seconde approche consiste à **lancer un rayon (*ray shooting*) d'orientation 0 radian (vers la droite ...) à partir de l'élément donné et à compter le nombre d'intersections avec les segments de l'enveloppe convexe**.

En effet, une courbe planaire simple et fermée délimite deux régions connexes du plan : l'intérieur et l'extérieur de cette courbe. Si le nombre d'intersections est impair, l'élément donné est forcément interne (figure 4.10).

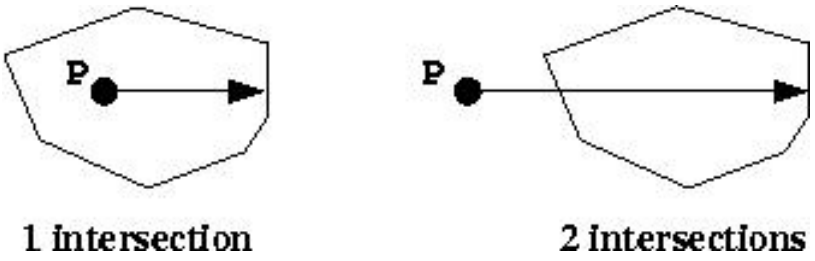


FIG. 4.10 – Test d'inclusion (enveloppe convexe), seconde approche

L'invariant de cet algorithme est : la variable cn compte le nombre d'intersections du rayon - lancé vers la droite à partir de l'élément donné - avec les segments déjà parcourus $[0..i-1]$. La terminaison suit et le résultat final est déterminé par la parité de la variable cn.

Notons toutefois que **ce second algorithme n'est pas pleinement satisfaisant**. La figure 4.11 illustre le problème. Un choix doit être réalisé par rapport à la comptabilisation des intersections.

Si une intersection avec un segment de l'enveloppe convexe est comptée uniquement lorsqu'elle est strictement à droite de l'élément donné, alors P_1 est interne (1 intersection) et P_2 est externe (0 intersection).

Par contre, si une intersection avec un segment de l'enveloppe convexe est comptée lorsqu'elle est à droite ou bien de même ordonnée que l'élément (ce dernier cas se produit lorsque l'élément est situé sur le segment de l'enveloppe convexe), alors P_1 est externe (2 intersections) et P_2 est interne (1 intersection).

Quel que soit le choix, les éléments situés à la frontière de l'enveloppe convexe, sur l'un de ses segments, sont renseignés par l'algorithme comme étant tantôt internes, tantôt externes, selon le positionnement de l'élément par rapport à l'enveloppe convexe. Ce qui n'est évidemment pas idéal du point de vue topologique.

FIG. 4.11 – Test d'inclusion (enveloppe convexe), seconde approche : interne ou externe ?

`isInsideConvexHull`

```
{ Première approche }

isInsideConvexHull(v : Vertex ;
                   data : array[0 .. size−1] of Vertex ; size : int ) : boolean
var
  flag : boolean;
  i : int ;
|[ if (size < 3) → isInsideCH := false
 [] (size ≥ 3) →
    i, flag := 0, true
    ;do ((i < size−1) and flag) →
        if (Δ(v,data[i],data[i+1]) ≤ 0) → flag := false
        [] (Δ(v,data[i],data[i+1]) > 0) → skip
        fi
        ;i := i+1
     od
    ;if (flag) → isInsideCH := (Δ(v,data[size−1],data[0]) > 0)
     [] (not flag) → isInsideCH := false
     fi
 fi ]|

{ Seconde approche }

isInsideConvexHull_RS(v : Vertex ;
                      data : array[0 .. size−1] of Vertex ; size : int ) : boolean
var
  flag : boolean;
  cn, i : int ;
  vt : float ;
|[ if (size < 3) → isInsideCH := false
 [] (size ≥ 3) →
    cn,i := 0,0 { cn : compteur d'intersections }
    ;do i < size → { Parcours des arêtes de l'enveloppe convexe }
        if (((data[i].y ≤ v.y) and
             (data[(i+1) mod size].y > v.y)) { Intersect.−>haut }
            or
            ((data[i].y > v.y) and
             (data[(i+1) mod size].y ≤ v.y))) { Intersect.−>bas } →
           { Calcul de la coord. en x de l'intersection avec l'arête [i .. i+1], dont
             on sait (vu les 2 sous−gardiens) qu'elle n'est pas horizontale }
           vt := (v.y − data[i].y) / (data[(i+1) mod size].y − data[i].y)
           ;if (v.x < (data[i].x + vt ∗ (data[(i+1) mod size].x − data[i].x))) →
               cn := cn+1 { Intersect. valide à la droite de v }
            [] not (...) → skip
            fi
        [] not (...) → skip
        fi
        ;i := i+1
    od
    ;isInsideCH_RS := (cn mod 2 = 1)
 fi ]|
```

Algorithme 4.3 – `isInsideConvexHull`

4.4 Conclusion

La **structure générale de l'algorithme** `DGSP` a été présentée, ainsi que les **choix de conception** qui ont été réalisés.

Les premières étapes de résolution de `DGSP` ont été étudiées : il a été montré comment **détecter, collecter et structurer les données**.

En outre, il a été montré comment **intégrer les processus de détection, collecte et approximation d'obstacles dans une chaîne d'algorithmes**.

Aucune des opérations prise isolément n'est véritablement compliquée, mais le raisonnement et les interrogations conduisant à un processus de traitement globalement optimisé ne sont pas immédiats.

En effet, si le choix de chaque algorithme intervenant dans le processus de traitement est effectué séparément, selon une perspective locale, les performances globales du processus ne seront pas aussi bonnes qu'en ayant tenu compte des résultats en amont et en aval à chaque étape du processus.

En d'autres termes, cela constitue un bon exemple de chaîne algorithmique dont le développement n'est pas optimal s'il est effectué uniquement selon une stratégie *boîte noire* isolant le développement de chaque composant.

Ce n'est pas la séparation des problèmes en vue de mieux les résoudre qui est ici mise en défaut mais bien le fait de ne pas étudier les interactions entre les processus résolutifs développés.

En particulier, voici un rappel des choix réalisés pour intégrer les différents algorithmes :

- sélection de l'algorithme `Mark and Explore` plutôt que `Sweep and Merge`, ce qui permet d'obtenir une version filtrée et triée des éléments à convexifier, en bénéficiant de plus d'une meilleure complexité dans le pire cas et en facilitant le stockage des données dans des zones mémoire contiguës ;
- sélection de l'enveloppe convexe comme structure d'approximation d'un obstacle ;
- sélection de l'algorithme `SortHull` plutôt que `QuickHull`, permise par la sélection précédente et souhaitable étant donné les caractéristiques des données à convexifier.

Du point de vue de l'objectif de contournement de barrières, ce chapitre a montré comment structurer les données de manière à appliquer la suite des traitements proposés : reconditionnement de l'espace de recherche et calcul des chemins les plus courts.

Chapitre 5

Calcul des requêtes

Dans ce chapitre, la démarche permettant de reconditionner l'espace de recherche d'un chemin le plus court sera détaillée.

L'approche envisagée est d'adopter une **perspective d'observation en de nombreuses localisations en vue d'acquérir par avance une connaissance totale de tous les chemins** intermédiaires permettant de contourner toute combinaison d'obstacles de la zone géographique donnée.

La construction d'un chemin peut ainsi être ramenée à une **opération de sélection parmi tous les changements de direction possibles** dus à des contournements d'obstacles.

5.1 Modèle de visibilité

Comment définir le fait qu'un élément est visible par un autre élément (et réciproquement) ?

Nous proposons d'utiliser le modèle de visibilité suivant [Hav00] : deux points $P_1 = (x_1, y_1)$ et $P_2 = (x_2, y_2)$ sont mutuellement visibles si le segment de droite $\overline{P_1 P_2}$ n'intersecte *aucun objet localisé dans la scène.*

Le modèle générique de [Hav00] est très complet. En se restreignant au contexte de ce document, un objet se limite à une barrière et la scène est constituée des barrières de la matrice B qui est à l'origine du problème posé.

On définit le **problème de visibilité** comme étant la détermination de la présence d'au moins un objet intersectant le segment de droite reliant un point source et un point destination. Un tel objet est dit *bloquant.*

On définit une **requête de visibilité** comme étant la résolution du problème de visibilité sous-jacent.

Par ailleurs, le **problème de lancer de rayon** (*ray shooting*), qui est un problème de visibilité le long d'une droite, peut être défini comme suit, toujours en suivant [Hav00] : pour une demi-droite orientée et définie par son point d'application et son vecteur de direction, on recherche l'objet le plus proche intersecté par la demi-droite, si un tel objet existe.

Comment exploiter les requêtes de visibilité ?

La structure de données à travers laquelle l'espace de recherche sera abstrait et perçu est le graphe de visibilité. Il permettra de sélectionner le chemin le plus court parmi tous les chemins intermédiaires implicitement stockés dans le graphe de visibilité.

Dans un premier temps, nous travaillerons dans un contexte restreint de manière à concentrer notre attention sur l'essentiel. Ainsi, les deux hypothèses suivantes seront tenues pour vraies dans un premier temps, puis levées en fin de chapitre :

- les enveloppes convexes approximant les barrières sont toutes disjointes et non emboîtées ;
- les éléments inclus dans une enveloppe convexe approximant une barrière ne seront pas considérés comme source ou destination d'un chemin.

5.2 Angle enveloppant

Introduisons maintenant la notion d'**angle enveloppant**, dont le but est de découvrir si un obstacle ou un morceau d'obstacle (par exemple, un arc d'une enveloppe convexe approximant un obstacle) limite le champ de visibilité d'un point. Autrement dit, on **résoud le problème de visibilité**.

5.2.1 Notion d'angle enveloppant

Le champ de visibilité d'un élément est l'union d'angles issus de cet élément tels que celui-ci n'y perçoit la présence d'aucun objet de la scène. L'amplitude du champ de visibilité d'un élément non entouré d'obstacles vaut évidemment 2π.

Etant donné un point et un obstacle, définissons la notion d'angle enveloppant l'obstacle à partir du point (figure 5.1) comme étant l'angle (unique) dont les branches sont tangentes à l'obstacle. L'union des angles enveloppants les obstacles voisinant un élément est le complémentaire du champ de visibilité.

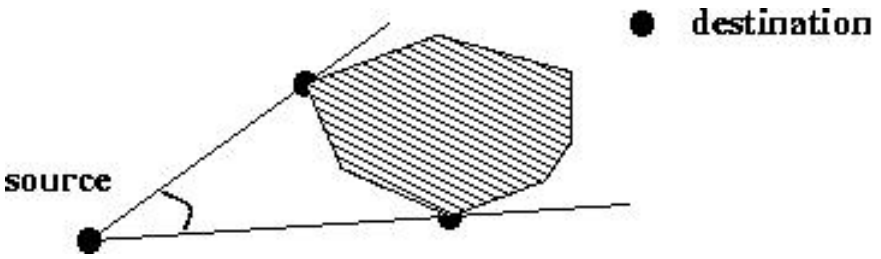


FIG. 5.1 – Angle enveloppant

Soit l'angle dont l'origine est le point source du problème de visibilité, dont une des branches est la demi-droite horizontale dirigée vers la droite et dont l'autre branche est la demi-droite passant par le point destination (figure 5.2). Appelons angle cible cet angle.

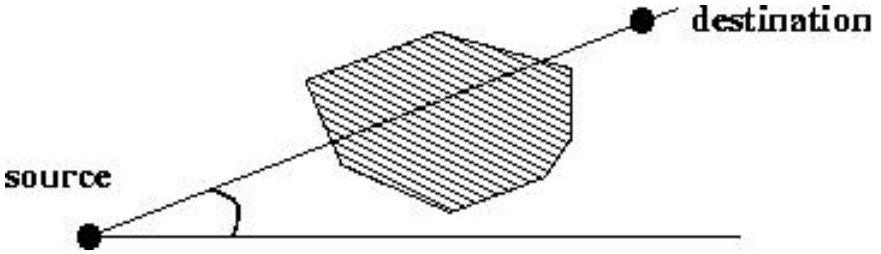


FIG. 5.2 – Angle cible

Soient les deux angles dont l'origine est le point source du problème de visibilité, dont une des branches (qui leur est commune) est la demi-droite horizontale dirigée vers la droite et dont l'autre branche est, respectivement, une des deux branches de l'angle enveloppant (figure 5.3). Appelons angles bornants ces deux angles.

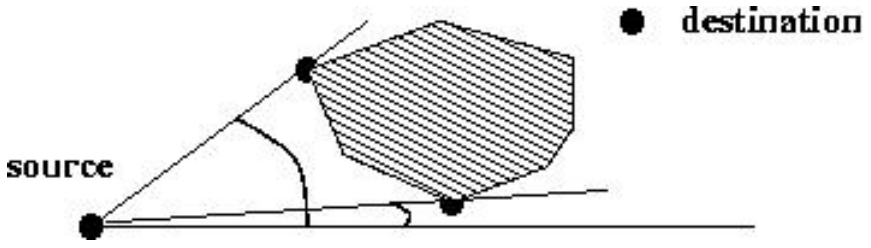


FIG. 5.3 – Angles bornants

Comment calculer si un obstacle limite le champ de visibilité d'un élément (5.4) ?

Après avoir calculé les deux points de tangence de l'angle enveloppant (qui sont des sommets de l'enveloppe convexe approximant l'obstacle), on vérifie si l'amplitude de l'angle cible est comprise dans l'intervalle délimité par les amplitudes des angles bornants (définis par les deux points de tangence calculés).

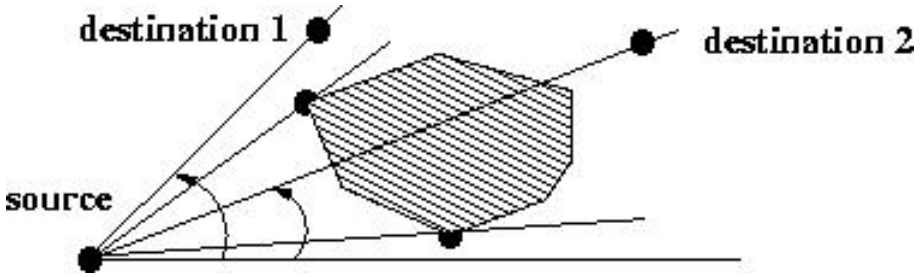


FIG. 5.4 – Angles cibles et angle enveloppant : destination1 est visible, destination2 n'est pas visible

5.2.2 Recherche linéaire

Définissons un test de visibilité pour un élément v donné et l'élément i d'un obstacle (cf. figure 5.5) :

Test de visibilité

```
isVisible (v : Vertex ; i : int ;
           data : array[0 .. size−1] of Vertex ; size : int ) : boolean
var
  im1, ip1 , t1 , t2 : int ;
|[ if (i = 0) → im1,ip1 := size−1,1
 [] (i = size−1) → im1,ip1 := size−2,0
 [] ((i > 0) and (i < size−1)) → im1,ip1 := i−1,i+1
 fi
 ;t1,t2 := Δ(v,data[i],data[ip1] ),Δ(v,data[im1],data[i])
 ; isVisible := ((t1 < 0) or (t2 < 0) or
                ((t1 = 0) and (t2 ≤ 0)) or ((t2 = 0) and (t1 ≤ 0))) ]|
```

Algorithme 5.1 – Test de visibilité

Pour découvrir les indices limites de la zone visible, une recherche linéaire est effectuée à partir d'un sommet de l'enveloppe convexe (choisi arbitrairement). On peut classer selon le résultat du test de visibilité les sommets de l'enveloppe convexe en

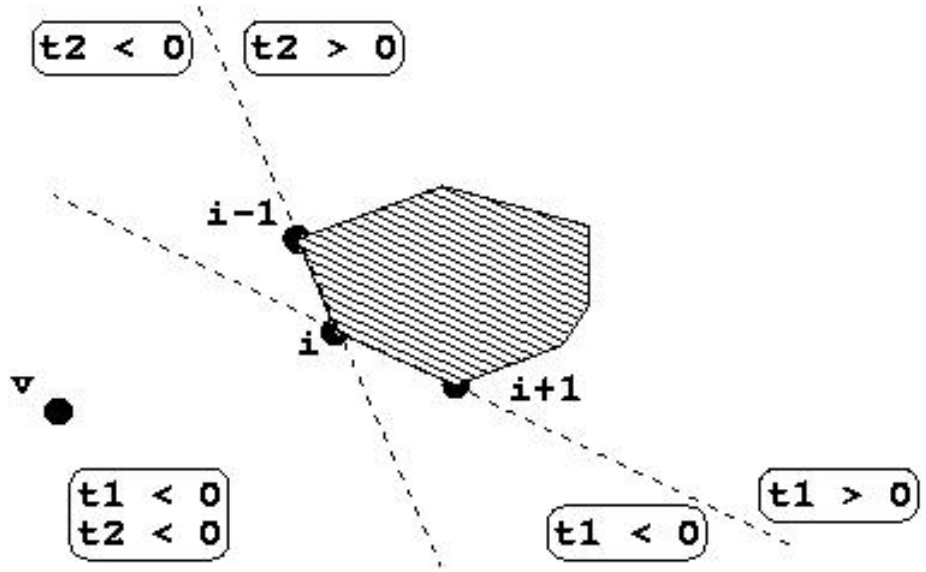


FIG. 5.5 – Test de visibilité

deux zones : la zone visible et la zone cachée.

On commence pour cela par déterminer dans quelle zone se trouve le premier sommet examiné. On parcourt ensuite séquentiellement les sommets en détectant les changements de zone : on découvre ainsi les deux sommets recherchés. La situation où le premier élément est dans la zone visible est représentée sur la figure 5.6.

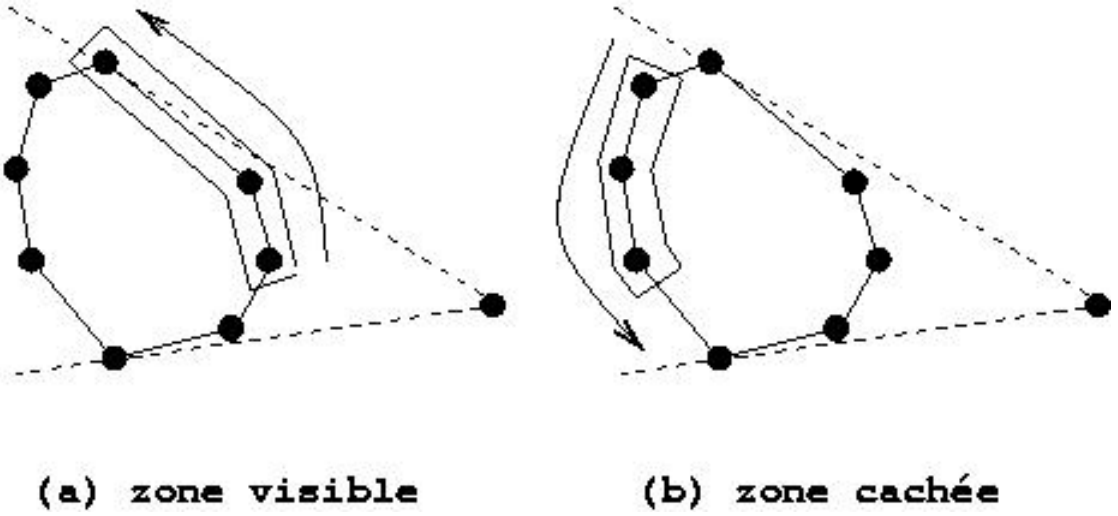


FIG. 5.6 – Angle enveloppant (a) parcours de la zone visible, puis (b) parcours de la zone cachée

Invariant (cas où le premier élément est dans la zone visible) :

	0	i	$size$
data	Zone visible	Zone à explorer	

	0	$iright + 1$	i	$size$
data	Zone visible	Zone cachée	Zone à explorer	

Postcondition (cas où le premier élément est dans la zone visible) :

	0	$iright + 1$	$ileft$	$size$
data	Zone visible	Zone cachée	Zone visible	

On raisonne par symétrie pour le cas où le premier élément est dans la zone cachée.

Ensuite, soulignons qu'il y a toutefois deux cas particuliers à ne pas perdre de vue.

Premièrement, lorsque tous les éléments sont visibles, l'algorithme proposé est incorrect. Or, une telle situation ne peut être détectée qu'en ayant parcouru tous les sommets de l'obstacle. Après détection d'une telle situation, un algorithme auxiliaire traitant uniquement ce cas particulier doit être mis en oeuvre.

Deuxièmement, lorsque tous les éléments sont cachés, l'algorithme renvoie une valeur d'échec. Cela se produit lorsque l'élément source est interne à l'enveloppe convexe donnée.

Les algorithmes proposés supposent qu'il y a au moins 3 sommets à traiter (étant donné les calculs d'aire signée). Le traitement des cas comportant moins de 3 sommets est immédiat.

Angle enveloppant (recherche linéaire)

```
computeWrappingAngle_lin(source : Vertex ;
                         data : array[0 .. size−1] of Vertex ; size : int ;
                         var ileft : int ; var iright : int ) : boolean

var
  i : int ;

|[ i := 0
 ; if isVisible (source, i , data, size ) →
      { zone visible }
      ; do ((i < size ) cand isVisible(source, i , data, size )) →
           i := i+1
       od
      ; if (i = size ) →
           { tous visibles ! }
           computeWrappingAngle_lin :=
             computeWrappingAngle_vis(source,data,size,ileft, iright )
       [] (i < size ) →
           iright := i−1
           { zone cachée }
           ; i := i+1
           ; do ((i < size ) cand (not isVisible(source, i , data, size ))) →
                i := i+1
            od
           ; ileft := i mod size
           ; computeWrappingAngle_lin := true
       fi
  [] not isVisible (source, i , data, size ) →
      { zone cachée }
      ; do ((i < size ) cand (not isVisible(source, i , data, size ))) →
           i := i+1
       od
      ; if (i = size ) →
           { tous cachés ! }
           computeWrappingAngle_lin := false
       [] (i < size ) →
           ileft := i
           { zone visible }
           ; i := i+1
           ; do ((i < size ) cand isVisible(source, i , data, size )) →
                i := i+1
            od
           ; iright := i−1
           ; computeWrappingAngle_lin := true
       fi
  fi ]|
```

Algorithme 5.2 – Angle enveloppant (recherche linéaire)

Résolvons maintenant le cas particulier où tous les sommets de l'obstacle sont visibles de l'observateur. La figure 5.7 illustre la propriété suivante : un des deux points de tangence recherchés est le sommet de l'obstacle le plus éloigné de l'élément source. Si le sommet le plus éloigné de l'observateur n'était pas un des points de tangence, soit l'obstacle serait concave, soit tous les sommets ne seraient pas visibles de l'observateur.

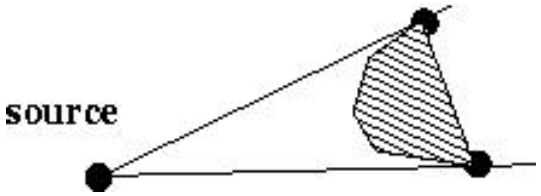


FIG. 5.7 – Angle enveloppant (tous les sommets visibles)

Un algorithme de coût linéaire consiste à rechercher le sommet le plus éloigné de l'observateur, puis de tester s'il s'agit du point de tangence gauche ou droit : une comparaison, d'une part, de l'angle formé par l'observateur et le sommet le plus éloigné, avec, d'autre part, l'angle formé par l'observateur et le voisin du sommet le plus éloigné, peut être effectuée.

Angle enveloppant (tous les sommets visibles)

```
computeWrappingAngle_vis(source : Vertex ;
                         data : array[0 .. size−1] of Vertex ; size : int ;
                         var ileft : int ; var iright : int ) : boolean
var
  i , imax, im1, ip1 : int ;
  d , dmax, a, b : float ;
|[ imax,dmax,i := 0,distance(source,data[0]),1
 ;do (i < size) → d := distance(source,data[i])
      ;if (d > dmax) → imax,dmax := i,d [] (d ≤ dmax) → skip fi
      ;i := i+1
 od
 ;if (imax = 0) → im1,ip1 := size−1,1
  [] (imax = size−1) → im1,ip1 := size−2,0
  [] ((imax > 0) and (imax < size−1)) → im1,ip1 := imax−1,imax+1
  fi

 ;a,b := angle(source,data[imax]),angle(source, data[ip1] )
 ;if (a < b) → ileft,iright := ip1,imax
  [] (a > b) → ileft,iright := imax,im1
 fi ]|
```

Algorithme 5.3 – Angle enveloppant (tous les sommets visibles)

Les angles sont calculés selon le schéma suivant (figure 5.8) :

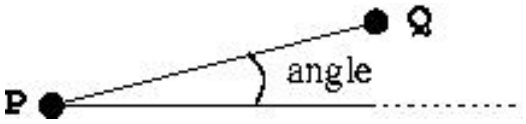


FIG. 5.8 – Calcul d'un angle (P et Q donnés)

5.2.3 Recherche dichotomique

Un autre point de vue qui viserait à effectuer une recherche dichotomique plutôt que linéaire peut-il être envisagé ?

D'une part, comme précédemment mentionné, l'ordre dans lequel sont stockés les sommets de l'obstacle est, arbitrairement, l'ordre de parcours convexe de ceux-ci dans le sens trigonométrique. D'autre part, le positionnement relatif du point source et de l'obstacle est quelconque et aucune information a priori n'est disponible.

Comment dès lors organiser les sommets de chaque obstacle dans une phase de prétraitement de manière à effectuer des requêtes de coût logarithmique indépendamment du positionnement relatif du point source et de l'obstacle ?

Reformulons (cf. figure 5.9) le problème de calcul d'un angle enveloppant comme suit : en considérant conceptuellement un tableau d'entiers dont chaque entrée correspond à un sommet de l'enveloppe convexe et a pour valeur 1 si celui-ci est visible depuis le point source et 0 s'il ne l'est pas, trouver les indices inférieur et supérieur (en arithmétique modulo la taille du tableau, c'est-à-dire le nombre de sommets de l'enveloppe convexe approximant l'obstacle) délimitant la zone de 1, c'est-à-dire la zone visible.

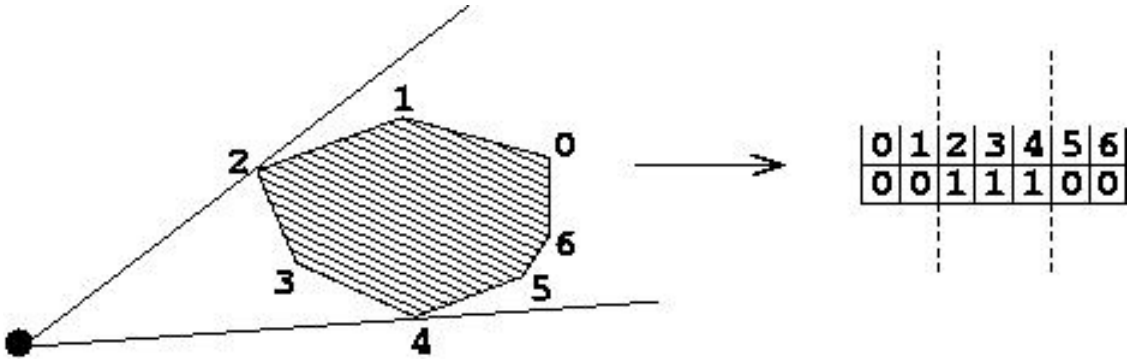


FIG. 5.9 – Reformulation du problème de calcul d'un angle enveloppant

Quelles observations peuvent être formulées à partir de cette perspective ?

Une paire de recherches dichotomiques (une par indice à calculer) serait possible dès lors que l'on connaîtrait les bornes de la zone de recherche.

En effet, par exemple, si la valeur associée à l'élément médian n'est pas 1, comment savoir si la recherche doit être poursuivie vers la partie basse ou vers la partie haute du tableau puisque la zone visible ne commence pas nécessairement au premier élément du tableau et que la zone cachée ne se prolonge pas nécessairement jusqu'au dernier élément du tableau.

Si on connaît de manière certaine les indices u et z d'un élément dont la valeur associée est 1 et d'un élément dont la valeur associée est 0, on peut effectuer une première recherche dichotomique de l'indice inférieur dans la zone $[z..u]$ du tableau (en arithmétique modulo sur les indices) et une seconde recherche dichotomique de l'indice supérieur dans la zone $[u..z]$ (en arithmétique modulo également).

Il convient d'être prudent pour la seconde recherche dichotomique étant donné que les indices de la zone à traiter présentent une discontinuité due au fait que les bornes de cette zone ne correspondent plus aux bornes du vecteur initial. Le fait que, initialement, $u < z$ ou $u > z$ a une influence sur les bornes des recherches dichotomiques et amène à considérer les deux cas séparément.

Voici les invariants et postconditions guidant la paire de recherches dichotomiques des indices de la zone visible, pour le cas $u < z$. Ceux pour le cas $u > z$ se déduisent par symétrie. Le cas où $u = z$ n'a pas de sens.

Invariant (recherche de la borne droite) :

		u		lo			hi		z	
data		1	...	1	$\longrightarrow$	$\longleftarrow$	0	...	0	

Post condition (recherche de la borne droite) :

		u		lo	hi		z	
data		1	...	1	0	...	0	

Invariant (recherche de la borne gauche) :

		z		lo			hi		u	
data		0	...	0	$\longrightarrow$	$\longleftarrow$	1	...	1	

Post condition (recherche de la borne gauche) :

		z		lo	hi		u	
data		0	...	0	1	...	1	

Angle enveloppant (recherche dichotomique)

```
computeWrappingAngle_bin( u : int ; z : int  ; source  : Vertex ;
                                  data : array[0 .. size−1] of Vertex ; size  : int  ;
                                  var  ileft  : int  ; var  iright  : int ) : boolean
var
  lo , hi , mid : int ;
|[ lo,hi := u,z
 ; if  (u < z) →
     ;do (lo < hi−1) → mid := (lo + hi) div 2
         ; if  ( isVisible (source,mid,data,size)) → lo := mid { Visible, 1 }
           []  (not  isVisible (source,mid,data,size)) → hi := mid { Caché, 0 }
           fi
      od
     ; iright  := lo

     lo , hi := z , u
     ;do (lo < (hi−1)+size) → mid := ((lo + hi + size) div 2) mod size
         ; if  (not  isVisible (source,mid,data,size)) → lo := mid { Caché, 0 }
           []  ( isVisible (source,mid,data,size)) → hi := mid { Visible, 1 }
           fi
      od
   []  (u > z) →
     ;do (lo < (hi−1)+size) → mid := ((lo + hi + size) div 2) mod size
         ; if  ( isVisible (source,mid,data,size)) → lo := mid { Visible, 1 }
           []  (not  isVisible (source,mid,data,size)) → hi := mid { Caché, 0 }
           fi
      od
     ; iright  := lo

     lo , hi := z , u
     ;do (lo < (hi−1)) → mid := (lo + hi) div 2
         ; if  (not  isVisible (source,mid,data,size)) → lo := mid { Caché, 0 }
           []  ( isVisible (source,mid,data,size)) → hi := mid { Visible, 1 }
           fi
      od
  fi
 ; ileft  := hi
 ;computeWrappingAngle_bin := true ]|
```

Algorithme 5.4 – Angle enveloppant (recherche dichotomique)

Il reste désormais à régler le problème d'orientation mentionné : en d'autres termes, **comment déterminer les indices u et z délimitant deux paires de séquences homogènes d'éléments** (suite de 0, suite de 1, d'une part et suite de 1, suite de 0, d'autre part) permettant d'effectuer des recherches dichotomiques.

Comment déterminer rapidement u et z d'après le positionnement relatif de l'élément donné et de l'obstacle ?

L'exploitation d'un partitionnement des sommets de l'enveloppe convexe répond à ce problème.

La figure 5.10 présente la décomposition d'une enveloppe convexe en quatre quadrants selon les deux médianes de la boîte couvrante minimum (le centre de gravité est également représenté). Nous appelons un tel partitionnement géométrique *unweigthed quadrant partitioning* (`UQP`).

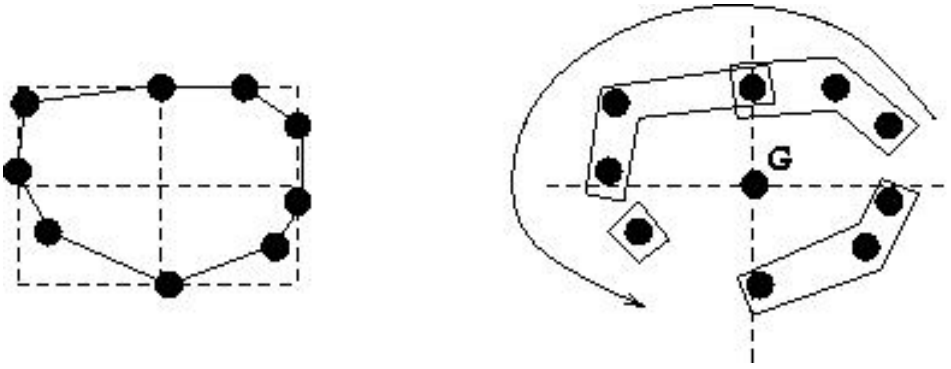


FIG. 5.10 – Unweigthed Quadrant Partitioning

Etant donné que, par définition, une enveloppe convexe ne comporte jamais 3 sommets alignés, la logique de partitionnement peut être basée sur des comparaisons avec les coordonnées du centre de gravité de la boîte couvrante minimum.

Cette décomposition est immédiate et est de complexité dans le pire cas $O(N)$ en effectuant un simple parcours du vecteur.

L'invariant des quatre boucles successives de l'algorithme de partitionnement, visant chacune à collecter les sommets d'un quadrant différent, est : les sommets parcourus appartiennent au quadrant courant. La terminaison est assurée car les sommets des différents quadrants sont collectés dans le même ordre - trigonométrique - que l'ordre de stockage des sommets de l'enveloppe convexe.

`Unweighted Quadrant Partitioning`

```
type Quadrant = record
  begin
   i_first  : int ; { Indice dans data du premier élément du Quadrant }
   i_last   : int ; { Indice dans data du dernier élément du Quadrant }
   qsize  : int ;   { Nombre d'éléments du Quadrant }
   z  : int ;       { Indice dans data d'un élément non visible ( si ≥ 0) }
  end;

uqp(data : array[0 .. size−1] of Vertex ; size  : int  ;
      bb_e : int  ; bb_n : int  ; bb_w : int  ; bb_s : int  ;
      var G : Vertex ; { Centre de gravité }
      var q_ne : Quadrant ; var q_nw : Quadrant ;
      var q_sw : Quadrant ; var q_se : Quadrant)
{ Les variables bb_ ... contiennent les coordonnées
  extrémales de la boîte couvrante minimum
  pour les 4 points cardinaux }

var
  flag  : boolean;
  i , i_xopt , i_yopt , a , b  : int ;

|[ G.x,G.y := bb_w+((bb_e − bb_w) div 2),bb_s+((bb_n − bb_s) div 2)

 ; i , flag  := 0,true { Faire en sorte que i référence le 1er élément
                        du quadrant N−W }
 ;do flag →
      if ( data[i] .x > G.x) → i := i+1
      [] ( data[i] .x ≤ G.x) → flag := false
      fi
  od
 ; i := i  mod size

 ; flag ,q_nw.qsize := true,0 { Quadrant N−W }
 ;i_xopt , i_yopt ,q_nw. i_first  := i , i , i
 ;do flag →
      if ( data[i] .y > G.y) →
         q_nw. i_last ,q_nw.qsize := i ,q_nw.qsize+1
         ; i := ( i+1) mod size
      [] ( data[i] .y = G.y) →
         q_nw. i_last ,q_nw.qsize := i ,q_nw.qsize+1
         ; flag  := false
      [] ( data[i] .y < G.y) → flag := false
      fi
      ; if ( data[i] .x < data[i_xopt] .x) → i_xopt := i
       [] ( data[i] .x ≥ data[i_xopt].x) → skip
       fi
      ; if ( data[i] .y > data[i_yopt] .y) → i_yopt := i
       [] ( data[i] .y ≤ data[i_yopt].y) → skip
       fi
  od
```

Algorithme 5.5 – `Unweighted Quadrant Partitioning`

`Unweighted Quadrant Partitioning` **(suite)**

```
;a,b := i_yopt,i_xopt
;if ((q_nw.qsize ≥ 3) and
      (((a ≤ b) and (b−a > 1)) or ((a > b) and (b+size−a > 1)))) →
    q_nw.z := (a+1) mod size
 [] (not ...) → q_nw.z := −1
 fi

;flag,q_sw.qsize := true,0 { Quadrant S−W }
;i_xopt,i_yopt,q_sw. i_first := i,i,i
;do flag →
    if ( data[i] .x < G.x) →
       q_sw. i_last ,q_sw.qsize := i,q_sw.qsize+1
       ;i := (i+1) mod size
    [] ( data[i] .x = G.x) →
       q_sw. i_last ,q_sw.qsize := i,q_sw.qsize+1
       ;flag := false
    [] ( data[i] .x > G.x) → flag := false
    fi
    ;if ( data[i] .x < data[i_xopt] .x) → i_xopt := i
     [] ( data[i] .x ≥ data[i_xopt].x) → skip
     fi
    ;if ( data[i] .y < data[i_yopt] .y) → i_yopt := i
     [] ( data[i] .y ≥ data[i_yopt].y) → skip
     fi
 od
;a,b := i_xopt,i_yopt
;if ((q_sw.qsize ≥ 3) and
      (((a ≤ b) and (b−a > 1)) or ((a > b) and (b+size−a > 1)))) →
    q_sw.z := (a+1) mod size
 [] (not ...) → q_sw.z := −1
 fi

;flag,q_se.qsize := true,0 { Quadrant S−E }
;i_xopt,i_yopt,q_se. i_first := i,i,i
;do flag →
    if ( data[i] .y < G.y) →
       q_se. i_last ,q_se.qsize := i,q_se.qsize+1
       ;i := (i+1) mod size
    [] ( data[i] .y = G.y) →
       q_se. i_last ,q_se.qsize := i,q_se.qsize+1
       ;flag := false
    [] ( data[i] .y > G.y) → flag := false
    fi
    ;if ( data[i] .x > data[i_xopt] .x) → i_xopt := i
     [] ( data[i] .x ≤ data[i_xopt].x) → skip
     fi
    ;if ( data[i] .y < data[i_yopt] .y) → yopt := i
     [] ( data[i] .y ≥ data[i_yopt].y) → skip
     fi
 od
```

Algorithme 5.6 – `Unweighted Quadrant Partitioning` (suite)

`Unweighted Quadrant Partitioning` **(suite)**

```
;a,b := i_yopt,i_xopt
;if ((q_se.qsize ≥ 3) and
      (((a ≤ b) and (b−a > 1)) or ((a > b) and (b+size−a > 1)))) →
    q_se.z := (a+1) mod size
 [] (not ...) → q_se.z := −1
 fi

;flag,q_ne.qsize := true,0 { Quadrant N−E }
;i_xopt,i_yopt,q_ne. i_first := i,i,i
;do flag →
    if (data[i].x > G.x) →
      q_ne. i_last ,q_ne.qsize := i,q_ne.qsize+1
      ;i := (i+1) mod size
    [] (data[i].x = G.x) →
      q_ne. i_last ,q_ne.qsize := i,q_ne.qsize+1
      ;flag := false
    [] (data[i].x < G.x) → flag := false
    fi
    ;if (data[i].x > data[i_xopt].x) → i_xopt := i
     [] (data[i].x ≤ data[i_xopt].x) → skip
     fi
    ;if (data[i].y > data[i_yopt].y) → i_yopt := i
     [] (data[i].y ≤ data[i_yopt].y) → skip
     fi
 od ]|
;a,b := q_ne.i_xopt,q_ne.i_yopt
;if ((q_ne.qsize ≥ 3) and
      (((a ≤ b) and (b−a > 1)) or ((a > b) and (b+size−a > 1)))) →
    q_ne.z := (a+1) mod size
 [] (not ...) → q_ne.z := −1
 fi

{ Notes
  * l'arrêt des boucles est conditionné par le fait
    que data est initialement bien ordonné
  * pour préserver l'ordre des sommets lors du stockage
    dans les quadrants, il convient de commencer par
    le quadrant N−W car le premier sommet de data
    n'est pas nécessairement dans le quadrant N−E
    et, réciproquement, le premier sommet du quadrant N−E
    n'est pas nécessairement le premier sommet de data;
    la seule certitude est que le premier sommet de data
    n'appartient pas à un des quadrants N−W ou S−W
    (sauf si size = 1 ...)
  * les sommets pouvant appartenir à plusieurs quadrants
    sont effectivement inclus dans chacun de ceux−ci }
```

Algorithme 5.7 – `Unweighted Quadrant Partitioning` (suite)

Le partitionnement en quatre séquences, une par quadrant, des sommets formant l'enveloppe convexe approximant un obstacle donné peut être effectué pour chaque obstacle lors d'une phase d'initialisation.

On peut exhiber les 2 propriétés suivantes de l'`UQP` pour un observateur donné :

- on peut toujours calculer en temps logarithmique un sommet de l'obstacle visible par l'observateur (complexité dans le pire cas $O(\log N)$, où N représente le nombre de sommets de l'enveloppe convexe) ;
- on peut souvent calculer en temps constant un sommet de l'obstacle caché à l'observateur (complexité $O(1)$ dans le pire cas).

Ces deux propriétés sont vérifiées grâce au caractère convexe des quatre séquences de sommets résultant du partitionnement.

La première propriété se démontre comme suit. L'observateur est dans un des quatre quadrants. Il est forcément situé dans une zone limitée par un des angles formés par les demi-droites qui sont issues du centre de gravité et qui passent par les sommets de ce quadrant. La figure 5.11 illustre cela.

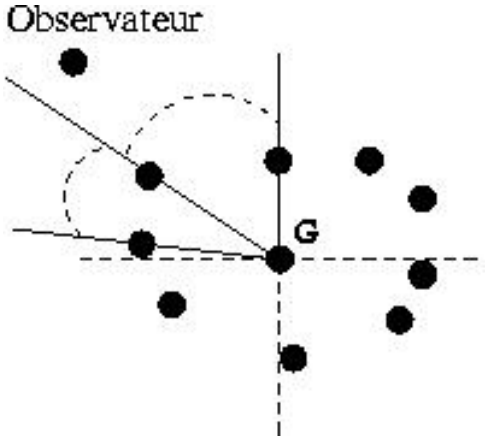
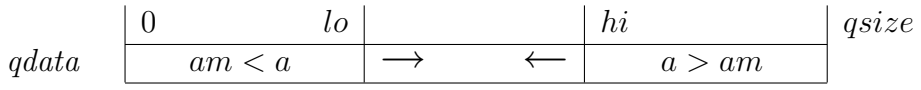


FIG. 5.11 – Calcul rapide d'un sommet visible

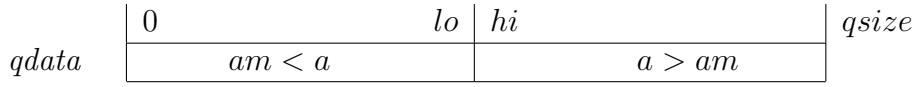
Les sommets supportant l'angle dans lequel est situé l'observateur sont tous deux visibles : on peut assigner à u l'indice de l'un ou l'autre. S'il vient que l'observateur est situé entre une des médianes et la plus proche demi-droite issue de G, il n'y a qu'un des sommets du quadrant qui est visible, mais cela est suffisant puisqu'on recherche un seul indice u.

L'algorithme pour calculer u se résume donc à une recherche dichotomique de l'angle dans lequel est situé l'observateur, garantissant une complexité dans le pire cas logarithmique en fonction du nombre d'éléments de l'enveloppe convexe.

Invariant (recherche de u) :



Postcondition (recherche de u) :



Angle enveloppant 2.0, calcul de u

```
compute_u(source : Vertex ; G : Vertex ; quad : Quadrant ;
          data : array[0 .. size−1] of Vertex ; size : int ;
          var u : int)

var
  a, am: float;
  lo, hi, mid : int;
  flag : boolean;

|[ if (quad.qsize = 0) → u := −1
 [] (quad.qsize ≥ 1) →
    a := angle(G,source)
    ;lo,hi := quad. i_first ,quad. i_last
    ;if (angle(G,data[lo] ) ≥ a) → u := lo
     [] (angle(G,data[hi] ) ≤ a) → u := hi
     [] ((angle(G,data[lo]) < a) or (angle(G,data[hi]) > a)) →
        flag := false
        ;do (lo < hi+1) and (not flag) →
             if (lo ≤ hi) →
                mid := ((lo+hi) div 2)
             [] (lo > hi) →
                mid := ((lo+hi+size) div 2) mod size
             fi
             ;am := angle(G,data[mid])
             ;if (am > a) → hi := mid
              [] (am < a) → lo := mid
              [] (am = a) → flag := true
              fi
         od
        ;if (flag) → u := mid
         [] (not flag) → u := lo { lo ou hi, au choix }
         fi
  fi
 fi ]|
```

Algorithme 5.8 – Angle enveloppant 2.0, calcul de u

La seconde propriété se démontre comme suit. Nous allons travailler dans le quadrant opposé par symétrie centrale autour du centre de gravité par rapport au quadrant où est situé l'observateur.

Considérons deux sommets remarquables du quadrant opposé :

- le sommet d'abscisse extrémale parmi les sommets du quadrant opposé (maximale si le quadrant opposé est à l'Est, minimale s'il est à l'Ouest),
- le sommet d'ordonnée extrémale parmi les sommets du quadrant opposé (maximale si le quadrant opposé est au Nord, minimale s'il est au Sud).

On peut commenter la figure 5.12 comme suit. Introduisons le concept de quadrilatère caché : il s'agit du quadrilatère passant par le centre de gravité de la boîte couvrante minimum, les deux sommets remarquables et le sommet de la boîte couvrante minimum situé dans le même quadrant.

Le quadrilatère caché délimite le lieu du plan où aucun élément ne peut être visible de l'observateur, comme cela est illustré sur la figure 5.12.

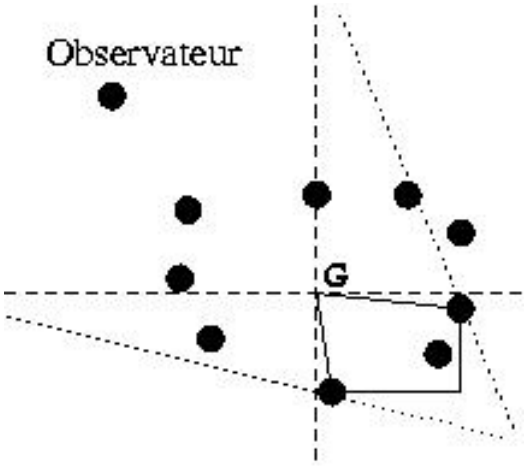


FIG. 5.12 – Quadrilatère caché et sommets remarquables du quadrant opposé à celui de l'observateur

On peut démontrer cela comme suit. Les deux droites en traits pointillés sur la figure 5.12 représentent les lignes de vue d'observateurs fictifs placés dans le quadrant de l'observateur réel. Ces observateurs fictifs, même avec une de leurs coordonnées à l'infini, ne pourraient avoir de ligne de vue intersectant l'intérieur du quadrilatère caché.

En effet, les deux sommets remarquables - qui sont aussi des sommets de l'enveloppe convexe - ont la propriété d'être extrémaux par rapport aux autres sommets situés dans le quadrilatère caché.

Ainsi, n'importe lequel des sommets internes au quadrilatère caché est forcément caché de l'observateur.

On peut donc calculer un sommet caché en temps constant. z peut d'ailleurs être précalculé lors du partitionnement de l'enveloppe convexe (cf. `UQP`).

Angle enveloppant 2.0, calcul de z

compute_z(source : Vertex ; iquad_opp : Quadrant ; **var** z : int)

|[z := iquad_opp.z]|

Algorithme 5.9 – Angle enveloppant 2.0, calcul de z

Il ne faut toutefois pas perdre de vue les cas particuliers suivants :

- la séquence de sommets internes au quadrilatère caché est vide ;
- les sommets remarquables sont confondus (c'est-à-dire qu'un seul sommet présente simultanément les caractéristiques des deux sommets remarquables) ;
- il n'y a pas de sommet remarquable.

Nous choisissons de ne pas calculer l'angle enveloppant par une recherche dichotomique lorsqu'un de ces cas particuliers se présente. En effet, le calcul de u, bien que performant, n'est pas négligeable. Si on doit de plus effectuer encore une recherche dichotomique pour trouver z, les gains temporels que l'on obtiendrait en pratique seraient moins importants. La propriété intéressante des enveloppes convexes est que le nombre de noeuds les composant est petit. Dès lors, effectuer plusieurs recherches dichotomiques et effectuer de nombreux tests ne sera pas tellement moins coûteux, en pratique et pour des tailles d'enveloppe convexe peu importantes, qu'une simple recherche linéaire.

Pour conclure, présentons l'algorithme de calcul d'angle enveloppant décidant s'il convient d'appliquer la recherche linéaire ou la recherche dichotomique.

Angle enveloppant 2.0

```
computeWrappingAngle(source : Vertex ; quads : array[0..3] of Quadrant;
                     data : array[0 .. size−1] of Vertex ; size : int ;
                     var ileft : int ; var iright : int ) : boolean
{ Les quadrants (précalculés) sont stockés dans quads (N−E, N−W, S−W, S−E) }

var
  u, z, iquad, iquad_opp, iquad_next, iquad_prev : int;

|[ { Détermination du quadrant courant }
 if (source.x ≥ G.x) →
    if (source.y ≥ G.y) → { NE } iquad,iquad_opp,iquad_next,iquad_prev := 1,3,2,4
    [] (source.y ≤ G.y) → { SE } iquad,iquad_opp,iquad_next,iquad_prev := 4,2,1,3
    fi
 [] (source.x ≤ G.x) →
    if (source.y ≥ G.y) → { NW } iquad,iquad_opp,iquad_next,iquad_prev := 2,4,3,1
    [] (source.y ≤ G.y) → { SW } iquad,iquad_opp,iquad_next,iquad_prev := 3,1,4,2
    fi
 fi

 ;compute_z(iquad_opp,z)
 ; if (z > −1) →
     compute_u(source,G,quads[iquad−1],data,size,u)
     ; if (u > −1) → skip
      [] (u = −1) → { Le quadrant est vide ! }
         { On est certain que (quads[iquad_next−1].qsize > 0)
           et que data[quads[iquad_next−1]. i_first] est visible
           car 2 quadrants voisins ne peuvent être simultanément vides. }
         u := quads[iquad_next−1]. i_first
      fi
     ;computeWrappingAngle :=
        computeWrappingAngle_bin(u,z,source,data,size,ileft, iright )
  [] (z = −1) → { Potentiellement tous visibles ! }
     computeWrappingAngle :=
        computeWrappingAngle_lin(source,data,size,ileft, iright )
 fi ]|
```

Algorithme 5.10 – Angle enveloppant 2.0

La complexité attendue et dans le pire cas pour calculer l'angle enveloppant est O(calcul de u+ calcul de z+recherche dichotomique), soit $O(\log N)$, où N représente le nombre de sommets de l'enveloppe convexe. La complexité du partitionnement `UQP` de chaque obstacle - réalisé lors d'une phase d'initialisation - est $O(N)$.

Etant donné les calculs d'aire signée qui interviennent, les algorithmes proposés pour le calcul d'angles enveloppants supposent qu'il y a au moins 3 sommets à traiter dans l'obstacle.

5.2.4 Perspectives

Une autre voie de recherches envisageable serait d'étudier comment accélérer les requêtes de visibilité en fonction des résultats des requêtes précédentes : en effet, deux tableaux indiquant la visibilité des sommets d'une enveloppe convexe à partir de deux points sources voisins seront souvent peu différents. Il serait donc intéressant, en supplément, d'introduire des méthodes heuristiques basées sur des probabilités et visant à réduire le nombre d'éléments à balayer.

Dans une autre perspective encore (figure 5.13), remarquons que les deux sommets recherchés sont les voisins du point source dans l'enveloppe convexe de l'ensemble de sommets de l'obstacle augmenté du point source. Evidemment, reconvexifier l'obstacle après ajout du point source n'est pas la méthode la plus efficace (complexité attendue $O(N \log N)$, où N représente le nombre de sommets de l'enveloppe convexe).

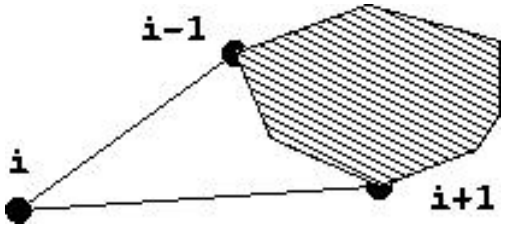


FIG. 5.13 – Angle enveloppant déduit de l'opération de reconvexification de l'obstacle après ajout du point source

---

Ensuite, indépendamment du contexte de calcul d'angle enveloppant, une version généralisée du problème peut être posée comme suit : détecter les indices délimitant les n zones d'un tableau (cf. figure 5.14).

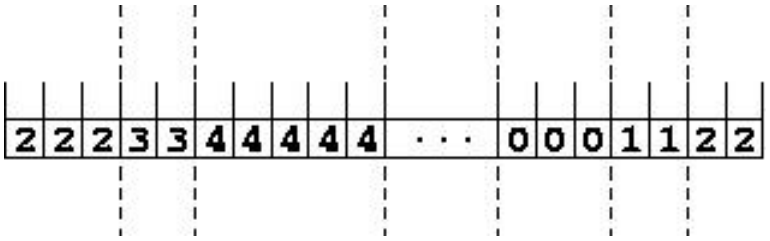


FIG. 5.14 – Généralisation du problème de détection de zones contiguës

Un contexte, géométrique ou autre, devrait absolument être défini de manière à estimer, pour chaque recherche des indices délimitant une zone, des bornes délimitant

des paires de séquences homogènes d'éléments.

Cela constitue un problème connexe qui ne sera pas étudié dans ce document mais qui méritait d'être mentionné.

5.2.5 Requête de visibilité

Il reste à développer un algorithme testant si un obstacle donné (appartenant à un ensemble d'obstacles) limite la portion du champ de visibilité de l'élément source dans laquelle se trouve l'élément destination, autrement dit **si cet obstacle doit être contourné**.

Cet algorithme repose sur la méthode proposée en début de section : après avoir déterminé les points de tangence de l'angle enveloppant, vérifier si l'angle cible est situé dans l'intervalle délimité par les angles bornants.

Puis, si l'angle cible est situé dans l'intervalle délimité par les angles bornants, il faut encore vérifier si l'élément destination est situé derrière ou devant l'obstacle pour conclure que celui-ci est bloquant ou ne l'est pas.

Un test de type *beneath/beyond* doit donc être effectué pour chaque arc visible de l'obstacle. Les arcs visibles d'un obstacle donné sont représentés sur la figure 5.15 et le test de type *beneath/beyond* est inclus dans les annexes.

En résumé, pour chaque barrière, on effectue un calcul d'angle enveloppant et, le cas échéant, un test *beneath/beyond* pour chaque segment de l'enveloppe convexe approximant l'obstacle.

La complexité dans le pire cas de cet algorithme est donc $O(N_b \times (\log N_h + N_h))$, où N_b est le nombre de barrières et N_h le nombre moyen (dans une perspective d'analyse amortie) de sommets dans une enveloppe convexe approximant un obstacle. Cette complexité peut se simplifier en $O(N_b\ N_h)$, c'est-à-dire le nombre total de sommets des enveloppes convexes approximant les obstacles, que nous nommerons N_{bh}.

`isAvoidanceRequired`

```
isAvoidanceRequired(v_src : Vertex ; v_dest : Vertex ;
                    regions : array[0 .. rs−1] of Region ;
                    rs : int ) : boolean

var
  i , ileft , iright : int ;
  targetAngle : float ;
  b, flag : boolean;

|[ targetAngle := angle(v_src,v_dest)
 ; i , flag := 0, false
 ;do ((i < rs) and (not flag)) →
     b := computeWrappingAngle(v_src,regions[i].data, regions[i] . size , ileft , iright )
     ; if (not b) → flag := true { Aucun sommet visible }
          { Ce cas sera justifié en fin de chapitre }
       [] (b) →
          if ((targetAngle ≥ angle(v_src, regions[i] . data[ileft] )) and
              (targetAngle ≤ angle(v_src, regions[i] . data[iright] ))) →
             if (isBarringRegion(v_src, v_dest , regions[i] , ileft , iright )) →
                flag := true
             [] (not . . .) → skip
             fi
          [] (not . . .) → skip
          fi
          ; i := i+1
     fi
 od
 ;isAvoidanceRequired := flag ]|
```

Algorithme 5.11 – `isAvoidanceRequired`

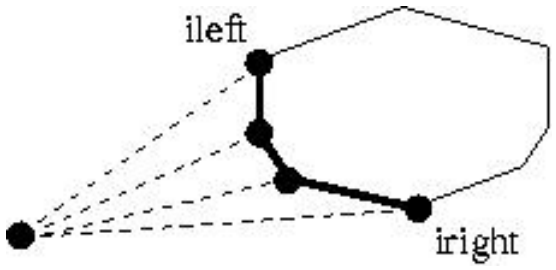


FIG. 5.15 – Arcs visibles

`isBarringRegion`

```
isBarringRegion(v_src : Vertex ; v_dest : Vertex ;
                data : array[0 .. size−1] of Vertex ; size : int ;
                ileft : int ; iright : int ) : boolean
var
  i : int ;
  flag : boolean;
|[ flag := false
 ; if ( ileft ≤ iright) →
      i := ileft
      ;do ((i < iright ) and (not flag)) →
          if (isBarringEdge(v_src,v_dest , data[i] ,data[(i+1) mod size])) →
             flag := true
          [] (not . . .) → skip
          fi
          ;i := i+1
      od
   [] ( ileft > iright ) →
      i := ileft
      ;do ((i < size ) and (not flag)) →
          if (isBarringEdge(v_src,v_dest , data[i] ,data[(i+1) mod size])) →
             flag := true
          [] (not . . .) → skip
          fi
          ;i := i+1
      od
      ;i := 0
      ;do ((i < iright ) and (not flag)) →
          if (isBarringEdge(v_src,v_dest , data[i] , data[i+1])) →
             flag := true
          [] (not . . .) → skip
          fi
          ;i := i+1
      od
 fi
;isBarringRegion := flag ]|
```

Algorithme 5.12 – `isBarringRegion`

5.3 Graphe de visibilité

5.3.1 Intérêt du graphe de visibilité

Qu'est-ce que le graphe de visibilité d'un ensemble d'éléments ?

En tant que graphe, c'est une relation des éléments de cet ensemble vers les éléments de cet ensemble. La relation d'un élément vers un autre élément est définie si le second est visible à partir du premier suivant le modèle de visibilité qui a été établi.

Les noeuds du graphe de visibilité sont les sommets des enveloppes convexes approximant chaque région. Les arcs (non orientés) du graphe de visibilité relient les noeuds (couples d'éléments) mutuellement visibles selon la définition qui vient d'être proposée.

La figure 5.16 représente plusieurs obstacles ainsi que le graphe de visibilité correspondant. En particulier, les zones hachurées représentent les obstacles et les arcs en pointillés représentent les segments de droite (forcément inclus dans l'ensemble des arcs du graphe de visibilité) constituant les obstacles.

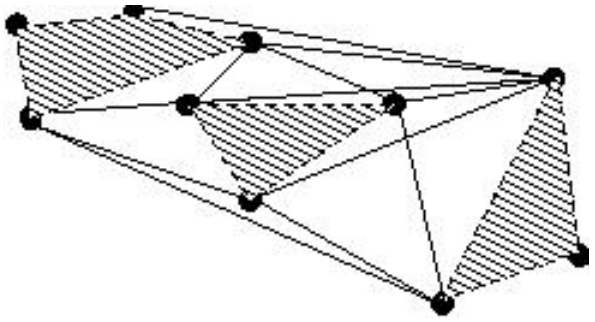


FIG. 5.16 – Graphe de visibilité d'un ensemble d'obstacles

L'**intérêt du graphe de visibilité**, lorsque des obstacles doivent être contournés, est le suivant : tout chemin géodésique discret le plus court, empruntera forcément les voies du réseau de déplacement optimal qu'est le graphe de visibilité.

En effet, chaque segment d'un chemin le plus court est un segment de droite (puisque tout chemin optimal est localement optimal) et que les points de jonction des segments d'un chemin le plus court doivent être des sommets des enveloppes convexes approximant les obstacles (par définition d'un chemin le plus court). Or, les noeuds du graphe de visibilité sont précisément les sommets des enveloppes convexes approximant les obstacles.

5.3.2 Construction du graphe de visibilité

Il existe plusieurs méthodes de construction d'un graphe de visibilité. Les méthodes directes sont cubiques : leur complexité dans le pire cas est $O(V^3)$, V représentant le nombre de noeuds potentiels du graphe de visibilité, c'est-à-dire, dans le contexte de ce document, le nombre total de sommets des enveloppes convexes approximant les obstacles. Le raisonnement est basé sur une énumération exhaustive de tous les segments de tous les obstacles pour vérifier si chacun d'eux est bloquant ou non.

Toutefois, plusieurs algorithmes [DV+00, Mou02] de construction d'un graphe de visibilité sont basés sur un balayage angulaire de l'espace des points constituant les sommets du graphe.

Pour construire tous les arcs de visibilité issus d'un point source donné, on effectue des tests d'intersection avec les segments de droite constituant les enveloppes convexes approximant les obstacles. La différence avec une approche directe est que l'on n'effectue qu'un nombre restreint de tests d'intersection.

Pour cela, on maintient en permanence un ensemble des segments potentiellement bloquants : en parcourant par ordre angulaire tous les noeuds (figure 5.17), il vient que les segments qui sont totalement inclus dans la partie balayée ne sont certainement plus bloquants. En implémentant cet ensemble à l'aide d'un arbre binaire équilibré (en utilisant l'ordre angulaire des noeuds comme clé), on pourra effectuer des requêtes de coût $O(\log V)$ pour trouver le plus proche segment bloquant. Le coût de balayage de tous les noeuds à partir du noeud source est donc $O(V \log V)$.

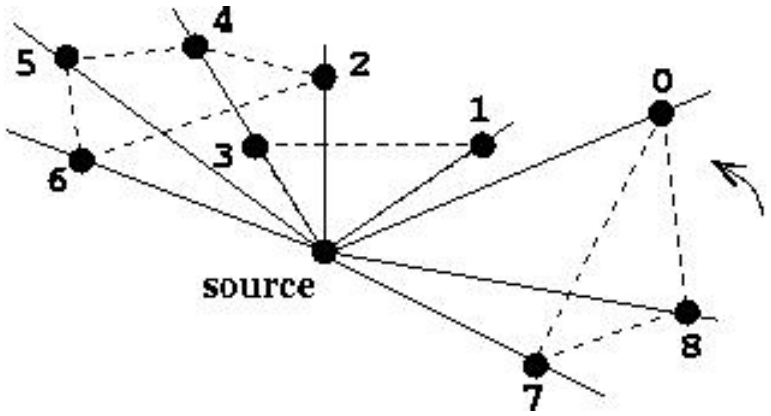


FIG. 5.17 – Calcul des arcs de visibilité issus du noeud source vers tous les autres, en effectuant un balayage angulaire (obstacles représentés en pointillés)

Puisqu'il faut construire des arcs de visibilité pour V noeuds, de tels algorithmes exhibent une complexité attendue et dans le pire cas de $O(V^2 \log V)$.

Cette complexité peut être réduite à $O(V^2)$ en utilisant un raisonnement basé sur des arguments de dualité [Ede87, Mou02, Wel85] entre un ensemble d'équations de droites et un ensemble de coordonnées de points.

Remarquons que le fait que ces algorithmes soient de complexité quadratique n'est pas surprenant puisque la taille du résultat est bornée par V^2.

L'implémentation d'algorithmes basés sur ces concepts est délicate car, comme souvent dans le domaine de la géométrie algorithmique, de nombreux cas non standards ou dégénérés peuvent se présenter et doivent être traités de manière particulière.

Des algorithmes exhibant une complexité attendue plus restreinte mais cependant sensible à la taille des résultats obtenus (*output-sensitive algorithms*) sont également connus [GM91]. Leur complexité attendue peut diminuer jusque $O(E + V \log V)$, E représentant le nombre d'arcs du graphe de visibilité résultant des calculs effectués. Leur complexité dans le pire cas est toutefois $O(V^2 + V \log V)$.

Ces algorithmes sont souvent basés sur l'idée d'une triangulation (triangulation de Delaunay complète ou partielle ou d'autres types de triangulation) de l'espace où sont situés les noeuds du graphe de visibilité. L'originalité de ces méthodes réside dans la manière d'effectuer des opérations sur le graphe dual de la triangulation calculée pour obtenir le graphe de visibilité.

De plus, l'implémentation de ces algorithmes n'est pas, parfois de l'aveu même de leurs auteurs [GM91], aisée à réaliser, même en se plaçant dans le contexte de la géométrie algorithmique. Il s'agit même d'un véritable problème en soi dans certains cas.

Pour terminer, en observant que le calcul du graphe de visibilité dans le cadre de `DGSP` est effectué une seule fois au cours d'une phase d'initialisation et ne constitue pas de goulot d'étranglement (*bottleneck*), nous avons implémenté une méthode directe (de complexité cubique), testant pour chaque couple de noeuds si chaque segment de l'ensemble de toutes les enveloppes convexes approximant les obstacles est bloquant ou non.

Un filtrage est toutefois effectué, permettant de conclure directement que tout couple de noeuds éloignés d'une distance $> R$ ne sont pas visibles. Dès lors, le nombre de segments dont il faut vérifier s'ils sont bloquants ou non est en pratique fort limité ($< 10\%$).

5.4 Calcul d'un chemin le plus court

L'espace de recherche ayant été reconditionné, montrons maintenant comment en tirer parti.

Si un ou plusieurs obstacles sont présents entre les éléments source et destination considérés, un calcul de chemin le plus court est nécessaire de manière à calculer le chemin optimal contournant ces obstacles.

Cette section suppose donc qu'une requête de visibilité a révélé la présence d'obstacles entre source et destination.

5.4.1 Algorithme de Dijkstra

Le chemin le plus court d'un élément vers un autre élément, en évitant les barrières, passe par le graphe de visibilité calculé à partir de la matrice des barrières, tel que précédemment mis en évidence.

Si l'élément source et l'élément destination sont tous deux des noeuds du graphe de visibilité, un chemin peut être recherché directement dans le graphe de visibilité.

Si, par contre, l'élément source n'est pas un noeud du graphe de visibilité, il faut ajouter temporairement (le temps d'effectuer le calcul du chemin le plus court) à celui-ci tous les arcs de visibilité qui le relient à tous les noeuds du graphe de visibilité. Il faut faire de même avec l'élément destination du chemin si celui-ci n'est pas non plus un noeud du graphe de visibilité.

Dans l'un ou l'autre cas où des arcs de visibilité sont ajoutés temporairement, c'est dans le **graphe de visibilité augmenté** de ceux-ci que le chemin le plus court doit être recherché.

L'algorithme classique pour résoudre un problème de calcul de chemins les plus courts dans un graphe dirigé - dont les valeurs des arcs sont non négatives - est celui de Dijkstra [CLR92]. Etant donné le contexte dans lequel ce problème apparaît, cet algorithme performant et de complexité dans le pire cas $O(E \log V)$ est adéquat (pour rappel, V est le nombre de noeuds du graphe et E est le nombre d'arcs).

5.4.2 Ensemble-frontière

L'algorithme de Dijkstra calcule le chemin le plus court du noeud source donné vers tous les autres noeuds du graphe auquel il est appliqué.

C'est un algorithme qui fonctionne par marquages provisoire et définitif. Il examine tous les noeuds du graphe comme suit : à chaque itération, il marque définitivement le noeud marqué provisoirement qui est le plus proche (en distance) du noeud source. Pour chaque voisin du noeud qui vient d'être définitivement marqué, il met à jour la distance qui les sépare du noeud source. Cette mise à jour consiste soit à conserver la distance la plus courte déjà calculée, soit à remplacer celle-ci par la distance - plus courte - obtenue en effectuant un détour par le noeud définitivement marqué.

L'algorithme de Dijkstra introduit le concept d'**ensemble-frontière** : il s'agit de l'ensemble de noeuds provisoirement marqués mais pas encore définitivement marqués. On connaît pour ces noeuds marqués non définitivement une distance à partir du noeud source, mais on n'est pas encore certain que ce soit la plus courte. Le concept d'ensemble-frontière est donc un concept d'ensemble de noeuds pour lesquels une distance du chemin provenant du noeud source est connue mais provisoire (car non définitive et peut-être pas optimale).

Plusieurs opérations d'accès à l'ensemble-frontière sont définies :

- retirer de l'ensemble le noeud le plus proche du noeud source (marquage définitif) ;
- insérer dans l'ensemble un noeud (marquage provisoire) ;
- mettre à jour la distance séparant un noeud du noeud source lorsque celle-ci a pu être améliorée en effectuant un détour par le noeud de ses voisins qui vient d'être définitivement marqué.

Soient deux vecteurs SP et VAL, de mêmes dimensions que le nombre de noeuds du graphe. $SP[i]$ représente la longueur du chemin optimal entre le noeud source et le noeud i. On a $SP[i] < 0$ si un tel chemin n'est pas défini. $VAL[i]$ représente la longueur d'un chemin (peut-être non optimal) entre le noeud source et le noeud i, le meilleur parmi les chemins déjà évalués pour le noeud i. On a $VAL[i] = 0$ si un tel chemin n'est pas défini (par construction du graphe de visibilité sur base des sommets des enveloppes convexes approximant les obstacles, il n'y a pas de noeuds confondus, de mêmes coordonnées).

Décrivons maintenant l'algorithme proprement dit. Il commence par une boucle d'initialisation qui marque provisoirement chaque noeud voisin du noeud source en les plaçant dans l'ensemble-frontière (initialement vide).

La boucle principale de l'algorithme a pour invariant :

$\mathcal{SP} = \{i \mid SP[i] \geq 0\}$ est l'ensemble des noeuds pour lesquels un chemin optimal issu du noeud source a été définitivement calculé. C'est l'ensemble des noeuds définitivement marqués.

$\mathcal{F} = \{i \mid (SP[i] < 0)\ and\ (VAL[i] > 0)\}$ est l'ensemble des noeuds pour lesquels un chemin provisoire (non nécessairement le plus court) a déjà été calculé. C'est l'ensemble des noeuds provisoirement marqués.

$\mathcal{U} = \{i \mid (SP[i] < 0)\ and\ (VAL[i] = 0)\}$ est l'ensemble des noeuds pour lesquels un chemin n'a pas encore été calculé. C'est l'ensemble des noeuds non encore marqués.

Ces 3 ensembles $\mathcal{SP}, \mathcal{F}, \mathcal{U}$ forment une partition de l'ensemble des noeuds accessibles (directement et indirectement) à partir du noeud source. La boucle va déplacer, à chaque itération :

- 1 noeud de $\mathcal{F}$ vers $\mathcal{SP}$
- 0, 1 ou plusieurs noeuds de $\mathcal{U}$ vers $\mathcal{F}$

La boucle s'arrête quand $\mathcal{F}$ est vide, c'est-à-dire quand tous les noeuds sont dans $\mathcal{SP}$.

La terminaison de la boucle principale est donc assurée car, étant donné qu'une insertion dans l'ensemble-frontière est réalisée pour un noeud non encore présent dans celui-ci (il y a donc au plus V insertions) et que, à chaque itération, on en retire un noeud (c'est-à-dire V suppressions car il y a au plus V noeuds dans le graphe et qu'un noeud est inséré une seule fois dans l'ensemble-frontière).

Les mises à jour n'influencent pas le nombre d'éléments de l'ensemble-frontière et n'influencent pas la terminaison de la boucle principale. On déduit le nombre d'itérations de la boucle principale en comptant le nombre d'insertions et de suppressions de noeuds de l'ensemble-frontière. Comme déjà montré, ce nombre est borné et l'algorithme se termine.

Par ailleurs, les mises à jour, qui n'ont pas d'impact sur la taille de l'ensemble-frontière, influencent par contre la complexité de l'algorithme. Elles sont en nombre E au maximum, dans le cas d'un graphe complètement connecté.

La complexité de l'algorithme dans le pire cas est $O(E\times$ coût de mise à jour$) + O(V\times$ coût d'insertion$) + O(V\times$ coût de suppression$)$.

Une structure de données classique convenant parfaitement pour implémenter les opérations relatives à l'ensemble-frontière est le **tas (*binary heap*)**.

Un tas est (conceptuellement) un arbre binaire équilibré dont tous les noeuds respectent la *propriété de tas* suivante : la clé de tout noeud n'est pas plus grande que celle d'aucun de ses descendants. La clé sélectionnée dans le contexte de l'algorithme de Dijkstra est la longueur du meilleur chemin provisioire reliant le noeud au noeud source : c'est donc $VAL[i]$ pour le noeud i.

Les opérations sur le tas sont décrites dans les annexes. Notons que celles-ci sont nommées `heap_delete_min` (suppression du noeud de plus petite clé, le plus proche du noeud source parmi les noeuds présents dans $\mathcal{F}$), `heap_insert` (insertion d'un noeud dans $\mathcal{F}$) et `heap_decrease_key` (mise à jour de la clé d'un noeud de $\mathcal{F}$).

En utilisant un tas (*binary heap*), on peut garantir un coût de $O(\log V)$ pour chacune des opérations d'insertion, mise à jour et suppression. Cela conduit à une complexité dans le pire cas de $O(E \log V + V \log V)$.

Avant de poursuivre, on peut observer que seuls les chemins de longueur inférieure à R (le rayon d'accessibilité) sont intéressants. Dès lors, un débranchement anticipé de la boucle principale de l'algorithme peut être effectué dès qu'un chemin de longueur supérieure à R est calculé. La complexité attendue de l'algorithme devient donc $O(U \log V)$, avec $V \leq U \leq E$.

En pratique, pour les données à notre disposition et un rayon d'accessibilité raisonnable ($R = 32$), le nombre de chemins de longueur supérieure à R constituent 90% à 95% des chemins. **La complexité observée de l'algorithme est donc beaucoup plus proche de $O(V \log V)$ que $O(E \log V)$.**

Cette complexité dépend évidemment de R mais également de la dispersion des obstacles : nous pensons que si l'écart-type des distances entre couples de noeuds du graphe de visibilité est faible (c'est-à-dire lorsque les obstacles sont uniformément répartis), la complexité de l'algorithme sera certainement fonction de R^2. Elle sera sans doute beaucoup plus importante si les obstacles sont groupés en clusters (dès que R augmente un peu, la longueur de 100% des chemins devient inférieure à R). Nous n'avons pas pu vérifier cette proposition car la quantité de données à notre disposition n'est pas suffisament importante pour tirer des conclusions fiables à ce sujet.

Ainsi, l'hypothèse que R sera souvent petit a été mise à profit et permet de diminuer fortement la complexité de l'algorithme de recherche des chemins les plus courts.

Par ailleurs et avant de clôturer la discussion menée au sujet des calculs de chemins les plus courts, mentionnons l'existence d'autres structures de données [CLR92, FT87, Sau99] permettant une implémentation de l'ensemble-frontière plus performante que le *binary heap* : il s'agit des *Fibonacci heap* et *2-3 heap*.

En se basant sur une analyse amortie, ces structures exhibent des coûts constants pour les opérations d'insertion et de mise à jour et un coût logarithmique $O(\log V)$ pour l'opération de suppression. Leur principe repose essentiellement sur une structuration plus souple que les *binary heaps*.

Cela conduit à une complexité amortie de $O(E + V \log V)$.

Toutefois, l'observation suivante [Sau99] ne devrait pas être négligée : dans le cas de graphes de petite taille ou de faible connectivité, une bonne implémentation d'un *binary heap* pourrait, en pratique, conduire à des performances meilleures que ce qu'on observerait avec d'autres types de tas. En effet, l'implémentation de ces structures de données alternatives peut être relativement lourde et ne se montrer intéressante en pratique que de manière asymptotique (pour de grandes tailles de données) ...

Pour les données à notre disposition, la taille du graphe (< 10000 noeuds) et sa connectivité ($< 10\%$) sont petites, permettant de prendre en considération l'observation de [Sau99].

De plus, il sera montré qu'il est possible de précalculer une bonne partie des chemins les plus courts : le calcul de ceux-ci ne constituera donc pas in fine un goulot d'étranglement (*bottleneck*) dans la résolution de `DGSP`.

Algorithme de Dijkstra

```
type Node = record
  begin
  id   : integer; { Identificateur unique }
  nc   : integer; { Nombre de voisins }
  nb   : array[0 .. nc−1] of Node; { Voisins }
  { Il existe VGSIZE variables de type Node, une par noeud existant }
  end;

ShortestPaths(s : Node ; var SP : array[0 .. VGSIZE−1] of float ; VGSIZE : int)
var
  v : Node;
  i , size : integer;
  VAL : array[0..VGSIZE−1] of float;
  oldval , newval : float ;
|[ i := 0 ;do i < VGSIZE → SP[i] := −1.0 ;VAL[i] := 0.0 ;i:= i+1 od

;SP[s. id] , size , i := 0.0,0,0
;do i < s.nc → { Insérer les voisins du noeud source }
    VAL[s.nb[i] . id] := distance(s,s . nb[i] )
    heap_insert(VAL[s.nb[i] . id] ,s . nb[i] )
    ; size , i := size+1,i+1
 od

;do (size > 0) → { Tant qu'il reste des noeuds pas encore calculés }
    heap_delete_min(v) { Extraire le plus proche noeud pas encore calculé }
    ; size := size−1
    ; if (VAL[v.id] > R) → size := 0 { Les chemins ≤ R sont tous calculés }
     [] (VAL[v.id] ≤ R) →
        SP[v. id] := VAL[v.id]
        ; i := 0
        ;do i < v.nc → { Traiter les voisins du noeud définitivement marqué }
            if SP[v. nb[i] . id] = −1.0 → { Voisin pas encore définitivement marqué }
               oldval := VAL[v.nb[i]. id]
               ;newval := VAL[n.id]+distance(v,v.nb[i])
               ; if oldval = 0.0 → { Voisin pas encore marqué }
                    VAL[v.nb[i]. id] := newval
                    ; size := size+1
                    ; heap_insert(newval,v. nb[i] )
                 [] oldval ≠ 0.0 → { Voisin déjà provisoirement marqué }
                    if newval < oldval →
                       VAL[v.nb[i]. id] = newval
                       ;heap_decrease_key(newval,v.nb[i] )
                    [] newval ≥ oldval → skip
                    fi
               fi
            [] SP[v. nb[i] . id] > −1.0 → skip { Voisin déjà calculé }
            fi
            ; i := i+1
        od
    fi
 od ]|
```

Algorithme 5.13 – Algorithme de Dijkstra

5.4.3 Une solution intermédiaire pour la prise en compte de la troisième dimension spatiale

La situation représentée sur la figure 2.15 pose un problème que nous avions choisi de ne pas résoudre intialement, à savoir le fait que l'on ne tient pas compte de situations où des différences d'altitude supérieures au seuil autorisé $\bar{Z}$ se produisent entre des points intermédiaires d'un chemin, mais pas entre les points source et destination.

Cela a pour conséquence que les chemins calculés seront, dans cette situation, plus courts qu'en réalité.

Après une discussion avec notre promoteur, il est apparu qu'une solution intermédiaire existait pour résoudre ce problème.

Nous avions initialement suggéré d'effectuer des tests d'altitude entre plusieurs points de contrôle intermédiaires.

En fait, il est apparu que les noeuds du graphe de visibilité constituent de bons candidats pour jouer le rôle de points de contrôle. Mieux, il existe un moyen de prendre en considération les différences d'altitude entre les différents noeuds consécutifs d'un chemin sans modifier la complexité des algorithmes de calcul de chemin le plus court présentés.

Il s'agit de remplacer la longueur de chaque arc du graphe de visibilité. En effet, cette longueur est la distance entre les noeuds de cet arc dans un espace à deux dimensions. L'idée est de la remplacer par la distance entre les noeuds dans un espace à trois dimensions, ce qui permet de prendre en compte la différence d'altitude qui existe entre ces noeuds (figure 5.18).

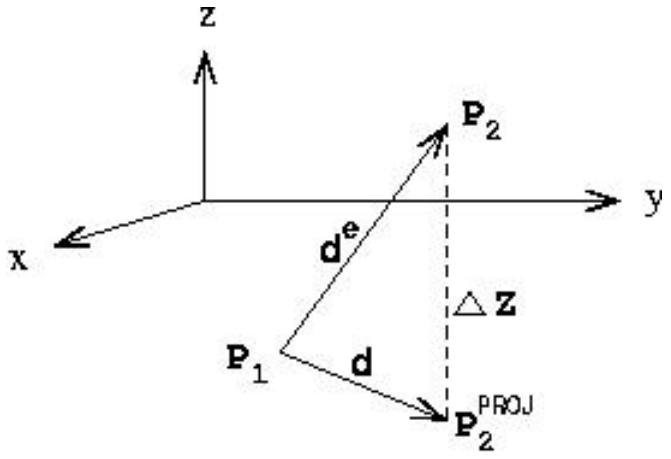


FIG. 5.18 – Distance dans un espace à 3 dimensions

d est la distance classique entre P_1 et la version P_2^{PROJ} de P_2 projetée dans le plan de même élévation que P_1.

d^e est la distance non projetée entre P_1 et P_2. On a : $d^e = \sqrt{(d)^2 + (\Delta Z)^2}$.

Il est donc très simple de remplacer d par d^e.

Evidemment, un problème subsiste : les noeuds du graphe de visibilité ne sont peut-être pas les meilleurs points de contrôle qu'il est possible de prendre en considération et des différences d'altitude inacceptables pourraient tout de même se produire entre les différents noeuds d'un chemin.

Toutefois, nous pensons qu'il s'agit d'une solution élégante, très performante et, finalement, intermédiaire entre le fait de ne pas prendre en considération le problème et étudier longuement le placement de points de contrôle.

5.5 Levée des hypothèses relatives aux obstacles

5.5.1 Hypothèse de masquage de l'intérieur des obstacles

Il a été décidé dans un premier temps d'ignorer complètement l'intérieur des obstacles.

Comment en effet un observateur peut-il contourner un obstacle à l'intérieur duquel il se trouve ? Une solution simple et raisonnable est de considérer que tous les sommets de l'enveloppe approximant l'obstacle sont visibles par l'observateur. Ainsi, le contournement de l'enveloppe convexe qui l'entoure se fera en suivant les sommets de celle-ci.

En pratique, une bonne solution est d'agir au niveau des requêtes de visibilité : les requêtes de calcul d'angle enveloppant retourneront une valeur spéciale pour indiquer que l'élément donné est interne à l'obstacle donné. Ainsi, l'algorithme `isAvoidanceRequired` présenté précédemment n'effectue pas de tests *beneath/beyond* et conclut directement que des obstacles doivent être contournés.

Cette solution a l'avantage de ne présenter aucun coût spatial ou temporel supplémentaire.

Par ailleurs, seule une étude des propriétés des polygones non convexes permettrait d'améliorer considérablement le degré de précision du cheminement interne à un obstacle. Si la résolution des données était beaucoup plus grande, cela deviendrait impératif, mais, vu que ce n'est pas le cas, cela sort du cadre de cette étude.

5.5.2 Hypothèse de disjonction spatiale des obstacles

La **possibilité que les enveloppes convexes approximant deux (ou plusieurs) obstacles ne soient pas disjointes** (illustration à la figure 5.19) a été écartée en première analyse alors que les éléments non accessibles des deux obstacles sont tout à fait disjoints.

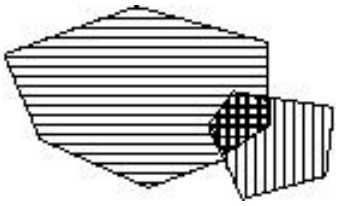


FIG. 5.19 – Enveloppes convexes non disjointes

En effet, lorsqu'un élément est situé dans l'intersection de plusieurs enveloppes convexes, un problème pourrait se poser pour déterminer comment effectuer les contournements nécessaires.

Une solution radicale serait de segmenter les enveloppes convexes en plusieurs enveloppes convexes disjointes de manière à obtenir des obstacles parfaitement disjoints. Cela aurait pour conséquence qu'un obstacle serait approximé simultanément par plusieurs enveloppes ... et que de nombreux éléments devraient être intégrés au graphe de visibilité, ce qu'il est préférable d'éviter.

La solution sélectionnée est beaucoup plus simple : les arcs du polygone résultant de l'intersection des enveloppes convexes entourant l'élément donné sont tous bloquants et le champ de visibilité de celui-ci est donc indirectement nul. Tout chemin issu de cet élément passera donc par les sommets du polygone-intersection. La recherche d'un chemin, à partir d'un de ces sommets pourra, dès lors, être effectuée de la même manière que lorsqu'une seule enveloppe convexe entoure l'élément donné.

On peut donc conclure que les enveloppes convexes des obstacles ne doivent pas être nécessairement disjointes, grâce à la décison prise par ailleurs de lever l'hypothèse de masquage de l'intérieur des obstacles.

5.6 Conclusion

La structure d'un algorithme résolvant `DGSP` avait été proposée au chapitre précédent. Ce chapitre a montré comment reconditionner et exploiter l'espace de recherche et y calculer des chemins les plus courts.

Un **modèle de visibilité** a été présenté, permettant de développer des algorithmes pour déterminer si un coutournement de barrières est nécessaire pour une source et une destination données.

Le **concept d'angle enveloppant**, important pour la résolution du problème, a été introduit. Un algorithme répondant aux requêtes de visibilité a été développé. Il est-sous tendu par ce concept d'angle enveloppant. Remarquons que, comme souvent en géométrie algorithmique, plusieurs cas particuliers ont dû être pris en considération pour pouvoir résoudre un problème qui est a priori raisonnablement simple.

Le **graphe de visibilité**, une **structure de données reconditionnant l'espace de recherches**, a été présenté. C'est dans celui-ci que sont recherchés les chemins les plus courts.

Le très classique **algorithme de Dijkstra** est ensuite revisité, avec une attention particulière accordée à l'implémentation de l'**ensemble-frontière**, qui influence fortement la complexité totale de l'algorithme.

Avant de poursuivre, soulignons l'existence d'**algorithmes recherchant des chemins les plus courts dans un espace planaire comportant des obstacles**. La différence par rapport à l'approche envisagée dans ce document est qu'une construction explicite du graphe de visibilité n'est pas nécessaire. Le lecteur intéressé pourra consulter [HS99], [KMM97], [Mit98] et [MO01].

Ils ne seront pas présentés dans ce document étant donné qu'une exploitation de résultats intermédiaires ne semble pas être aisément réalisable (notre approche exploite massivement le graphe de visibilité), d'autant plus que les auteurs de ces algorithmes ont insisté essentiellement sur une étude de leur complexité sans véritablement discuter des aspects propres à leur programmation.

L'étude et la prise en considération de ces algorithmes seraient sans aucun doute très intéressantes et constitueraient une voie de recherches parallèle à ce qui a été envisagé dans ce document.

Le chapitre suivant étudiera plusieurs idées pour optimiser les requêtes de calcul de chemin le plus court.

Chapitre 6

Optimisation des requêtes

Nous allons maintenant optimiser les requêtes de calcul :

- en filtrant les barrières intervenant dans les requêtes de visibilité ;
- en précalculant un certain nombre de données en vue d'accélérer les requêtes de calcul d'un chemin le plus court.

6.1 Filtrage des barrières

L'algorithme `isAvoidanceRequired` présenté dans une section précédente permet de déterminer si un contournement d'obstacles est nécessaire pour établir un chemin entre un élément source et un élément destination.

Cette requête ne doit pas nécessairement être appliquée à tous les obstacles de B : seule une fraction de ceux-ci est réellement intéressante.

En effet, si un obstacle est éloigné de l'élément source ou de l'élément destination d'une distance plus grande que le rayon d'accessibilité R, il ne gênera en rien la visibilité entre source et destination, puisque `DGSP` n'est calculé que pour des éléments source et destination éloignés d'une distance inférieure à R.

Notons $\mathcal{B}_\cap$ (figure 6.1) l'ensemble des obstacles résultant de l'intersection

- de l'ensemble d'obstacles dont un ou plusieurs sommets est localisé dans le voisinage de la source ;
- et de l'ensemble d'obstacles se trouvant dans le voisinage de la destination.

Cet ensemble $\mathcal{B}_\cap$ est propre à chaque couple d'éléments source/destination.

On peut donc tirer parti de la notion de rayon d'accessibilité R en filtrant les obstacles à considérer par une requête de visibilité.

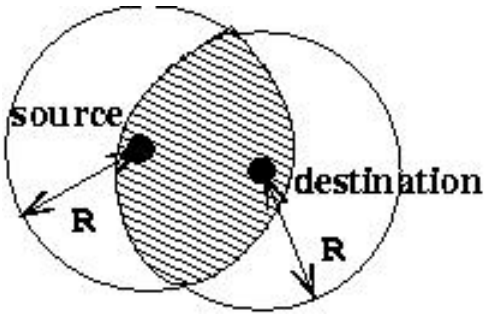


FIG. 6.1 – Limitation de l'espace des obstacles potentiellement bloquants à $\mathcal{B}_\cap$

Pour une source v_{src} et une destination v_{dest} fixées, comment calculer rapidement si un obstacle donné appartient à $\mathcal{B}_\cap(v_{src}, v_{dest})$?

6.1.1 Positionnement des barrières par rapport à la boîte couvrante minimum

Approximer chaque obstacle par sa boîte couvrante minimum permet de calculer rapidement si un obstacle appartient à $\mathcal{B}_\cap$.

La décomposition suivante (figure 6.2) de l'espace planaire selon des droites prolongeant les segments de la boîte couvrante minimum d'un obstacle permet de déterminer en très peu d'opérations si cet obstacle est situé non loin (distance $\leq R$) d'un élément donné. Les quatre sommets de la boîte couvrante minimum sont mis en évidence.

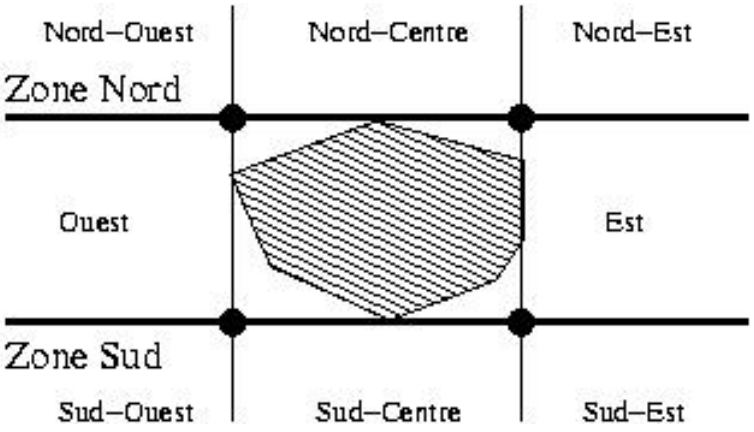


FIG. 6.2 – Décomposition du plan selon les segments d'une boîte couvrante minimum

Voici l'algorithme exploitant cette décomposition :

`isInsideBInter`

```
isInInsideBInter (v_src : Vertex ; v_dest : Vertex ;
                  bb_e : int ; bb_n : int ;
                  bb_w : int ; bb_s : int ) : boolean
var
  is_e , is_n , is_w , is_s : boolean;
|[ is_e,is_n := (v_src.x > bb_e),(v_src.y > bb_n)
 ;is_w, is_s := (v_src.x < bb_w),(v_src.y < bb_s)
 { v_src est-il proche de l'obstacle ? }
 ;if (is_n) → { Zone Nord }
10     if (is_w) → d := distance(v_src.x,v_src.y,bb_w,bb_n,d)
       [] (is_e) → d := distance(v_src.x,v_src.y,bb_e,bb_n,d)
       [] ((not is_e) and (not is_e)) → d := abs(v_src.y-bb_n)
       fi
  [] (is_s) → { Zone Sud }
       if (is_w) → d := distance(v_src.x,v_src.y,bb_w,bb_s,d)
       [] (is_e) → d := distance(v_src.x,v_src.y,bb_e,bb_s,d)
       [] ((not is_e) and (not is_e)) → d := abs(v_src.y-bb_s)
       fi
  [] ((not is_n) and (not is_s) and (is_e)) → { Est }
20     d := abs(v_src.x-bb_e)
  [] ((not is_n) and (not is_s) and (is_w)) → { Ouest }
     d := abs(v_src.x-bb_w)
  [] ((not is_n) and (not is_s) and (not is_e) and (not is_w)) →
     d := R { v_src est interne à la boîte couvrante }
  fi

 ;if (d > R) → isInInsideBInter := false
  [] (d ≤ R) →
     is_e , is_n := (v_dest.x > bb_e),(v_dest.y > bb_n)
30   ;is_w, is_s := (v_dest.x < bb_w),(v_dest.y < bb_s)
     { v_dest est-il proche de l'obstacle ? }
     ;if (is_n) → { Zone Nord }
          if (is_w) → d := distance(v_dest.x,v_dest.y,bb_w,bb_n,d)
          [] (is_e) → d := distance(v_dest.x,v_dest.y,bb_e,bb_n,d)
          [] ((not is_e) and (not is_e)) → d := abs(v_dest.y-bb_n)
          fi
       [] (is_s) → { Zone Sud }
          if (is_w) → d := distance(v_dest.x,v_dest.y,bb_w,bb_s,d)
          [] (is_e) → d := distance(v_dest.x,v_dest.y,bb_e,bb_s,d)
40        [] ((not is_e) and (not is_e)) → d := abs(v_dest.y-bb_s)
          fi
       [] ((not is_n) and (not is_s) and (is_e)) → { Est }
          d := abs(v_dest.x-bb_e)
       [] ((not is_n) and (not is_s) and (is_w)) → { Ouest }
          d := abs(v_dest.x-bb_w)
       [] ((not is_n) and (not is_s) and (not is_e) and (not is_w)) →
          d := R { v_dest est interne à la boîte couvrante }
       fi
  fi

50
 ;isInInsideBInter := (d <= R) ]|
```

Algorithme 6.1 – `isInsideBInter`

On commence par déterminer dans laquelle des neuf zones de la décomposition l'élément source se trouve. On effectue ensuite un test de distance.

Ce test de distance varie selon la zone où se trouve l'élément source. La figure 6.3 illustre (en traits pointillés) la courbe qui se situe à une distance R de la boîte couvrante minimum.

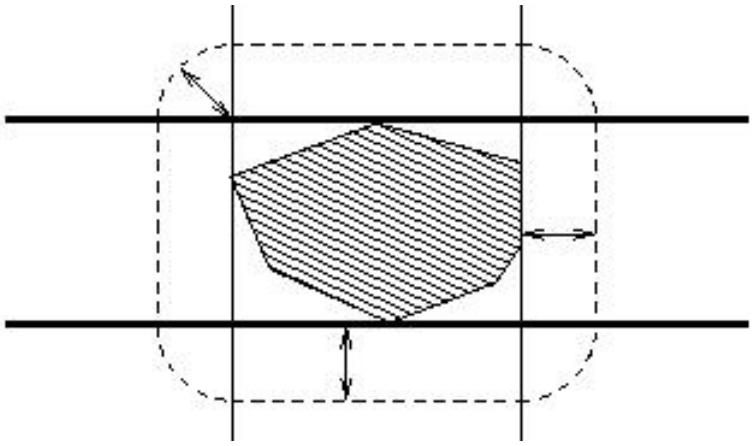


FIG. 6.3 – Courbe équidistante de la boîte couvrante minimum

Si la source et l'obstacle sont éloignés d'une distance $\leq R$, on recommence cette procédure pour la destination (détermination de la zone, test de distance).

On sait alors si l'obstacle est proche à la fois de la source et de la destination : dans ce cas, on peut l'inclure dans $\mathcal{B}_\cap$.

Un tel algorithme présente un aspect *étude de cas* assez lourd. Nous l'avons tout de même considéré car il est très performant et approxime plus ou moins bien l'obstacle.

Notons que, d'un certain point de vue, des *false positive* seront générés : il y a un risque d'inclure dans $\mathcal{B}_\cap$ des obstacles dont la boîte couvrante minimum est proche de la source et de la destination mais dont aucun sommet n'est proche ni de l'une ni de l'autre.

Le filtrage géométrique réalisé est donc approximatif.

Il est toutefois performant puisqu'il a été observé qu'il permet de n'inclure dans $\mathcal{B}_\cap$ que de 1% à 10% des obstacles (en ce qui concerne les données dont nous disposons). La complexité dans le pire cas de l'algorithme de construction de $\mathcal{B}_\cap$ est linéaire en fonction du nombre d'obstacles (application de `isInsideBInter` à chaque obstacle).

6.1.2 Indexation des barrières

Indexer globalement les obstacles permet également de calculer rapidement si un obstacle appartient à $\mathcal{B}_\cap$.

Plusieurs structures de données permettent une indexation spatiale d'objets dans un espace planaire.

Citons notamment les *R-trees* et *kd-trees*, basés sur un partitionnement selon le positionnement des objets indexés et les *quadtrees* et *grid files*, basés sur un partitionnement absolu de l'espace englobant les objets indexés. Il s'agit de structures de données classiques dans le domaine des Systèmes d'Information Géographique [Don02b].

Indexer les barrières permet de répondre à la question : parmi toutes les barrières indexées, lesquelles sont *proches* du point donné ?

Une indexation simple dérivée d'une indexation par *grid file* a été développée.

Un *grid file* est une méthode d'accès à des points (de coordonnées multidimensionnelles) qui divise l'espace en une grille non périodique de telle manière que chaque dimension spatiale est divisée par un hachage linéaire [Nis01]. De petits ensembles de points sont référencés par une ou plusieurs cellules de la grille.

La différence est que nous avons divisé l'espace de manière continue. Cela donne l'avantage de pouvoir accéder directement à une information d'après son index sans devoir hasher celui-ci. Evidemment, cela entraîne une perte énorme d'espace.

Nous avons opéré un tel choix car il est intervenu assez tard dans la rédaction de ce document. L'étude de techniques d'indexation spatiale serait vraiment très enrichissante mais nous manquons d'espace rédactionnel pour effectuer ces développements de manière satisfaisante.

Voici ce que réalise (cf. figure 6.4) l'algorithme de construction d'une telle structure de données :

- division de la matrice des barrières en cellules de dimensions $R \times R$ au plus ;
- stockage dans chacune de ces cellules d'une référence sur chaque barrière dont la boîte couvrante minimum intersecte l'espace représenté par la cellule ou par une ou plusieurs de ses voisines en 8-connexité.

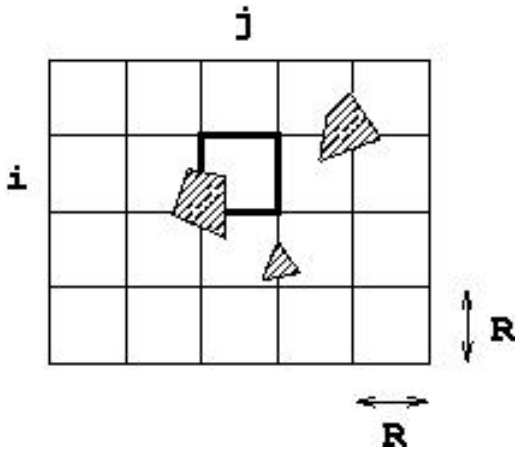


FIG. 6.4 – Construction de la cellule d'index $[i][j]$: référencement de trois barrières

Une requête adressée à cette structure de données est évidemment très performante : en temps constant, on obtient la liste des barrières dont les boîtes couvrantes minimum sont proches du point donné.

Une telle requête est de plus efficace puisqu'elle permet de n'inclure dans $\mathcal{B}_\cap$ que de 1% à 20% des obstacles (en ce qui concerne les données dont nous disposons).

Avant de poursuivre d'autres développements, notons que, combinée avec un calcul de positionnement des barrières par rapport à leur boîte couvrante minimum, une requête spatiale exploitant la structure de données que nous avons décrite est très efficace au niveau du temps d'exécution.

En effet, si on applique d'abord la requête spatiale puis le calcul de positionnement, ce dernier sera effectué sur un ensemble de barrières déjà considérablement réduit. Les temps d'exécution seront toujours meilleurs lorsqu'on combine les deux méthodes et toujours beaucoup plus lents lorsqu'on n'en applique aucune.

Les temps d'exécution lorsqu'on applique l'une ou l'autre méthode sont tantôt meilleurs pour l'application d'uniquement l'une ou l'autre méthode. Nous n'avons pas suffisamment de données à notre disposition pour tirer des conclusions à ce sujet.

6.2 Graphe des chemins les plus courts

Les chemins les plus courts de chaque noeud du graphe de visibilité vers tous les autres noeuds de celui-ci peuvent être calculés comme cela a été montré. Ces résultats ne varient pas au cours du temps puisque les matrices de données ne sont pas mises à jour dynamiquement et que, par conséquent, une fois construit, le graphe de visibilité n'est plus modifié.

Les chemins optimaux entre noeuds du graphe de visibilité peuvent donc être précalculés au cours d'une phase d'initialisation, comme le graphe de visibilité lui-même.

Chaque chemin optimal peut être représenté de manière compacte par un arc unique de longueur égale à la longueur du chemin entre sa source et sa destination, puisque c'est, in fine, à la longueur et non aux détours effectués par le chemin que l'on s'intéresse. Nommons **arc optimal** un tel arc.

Lors du traitement de chaque requête de calcul de chemin optimal pour un couple d'éléments source/destination, on peut construire un graphe de manière à exploiter le fait que les chemins optimaux entre noeuds du graphe de visibilité sont déjà calculés.

Nous appelons **graphe des chemins les plus courts** ce graphe, représenté sur la figure 6.5. Il exhibe les caractéristiques suivantes :

- Noeuds : la source, les noeuds du graphe de visibilité, la destination.
- Arcs :
  - ◇ (a) Arcs de liaison entre le noeud source et chaque arc optimal appartenant à $\mathcal{B}_\cap$ et dont un noeud est visible depuis la source.
  - ◇ (b) Arcs optimaux appartenant à $\mathcal{B}_\cap$, dont un noeud est visible depuis la source et dont l'autre noeud est visible depuis la destination.
  - ◇ (c) Arcs de liaison entre le noeud destination et chaque arc optimal appartenant à $\mathcal{B}_\cap$ et dont un noeud est visible depuis la destination.

La figure 6.5 représente dans la partie (a) les arcs de liaison entre la source et les arcs optimaux. Elle représente dans la partie (c) les arcs de liaison entre la destination et les arcs optimaux et représente ces derniers dans la partie (b).

Pour une meilleure lisibilité, les noeuds du graphe de visibilité ont été reproduits deux fois, en deux rangées parallèles dessinées entre les zones (a) et (b) ainsi que (b) et (c).

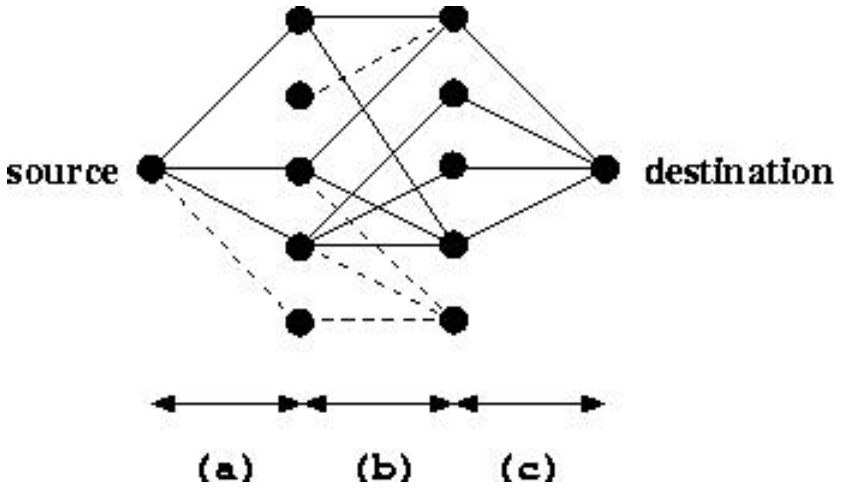


FIG. 6.5 – Graphe des chemins les plus courts

On remarque de plus que certains arcs sont en traits pointillés : il s'agit des arcs ne participant à aucun chemin complet entre source et destination pour des raisons de non visibilité des noeuds de certains arcs optimaux à partir de la source et/ou de la destination.

La longueur d'un chemin complet (source → destination) du graphe des chemins les plus courts est calculée en additionnant la longueur de l'arc de liaison reliant source et arc optimal, la longueur de l'arc optimal et la longueur de l'arc de liaison reliant arc optimal et destination.

Résoudre une requête de calcul de chemin le plus court se réduit donc à construire le graphe des chemins les plus courts et sélectionner le chemin le plus court entre source et destination.

La longueur du chemin le plus court du graphe des chemins les plus courts est la longueur du chemin le plus court entre source et destination, notée δ_k(source,destination) dans le modèle, en tenant compte de la présence de barrières.

La figure 6.6 représente un chemin du graphe des chemins les plus courts (pas nécessairement le plus court, par ailleurs) ainsi que la correspondance visuelle avec les différentes zones du graphe des chemins les plus courts présentées sur la figure 6.5.

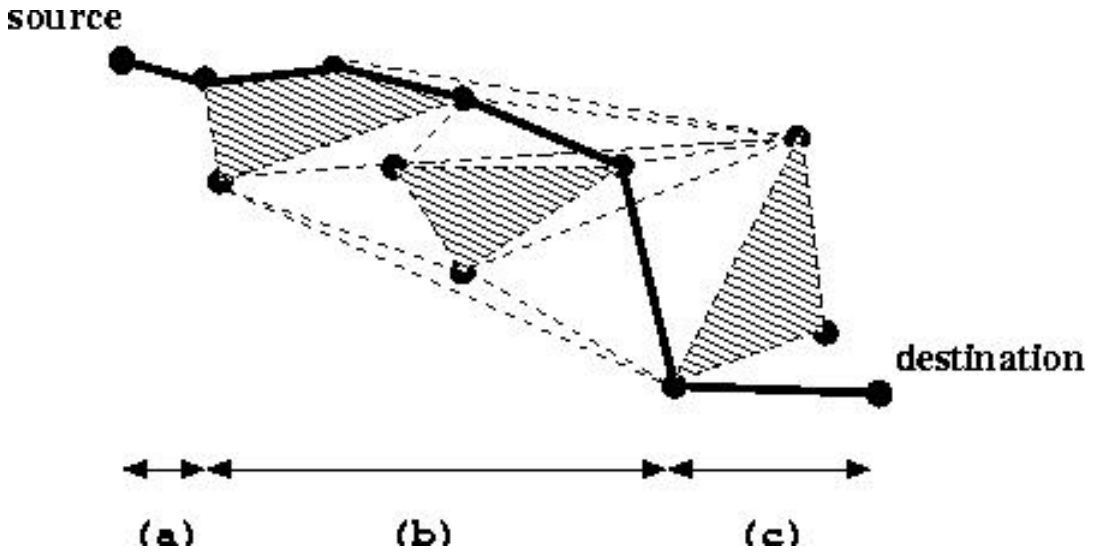


FIG. 6.6 – Chemin du graphe des chemins les plus courts

Voyons maintenant comment organiser ces idées sous forme algorithmique.

L'algorithme `ba_distance` calcule la distance entre deux éléments source et destination en évitant les barrières bloquantes.

Le principe est de constuire les chemins source/destination du graphe des chemins les plus courts et de mémoriser au fur et à mesure celui de plus courte longueur.

Dans une première phase, l'algorithme stocke les distances entre l'élément destination et chaque noeud des arcs optimaux appartenant à $\mathcal{B}_\cap$. Les résultats des requêtes de visibilité à partir de la destination sont également mémorisés pour ces noeuds.

Dans une seconde phase, l'algorithme passe en revue tous les noeuds des arcs optimaux appartenant à $\mathcal{B}_\cap$. Un calcul de distance ainsi qu'une requête de visibilité à partir de la source sont effectués pour chacun de ces noeuds.

Un arc du graphe des chemins les plus courts est détecté lorsque :

- un des deux noeuds de l'arc optimal évalué est éloigné de la source d'une distance inférieure à R, et aucun obstacle n'est présent entre ce noeud et la source,
- l'autre noeud de l'arc optimal évalué est éloigné de la destination d'une distance inférieure à R, et aucun obstacle n'est présent entre ce noeud et la destination.

Le meilleur chemin est retenu au fur et à mesure qu'on détecte des chemins les plus courts.

`Barriers-Avoiding Distance`

```
type Region = record
  begin
  data : array[0 .. size−1] of Vertex;
  size : int;
  end;

ba_distance(v_src : Vertex ; v_dest : Vertex ;
            regions : array[0 .. rs−1] of Region ; rs : int ) : float
{ regions contient toutes les régions résultant
  du processus de convexification des obstacles }

var
  v_sp_src, v_sp_dest : Vertex;
  i, j, vid : int;
  dmin, d, dsrc, ddest : float;
  dtmp : array[0 .. VGSIZE−1] of float;

|[ dmin := distance(v_src,v_dest)

;“ filtrer les barrières, retenir celles appartenant à B∩”

;if (isAvoidanceRequired(v_src,v_dest,regions, rs)) →

    { Des barrières doivent être contournées ! }

    { ** PHASE I **
      On stocke dans dtmp les distances de la destination
      aux éléments du graphe de visibilité,
      en mémorisant s'ils sont visibles par la destination }
    i := 0
    ;do (i < rs) → →
        vid := 0
        ;do (vid < regions[i] . size) →
            v_sp_dest := region[i] . data[vid] { Destination de l'arc optimal }
            ;if (isAvoidanceRequired(v_dest,v_sp_dest,regions, rs)) →
                dtmp[v_sp_dest.ID] := −1.0
             [] (not isAvoidanceRequired(v_dest,v_sp_dest,regions,rs)) →
                dtmp[v_sp_dest.ID] := distance(v_dest, v_sp_dest)
             fi
             { dtmp[id] = distance(v,v_dest) avec v tel que v.ID = id }
            ;vid := vid+1
         od
        ;i := i+1
     od
    ;“<PHASE II : cf. page suivante>”

 [] not (isAvoidanceRequired(v_src,v_dest,regions, rs)) → skip
    { Pas de barrières à contourner }
 fi
;ba_distance := dmin ]|
```

Algorithme 6.2 – `Barriers-Avoiding Distance`

Barriers-Avoiding Distance (suite)

```
{ ** PHASE II **
  Les 3 tableaux SP_XXX de taille VGSIZE contiennent
  des informations au sujet des arcs optimaux.
  Pour l'élément dont l'ID est vid :
    SP_VAL[vid][j], SP_DEST[vid][j] =
      valeur et élément destination du j−ème arc optimal
    SP_SIZE[vid] = nombre d'arcs optimaux }

;i := 0
;do (i < rs) →
    vid := 0
    ;do (vid < regions[i].size) →
        v_sp_src := regions[i].data[vid] { Source de l'arc optimal }
        ;if (isAvoidanceRequired(v_src,v_sp_src,regions,rs)) → skip
            { v_sp_src n'est pas visible par la source −>
              pas de chemin du graphe des chemins les plus courts }
         [] (not isAvoidanceRequired(v_src,v_sp_src,regions,rs)) →
            { v_sp_src est visible par la source}

            dmin := R
            ;dsrc := distance(v_src,v_sp_src)
            ;j := 0
            ;do (j < SP_SIZE[v_sp_src.ID]) →
                { On passe en revue tous les arcs optimaux
                  issus de v_sp_src }

                v_sp_dest := SP_DEST[v_sp_src.ID][j] { Destination de l'arc optimal }
                ;ddest := dtmp[v_sp_dest.ID]
                ;if (ddest ≠ −1.0) →

                    { v_sp_dest est visible par la destination : un chemin
                      du graphe des chemins les plus courts a été détecté −>
                      évaluer la longueur des 3 composantes }
                    d := dsrc + SP_VAL[v_sp_src.ID][j] + ddest
                    ;if (d < dmin) → dmin := d { Plus court, mémoriser ! }
                     [] (d ≥ dmin) → skip
                     fi

                 [] (ddest = 1.0) → skip { Destination de l'arc optimal pas visible }
                 fi                      { ou trop loin de l'élément destination }
                ;j := j+1

            od

        fi
        ;vid := vid+1
    od
    ;i := i+1
od
```

Algorithme 6.3 – `Barriers-Avoiding Distance` (suite)

La complexité dans le pire cas de l'algorithme est $O(E\ N_{bh})$, où E représente le nombre d'arcs optimaux et où N_{bh} représente la complexité dans le pire cas de l'algorithme `isAvoidanceRequired`, qui détermine si un obstacle bloque la visibilité mutuelle de deux éléments donnés. En d'autres termes, N_{bh} est le nombre total de sommets des enveloppes convexes approximant les obstacles. N_{bh} correspond également au nombre de noeuds V du graphe de visibilité.

La complexité de `ba_distance` est donc $O(E\,V)$.

En pratique, il y a en moyenne de 500 à 2000 chemins du graphe des chemins les plus courts qui sont construits pour chaque requête de calcul d'un chemin le plus court. Le nombre de noeuds du graphe de visibilité est $O(10^3)$.

La complexité attendue est toutefois nettement meilleure, grâce aux opérations de filtrage des barrières présentées en début de chapitre.

En résumé, `ba_distance` répond aux requêtes de calcul de chemin le plus court en exploitant le fait que les chemins optimaux entre noeuds du graphe de visibilité sont précalculés et stockés sous une forme compacte (arcs optimaux). La recherche de chemin optimal est donc effectuée dans le graphe des chemins les plus courts plutôt que dans le graphe de visibilité augmenté d'arcs de liaison avec la source et la destination.

6.3 Conclusion

Plusieurs idées ont été proposées pour accélérer les requêtes de visibilité en filtrant les barrières qui n'interviennent certainement pas dans celles-ci.

Des techniques d'indexation spatiales plus sophistiquées que celles qui ont été présentées apporteront certainement des gains de performance supplémentaires (ainsi qu'une économie certaine d'espace mémoire) et constitueraient une voie de recherches très intéressante, un prolongement naturel de ce travail.

Une analyse plus poussée a montré comment il est possible d'accélérer les requêtes de calcul d'un chemin le plus court en précalculant un certain nombre de données.

En particulier, il a été proposé de construire un graphe - le graphe des chemins les plus courts - lors de chaque requête de calcul d'un chemin le plus court en se basant sur l'information contenue dans les arcs optimaux, qui sont des arcs résumant l'information contenue dans les chemins les plus courts entre noeuds du graphe de visibilité.

Finalement, les opération de filtrage sont liées à la complexité attendue de l'algorithme `ba_distance` et l'introduction du graphe des chemins les plus courts est liée à sa complexité dans le pire cas.

Le chapitre suivant, le dernier avant de conclure ce document, abordera les possibilités de distribution des calculs étant donné que la structure du problème `DGSP` est naturellement décomposable comme cela va enfin être montré.

Chapitre 7

Distribution des calculs

Les algorithmes résolvant `DLOSP` et `DGSP` ont été présentés. Une distribution des calculs requis pour exécuter ces algorithmes plus rapidement sera envisagée dans ce chapitre. **Le terme `DxSP` sera utilisé pour désigner simultanément `DLOSP` et `DGSP`.**

7.1 Décomposition en domaines de calcul

L'algorithme général de calcul de la matrice α d'indice d'accessibilité, tel que présenté au chapitre 3, consiste essentiellement en un remplissage séquentiel des éléments de la matrice.

Considérons comme élément source l'élément d'indice de ligne i et d'indice de colonne j. Considérons les éléments de son voisinage qui sont d'indice de ligne inférieur à i (ou égal à i mais d'indice de colonne inférieur à j), tels qu'illustrés en noir sur la figure 7.1. Les termes d'accessibilité de l'élément (i, j) vers ces éléments sont déjà calculés.

Comme expliqué au chapitre 3, résoudre `DxSP` pour l'élément (i, j) revient donc à ajouter à $\alpha[i][j]$ les valeurs des termes d'accessibilité relatifs aux éléments de son voisinage qui sont d'indice de ligne supérieur à i (ou égal à i mais d'indice de colonne supérieur ou égal à j), tels qu'illustrés par des X sur la figure 7.1.

Supposons maintenant qu'**un découpage de α est effectué suivant des bandes de R lignes**[1] (R = rayon d'accessibilité = distance maximale d'influence d'un élément), tel qu'illustré sur la figure 7.2. Une telle méthode est appelée *Global Domain Decomposition Method* [CES97].

[1] Le découpage aurait pu être également arbitrairement effectué sur les colonnes.

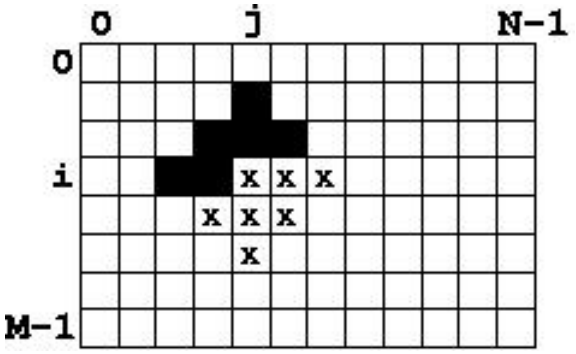


FIG. 7.1 – Partie utile du voisinage de l'élément source

FIG. 7.2 – Découpage de α en bandes de hauteur R

Mettons en évidence la propriété suivante, déduite des observations effectuées au sujet des figures 7.1 et 7.2 : **le calcul de `DxSP` pour tous les éléments d'une bande dépend uniquement des données relatives aux bandes précédente, courante et suivante**.

Toute bande peut donc être calculée séparément des autres à condition de disposer également des données des bandes voisines. **Une distribution des calculs sur plusieurs noeuds de calcul est réalisable sans problème de synchronisation.**

Le nombre de noeuds de calcul disponibles et le nombre total de bandes à distribuer permettent de calculer le nombre de bandes à calculer sur chaque noeud de calcul disponible. On communique à chaque noeud de calcul l'intervalle de lignes de α à calculer ainsi que l'intervalle de lignes de données à prendre en considération.

L'algorithme `DistributeBands` va répartir la charge le plus uniformément possible entre les différents noeuds de calcul. Les morceaux de α calculés sur les différents noeuds seront récupérés et rassemblés sur le serveur web.

`DistributeBands`

```
DistributeBands(<DONNEES ADEQUATES> ; M : int ; R : int)

var
  n, nbt, nbn, i, ni, clo, chi, dlo, dhi : int;

|[ n := "nombre de noeuds de calcul disponibles"
 ;if (n < 1) → "pas de calcul possible !"
  [] (n ≥ 1) →
     nbt := max(M div R,1) { Nombre de Bandes Total }
     ;if (nbt ≤ n) →
         nbn := 1
      [] (nbt > n) →
         if (nbt mod n = 0) → nbn := nbt div n
         [] (nbt mod n > 0) → nbn := (nbt div n) + 1
         fi
      fi
     { nbn = Nombre de Bandes (maximum) par Noeud }
     ;i,ni := 0,0
     ;do i < nbt → { nbt − i = nombre de bandes à distribuer }
         if ((i = 0) and (i < nbt−nbn)) →
            clo,chi := 0,(nbn ∗ R)−1
            ;dlo,dhi := clo,chi+R
         [] ((i > 0) and (i < nbt−nbn)) →
            clo,chi := (i ∗ R),((i+nbn) ∗ R)−1
            ;dlo,dhi := clo, chi+R
         [] ((i ≥ nbt−nbn) and (i > 0)) →
            clo,chi := (i ∗ R),M−1
            ;dlo,dhi := clo,chi
         [] ((i = 0) and (i ≥ nbt−nbn)) →
            clo,chi := 0,M−1
            ;dlo,dhi := clo,chi
         fi
         { [clo .. chi] = lignes à calculer
           [dlo .. dhi] = lignes de données à considérer }
         ;" lancer l'exécution asynchrone de DxSP sur le noeud n_i"
         ;ni := ni+1
         ;i := i+nbn
      od
     { ni = nombre de noeuds de calcul utilisés }
     ;i := 0
     ;do i < ni →
          "attendre la fin des calculs sur le noeud i"
          ;" récupérer les lignes [clo .. dhi] sur le noeud i"
          ;" ajouter les valeurs des lignes récupérées
            aux valeurs de α correspondantes"
          ;i := i+1
      od
 fi ]|
```

Algorithme 7.1 – `DistributeBands`

Il faut remarquer qu'il est nécessaire de récupérer les lignes $[clo..dhi]$ de la matrice calculées par le noeud n_i, alors qu'on s'attendrait à récupérer seulement les lignes $[clo..chi]$.

Il ne faut pas perdre de vue le principe *calculer une fois, stocker deux fois*. Cela implique que les dernières lignes de la zone $[clo..chi]$ contribuent aux valeurs des premières lignes de $[chi + 1..dhi]$. C'est pourquoi il est essentiel de les récupérer.

7.2 Mesure de performances

Il est clair que si un seul noeud de calcul est disponible, l'overhead en données transmises sur le réseau est nul, tout comme le gain de temps. De même, si α est décomposée en nbt bandes et si $n = nbt$ noeuds de calcul sont disponibles, l'overhead en données transmises sur le réseau est de 100% (puisque deux bandes sont envoyées à chaque noeud) mais le gain de temps pourrait être proportionnel à n, si les temps de transfert et de rassemblement des données ne sont pas trop pénalisants.

Les mesures suivantes [dK95] seront utilisées pour comparer les temps de calcul :

$$A = \frac{T_s}{T_d}$$

$$E = \frac{A}{n}$$

où A est le facteur d'accélération obtenu grâce à la distribution des calculs, E est l'efficacité de la distribution, n est le nombre de noeuds de calcul, T_s est le temps total d'exécution de la version séquentielle et T_d le temps total d'exécution de la version distribuée.

L'équipement utilisé est un PC doté d'un CPU Athlon XP 1600+ et de 512 Mo RAM. Le compilateur utilisé est `gcc`.

Algorithme	Zone	R	T_s	T_d	n	A	E
`DLOSP`	Urbaine	32	2.9 s	1.6 s	6	1.8	30.2%
`DLOSP`	Urbaine	64	11.93 s	4.85 s	3	2.5	82%
`DLOSP`	Rurale	32	31.08 s	8.806 s	18	3.5	19.6%
`DLOSP`	Rurale	64	147.89 s	19.24 s	9	7.7	85.4%
`DGSP`	Urbaine	16	58.52 s	10.76 s	12	5.4	45.3%
`DGSP`	Urbaine	32	641.57 s	170.82 s	6	3.8	62.6%
`DGSP`	Rurale	16	165.73 s	22.526 s	19	7.4	38.7%
`DGSP`	Rurale	32	1255.5 s	114.22 s	18	11	61%

FIG. 7.3 – Tableau récapitulatif des temps de calcul

On constate que l'efficacité est d'autant plus acceptable que le nombre de calculs à effectuer est grand.

Un élément à ne pas négliger outre l'influence du rayon d'accessibilité est que les requêtes de visibilité sont certainement plus performantes lorsque le calcul est distribué : en effet, le nombre de barrières à prendre en considération sur chaque noeud est d'autant plus petit que le nombre de noeuds de calcul impliqués est grand.

Quel est l'impact des prétraitements sur le temps total d'exécution de `DGSP` ? Voici un résumé pour la version séquentielle :

Opérations ($R = 32$)	Zone Urbaine	Zone Rurale
Lecture des données	70 ms	540
Construction des régions	10 ms	120 ms
Graphe de visibilité et arcs optimaux ($R = 16$)	670 ms	2170 ms
Graphe de visibilité et arcs optimaux ($R = 32$)	1630 ms	4310 ms
Graphe de visibilité et arcs optimaux ($R = 64$)	4530 ms	9340 ms
Total ($R = 16$)	750 ms	2830 ms
Total ($R = 32$)	1710 ms	4970 ms
Total ($R = 64$)	4610 ms	10000 ms

FIG. 7.4 – Temps de calcul des prétraitements de `DGSP`

Ces temps sont tout à fait négligeables par rapport au temps de calcul total de `DGSP`, même distribué.

7.3 Conclusion

Une procédure de **distribution des calculs sous-tendant `DLOSP` et `DGSP`** a été développée, permettant de **diminuer les temps de calcul grâce à la décomposition** du problème.

Un **système distribué** implémentant la distribution des calculs proposée est présenté dans les annexes.

Quels enseignements peut-on tirer de ce travail ?

- L'impact de la longueur du rayon d'accessibilité est important mais le nombre et la dispersion des barrières ont également une influence évidente sur le temps de calcul global. Nous n'avons pas pu établir de manière claire ces relations, essentiellement en raison du manque de données réelles à notre disposition.

- Sans les différents prétraitements et optimisations proposés, la résolution de `DGSP` n'aurait pu se concrétiser en des temps de calcul raisonnables. On a parfois l'impression que beaucoup de temps est consacré à des concepts de portée fort limitée, mais on s'aprerçoit au final que c'est la combinaison et l'intégration de tous les efforts de calcul qui permettent d'obtenir un résultat intéressant, meilleur que si on considère toutes les parties séparément.

Chapitre 8

Conclusions

Ce travail a été réalisé en vue d'apporter des solutions à un problème de calcul du Laboratoire d'Etude en Planification Urbaine et Rurale de l'Université de Liège.

Le but de ce travail est le calcul d'un indice d'accessibilité des modes de transport lents (marche à pied, vélo, ...), dans une perspective d'aide à la décision.

Cet indice est calculé à partir de plusieurs couches de données matricielles : densité moyenne de population, modèle numérique de terrain et barrières (obstacles non franchissables naturels ou issus de l'activité humaine). Le résultat est également une couche de données matricielles.

La prise en considération des barrières constitue le problème central du calcul de cet indice.

Pour modéliser correctement un chemin entre deux points, un processus de contournement des barrières doit être développé. Il doit de plus être efficace en raison de la grande quantité de chemins qui sont calculés.

Nous avons développé un modèle, étendant le modèle original développé au LEPUR, présenté les deux problèmes `DLOSP` (*Discrete Line-Of-Sight Paths*, ne prenant pas en compte les barrières) et `DGSP` (*Discrete Geodesic Shortest Paths*, gérant le contournement des barrières) et développé puis implémenté des algorithmes pour résoudre ces problèmes.

Le contournement de barrières imposé par `DGSP` est, comme annoncé, le problème central de ce travail. Une chaîne d'algorithmes performante a été développée : nous avons essentiellement revisité des grands classiques algorithmiques en montrant comment les intégrer harmonieusement et en conservant constamment un objectif de diminution des temps de calcul.

Parmi les concepts abordés, citons notamment :

- détecter les barrières,
- approximer les barrières détectées,
- développer un modèle de visibilité,
- répondre aux requêtes de visibilité,
- reconditionner l'espace de recherche,
- répondre aux requêtes de calcul de chemin le plus court,
- optimiser les requêtes critiques (*bottlenecks*).

Une opération très importante du point de vue des performances observées et de la complexité attendue de `ba_distance` (l'algorithme de calcul de distance prenant en compte les barrières) est le filtrage des barrières. En effet, partant du principe que certaines barrières ne sont certainement pas bloquantes pour un couple d'éléments source/destination, on peut tenter de les filtrer rapidement avant d'effectuer les requêtes de visibilité et de recherche d'un chemin le plus court.

Une distribution des calculs de `DLOSP` et `DGSP` a été conçue, permettant d'exploiter un réseau de stations de travail et de diminuer les temps de calcul. Cette distribution est rendue possible par la structure des problèmes à traiter.

Enfin, un prototype de serveur d'images a été présenté, montrant comment diffuser les résultats des calculs d'indice d'accessibilité effectués. Une interface simple (un navigateur HTTP) permet d'accéder à ce serveur de manière conviviale. L'utilisation d'un cache d'images permet une semi-persistance des données récemment ou fréquemment consultées.

Plusieurs voies de recherche sont envisageables comme continuité à ce travail :

- l'étude approfondie de structures de données permettant d'indexer spatialement des objets,
- la mise au point de méthodes nouvelles de résolution du problème de visibilité, voire l'extension du modèle de visibilité,
- l'amélioration du processus de distribution des calculs,
- l'étude de la persistance des résultats calculés (sous forme matricielle et sous forme d'image) ainsi que des structures de données intermédiaires, qui pourraient être stockés dans une base de données,
- la prise en compte totale de la troisième dimension spatiale (utilisation de TIN, par exemple),
- l'étude de la possibilité et du bien-fondé d'utiliser des méthodes heuristiques de recherche de chemins optimaux en vue d'affiner l'approximation des obstacles.

Bibliographie

[Ber98] L. Berden. *Parallélisation d'algorithmes d'enveloppes convexes.* Travail de fin d'études, Université de Liège, 1998.

[Bri01] C. Briquet. *Conception d'algorithmes pour la construction d'un ensemble emboîté de contours polygonaux convexes.* Travail de fin d'études, Université de Liège, 2001.

[Bri02] C. Briquet. *Simulation : HTTP Image Server.* Travail réalisé dans le cadre du cours de Performances des Systèmes Informatiques, Université de Liège, 2002.

[CES97] Rapport du CES. *SALMON (Sea Air Land Modeling Operational Network) Technical Report 1996-1997.* Center for Environmental Studies, Université de Liège, 1997.

[CLR92] T.H. Cormen, C.E. Leiserson & R.L. Rivest. *Introduction to Algorithms (6th printing).* MacGraw-Hill, 1992.

[Cui99] Olivier Cuisenaire. *Distance Transformations : Fast Algorithms and Applications to Medical Image Processing.* Ph.D. Thesis, Université Catholique de Louvain, 1999.

[DV+00] M. de Berg, M. van Kreveld, M. Overmars & O. Schwarzkopf. *Computational Geometry : Algorithms and Applications.* Springer-Verlag, 2000.

[deM01] P.-A. de Marneffe. *Algorithmique Avancée.* Notes de cours, Université de Liège, 2001.

[dK95] P.-A. de Marneffe & D. Korthoudt. *Conception d'un solveur direct sur machine parallèle à mémoire distribuée.* Université de Liège, 1995.

[Don02a] J.-P. Donnay. *Analyse Spatiale.* Notes de cours, Université de Liège, 2002.

[Don02b] J.-P. Donnay. *Systèmes d'Information Géographique.* Notes de cours, Université de Liège, 2002.

[Ede87] H. Edelsbrunner. *Algorithms in Combinatorial Geometry.* Springer-Verlag, 1987.

[FT87] M.L. Fredman & R.E. Tarjan. *Fibonacci heaps and their uses in improved network optimization algorithms.* Journal of the ACM 34, 1987.

[GM91] S.K. Ghosh & D.M. Mount. *An Output-Sensitive Algorithm for Computing Visibility Graphs.* SIAM Journal on Computing 20, 1991.

[Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* Ph.D. Thesis, Czech Technical University, 2000.

[HS99] J. Hershberger & S. Suri. *An optimal algorithm for Euclidian shortest paths in the plane.* SIAM Journal on Computing 28, 1999.

[KMM97] S. Kapoor, S.N. Maheshwari & J.S.B. Mitchell. *An Efficient Algorithm for Euclidean Shortest Paths Among Polygonal Obstacles in the Plane.* Discrete Computational Geometry 18, 1997.

[LEP02] Rapport du LEPUR. *Etude complémentaire relative aux profils d'accessibilité - Profils de mobilité.* Université de Liège, 2002.

[Mit98] J.S.B. Mitchell. *Geometric Shortest Paths and Network Optimization.* State University of New York, 1998.

[MO01] J.S.B. Mitchell & J. O'Rourke. *Computational Geometry Column 42.* SIGACT News 32 (3), 2001.

[Mou02] D.M. Mount. *Computational Geometry.* Lecture notes, University of Maryland, 2002.

[Nis01] National Institute of Standards and Technology. *Dictionary of Algorithms and Data Structures.* NIST website, 2001.

[NP96] P. Niemeyer & J. Peck. *Java par la pratique.* Paris, O'Reilly, 1996.

[Oro98] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1998.

[Rif96] J.-M. Rifflet. *La Communication sous UNIX - Applications réparties, 2ème édition.* Ediscience international, 1996.

[Sau99] S. Saunders. *A Comparison of Data Structures for Dijkstra's Single Source Shortest Path Algorithm.* Honours project, Univeristy of Canterbury, 1999.

[Vdb00] M. Van Droogenbroeck. *Traitement Numérique des Images.* Notes de cours, Université de Liège, 2000.

[Vit01] J.S. Vitter. *External Memory Algorithms and Data Structures : Dealing with Massive Data.* ACM Computing Systems 33 (2), 2001.

[Wel85] E. Welzl. *Constructing the Visibility Graph for n-line segments in $O(n^2)$ time.* Information Processing Letters 20, 1985.

Annexe A

Notations

- B : matrice de données (couche de données topographiques indiquant la présence d'une barrière pour chaque pixel)
- Alt : matrice de données (couche de données attributaires estimant l'élévation moyenne de chaque pixel, il s'agit du modèle numérique de terrain)
- Pop : matrice de données (couche de données attributaires estimant la densité de population moyenne de chaque pixel urbanisé)
- α : matrice de résultats contenant les valeurs calculées de l'indice d'accessibilité

- M : nombre de lignes des matrices de données et de résultats
- N : nombre de colonnes des matrices de données et de résultats

- R : rayon d'accessibilité

- d : distance entre deux éléments
- d^e : distance entre deux éléments (tenant compte de la différence d'élévation entre ceux-ci)

- ΔZ : différence d'élévation
- $\bar{Z}$: seuil d'élévation

- C : chemin entre deux éléments
- N_C : nombre d'éléments du chemin entre deux éléments
- L_C : longueur du chemin entre deux éléments

- C^* : chemin optimal (géodésique discret le plus court) entre deux éléments
- N_{C^*} : nombre d'éléments du chemin optimal entre deux éléments
- L_{C^*} : longueur du chemin optimal entre deux éléments

- V : voisinage (ensemble d'éléments) d'un élément source
- N_V : nombre d'éléments du voisinage d'un élément source

- D : ensemble des destinations possibles à partir d'un élément source
- N_D : nombre d'éléments de l'ensemble des destinations possibles

- $\Gamma(i, j)$: ensemble des coordonnées des éléments appartenant (selon le contexte) au voisinage ou à l'ensemble des destinations d'un élément source de coordonnées (i, j)
- $\gamma_k(i, j)$ coordonnées du k-ème élément du voisinage ou (selon le contexte) de l'ensemble des destinations d'un élément source de coordonnées (i, j)

- $d_k(P, P_k)$: longueur (en fait, une simple distance, utilisée dans `DLOSP`) du chemin optimal entre P et P_k (un élément source et le k-ème élément de son voisinage), ne prenant pas en compte la présence de barrières
- $\delta_k(P, P_k)$: longueur (utilisée dans `DGSP`) du chemin optimal entre P et P_k (un élément source et le k-ème élément de son ensemble des destinations), prenant en compte la présence de barrières

Annexe B

Calcul d'enveloppe convexe

Le calcul de l'enveloppe convexe d'un nuage de points est souvent réalisé à partir des enveloppes convexes inférieure et supérieure du nuage.

On commence dans ce cas par identifier les points E et W qui, appartenant de manière certaine à l'enveloppe convexe à calculer, permettent de séparer le nuage de points en deux régions auxquelles on peut appliquer les routines de calcul d'enveloppe convexe supérieure et inférieure, comme on peut le voir sur la figure B.1 :

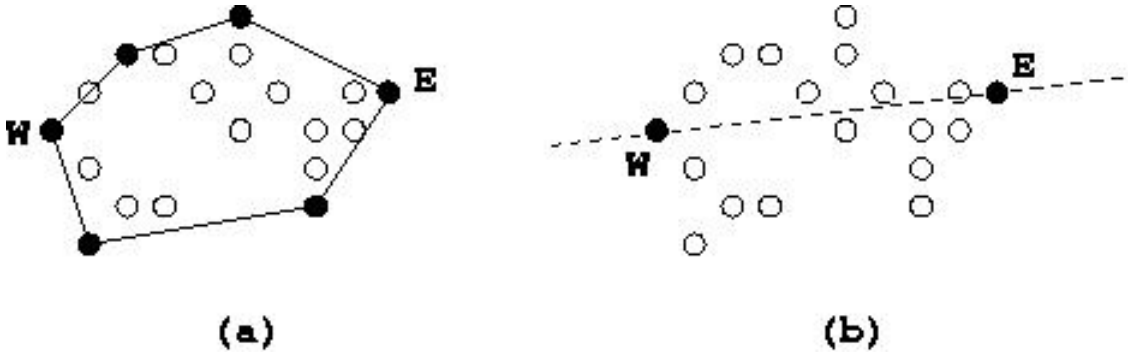


FIG. B.1 – (a) Enveloppe convexe (b) Séparation en deux régions à convexifier

Deux algorithmes classiques de calcul d'enveloppe convexe *in situ* (ne nécessitant pas de structures de données auxiliaires) vont être présentés : le QuickHull et le SortHull.

B.1 Principe algorithmique du QuickHull

L'algorithme connu comme étant le plus performant pour le calcul de l'enveloppe convexe d'un nuage de points est le QuickHull [deM01, Ber98]. Il ne requiert pas de tri préalable des données à convexifier.

La complexité attendue du QuickHull est $O(N \log N)$, N représentant la taille du nuage de points à convexifier.

Le calcul de l'enveloppe convexe supérieure est réalisé sur la région supérieure stockée comme suit :

	p	$p+1$	$q-2$		q
V	E	Sac U		W	

Après application du QuickHull pour la partie supérieure (algorithme `uQuickHull`), V sera réarrangée comme suit (L représente la partie de contour relative aux points compris entre les indices p et q initiaux) :

	p			h	q
V	E	L	W	Sac I	

Le QuickHull est un algorithme récursif opérant d'une façon assez similaire à son alter ego pour les opérations de tri le QuickSort. Initialement, les bornes p et q des 2 schémas précédents valent 0 et u. Les points E et W sont les points respectivement le plus à l'Est et le plus à l'Ouest des points contenus dans V lors de l'application de l'algorithme.

Le QuickHull va effectuer une partition des données en 3 zones (figure B.2) :

- points certainement internes (I),
- points pouvant appartenir au contour (R_0),
- points pouvant appartenir au contour (R_1).

Le QuickHull vise à éliminer tous les points n'appartenant certainement pas au contour (zone I) puis à se réappliquer sur les deux sous-zones restantes (R_0 et R_1).

Comment déterminer les sous-zones ? Il s'agit des régions qui se trouvent de part et d'autre des segments de droite MW et ME.

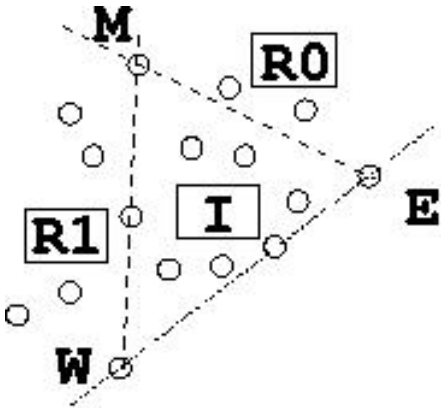


FIG. B.2 – Partition de la région supérieure en trois sous-zones

Comment déterminer le point M sur lequel s'appuient les deux segments séparateurs ? M **est l'équivalent du concept de pivot du QuickSort à la différence que son choix n'est pas arbitraire.** Il est déterminé comme étant le point tel qu'il forme une aire signée maximale avec les points W et E et qu'il n'est ni W ni E.

L'opération de partition réarrange V comme suit :

	p		m			n	q
V	E	R_0	M	R_1	W	Sac I	

Après application de `uQuickHull` à R_1 :

	p		m			j	n	q
V	E	R_0	M	L_1	W	I	Sac I	

Après application de `uQuickHull` à R_0 :

	p			i	$m+1$		j	n	q
V	E	L_0	M	I	L_1	W	I	Sac I	

Après l'application de l'algorithme aux deux sous-zones R_0 et R_1, il suffit de concaténer les deux parties de contour correspondantes (L_0 et L_1) :

	p					h	q
V	E	L_0	M	L_1	W		

On obtient ainsi le réarrangement souhaité tel qu'exprimé en début de section.

L'algorithme se termine car les deux zones de V auxquelles on applique récursivement le QuickHull sont disjointes à l'exception du point M qui est commun. Par ailleurs, la taille de ces zones est de plus en plus réduite au fur et à mesure de l'application répétée du QuickHull. Le cas de base correspond à une zone contenant uniquement les trois éléments E, M et W formant naturellement une partie de contour convexe.

`uQuickHull`

```
uQuickHull(p : int ; q : int ; var h : int)

var
    as, i , j , m,n : int ;

|[ if p+2 = q →
    h := q
  [] p+2 < q →
    <Sélectionner un point extrême (uQuickHull)>
    {Appel : uSelect2(p,q,m)}
    ;as := Δ(V[p],V[m],V[q−1])
    ; if as = 0 →
        V:swap(p+1,q−1)
        ;h := p+2
      [] as > 0 →
        <Partitionner (uQuickHull)>
        {Appel : uPartition3(p,q,m,n)}
        ;uQuickHull(m,n,j)
        ;uQuickHull(p,m+1,i)
        ;m := m+1
        ;do m < j →
            V:swap(i,m)
            ;i , m := i+1,m+1
         od
        ;h := i
    fi
 fi ]|
```

Algorithme B.1 – `uQuickHull`

Notons que cet algorithme est défini pour un nombre d'éléments supérieur ou égal à deux.

Il nous reste à présenter les opérations utilisées par `uQuickHull`.

B.2 Opérations du QuickHull

Commençons par l'**opération de sélection d'un point extrême**. On se base sur l'invariant suivant :

	p	m	t	q
V	W	M	E	

Cette opération consiste à évaluer les points de la zone de manière à déterminer lequel forme la plus grande aire signée avec les points E et W. L'algorithme est assez direct et sa terminaison est manifestement assurée par la commande d'incrémentation de la variable t en fin de boucle.

`uSelect2`

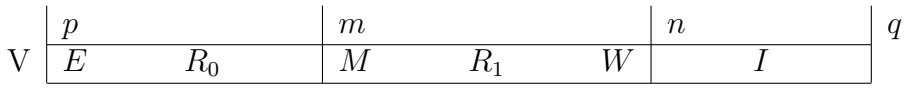
```
int uSelect2(p : int ; q : int ; var m : int)

var
    t,asmas,as : int;
    vm,vt : TPoint;

|[ m := p+1
;t := p+2;
;asmax := Δ(V[p],V[m],V[q−1])
{asmax ≥ 0}

;vm := V[m]
;do t < q−1 →
    vt := V[t]
    ;as := Δ(V[p],vt,V[q−1])
    ;if as > asmax →
        m := t
        ;vm := V[m]
        ;asmax := as
     [] as = asmax →
        if (vt.X < vm.X) or ((vt.X = vm.X) and (vt.Y < vm.Y)) →
            m := t
            ;vm := V[m]
        [] (vt.X > vm.X) or ((vt.X = vm.X) and (vt.Y ≥ vm.Y)) → skip
        fi
     [] as < asmax → skip
     fi
    ;t := t+1
od ]|
```

Algorithme B.2 – `uSelect2`

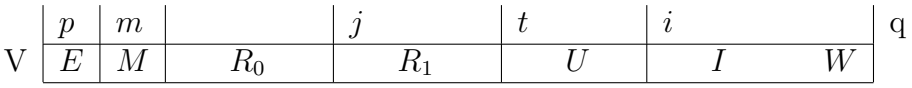
Pour conclure, détaillons maintenant l'**opération de partition d'une zone**. Comme on connaît le point M pour la zone (calculé par l'opération de sélection d'un point extrême), **on va pouvoir classer les points de la zone en trois sous-zones** :

- points internes, c'est-à-dire compris dans le triangle de sommets W, M et E,
- points susceptibles d'appartenir au contour car compris dans la zone R_0,
- points susceptibles d'appartenir au contour car compris dans la zone R_1.

On va parcourir toute la zone en réarrangeant les éléments en sous-zones pour arriver au résultat suivant



en suivant cet invariant, visant à réduire la taille de la zone U au profit des zones I, R_0 et R_1 (qui sont toutes les trois initialement vides) :



Pour rappel, lorsqu'on appelle cette routine, la zone comporte au moins trois éléments et le point M est différent des points E et W.

`uPartition3`

```
uPartition3(p : int ; q : int ; var m : int ; var n : int)

var
    i,j,t : int;

|[ V:swap(m,p+1)
;j,t := p+2,p+2
;m,i := p+1,q−1

;do t ≠ i →
    if Δ(V[p],V[t],V[m]) > 0 →
        V:swap(j,t)
        ;j,t := j+1,t+1
    [] Δ(V[m],V[t],V[q−1]) > 0 →
        t := t+1
    [] Δ(V[p],V[t],V[m]) ≤ 0 and
       Δ(V[m],V[t],V[q−1]) ≤ 0 →
        V:swap(t,i−1)
        ;i := i−1
    fi
 od

;V:swap(m,j−1)
;V:swap(q−1,i)
;m,n := j−1,i+1 ]|
```

Algorithme B.3 – `uPartition3`

La terminaison de la boucle est assurée car on diminue à chaque itération la taille de la zone à parcourir (avec une instruction d'incrémentation de la variable t ou avec une instruction de décrémentation de la variable i). Après cette boucle, on doit effectuer les permutations adéquates d'éléments de manière à rétablir l'invariant.

B.3 Principe algorithmique du SortHull

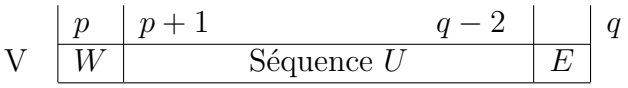
Le SortHull, ou Marche de Graham, est un algorithme classique de calcul d'enveloppe convexe d'un nuage de points. Cet algorithme suppose que les points à convexifier sont triés par ordre lexicographique.

La complexité attendue du SortHull est $O(N \log N)$, N représentant la taille du nuage de points à convexifier. Une constante multiplicative positive est à prendre en considération par rapport à la complexité du QuickHull.

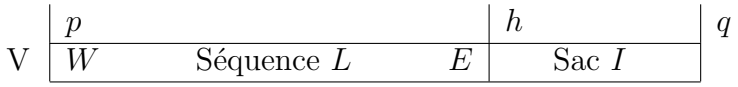
Le calcul de l'enveloppe convexe s'effectue à nouveau en calculant les enveloppes inférieure et supérieure : ici, W et E sont les points minimal et maximal par ordre lexicographique.

Le calcul de l'enveloppe convexe inférieure est réalisé sur la région inférieure stockée comme suit (le tri des données par ordre lexicographique étant supposé réalisé) :

Situation initiale :

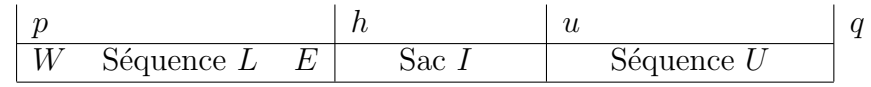


Postcondition :



- Une séquence U est une séquence de points triés par ordre lexicographique comportant éventuellement des répétitions de points (points confondus).
- Une séquence L est une séquence de points formant une partie de contour (convexe et ne comportant ni points alignés ni points confondus).
- Un sac I est une suite de points qui n'est plus triée.

Invariant :



On va construire le contour en effectuant un parcours de V : à chaque itération, **on intègre au contour le point u lorsque le triplet ($h-2$,$h-1$,u) forme une partie de contour convexe**, autrement dit que son aire signée est strictement supérieure à 0. **Lorsque ce n'est pas le cas, on rejette vers l'intérieur du contour le point $h-1$ responsable de la concavité** puis on recule d'une unité dans le contour (le retrait du point en $h-1$ peut provoquer une concavité amenée par le point qui était noté $h-2$ avant le retrait).

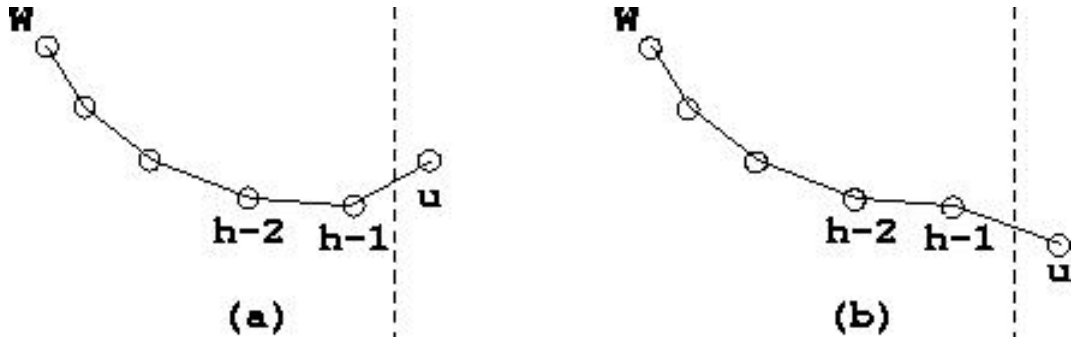


FIG. B.3 – Marche de Graham (a) u intègre le contour (b) $h-1$ devient interne

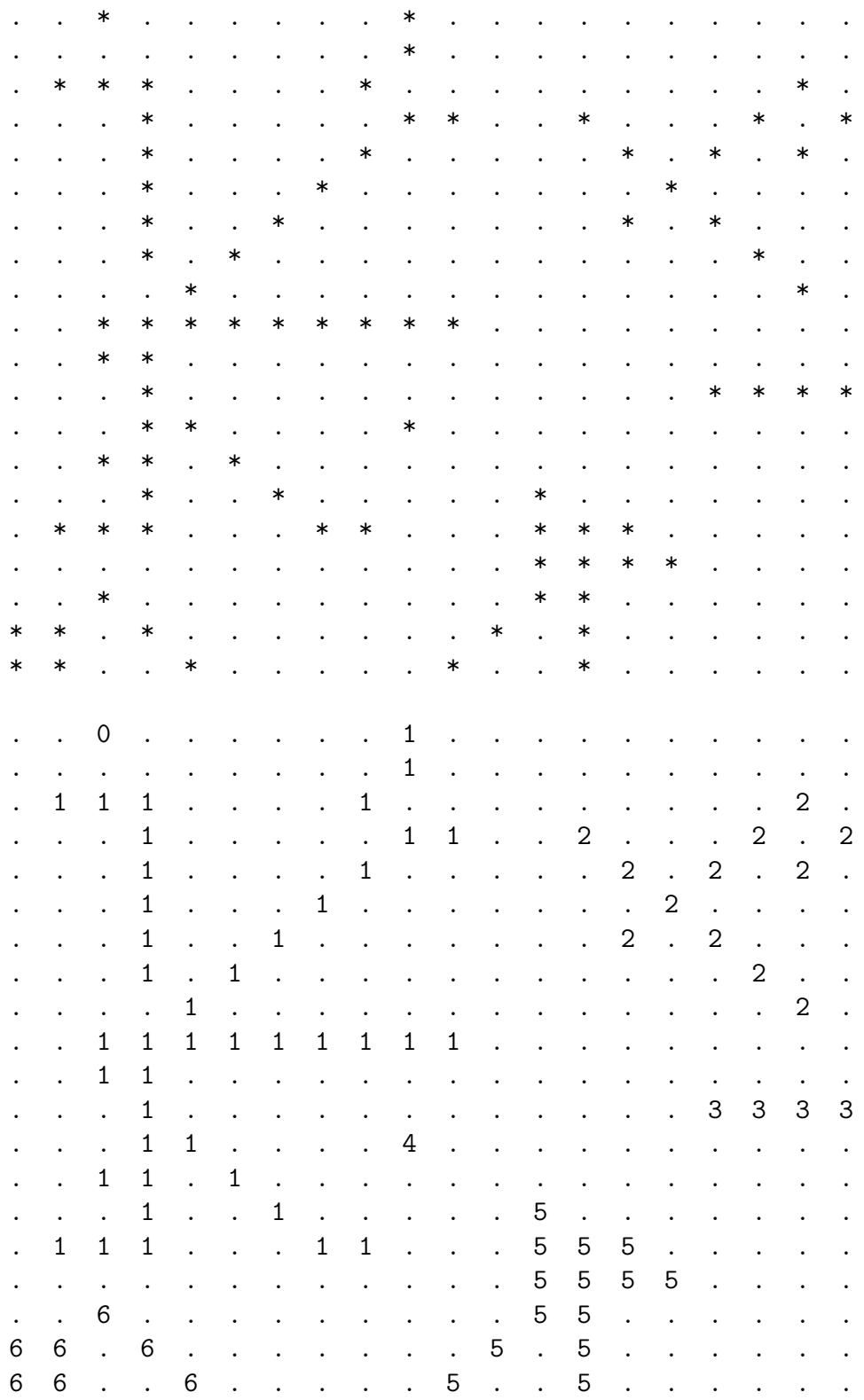
`lSortHull`

```
{Marche de Graham}

lSortHull(p : int ; q : int ; var h : int)

var
    u : int;

|[ h,u := p+1,p+1
;do u < q →
    if (V[u].X = V[h−1].X) and (V[u].Y = V[h−1].Y) →
        u := u+1 {Points confondus}
    [] (V[u].X ≠ V[h−1].X) or (V[u].Y ≠ V[h−1].Y) →
        {On essaye d'ajouter le point V[u] au contour}
        do (h−1 > p) cand Δ(V[h−2],V[h−1],V[u]) ≤ 0 →
            h := h−1 {V[h−1] est interne par rapport à la partie}
        od          {de contour déjà construite}
        ;V:swap(h,u)
        ;h,u := h+1,u+1
    fi
od ]|
```

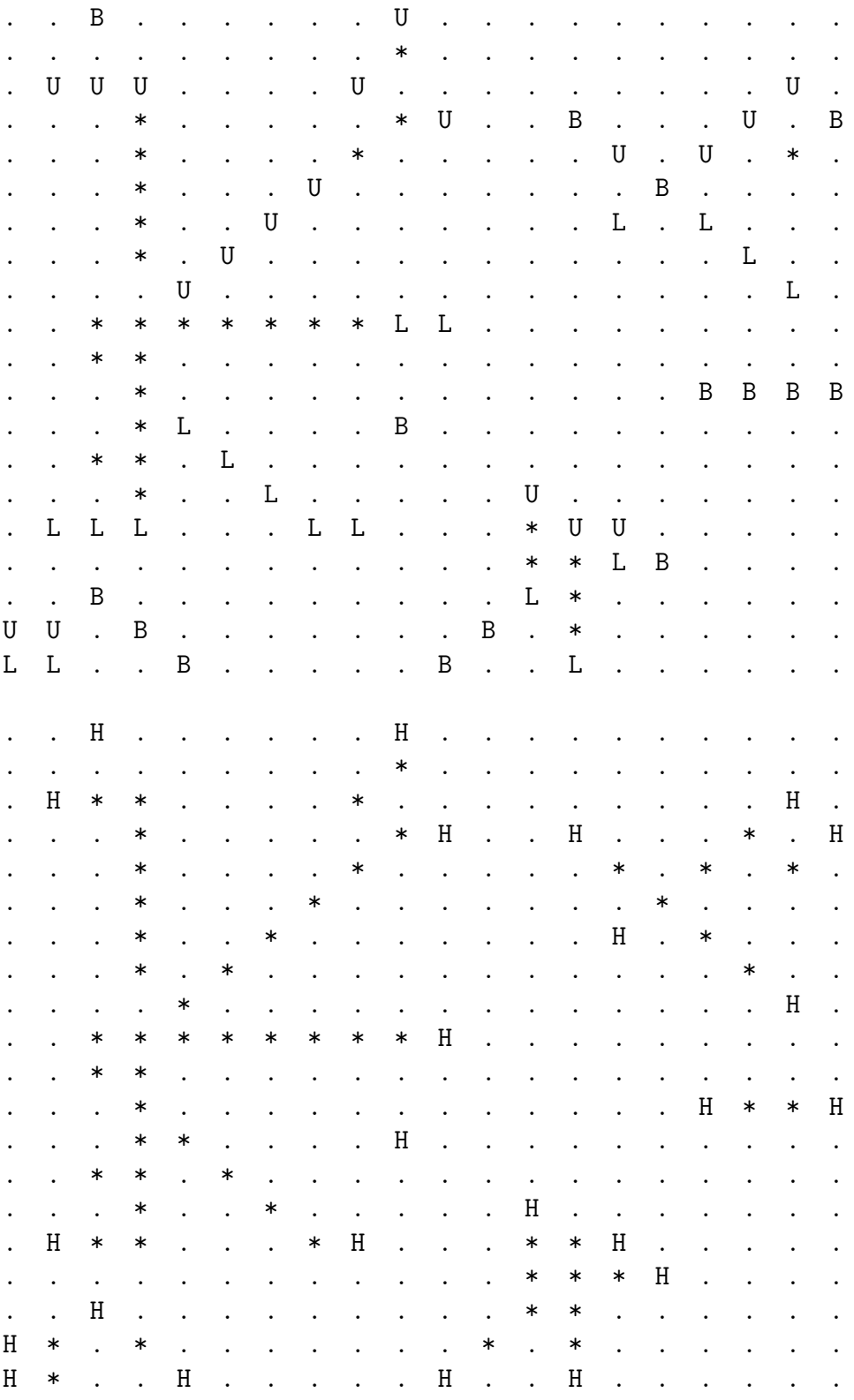
Algorithme B.4 – `lSortHull`

B.4 Construction des régions

Barrières (avant et après détection)



Enveloppes convexes (sommets candidats, sommets définitifs)



⇒ 24 points résument l'information contenue dans les 7 barrières

Annexe C

Test *beneath/beyond*

Voici l'algorithme testant si l'élément destination est devant ou derrière le segment d'obstacle donné.

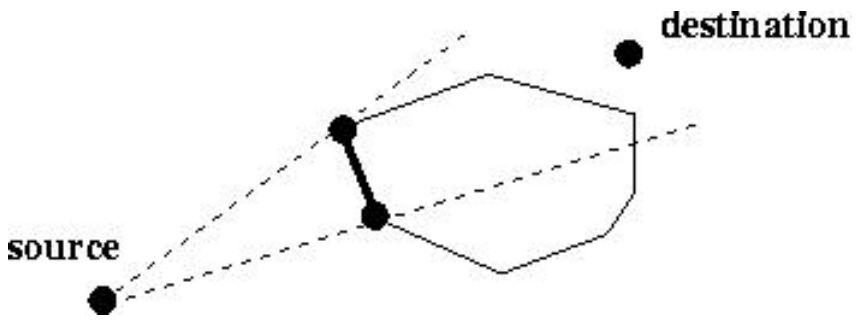


FIG. C.1 – Test *beneath/beyond*

Les notations utilisées correspondent au schéma suivant :

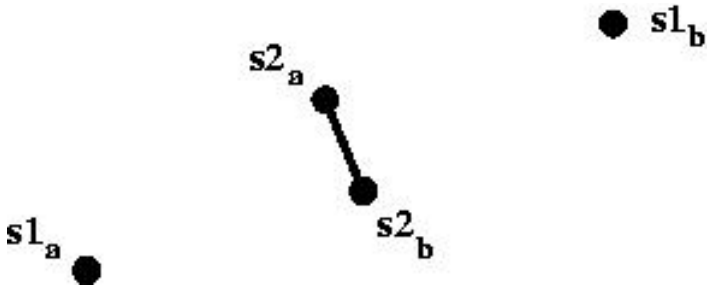


FIG. C.2 – Test *beneath/beyond* (notations)

`isBarringEdge`

```
isBarringEdge(s1_a : Vertex ; s1_b : Vertex ; s2_a : Vertex ; s2_b : Vertex ) : boolean

var
  dx1, dx2, dy1, dy2, m1, p1, m2, p2, x_inter , x1_lo , x1_hi , x2_lo , x2_hi ,
  y , xmin, xmax, ymin, ymax : float;

|[ dx1,dx2 := s1_b.x − s1_a.x,s2_b.x − s2_a.x
 ;dy1,dy2 := s1_b.y − s1_a.y,s2_b.y − s2_a.y

 ; if ((dx1 = 0) and (dx2 ≠ 0)) →
     m2 := dy2 / dx2 ;p2 := (s2_a.y) − (s2_a.x) ∗ m2 ;y := m2 ∗ s1_a.x + p2
     ;ymin,ymax := min(s1_a.y,s1_b.y),max(s1_a.y,s1_b.y)
     ;xmin,xmax := min(s2_a.x,s2_b.x),max(s2_a.x,s2_b.x)
     ;isBarringEdge := ((y ≥ ymin) and (y ≤ ymax) and
                        (s1_a.x ≥ xmin) and (s1_a.x ≤ xmax))

  [] ((dx1 ≠ 0) and (dx2 = 0)) →
     m1 := dy1 / dx1 ;p1 := (s1_a.y) − (s1_a.x) ∗ m1 ;y := m1 ∗ s2_a.x + p1
     ;ymin,ymax := min(s2_a.y,s2_b.y),max(s2_a.y,s2_b.y)
     ;xmin,xmax := min(s1_a.x,s1_b.x),max(s1_a.x,s1_b.x)
     ;isBarringEdge := ((y ≥ ymin) and (y ≤ ymax) and
                        (s2_a.x ≥ xmin) and (s2_a.x ≤ xmax))

  [] ((dx1 = 0) and (dx2 = 0)) →
     ymin,ymax := min(s2_a.y,s2_b.y),max(s2_a.y,s2_b.y)
     ;isBarringEdge :=
        ((s1_a.x = s2_a.x) and
          (((s1_a.y ≥ ymin) and (s1_a.y ≤ ymax)) or
           ((s1_b.y ≥ ymin) and (s1_b.y ≤ ymax)) or
           ((s1_a.y ≤ ymin) and (s1_a.y ≥ ymax)) or
           ((s1_b.y ≤ ymin) and (s1_b.y ≥ ymax))))

  [] ((dx1 ≠ 0) and (dx2 ≠ 0)) →
     m1 := dy1 / dx1 ;p1 := (s1_a.y) − (s1_a.x) ∗ m1
     ;m2 := dy2 / dx2 ;p2 := (s2_a.y) − (s2_a.x) ∗ m2
     ; if (m1 ≠ m2) →
          x_inter := (p2−p1) / (m1−m2)
        ;x1_lo,x1_hi := min(s1_a.x,s1_b.x),max(s1_a.x,s1_b.x)
        ;x2_lo,x2_hi := min(s2_a.x,s2_b.x),max(s2_a.x,s2_b.x)
        ;isBarringEdge :=
           ((((x_inter > x1_lo) and (x_inter ≤ x1_hi)) or
             ((x_inter ≥ x1_lo) and (x_inter < x1_hi))) and
            (((x_inter > x2_lo) and (x_inter ≤ x2_hi)) or
             ((x_inter ≥ x2_lo) and (x_inter < x2_hi))))
      [] (m1 = m2) → isBarringEdge := false { Les 2 segments sont // }
      fi

  fi ]|
```

Algorithme C.1 – `isBarringEdge`

Annexe D

Opérations sur un `binary heap`

Les opérations relatives à un `binary heap` sont présentées. Elle permettent l'implémentation de l'ensemble-frontière sur lequel repose l'algorithme de Dijkstra.

Cette implémentation est relativement directe : les données peuvent en effet être stockées simplement sous forme de vecteurs.

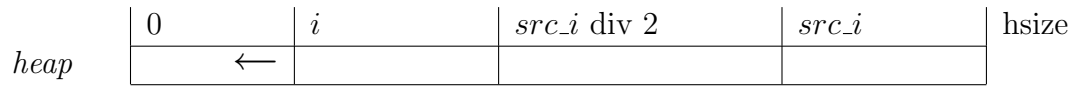
Insérer un élément revient à l'ajouter en fin de tas (en fin de vecteur de stockage), ce qui est immédiat. Il convient alors de rétablir la condition de tas, ce qui coûte $O(\log V)$ dans le pire cas. Cette opération est appelée `heap_delete_min`.

Mettre à jour la distance d'un élément est immédiat, si chaque élément du tas conserve une référence sur sa position dans le vecteur de stockage (ce qui est immédiat lorsque c'est réalisé au fur et à mesure des insertions et permutations lors des recouvrements de la condition de tas). Il reste alors à rétablir la condition de tas. Cette opération est appelée `heap_insert`.

Retirer l'élément de plus courte distance revient à retirer l'élément au sommet du tas (au début du vecteur de stockage). Après le retrait, on déplace (car c'est ce qu'il y a de plus immédiat) l'élément en fin de tas vers le sommet du tas, puis on rétablit ensuite la condition de tas (d'où le coût logarithmique de l'opération). Cette opération est appelée `heap_decrease_key`.

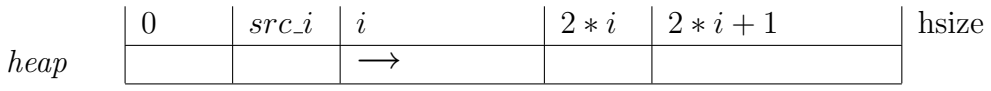
La technique de rétablissement de la condition de tas est appelée *sifting* : on examine à partir de l'élément modifié jusqu'au sommet du tas (*siftDown*) ou jusque la fin de tas (*siftUp*), tant que la condition de tas n'est pas localement respectée. L'avantage de la condition de tas est que, si on ne modifie qu'un élément, le rétablissement de celle-ci implique uniquement le passage en revue d'un seul chemin jusqu'à une des extrémités du tas.

Invariant (siftDown) :



A chaque itération, on maintient la propriété suivante : la condition de tas est respectée pour la partie $[i..hsize-1]$ (de la zone de stockage) du tas et i est mis à jour en étant remplacé par l'indice du père. La terminaison est assurée car la zone du tas traitée à chaque itération augmente et les divisions successives amèneront (au maximum) i à zéro.

Invariant (siftUp) :



A chaque itération, on maintient la propriété suivante : la condition de tas est respectée pour la partie $[0..i]$ (de la zone de stockage) du tas et i est mis à jour en étant remplacé par l'indice du plus petit de ses fils (si le père est plus petit que le plus petit de ses fils, la condition de tas est respectée ...). La terminaison est assurée car la zone du tas traitée à chaque itération augmente et les divisions successives amèneront (au maximum) i à l'indice maximum (de la zone de stockage) du tas.

Binary Heap

```
var
  hkey : array[0 .. VGSIZE−1] of float;
  hval : array[0 .. VGSIZE−1] of Node;
  hsize : int ; { −> valeur initiale : 0 }
  hsize_init : int ; { −> valeur initiale : VGSIZE }

{ On effectue la double hypothèse qu'on n'essaiera pas de stocker
  simultanément plus de VGSIZE éléments
  et que l'on ne tentera pas de retirer d'élément si hsize vaut 0. }

heap_insert(key : float ; value : Node)
|[ if (hsize < hsize_init−1) →
     hkey[hsize] , hval[hsize] ,value.heapIndex := key,value,hsize
     ;hsize := hsize+1
     ;siftDown(hsize−1) ]| { Rétablissement de la condition de tas }
  [] (hsize ≥ hsize_init−1) → skip
  fi ]|

heap_decrease_key(newkey : float ; value : Node)
|[ hkey[value.heapIndex] := newkey
 ;siftDown(value.heapIndex) { Rétablissement de la condition de tas } ]|

heap_delete_min(var min : Node)
|[ if (hsize > 0) →
     min := hval[0];
     ;min.heapIndex := −1
     ;hkey[0], hval[0] , hval[0] .heapIndex := hkey[hsize−1],hval[hsize−1],0
     ;hsize := hsize−1
     ;siftUp(0) { Rétablissement de la condition de tas }
  [] (hsize ≤ 0) → skip
  fi ]|

heapSwap(a : int ; b : int)
|[ hkey[a],hkey[b] := hkey[b],hkey[a]
 ; hval[a] , hval[b] := hval[b] , hval[a]
 ; hval[a] .heapIndex,hval[b].heapIndex :=
     hval[b] .heapIndex,hval[a].heapIndex
 ]|

siftDown(src_i : int)
var
  i , p_i : integer;
|[ i := src_i
 ;do i > 0 →
     p_i := i div 2;
     if ( hkey[i] < hkey[p_i]) → { Réordonner }
        heapSwap(i,p_i)
        ;i := p_i
     [] ( hkey[i] ≥ hkey[p_i]) → { Tas OK }
         ;i := −1
     fi
 od ]|
```

Algorithme D.1 – Binary Heap

Binary Heap (suite)

```
siftUp( src_i : int )
var
  i , l_i , r_i : int ;
  flag : boolean;
|[ i := src_i
 ; flag := true
 ;do flag →
      l_i , r_i := 2∗i,2∗i+1
      ; if ( l_i < hsize) →
          if ( r_i < hsize) → { 2 fils }
             if ( hkey[l_i] < hkey[r_i] ) → { Fils gauche + petit }
                if ( hkey[i] > hkey[l_i] ) → { Réordonner }
                   heapSwap(i,l_i)
                   ; i := l_i
                [] ( hkey[i] ≤ hkey[l_i]) → { Tas OK }
                   flag := false
                fi
             [] ( hkey[l_i] ≥ hkey[r_i]) → { Fils droit + petit }
                if ( hkey[i] > hkey[r_i] ) → { Réordonner }
                   heapSwap(i,r_i)
                   ; i := r_i
                [] ( hkey[i] ≤ hkey[r_i]) → { Tas OK }
                   flag := false
                fi
             fi
          [] ( r_i ≥ hsize) → { 1 fils }
             if ( hkey[i] > hkey[l_i] ) → { Réordonner }
                heapSwap(i,l_i)
                ; i := l_i
             [] ( hkey[i] ≤ hkey[l_i]) → { Tas OK }
                flag := false
             fi
          fi
      [] ( l_i < hsize) → { 0 fils, Tas OK } flag := false
      fi
  od ]|
```

Algorithme D.2 – `Binary Heap` (suite)

Annexe E

Système distribué

E.1 Architecture du système distribué

Nous avons implémenté un **système distribué moderne dans son interface mais relativement classique dans ses principes**. Son schéma est représenté sur la figure E.1.

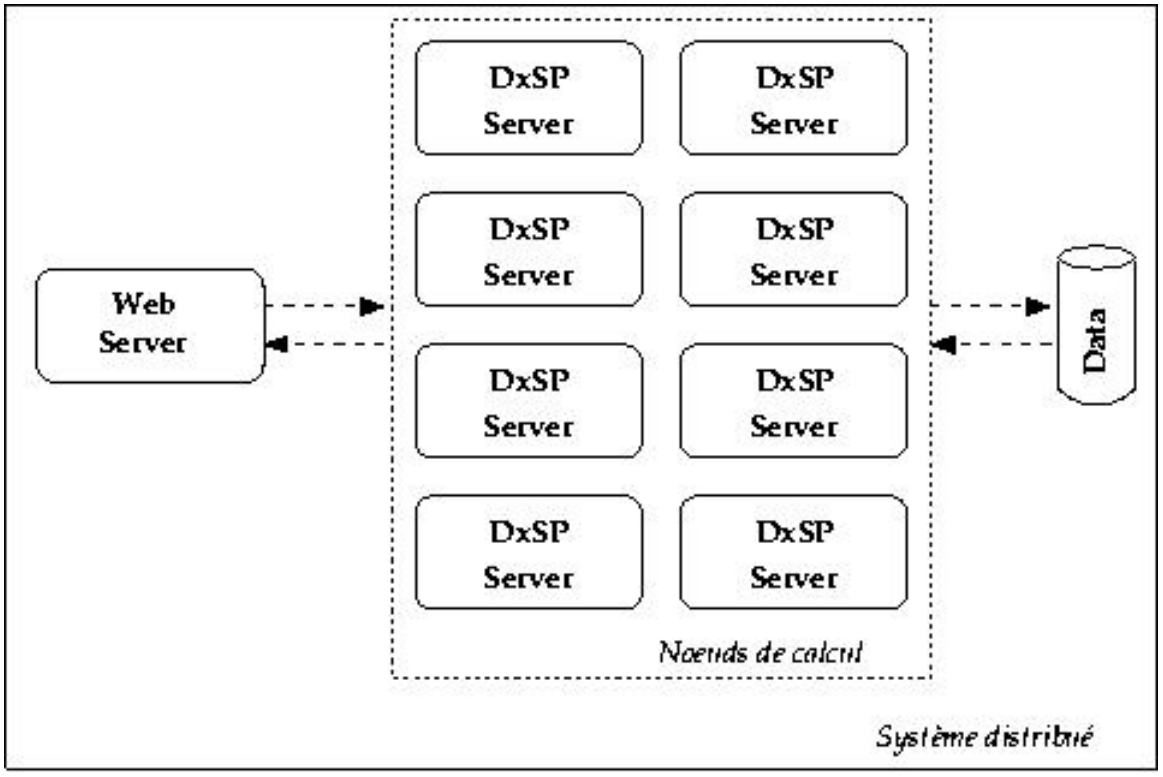


FIG. E.1 – Schéma du système distribué

Le système est composé de :

- **1 serveur web**
- **plusieurs serveurs de calcul**
- **1 serveur de bases de données**

reliés entre eux par des connexions TCP/IP non permanentes [Rif96].

E.1.1 Serveur web

Le rôle du serveur web est de **proposer aux décideurs souhaitant consulter les données traitées par `DxSP` une interface simple, pratique et indépendante du système distribué** effectuant les calculs dont on souhaite consulter les résultats.

Proposer à l'utilisateur des données issues de ce projet de surfer sur un site web et cliquer sur quelques liens, sans aucune configuration particulière de son navigateur, répond à ces critères.

Un **serveur HTTP** [NP96] a donc été implémenté. Il génère une page de liens vers toute une série de matrices d'indice d'accessibilité proposées sous forme de fichiers d'images PNG (*Portable Network Graphics*).

Un des travaux [Bri02] réalisé dans le cadre de notre DEA était l'étude par simulation des performances du cache d'un serveur d'images HTTP.

Nous disposions donc d'une base de code intéressante que nous avons exploitée au maximum, ne modifiant qu'un minimum de code.

Le schéma du serveur web modifié est présenté sur la figure E.2.

Décrivons succinctement les différents composants du serveur web.

Un démon HTTP est à l'écoute sur un port TCP fixé et traite les requêtes qui lui sont adressées de manière concurrente.

Un registre permet d'identifier les requêtes valides, ce qui isole le composant chargé de l'analyse de celles-ci du reste du serveur pour des raisons de sécurité regrettablement claires : le nombre d'attaques de serveurs webs par buffer overflow est chaque jour croissant. Il s'agit d'un premier pas vers un niveau de sécurité acceptable.

Une file d'attente séquentialise les requêtes. On peut fixer sa taille de manière à diminuer l'impact des attaques par déni de service, malheureusement très courantes également.

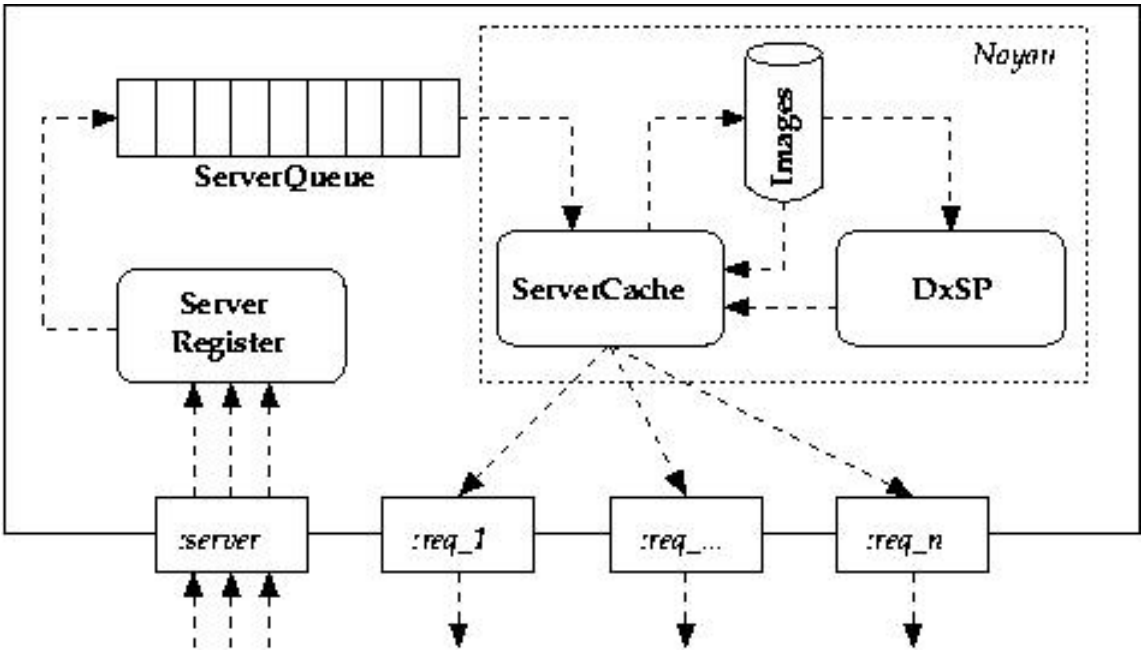


FIG. E.2 – Schéma du serveur web

Le noyau du serveur web est centré sur un cache d'images. Si la requête extraite de la file d'attente demande l'envoi d'une matrice présente dans le cache, celle-ci est envoyée immédiatement. Sinon, elle est générée en faisant appel aux noeuds de calcul. Dès que la collecte des morceaux de résultats envoyés par chaque noeud de calcul est terminée, la matrice d'indice d'accessibilité est créée et convertie en image PNG. Elle est ensuite envoyée au client.

La matrice et l'image PNG sont alors stockées dans le cache. Le service du client est alors terminé.

E.1.2 Serveurs de calcul

Le rôle de chaque serveur de calcul est d'**exécuter `DxSP` sur les données spécifiées par une requête du serveur web**, puis de renvoyer à celui-ci la matrice de résultats calculée.

La complexité de requêtes provenant du serveur web est vraiment réduite au strict minimum : il s'agit de plusieurs chaînes de caractères séparées par un caractère spécial. La forme des résultats renvoyés est tout aussi minimaliste car les valeurs calculées sont envoyées sous forme d'un flux continu d'octets (le serveur web connaît la structure et la taille des résultats à récupérer).

Comme toute l'implémentation de `DxSP` a été rédigée en C ANSI, un très grand nombre d'équipements peuvent faire office de serveur de calcul.

Etant donné que le concept de distribution des calculs n'était pas présent explicitement dans les demandes initiales du LEPUR, nous avons développé un logiciel opérationnel mais qui demandera peut-être certaines adaptations mineures pour fonctionner sur les équipements accessibles par le LEPUR, qui sont majoritairement des clients Windows.

Plus précisément, toute la partie *scientifique* (`DLOSP` + `DGSP` + gestion des données, de la mémoire, ...) est 100% portable (elle a été testée avec succès dans des environnements variés) mais toute la partie de communications réseaux nécessitera d'être adaptée si on souhaite la porter sur un système non POSIX, étant donné l'utilisation de sockets BSD. Cette partie du logiciel a toutefois été volontairement compartimentée par rapport à la base de code portable. Le nombre de lignes de code à réécrire est donc restreint (~ 200).

Un dernier point à préciser est qu'une couche logicielle dédiée à la gestion des allocations et désallocations de blocs de mémoire a été développée de manière à pouvoir à tout moment effectuer un bilan de celles-ci. En effet, chaque noeud de calcul doit pouvoir sans problème rester disponible pour une durée indéterminée. Dès lors, éviter les fuites de mémoire (*memory leaks*) est impératif si on souhaite que les noeuds de calcul restent disponibles pendant de très longues périodes.

E.1.3 Serveur de bases de données

Le rôle du serveur de bases de données est de **fournir les matrices de données aux serveurs de calcul**.

La solution choisie pour implémenter ce serveur est très simple : les serveurs de calcul accèdent directement aux matrices à travers leur système de fichiers. Celui-ci est évidemment distribué (ce qui évite la réplication des données) et les fichiers sont accessibles de manière transparente au travers du protocole NFS.

Deux motivations ont conduit à ce choix. D'une part, le volume de données de test ne nécessitait pas de solution plus élaborée. D'autre part, un serveur NFS était déjà installé au Laboratoire d'Algorithmique.

La mise en oeuvre d'une solution plus simple encore (système de fichiers local, par exemple) ne pose aucun problème. Par contre, une couche d'accès aux données devrait être développée si une vraie base de données était utilisée.

Avant de poursuivre, cette interrogation au sujet du stockage des résultats calculés mérite d'être soulevée : une fois calculés, est-il vraiment nécessaire de recalculer les résultats à nouveau quelques semaines plus tard ? Pourquoi ne pas stocker les

résultats et les diffuser avec un serveur HTTP standard ?

La réponse est simple : le nombre de combinaisons possibles des valeurs des paramètres est vraiment très grand et le but d'un serveur d'images tel qu'envisagé dans ce document est de permettre aux utilisateurs de tester de nombreuses hypothèses de manière aisée, en conservant un accès rapide aux résultats récemment ou le plus souvent calculés.

E.1.4 Synthèse

Les utilisateurs naviguent (protocole HTTP) sur le contenu généré par le serveur web développé en Java. Celui-ci distribue les calculs demandés sur plusieurs noeuds de calcul. Sur chacun de ceux-ci, un petit serveur développé en C POSIX est à l'écoute des requêtes (protocole *ad hoc*, très simple) du serveur web. Le serveur d'un noeud de calcul fait alors appel aux algorithmes `DxSP` développés en C ANSI et renvoie le morceau de matrice généré au serveur.

E.2 Perspectives

Voici plusieurs idées pour améliorer le système distribué ainsi que quelques perspectives qu'il semble intéressant d'investiguer.

E.2.1 Serveur web

La politique actuellement implémentée d'éjection d'une donnée du cache du serveur web est LRU (*Least Recently Used*), qui donne en général de bons résultats. En plus d'étudier le temps d'inutilisation des images stockées dans le cache, il pourrait s'avérer intéressant d'effectuer des statistiques sur les paramètres des images. Par exemple, modifier la priorité d'une image en fonction du type de calcul qui a été réalisé ou de l'origine géographique de la zone traitée, selon la fréquence de ces caractéristiques parmi les requêtes.

Ce type de démarche est à comparer, dans le domaine du traitement d'images, aux opérations de convolution (ayant un effet purement local) tenant compte de manière probabiliste du contexte local (le voisinage du point courant) mais également global (l'histogramme des niveaux de couleurs de l'image entière).

Par ailleurs, une amélioration de la convivialité et des fonctionnalités de l'**interface utilisateur** - autrement dit, les formulaires de requêtes générés par le serveur web - est nécessaire mais requiert une étude préalable des besoins des utilisateurs.

Enfin, une amélioration des performances et de la robustesse des **opérations de transfert de données** sur le réseau est indispensable.

E.2.2 Serveurs de calcul

Réécrire la partie réseau du serveur de calcul pour un environnement Windows serait certainement le bienvenu, de manière à élargir le nombre de systèmes capables de faire bénéficier de leur CPU le système distribué de calcul d'indices d'accessibilité.

Une autre approche à investiguer est le développement en Java de la partie réseau du serveur de calcul.

Par ailleurs, un système d'enregistrement automatique d'un serveur de calcul auprès du (ou des) serveur(s) web qu'il accepte de servir permettrait une plus grande souplesse de management du système distribué, mais s'éloigne de l'objet de ce document.

Par contre, une attribution dynamique et progressive des bandes aux noeuds de calcul permettrait de tenir compte de la disponibilité de nouveaux noeuds de calcul après la répartition initiale. Autrement dit, tant qu'il reste des bandes à distribuer, cela revient à n'attribuer qu'une seule bande à chaque noeud. Si des noeuds de calcul supplémentaires deviennent disponibles, cela permet de les prendre en compte. Dans le cas inverse, du temps et de la bande passante seront gaspillés.

Il est donc intéressant d'investiguer cette proposition d'**attribution dynamique et progressive des bandes** uniquement dans un contexte où le nombre de noeuds de calcul disponibles est fluctuant au cours du temps.

E.2.3 Serveur de bases de données

L'utilisation d'un **vrai serveur de bases de données**, résistant aux facteurs d'échelle (*scalable*), sera certainement nécessaire pour couvrir une zone géographique étendue, comme la Région wallonne par exemple, de manière totalement transparente pour l'utilisateur. **Un aspect souvent présent dans les bases de données spatiales et les GIS est le caractère massif de la quantité de données à traiter [Vit01]**.

Enfin, une option qui n'a pas été du tout considérée mais qui pourrait présenter une certaine utilité serait de **calculer sur un seul noeud de calcul le graphe de visibilité pour toutes les bandes d'une zone géographique, ainsi que les chemins les plus courts et autres données annexes**. Stockées sur le serveur de bases de données, elles pourraient être transmises aux serveurs de calcul de manière

très fine.

Cela consommerait de l'espace de stockage, de la bande passante sur le réseau, mais diminuerait les temps de calcul. Toutefois, les temps de prétraitement sont a priori peu élevés. Il faudrait donc comparer l'impact de la charge supplémentaire sur le réseau et le serveur de bases de données avec les gains temporels observés.

E.3 Conclusion

Un **système distribué** perçu par l'utilisateur au travers d'un serveur web d'images a été implémenté, **rendant transparent l'accès aux données** issues de la résolution de `DxSP`.

Des choix simples ont été effectués lorsqu'aucune contrainte ne se posait, permettant l'implémentation d'une solution satisfaisante, opérationnelle mais évidemment non taillée sur mesure par rapport à des besoins ciblés. Il s'agit d'une **approche raisonnable en l'absence de directives spécifiques**.