UMONS Université de Mons

Faculté des Sciences

COMOD

Algorithmes d'optimisation et modélisation mathématique pour des problèmes liés à l'urbanisme

Quentin Meurisse

Thèse présentée en vue de l'obtention du titre de *Docteur en Sciences*

Jury

Dr. Alessandro Araldi

Université Côte d'Azur, France

Pr. Thomas Brihaye (Président)

Université de Mons, Belgique (FS)

Dr. Isabelle De Smet (Secrétaire)

Université de Mons, Belgique (FA+U)

Pr. Alain Hertz

Polytechnique Montréal, Canada

Pr. David Laplume (Co-promoteur)

Université de Mons, Belgique (FA+U)

Pr. Hadrien Mélot (Promoteur)

Université de Mons, Belgique (FS)

Pr. Cédric Rivière

Université de Mons, Belgique (FS)

REMERCIEMENTS

Je voudrais, dans un premier temps, remercier mes co-promoteurs Hadrien Mélot et David Laplume. Merci de m'avoir encadré et conseillé pendant ces quatre années tout me laissant de l'indépendance dans mon travail.

Je remercie également les autres membres de mon jury : Alessandro Araldi, Thomas Brihaye, Isabelle De Smet, Alain Hertz et Cédric Rivière. Merci d'avoir accepté de relire cette thèse qui sort de l'ordinaire.

Je remercie les membres du projet ARC CoMod : Vincent Becue, Thomas Brihaye, Jérémy Cenci, Emeline Coszach, Isabelle De Smet, Sesil Koutra, David Laplume, Hadrien Mélot et Cédric Rivière. Merci pour votre ouverture d'esprit qui a fait de cette collaboration entre deux communautés de chercheurs une expérience très enrichissante. Merci pour les réunions où pour chaque question posée, j'avais dix mille réponses en retour.

Je tiens à remercier, en particulier, Thomas et Cédric. Merci de m'avoir proposé des sujets de projet de master et de mémoire sortant du stéréotype «les maths ce sont des théorèmes et des équations». Sans ces sujets, cette thèse n'aurait peut-être pas vu le jour.

Je remercie mes amis qui m'ont permis de souffler et de me changer les

idées pendant mes quatre années de thèse. Je remercie particulièrement Aline. Premièrement, je la remercie d'avoir accepté que je m'inspire du template de sa thèse pour créer le mien. Je la remercie surtout pour son soutien infaillible depuis toutes ces années.

Un grand merci à ma famille de m'avoir supporté et encouragé pendant mes années d'études et de doctorat.

Enfin, merci à toutes les personnes que j'ai croisées et côtoyées et qui ont fait que ces quatre années sont passées à toute vitesse.

TABLE DES MATIÈRES

CHAPITRE 1

INTRODUCTION

En vue d'une densification urbaine durable, un outil ayant pour but d'évaluer et d'assister la conception d'îlots urbains compacts pourvus d'une densité de population cible a été créé et testé dans le cadre du projet CoMod (Compacité urbaine sous l'angle de la modélisation mathématique) et dans le cadre des travaux de De Smet [De 18]. Le concept de compacité spatiale est appliqué ici, à l'échelle architecturale, sur le bâti, le non-bâti et leur combinaison. Cette approche encourage les typo-morphologies économes en terrains et en ressources matérielles, tout en étant efficaces d'un point de vue énergétique. Afin d'éviter les potentielles dérives résultant de l'emploi à outrance de ce concept, divers critères, notamment relatifs aux espaces verts, aux ombres portées ainsi que des distances et surfaces minimales sont considérés.

Cependant, viser la compacité urbaine entraîne une conciliation difficile entre les divers critères quantitatifs et qualitatifs. Dans ce contexte, un des objectifs du projet CoMod est d'étudier des problématiques liées à la compacité urbaine à travers le prisme de modèles mathématiques et d'outils informatiques ; tout cela dans le but d'automatiser l'optimisation et l'arbitrage des critères de compacité.

La philosophie de cette thèse est de montrer la pertinence et la richesse

d'une collaboration interdisciplinaire urbanisme/mathématiques. Dans le cadre de cette thèse, nous utilisons des algorithmes et des modèles très génériques mais ayant déjà fait leurs preuves. Nous faisons ainsi appel à, notamment, des notions de *théorie des jeux* [LCS16], de *théorie des graphes* [CLRS09], des algorithmes de *recherche locale* [EG09] et des *problèmes d'optimisation* tels que le *Facility Location Problem* [FH09] et le *Knapsack Problem* [KPP04].

Ces différents outils ont pour avantage d'être très flexibles. Ils sont donc facilement adaptables à diverses problématiques ; notamment celle de la compacité urbaine. Il est important de noter que ces modèles sont très généraux. Ainsi, bien que nous les appliquions ici à la problématique de la compacité urbaine, ils peuvent aider à répondre à d'autres questions, urbanistiques ou non.

Afin de valider la pertinence des modèles considérés dans le cadre de la compacité urbaine, nous avons implémenté un programme : CUBE (Compact Urban Block Explorer). CUBE est un outil possédant plusieurs facettes. Il fournit notamment un module proposant des configurations d'îlots optimisant certains critères de compacité [MDSM$^+$20, DSMB$^+$22]. CUBE possède également des modules traitant de l'affectation des bâtiments dans un quartier.

CUBE n'a pas la prétention de remplacer le travail de l'urbaniste. Il fournit des outils assistant l'utilisateur dans la conception d'îlots compacts et doit être perçu comme un terminal de contrôle. L'utilisateur reste libre d'entrer les paramètres des différents critères de compacité pris en compte. Le programme propose ensuite une solution optimisant au mieux les différents critères.

En plus de proposer et implémenter des modèles mathématiques adaptés à l'étude de la compacité, le troisième défi rencontré dans cette thèse est de maintenir le dialogue entre plusieurs communautés de chercheurs : les urbanistes, les informaticiens et les mathématiciens. Ainsi, le présent document est rédigé de sorte à être compréhensible par un maximum de lecteurs. Les spécialistes trouveront peut-être que certains concepts issus de leur domaine de prédilection sont abordés de manière succincte. Ils ne doivent cependant pas

oublier les autres communautés de chercheurs au bagage significativement différent, qui vont découvrir un domaine dans lequel ils ne sont pas familiers. Le lecteur est invité à se référer à la bibliographie pour approfondir les différents sujets rencontrés dans cette thèse.

Le chapitre 2 présente brièvement le problème de l'étalement urbain et la notion de compacité urbaine. Les chapitres suivants sont organisés de sorte à faire transparaître l'augmentation du nombre de critères de compacités pris en compte par CUBE, mais également l'intégration de l'échelle urbanistique.

Pour faciliter la lecture de ces chapitres, les titres sont syntaxiquement organisés comme suit : «notions d'urbanisme et modèle mathématique». Ainsi, le chapitre 3 traite de la problématique de la porosité. Cette notion liée à la surface non-bâtie de l'îlot est abordée via des algorithmes de recherche locale.

Dans le chapitre 4, nous considérons des critères de compacité supplémentaires. Ces nouveaux critères portent, notamment, sur la présence d'espaces verts au sein de l'îlot. Les espaces verts apportent les premiers conflits importants entre critères. Afin de gérer ces conflits, nos algorithmes se basent sur des notions de théorie des jeux.

Le chapitre 5 traite de l'affectation des bâtiments. Les travaux de De Smet préconisent que les citoyens aient un accès rapide à diverses facilités (écoles, commerces de proximité, *etc.*). Considérer l'affectation des bâtiments demande de quitter l'échelle de l'îlot pour celle du quartier, voire de la ville. Déterminer le meilleur emplacement pour ces différentes facilités se fait grâce à des problèmes d'optimisation.

Le chapitre 6 présente un mode d'emploi des différents modules de CUBE. Ce dernier chapitre permet également de rappeler brièvement les différents modèles et algorithmes présentés dans les chapitres précédents.

CHAPITRE 2

ÉTALEMENT URBAIN *VERSUS* COMPACITÉ

L'accroissement démographique, les modalités de la croissance économique, les modes de vie et de consommation contemporains entraînent des formes de développement urbain, nécessitant des besoins en fonciers, en matériaux et en énergies toujours grandissants. Partout dans le monde, en particulier en Wallonie [CR21], le phénomène *d'étalement urbain* est observé. Ce phénomène, bien qu'indicateur de croissance économique, présente de nombreux inconvénients, notamment une utilisation trop importante de ressources foncières.

Le but de ce chapitre est de présenter brièvement le problème de l'étalement urbain, ses inconvénients, ainsi qu'une des solutions envisagées dans la littérature : l'étude des villes *compactes*.

2.1 Étalement urbain

L'étalement urbain est une notion complexe pouvant être définie de plusieurs façons. Dans leur rapport de 2016 [HSO+16], l'European Environment Agency (EEA) et le Federal Office for the Environment (FEON) ont réalisé une revue de la littérature pour finalement retenir la définition suivante, également retenue par l'Institut wallon de la prospection et de la statistique (Iweps) dans son rapport de 2021 [CR21] :

Définition 2.1.1 (Étalement Urbain [HSO+16, CR21])**.** L'étalement urbain est un phénomène perceptible visuellement dans le paysage. Un paysage est affecté par l'étalement urbain lorsqu'il est imprégné par le développement urbain ou des bâtiments isolés et lorsque l'occupation des sols par habitant ou par emploi est élevée. Plus la surface bâtie est importante dans un paysage donné (quantité de surface bâtie) et plus cette surface bâtie est dispersée dans le paysage (configuration spatiale), plus la consommation de surface bâtie par habitant ou par emploi est élevée (intensité d'utilisation plus faible dans la zone bâtie), plus le degré d'étalement urbain est élevé.

Cette définition met en avant trois dimensions principales de l'étalement urbain :

- *L'artificialisation du sol*, *i.e.* la consommation de surfaces initialement dédiées à l'agriculture ou la forêt, entre autres, par des surfaces artificialisées bâties ou non bâties ;
- *Une faible efficience/intensité de l'utilisation* qui est faite du sol artificialisé par rapport au nombre d'habitants et au nombre d'emplois ;
- *La dispersion spatiale de l'artificialisation*, soit l'éparpillement sur le territoire des surfaces artificialisées et des activités humaines qui y sont liées. La dispersion peut être mesurée sans référent(s) spatial(aux) ou par rapport à des lieux particuliers définis et adaptés au contexte, par exemple des centralités [1].

La Figure 2.1 illustre ces trois dimensions. Les deux premiers schémas illustrent la tendance au desserrement urbain induite par l'étalement, les deux derniers la tendance à la dispersion.

Certains auteurs présentent l'étalement urbain comme une étape dans le développement naturel d'une ville. Celle-ci peut partir d'une forme compacte et ensuite se décentraliser suivant la croissance démographique et économique.

1. Nous qualifions de centralité la capacité d'action d'un élément central (le centre urbain, la métropole) sur sa périphérie en termes de desserte, de services, d'attractivité, et d'une manière générale, de polarisation [Gé19].

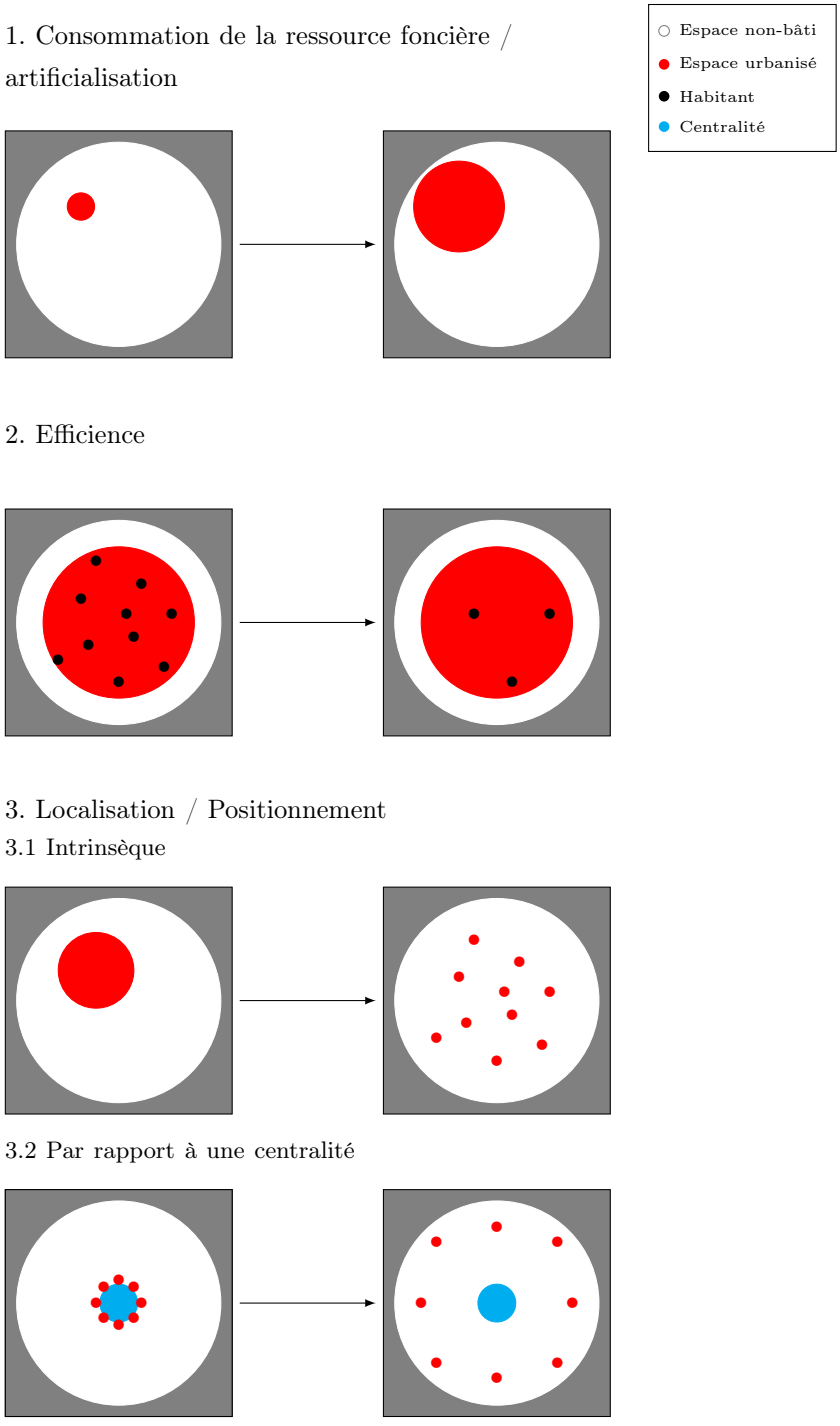


Figure 2.1 – Les différentes dimensions de l'étalement urbain [CR21, HSO+16]

Au bout d'un certain temps, les zones étalées peuvent redevenir plus compactes via un remplissage, la sous-division des parcelles et un développement plus dense [Tor06]. Cependant, ce niveau d'étalement ne peut être considéré comme naturel de manière significative seulement durant les périodes de faible urbanisation et peut être évité avec une planification adéquate.

De potentiels avantages de l'étalement urbain sont liés à l'économie et la pollution de l'air. Les entreprises peuvent avoir tendance à bouger vers la périphérie de la ville afin d'avoir accès à plus d'espace et d'améliorer les connexions avec les systèmes de transport. Même si cela peut augmenter les distances entre les zones résidentielles et les lieux de travail, donc augmenter le trafic en banlieue urbaine et la pollution atmosphérique, certains auteurs [Ing98] affirment que l'étalement urbain réduit les transports de marchandises en centre-ville, et donc la pollution de l'air et les embouteillages. La décentralisation des zones résidentielles et des lieux de travail est vue pour certains comme un moyen efficace de réduire la pollution de l'air en centre-ville et les trajets entre les banlieues résidentielles et les lieux de travail.

Malgré ces quelques avantages, l'étalement urbain présente de nombreux inconvénients environnementaux, économiques et sociaux. L'appropriation des terres est une des plus grandes menaces de l'étalement urbain. Les sols sont une ressource finie, leur perte ou leur destruction est dans la plupart des cas irréversible. Le rapport de l'EEA [HSO+16] donne certaines conséquences d'étalement urbain ainsi que des références pour approfondir le sujet. Les conséquences sont reprises par thématique dans la Table 2.1.

Table 2.1 – Conséquences environnementales, économiques et sociales de l'étalement urbain [HSO+16]

Thématique	**Conséquences**
Couverture des sols	• Assimilation des sols par les bâtiments et autres infrastructures et pertes de terres cultivables.
	Suite à la page suivante

Table 2.1 – suite

Thématique	**Conséquences**
	• Suppression et altération de terres cultivables sur de grandes étendues. • Tassement du sol, imperméabilisation des surfaces du sol, perte des fonctions écologiques du sol, perte de perméabilité à l'eau, réduction de la régénération des eaux souterraines et réduction de l'évapotranspiration, désertification .
Géomorphologie	• Modifications locales de la géomorphologie, (par exemple coupe ou stabilisation des pentes) sur de plus larges surfaces.
Changements climatiques et énergétiques	• Changement des conditions microclimatiques résultant de l'effet d'îlot de chaleur urbaine qui entraîne une réduction de la couverture végétale, une diminution de l'albédo (pouvoir réfléchissant d'une surface) et un réchauffement des surfaces et variabilité accrue des températures. • Baisse de l'humidité de l'air suite à un rayonnement solaire plus important, humidité stagnante en raison du tassement du sol et variabilité accrue de l'humidité. • Perturbation de la circulation du vent à cause de la suppression de la végétation et de la construction de bâtiments. • Accroissement des consommations d'énergie et des émissions de gaz à effet de serre par personne. • Réduction de l'absorption du CO_2 suite à l'élimination de la végétation telle que les forêts et les prairies sur de grandes surfaces.

Suite à la page suivante

Table 2.1 – suite

Thématique	**Conséquences**
Pollution atmosphérique, sonore et lumineuse	• Pollution atmosphérique plus élevée par habitant en raison des gaz d'échappement des véhicules, des engrais, de la poussière, du sel de déneigement, de l'huile, du carburant et d'autres substances qui provoquent la pollution de l'air et de l'eau ainsi que l'eutrophisation (accumulation de nutriments tels que l'azote et le phosphore dans l'environnement). • Plus grande pollution sonore (source d'insomnies). • Plus grande pollution lumineuse, altération des conditions lumineuses et autres stimuli visuels. • Allongement des distances pour le transport et le traitement des déchets, qui contrebalance les effets positifs du recyclage des matériaux.
Eau	• Altérations hydrologiques des bassins versants résultant de la diminution de la quantité et de la qualité des eaux souterraines et des changements de niveau des nappes phréatiques. • Modification des cours d'eau de surface. • Pollution des eaux de pluie par l'abrasion des pneus, la poussière et les métaux lourds qui se déversent dans les rivières. • Risques de fuites par habitant plus élevés suite à l'agrandissement du réseau de canalisations.
Suite à la page suivante	

Table 2.1 – suite

Thématique	**Conséquences**
	• Drainage, évacuation plus rapide de l'eau et augmentation du risque d'inondation en raison, entre autres, de l'étanchéité des surfaces. • Diminution de la dynamique hydrologique des zones humides autour des villes étalées. • Compétition entre l'irrigation agricole et la consommation d'eau par les citadins (*e.g.* pendant les étés secs).
Faune et flore	• Perte d'habitat pour les espèces indigènes et parfois création de nouveaux habitats avec des conditions spéciales. • Perte de la biodiversité des sols. • Augmentation du nombre d'espèces invasives et propagation de ces dernières en conséquence des changements des conditions climatiques. • Diminution de la résilience des écosystèmes. • Appauvrissement ou altération des communautés d'espèces. • Modification des réseaux alimentaires en raison de l'altération des disponibilités en nourriture. • Effet de barrière, fragmentation des habitats, perturbation des voies de migration, entrave à la dispersion, augmentation de la mortalité routière de la faune, isolement des populations, dégradation des réseaux écologiques, perte des infrastructures vertes existantes.

Suite à la page suivante

Table 2.1 – suite

Thématique	**Conséquences**
	• Isolation génétique et augmentation de la consanguinité de la faune. • Recolonisation restreinte des parcelles d'habitats vides.
Paysages	• Invasion des paysages par des zones bâties.
Économie	• Coûts de transport plus élevés liés aux déplacements domicile-travail pour les ménages. • Plus grande demande pour les transports, utilisation plus élevée de la voiture. • Augmentation des divers coûts liés aux infrastructures des transports. • Réduction de la production alimentaire et de l'autosuffisance, et une plus grande dépendance à l'égard des importations alimentaires. • Perte de revenus liés au tourisme pour les zones où les paysages ont été dégradés.
Impacts sociaux et qualité de vie	• Lieux de vie souhaités par de nombreuses personnes car les logements à faible densité offrent plus d'intimité et des jardins plus vastes que dans les quartiers densément bâtis des villes. • Proportion plus élevée de ménages individuels, ce qui entraîne un style de vie plus gourmand en ressources. • Plus grande ségrégation du développement résidentiel en fonction du revenu. • Allongement des trajets domicile-travail et réduction des interactions sociales.
Suite à la page suivante	

Table 2.1 – suite

Thématique	Conséquences
	• Problèmes respiratoires (*e.g.* asthme) suite à la pollution de l'air. • Augmentation de l'usage de la voiture au détriment de la marche augmentant l'obésité, le stress et diminuant l'activité physique.

Afin d'essayer de réduire, voire éviter les effets négatifs de l'étalement urbain, une des solutions possibles est d'étudier les villes *compactes*.

2.2 Compacité

La notion de *ville compacte* est construite en totale opposition à celle de l'étalement urbain. Une ville compacte se veut plus dense, plus économe en énergie et moins polluante de par sa haute densité. La compacité est un modèle dont la densité n'est qu'un de ses indicateurs. La ville compacte reprend les attributs de la ville pédestre, où se déplacer à pied est le moyen de transport principal [Pou04]. Neuman identifie les caractéristiques suivantes pour la ville compacte [Neu05] :

Définition 2.2.1 (Attributs de la ville compacte [Neu05])**.** Un ville compacte est caractérisée par :

- des densités résidentielles et d'emploi élevées ;
- une mixité des utilisations des sols ;
- une proximité d'utilisations variées et une taille modérée des parcelles de terrain ;
- une augmentation des interactions économiques et sociales ;
- un développement contigu (certaines parcelles peuvent être abandonnées, ou vacantes ou être des parkings de surfaces) ;
- un développement urbain contenu, délimité par des limites lisibles ;
- des transports multimodaux ;

- des hauts degrés d'accessibilité locaux et régionaux ;
- des hauts degrés de connectivité des rues (internes et externes), incluant les trottoirs et les pistes cyclables ;
- un faible ratio des espaces ouverts ;
- un contrôle unitaire de la planification de l'aménagement des sols ;
- une capacité fiscale suffisante du gouvernement pour financer les équipements et infrastructures urbains.

Un des avantages de la ville compacte est de permettre une économie des sols urbanisés, *i.e.* des terrains utilisés pour le développement de la ville. Ceci permet la protection des espaces verts et la préservation des sols cultivables. La ville compacte induit également une économie d'énergie due aux déplacements [Pou04]. Ce dernier point est appuyé par la courbe de Newman et Kenworthy (Figure 2.2) qui lie de façon inversement proportionnelle la densité résidentielle et la consommation d'énergie par habitant.

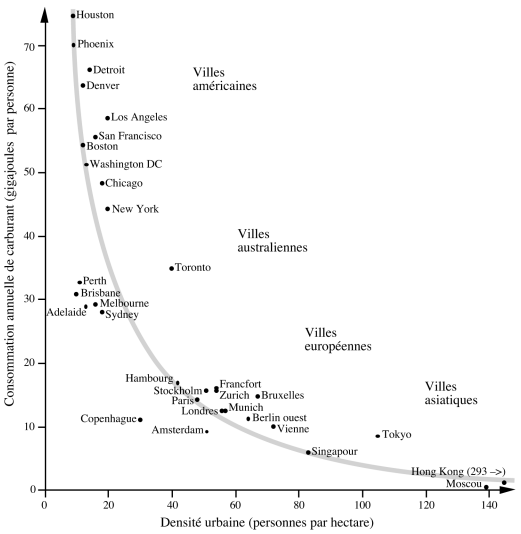


Figure 2.2 – Consommation de carburant et densité urbaine d'après Peter Newman et Jeffrey Kenworthy [Des11].

Nous étudions dans un premier temps la compacité à l'échelle de l'îlot urbain. En urbanisme, un îlot est défini comme un ensemble restreint de bâtiments et de parcelles non bâties entouré par des rues ou des avenues en réseau public (Figure 2.3). Une attention particulière est portée sur les répercussions spatiales dérivées du processus de compactification. En effet, un ensemble de contraintes spatiales et morphologiques doit être pris en compte lors de la création d'îlots de logements compacts afin d'éviter une augmentation exagérée de la compacité tout en favorisant les qualités spatiales des espaces compacts.

Figure 2.3 – À gauche un îlot de la ville de Dersden [Kö08]. À droite, Barcelone et le plan Cerdà où chaque «carré» de bâtiments est un îlot [Alh07].

Un ensemble de contraintes ont été définies [DSL19] et sont prises en considération dans notre étude :

- un volume et une superficie d'habitat minima par type de logement ;
- un pourcentage de superficie d'espaces verts par îlot, dont 50% sont constitués de maximum deux espaces de forme considérée comme compacte ;
- un apport de luminosité suffisant sur les façades, les espaces extérieurs et les espaces intérieurs ;
- une distance minimale entre façades du point de vue de la promiscuité et de l'accessibilité par les services de secours ;

- un sentiment de fermeture de l'ensemble et de ses composants, permettant notamment une certaine porosité entre l'intra-îlot et l'extérieur.

Cibler la compacité urbaine fait face à une conciliation difficile entre divers paramètres quantitatifs et qualitatifs. L'outil développé dans la thèse de De Smet [De 18] est un outil statique, dans lequel concilier l'ensemble des paramètres nécessite une utilisation «par tâtonnements», rendue fastidieuse par un encodage manuel des paramètres. Ces derniers doivent être préalablement calculés par l'utilisateur à chaque étape de la conception, et ce, en raison du caractère itératif du processus de conception. Une facilité considérable pourrait découler d'un interfaçage de l'outil avec un logiciel de représentation 3D. Cette disposition permettrait le calcul automatique de la majorité des indicateurs, sur la base d'un modèle DAO (dessin assisté par ordinateur) de l'îlot, dans toutes ses composantes. Il deviendrait alors possible de guider plus efficacement l'utilisateur de l'outil, sur base d'une configuration donnée d'un projet, quant à l'intérêt d'une modification spécifique envisagée. Cette approche nécessiterait un procédé permettant de hiérarchiser les indicateurs et contraintes. L'outil aurait alors toutes les propriétés nécessaires à la mise en œuvre d'une véritable démarche d'optimisation, au sens mathématique du terme [MDSM$^+$20, DSMB$^+$22].

Dans ce contexte, l'objectif principal du Projet CoMod et donc de la présente thèse est d'implémenter un outil générant des configurations d'îlots compacts afin d'assister l'urbaniste dans son travail de planification. Pour ce faire, nous utilisons divers modèles mathématiques et algorithmiques (la recherche locale, la théorie des jeux, la théorie des graphes, entre autres). Appliquer ces modèles à la question de la compacité de l'îlot urbain n'est qu'un exemple parmi toutes leurs autres utilisations potentielles de ces modèles.

2.3 Méthodologie et conception de CUBE

Le développement des nouvelles technologies augmente l'intérêt des chercheurs pour les outils informatiques dans l'étude de morphologies urbaines. Carpio-Pinedo *et al.* illustrent le potentiel du *Computational Design*, *i.e.* le design de formes urbaines à l'aide d'outils informatiques. Il ressort de leur article [CPRML20] que tels outils sont particulièrement adaptés pour créer, designer et évaluer de nouvelles formes urbaines.

Parmi les processus pouvant se retrouver sous la dénomination de *Computational Design*, citons le *Solid Modeling* de Shapiro [Sha02]. Le *Solid Modeling* consiste à représenter des objets urbanistiques comme des solides dans le sens géométrique du terme. Avoir un objet géométrique facilite l'utilisation d'algorithme de représentation de l'objet. De plus grâce à un tel algorithme, il est aisé de modifier les dimensions de l'objet.

Un autre exemple de *Computational Design* est la modélisation paramétrique utilisée par Moura [Mou15]. La modélisation paramétrique simule les résultats de paramètres urbain dans le paysage volumétrique d'une ville. La cartographie dynamique induite par la modélisation paramétrique permet d'explorer les impacts des conditions actuelles définies par la loi et les changements potentiels de ces paramètres. À l'aide de la modélisation paramétrique et d'outils informatiques, l'article de Moura [Mou15] présente une simulation du futur du paysage urbain du lac Pampulha au Brésil en considérant les paramètres urbains actuel ainsi que des études prédictives sur les possibles transformations de cette zone.

En géographie, dans le même état d'esprit, certains chercheurs s'intéressent à des méthodes dites multi-agents. Le multi-agent se base sur un ensemble de règles empiriques associées à chaque agent du système (qui peut représenter les individus, les bâtiments ou autre encore). Ces agents interagissent entre eux pour simuler l'évolution de processus urbains. Via un modèle multi-agents, Sanders *et al.* ont implémenté SIMPOP [SPM+97], un outil permettant de simuler l'évolution d'une colonie au cours du temps. Citons également les travaux

d'Ali *et al.* [ASL+20] et de Yang *et al.* [YGY19] qui présentent des modèles multi-agent permettant respectivement d'ajouter de la végétation et ville et de simuler des mouvements de foules.

Notre démarche s'intègre totalement dans le *Computational Design*. Elle partage de nombreux points commun avec les travaux mentionnés ci-dessus, via l'utilisation d'algorithmes et modèles multi-agents pour résoudre ou étudier des problématiques urbaines. Cependant, notre méthodologie se démarque sur plusieurs aspects.

Si nous devions la résumer très brièvement, la méthodologie des travaux précédents est la suivante : énoncé une problématique urbaine, concevoir un modèle et un algorithme *dédié* à cette problématique. Notre méthodologie se démarque par l'utilisation de modèles issus des sciences mathématiques pour étudier la problématique urbaine. Ces modèles sont très généraux et flexibles. Ainsi, nos modèles et algorithmes peuvent être aisément adaptés pour résoudre *d'autres problématique*. De plus, le modèle créé est le fruit de plusieurs échanges entre urbanistes, informaticiens et mathématiciens. Ces échanges permettent d'une part d'identifier des outils pertinents dans le cadre urbanistique et d'autre part de valider les solutions proposées par notre programme.

Notre outil générateur d'îlots, CUBE (Compact Urban Block Explorer), est composé de plusieurs modules. Chacun d'entre eux répond à une problématique liée à la compacité urbaine. CUBE est implémenté en suivant les principes de la programmation orientée objet. Ainsi, certaines classes composant ces modules sont présentées dans le texte afin d'assurer une bonne compréhension du fonctionnement de CUBE.

La méthodologie employée afin de construire CUBE est la suivante. Nous commençons par ne considérer qu'un seul et unique critère de compacité, simple à optimiser. Nous utilisons des modèles et méthodes mathématiques adaptés pour résoudre le problème d'optimisation ainsi défini. Nous ajoutons ensuite progressivement de nouveaux critères et contraintes afin d'améliorer la pertinence des solutions retournées par les modèles. Une fois qu'un modèle montre

ses limites pour le problème considéré, nous utilisons un autre modèle mathématique plus complexe, mais qui outrepasse les limitations rencontrées. Dans les chapitres qui suivent, nous présentons les modèles que nous avons utilisés dans le cadre du projet CoMod et les résultats obtenus grâce à ceux-ci.

Pour faciliter la lecture de ces chapitres, les titres sont syntaxiquement organisés comme suit : «notions d'urbanisme et modèle mathématique». Ainsi par exemple, le chapitre 3 traite de la problématique de la porosité. Cette notion liée à la surface non-bâtie de l'îlot est abordée via des algorithmes de recherche locale.

CHAPITRE 3

POROSITÉ DE L'ÎLOT ET RECHERCHE LOCALE

Le premier critère de compacité de l'îlot [2] considéré concerne la *porosité* de celui-ci (rapport entre l'aire non-bâtie et l'aire totale de l'îlot). Là où la compacité porte sur l'occupation des sols par du bâti, la porosité porte sur le vide. Essayer de minimiser la porosité augmente le nombre de logements potentiels et donc la densité de l'îlot. La présence d'espaces vides est cependant indispensable dans la composition d'une morphologie urbaine. Il reste donc nécessaire de conserver un minimum de porosité dans les îlots construits. Un outil basé sur des algorithmes de *recherche locale* [EG09], LSCUBE (Local Search Compact Urban Block Explorer), est implémenté , afin d'étudier ce critère. LSCUBE cherche à créer des configurations d'îlots les moins poreuses possibles, tout en respectant diverses contraintes permettant de guider l'outil vers des solutions plus pertinentes.

La première partie de ce chapitre est consacrée à la présentation des notions théoriques nécessaires à la compréhension des différents algorithmes de recherche locale. Dans la seconde partie, nous décrivons les différentes modélisations envisagées pour définir un problème exploitable par les algorithmes ainsi que les solutions obtenues.

2. Pour rappel, un îlot est défini comme un ensemble restreint de bâtiments et de parcelles non bâties entouré par des rues ou des avenues en réseau public.

3.1 Recherche locale

3.1.1 Le Hill Climbing

En informatique, la recherche locale est une méthode pour résoudre des *problèmes d'optimisation*. Intuitivement, un tel problème peut être formulé comme suit : étant donné un ensemble de solutions pour un problème, nous voulons trouver les solutions qui maximisent ou minimisent un certain critère appelé fonction objectif. Formellement, un tel problème peut être défini comme suit.

Définition 3.1.1 (Problème d'optimisation)**.** Soit S un ensemble de solutions pour un problème donné. Soit $f : S \longrightarrow \mathbb{R}$ une fonction appelée *fonction objectif*. Le but d'un problème d'optimisation est de trouver $x \in S$ tel que $x = \arg\max_{s \in S} f(s)$ ou $x = \arg\min_{s \in S} f(s)$.

Remarque 3.1.2. Les problèmes d'optimisation sont souvent NP-difficiles. De tels problèmes ont un très grand nombre de solutions. Énumérer chaque solution prendrait beaucoup trop de temps et cela même avec l'aide d'un outil informatique. C'est pourquoi il est nécessaire d'utiliser d'autres techniques. Certaines de ces méthodes, comme la recherche locale, sont dites approchées. Ces algorithmes ne retournent pas systématiquement la solution optimale.

Les algorithmes de recherche locale utilisent la fonction objectif d'un problème d'optimisation afin d'évaluer la qualité d'une solution. Pour sélectionner une solution, la recherche locale utilise une opération de *voisinage*.

Définition 3.1.3 (Voisinage [EG09])**.** Une opération de voisinage N est une fonction $N : S \longrightarrow 2^S$ qui assigne à chaque solution $s \in S$ un ensemble de solutions $N(s) \subseteq S$. Un élément de $N(s)$ est appelé un *voisin* de s.

Un voisin d'une solution s est traditionnellement obtenu en appliquant une petite transformation à s. Un exemple d'opération de voisinage sur l'ensemble de solutions de la Figure 3.1 est d'attribuer à chaque solution les solutions qui lui sont adjacentes. Ainsi le voisinage de la solution verte est l'ensemble des

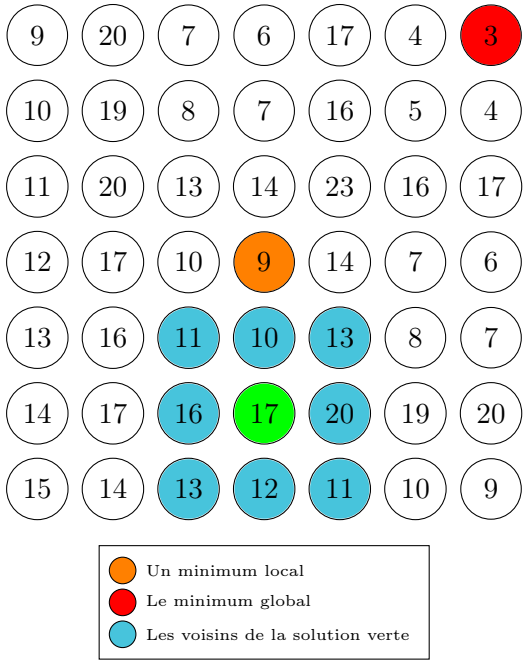


Figure 3.1 – Un ensemble de solutions (les cercles) avec leur évaluation par une fonction objectif.

solutions bleues.

L'algorithme de recherche locale le plus basique est connu sous le nom de *Hill Climbing* (Algorithme 1). En supposant vouloir minimiser la fonction objectif $f : S \longrightarrow \mathbb{R}$, cet algorithme fonctionne comme suit :

- L'algorithme démarre à partir d'une solution quelconque $s \in S$ (peut être spécifiée par l'utilisateur ou sélectionnée aléatoirement) ;
- L'algorithme génère un voisinage de s selon l'opération $N : S \longrightarrow 2^S$;
- Parmi les solutions de $N(s)$, la solution s' avec la plus petite valeur de f est sélectionnée ;
- Si $f(s') < f(s)$, alors s' devient la nouvelle solution courante ;
- Sinon, l'algorithme retourne s et s'arrête.

Le Hill Climbing s'arrête quand il a trouvé une solution meilleure que tous ses voisins. Afin d'éviter de potentielles boucles infinies (dans le cas où l'algorithme peut bouger d'une solution à une autre ayant la même valeur par exemple) ou un temps d'exécution trop long, le Hill Climbing peut également stopper la recherche quand un critère d'arrêt est satisfait (nombre d'itérations ou temps maximum).

Algorithme 1 : Hill Climbing

Entrées : Un ensemble de solutions S, une opération de voisinage $N : S \longrightarrow 2^S$ et une fonction objectif $f : S \longrightarrow \mathbb{R}$.

Sortie : La meilleure solution trouvée.

```
Soit s une solution quelconque de S
répéter
    s' ← arg min_{t∈N(s)} f(t)
    si f(s') < f(s) alors
        s ← s'
    sinon
        retourner s
jusqu'à la satisfaction de critères d'arrêt
retourner s
```

Exemple 3.1.4. Considérons l'ensemble de solutions de la Figure 3.1 et supposons que le Hill Climbing démarre de la solution verte de valeur 17. Le voisinage de la solution verte est l'ensemble des solutions bleues. La solution bleue avec la plus petite valeur est celle de valeur 10. Cette dernière solution a une meilleure évaluation que la solution de départ. Elle devient donc notre nouvelle solution courante. Le meilleur voisin de notre solution courante est la solution orange de valeur 9. La solution orange n'ayant pas de voisin de meilleure qualité qu'elle, cette dernière est retournée par l'algorithme.

Remarque 3.1.5. Comme dit précédemment, la recherche locale n'est pas une méthode exacte. Ainsi, dans l'exemple précédent, le Hill Climbing a retourné la solution orange de valeur 9. Cependant, ce n'est pas la solution avec la plus petite valeur. En effet, la meilleure solution de la Figure 3.1 est la solution rouge de valeur 3. La solution orange est un optimum *local* alors que la solution

rouge est un optimum *global*. Afin d'approcher au mieux l'optimum global d'un ensemble de solutions, nous avons besoin de méthodes plus complexes : *les métaheuristiques*.

3.1.2 Métaheuristiques

Définition 3.1.6 (Métaheuristiques [EG09])**.** Les métaheuristiques peuvent être définies comme des méthodologies générales de haut niveau pouvant être utilisées comme des guides pour résoudre des problèmes d'optimisation spécifiques.

Remarque 3.1.7. Les métaheuristiques sont des méthodes très générales. Nous les utilisons ici pour résoudre des problèmes liés à la compacité urbaine mais elles peuvent être utilisées pour résoudre d'autres types de problèmes, urbanistiques ou non.

Le *Hill Climbing* est une métaheuristique. Un autre exemple de métaheuristique est le *Random Descent* (Algorithme 2). Cet algorithme est très similaire au *Hill Climbing* dans son fonctionnement. La différence majeure réside dans la façon de sélectionner un voisin d'une solution. Au lieu de sélectionner le meilleur voisin, ici l'algorithme sélectionne un voisin de la solution courante *aléatoirement*. Si ce voisin est meilleur, il devient la nouvelle solution courante. Utiliser cette métaheuristique permet de gagner du temps de calcul lorsque les voisinages sont très grands.

Une autre métaheuristique bien connue et souvent utilisée est le *Simulated Annealing* (Algorithme 3). Une spécificité de cet algorithme est de parfois accepter, comme nouvelle solution courante, une solution de moins bonne qualité. Pour cela, l'algorithme utilise un paramètre T appelé la température du système. Notons s la solution courante trouvée par le *Simulated Annealing* et s' un des voisins s choisi aléatoirement. Si $f(s') - f(s) < 0$ *i.e.*, si s' est de meilleure qualité que s, alors s' devient la nouvelle solution courante comme pour l'Algorithme 2. Sinon, s' peut quand même devenir la nouvelle solution courante avec une probabilité de $\mathrm{e}^{-\frac{f(s')-f(s)}{T}}$. Plus la différence de qualité entre s et s' est petite, plus la probabilité que s' soit acceptée est grande. Après un

Algorithme 2 : Random Descent

Entrées : Un ensemble de solutions S, une opération de voisinage $N : S \longrightarrow 2^S$ et une fonction objectif $f : S \longrightarrow \mathbb{R}$.

Sortie : La meilleure solution trouvée.

```
1 Soit s une solution quelconque de S
2 répéter
3 |   Soit s' ∈ N(s) choisi aléatoirement
4 |   si f(s') < f(s) alors
5 |   |   s ← s'
6 jusqu'à la satisfaction de critères d'arrêt
7 retourner s
```

Algorithme 3 : Simulated Annealing

Entrées : Un ensemble de solutions S, une opération de voisinage $N : S \longrightarrow 2^S$ et une fonction objectif $f : S \longrightarrow \mathbb{R}$, T_0 la température initiale, T_{min} la température minimale que peut avoir le système.

Sortie : La meilleure solution trouvée.

```
1  Soit s une solution quelconque de S
2  T ← T_0
3  répéter
4  |   répéter
5  |   |   Soit s' ∈ N(s) choisi aléatoirement
6  |   |   si f(s') − f(s) < 0 alors
7  |   |   |   s ← s'
8  |   |   sinon
9  |   |   |   s ← s' avec une probabilité de e^(−(f(s')−f(s))/T)
10 |   jusqu'à la satisfaction de critères d'arrêt
11 |   Diminuer la valeur de T
12 jusqu'à T = T_min
13 retourner s
```

certain temps ou un certain nombre d'itérations, la température T diminue, diminuant également la probabilité qu'une moins bonne solution soit choisie.

Les métaheuristiques décrites dans cette section sont celles que nous avons testées dans LSCUBE. Cependant, il en existe d'autres, notamment, la *recherche tabou* [EG09] et les *algorithmes génétiques*. Ces derniers sont présentés dans la section 5.4.

3.2 Modélisation

Nous expliquons dans cette section comment nous modélisons notre problème de porosité afin de pouvoir utiliser des algorithmes de recherche locale. Nous décrivons ici comment nous définissons une solution de notre problème, la fonction objectif et l'opération de voisinage. Cette section se conclut par la présentation et l'analyse d'exemples de solutions retournées par LSCUBE, ici implémenté en Java 8.

3.2.1 Modélisation d'une solution

Pour calculer la porosité de l'îlot, nous n'avons besoin que de connaître l'aire totale de l'îlot et l'aire non-bâtie. Ainsi, à ce stade, il n'est pas nécessaire de prendre en compte la hauteur des bâtiments et donc de modéliser un îlot comme un objet en trois dimensions. Pour efficacement observer les différentes surfaces composant l'îlot, il est naturel de considérer la vue en plan de celui-ci.

Afin de pouvoir définir une opération de voisinage simple, de différencier deux îlots et d'identifier facilement les différents composants de l'îlot (bâtiments, espaces verts, *etc.*), la surface de l'îlot est simplement quadrillée. Les différentes cellules du quadrillage peuvent avoir différentes couleurs. Les cases blanches (dites vides), représentant du terrain à aménager. Les agglomérations de cellules de la même couleur symbolisent des bâtiments [3].

La Figure 3.2 illustre une telle modélisation. Notons que la taille de l'îlot ainsi que la taille des cellules n'est pas fixe. Lors de l'initialisation d'un pro-

3. La couleur verte est réservée à la future modélisation des espaces verts.

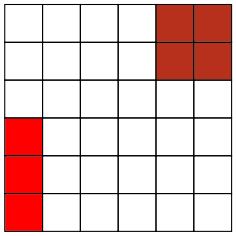


Figure 3.2 – Un exemple de modélisation d'un îlot avec deux bâtiments.

blème, l'utilisateur doit définir le nombre de cases en longueur et en largeur de l'îlot ainsi que l'aire d'une cellule du quadrillage. Dès lors, le quadrillage de la Figure 3.2 peut être composé aussi bien de cellules de 1 m sur 1 m que de 10 m sur 10 m.

La représentation d'un îlot sous forme de quadrillage permet à l'utilisateur de facilement en visualiser la configuration. La première approche utilisée pour implémenter ce quadrillage est d'utiliser une matrice. À chaque cellule de la matrice correspond une case du quadrillage. À chaque bâtiment est attribué un numéro d'identification. Toutes les cellules de la matrice ayant ce numéro sont considérées comme composant ce bâtiment. Par convention, une cellule valant 0 symbolise une case vide. Une telle matrice a l'avantage d'être facilement implémentable et utilisable par LSCUBE. Cela permet également de pouvoir facilement reconstruire le quadrillage, plus visuel. Par exemple, la Figure 3.3 est la matrice symbolisant le quadrillage de la Figure 3.2.

Cependant, utiliser une matrice pour représenter un îlot limite la forme de ce dernier à un rectangle avec un pavage carré. Or, il peut être intéressant d'avoir des géométries d'îlots plus variées et d'utiliser d'autres pavages (hexagonaux, triangulaires, *etc.*). Utiliser d'autres pavages permet d'obtenir des bâtiments avec des formes différentes et d'autres rapports entre le périmètre et l'aire qu'en considérant uniquement des bâtiments composés de modules carrés.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 3.3 – Matrice représentant le quadrillage de la Figure 3.2

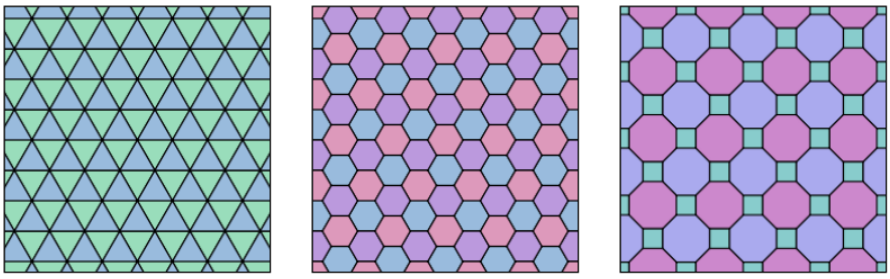


Figure 3.4 – De gauche à droite : un pavage triangulaire [Non09a], un pavage hexagonal [Non09b] et un pavage carré tronqué [Non09c].

Nous ne considérons que des pavages composés de polygones réguliers. Ce choix nous donne déjà accès à une grande variété de pavages possibles et facilite l'implémentation d'un pavage. Dans notre implémentation, pour définir un pavage, il est nécessaire de définir préalablement une cellule de ce pavage. Cela se fait via l'objet `Cell`.

Une instance de `Cell` prend en entrées le nombre de côtés de la cellule, la longueur d'un côté ainsi que le nombre maximum de cellules adjacentes que peut avoir notre instance dans un pavage. Le dernier paramètre doit être spécifié car, en fonction du type de pavage, deux cellules de même forme peuvent être adjacentes à un nombre différent de cellules. Par exemple, une cellule carrée dans un pavage carré (Figure 3.2) a, au plus, un point d'intersection

avec huit autres cellules. Alors que dans un pavage carré tronqué (Figure 3.4), elle n'en a que quatre. Un objet `Cell` possède une liste de références vers les cellules qui lui sont adjacentes lorsqu'il est utilisé dans un pavage. Enfin, remarquons que, comme les cellules sont des polygones réguliers, il est très aisé de connaître leur aire. Si nous notons $A(n, x)$ l'aire d'un polygone régulier à n côtés de longueur x, nous avons :

$$A(n, x) = \frac{x^2 n}{4 \tan(\frac{\pi}{n})}.$$

En outre, une instance de `Cell` contient l'identifiant du bâtiment auquel elle appartient, ainsi que sa couleur.

Un pavage (implémenté via l'objet `Tiling`) est simplement une liste d'objets `Cell` connectés entre eux[4]. La classe `Tiling` est une classe abstraite définissant une méthode abstraite `generate`. Chaque type de pavage est une sous-classe de `Tiling` redéfinissant la méthode `generate` en fonction de sa forme et de la forme de ses cellules.

Remarque 3.2.1. Si nous considérons que chaque cellule d'un pavage ne peut être que de deux couleurs (rouge ou blanche), alors pour un pavage ayant k cellules, il y a exactement 2^k configurations possibles. Pour donner un ordre d'idée, pour un pavage ayant 6 cellules en longueur et en largeur comme à la Figure 3.2, nous avons $2^{36} = 68\ 719\ 476\ 736$ configurations d'îlots possibles. Pour un pavage ayant 50 cellules en longueur et en largeur comme à la Figure 3.7, nous avons $2^{2500} \approx 3{,}76 \cdot 10^{752}$ solutions. Il est exclu d'énumérer toutes les solutions pour un tel problème.

3.2.2 Fonction objectif

Pour calculer la porosité de l'îlot, nous avons besoin de son *aire totale* et de son aire *non-bâtie*. Obtenir l'aire totale de l'îlot est très facile. En effet, chaque `Cell` connaissant sa surface, une instance de `Tiling` peut calculer sa surface totale en sommant cumulativement l'aire des objets `Cell` créés lors sa

4. Une instance de la classe `Tiling` peut être vue comme un graphe où les nœuds de ce graphe ont une forme géométrique spécifique et un degré fixe. La notion de graphe est détaillée à la section 5.1

génération.

Concernant l'*aire non-bâtie*, nous exploitons l'égalité suivante : *aire totale* = *aire non-bâtie* + *aire bâtie*. Nous en déduisons que $\frac{aire\ non\text{-}bâtie}{aire\ totale} = 1 - \frac{aire\ bâtie}{aire\ totale}$. Nous préférons utiliser le membre de droite de l'équation précédente. En effet, il est plus simple d'obtenir l'*aire bâtie* de l'îlot notamment grâce à l'objet `GroupOfCells`.

Un `GroupOfCells` contient un `set` (collection d'éléments non ordonnés et sans doublons) de `Cells` de même couleur et ayant le même identifiant. La classe `GroupOfCells` modélise les différents composants de l'îlot (bâtiments, espaces verts, *etc.*.). Une instance de `GroupOfCells` maintient l'aire et le périmètre de la forme qu'elle décrit sur l'îlot en fonction des cellules qui lui sont ajoutées ou retirées pendant la recherche locale. Ainsi pour calculer l'*aire bâtie* d'un îlot, il suffit de calculer la somme des aires des `GroupOfCells` identifiés comme des bâtiments.

3.2.3 Opération de voisinage

L'opération de voisinage de base consiste simplement à changer la couleur d'une cellule du pavage. Ainsi, le voisinage d'une solution donnée est l'ensemble des solutions dont la couleur d'une seule cellule diffère. Un exemple d'un tel voisinage est illustré à la Figure 3.5.

Générer les voisins d'une solution donnée se fait à l'aide de la classe `ChangeColorMove`. Une instance de cette classe prend en entrée deux paramètres : l'indice d'une cellule du pavage [5] et une chaîne de caractères contenant le code HTML d'une couleur. Si un tel objet reçoit en entrée un indice k et une couleur c, il peut, grâce à sa méthode `apply`, changer la couleur de la cellule k en la couleur c.

Conservons le code couleur de la Figure 3.5 : blanc pour les cellules vides et rouge pour les bâtiments. Pour obtenir un voisin d'une solution donnée, il suffit de changer une cellule blanche en cellule rouge ou inversement. Changer

5. Rappelons qu'un pavage est implémenté comme une liste de cellules interconnectées.

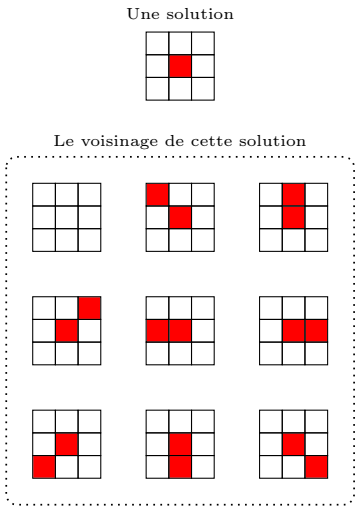


Figure 3.5 – Une solution et son voisinage.

une cellule rouge en blanche a pour effet de supprimer une cellule du bâtiment auquel elle appartenait. Si c'était la seule cellule composant un bâtiment, le bâtiment est supprimé. Lorsque qu'une cellule blanche est colorée en rouge, CUBE vérifie dans un premier temps les cellules adjacentes à la cellule visée. Si une de ces cellules appartient à un bâtiment, la nouvelle cellule rouge agrandit ce bâtiment. Sinon, un nouveau bâtiment est créé.

3.2.4 Contraintes

La solution optimale avec pour unique objectif de minimiser le rapport entre la surface non-bâtie et la surface de l'îlot est simplement l'îlot bâti à 100%. Cependant cette solution n'a que peu d'intérêt d'un point de vue urbanistique. Pour encourager l'algorithme à retourner des îlots avec des morphologies plus variées, nous devons ajouter des contraintes à notre modèle.

Les algorithmes de recherche locale peuvent travailler avec deux types de contraintes. Le premier type correspond à des contraintes dites *obligatoires*. Avec de telles contraintes, seules les solutions satisfaisant un certain critère

sont considérées comme valides et peuvent être retournées par l'algorithme, et cela même si certaines solutions ne satisfaisant pas la contrainte ont une meilleure évaluation par la fonction objectif.

Les travaux de De Smet [De 18] identifient la nécessité d'avoir une distance minimale entre les façades des bâtiments, afin de garantir un certain confort et le respect de la vie privée, par exemple 6 m entre les pièces de jour et 3,8 m pour les autres pièces. Ce critère nous sert de base à la création d'une contrainte obligatoire.

Intuitivement, vérifier si deux bâtiments sont suffisamment distants est facile. Il suffit d'identifier deux bâtiments qui se font face et de mesurer la distance entre les deux façades. Néanmoins, mettre en place tout un système de coordonnées dans l'espace ainsi que des procédés pour calculer la distance entre deux bâtiments demanderait un travail complexe et peu utile de par la construction cellule par cellule des composants de l'îlot.

En effet, nous avons connaissance des dimensions de chaque cellule. Ainsi en comptant simplement les cases vides entre chaque bâtiment, nous pouvons en déduire la distance entre ceux-ci. En pratique, l'utilisateur peut spécifier un nombre minimum r de cases devant être vides entre deux bâtiments. Lors de la création d'une cellule bâtie x durant la recherche locale, le programme vérifie si, parmi les cellules dans un rayon d'au plus r par rapport à la cellule x, il n'y aucune cellule appartenant à un autre bâtiment.

Exemple 3.2.2. Considérons la Figure 3.6 et supposons que la case centrale est la dernière colorée. Si le paramètre r de la contrainte vaut 1, alors cette solution est acceptable. Si r vaut 2, cette solution n'est pas acceptable.

N'ajouter que cette contrainte à la recherche locale n'empêche pas l'obtention d'un unique bâtiment occupant la totalité de l'îlot lors de l'utilisation d'un algorithme déterministe démarrant de l'îlot vide. Lors de l'utilisation d'algorithmes comme le *Random Descent*, les solutions retournées contiennent quantité de petits bâtiments éparpillés dans l'îlot comme illustré à la Figure 3.7. En fonction de l'échelle, une telle solution peut impliquer l'existence de bâtiments de $2\,\mathrm{m}^2$. Il est donc nécessaire d'implémenter des contraintes supplémentaires.

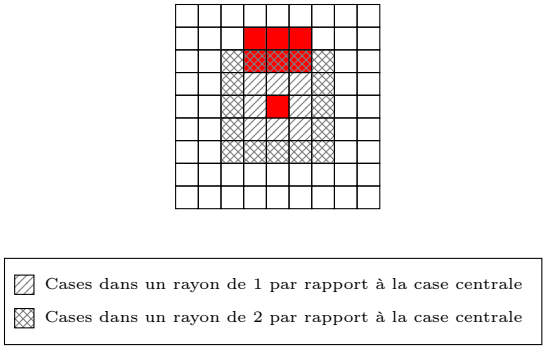


Figure 3.6 – Une case et les cellules positionnées dans un rayon d'au plus 2 par rapport à cette dernière.

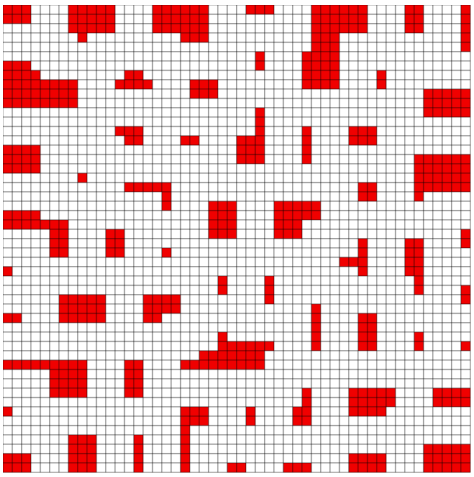


Figure 3.7 – Une solution obtenue par le *Random Descent* visant la porosité minimale et avec au moins 4 cases vides entre les bâtiments.

Les autres contraintes implémentées sont des contraintes dites *pénalisantes*. Elles ajoutent une pénalité à l'évaluation des solutions ne vérifiant pas un critère donné. La première contrainte pénalisante implémentée vérifie si le nombre appartient à un intervalle défini par l'utilisateur. Notons $b(s)$ le nombre de bâ-

timents présents dans une configuration d'îlot s. Soient $x, y \in \mathbb{N}$, les bornes de l'intervalle spécifié par l'utilisateur où $x \leqslant y$. Nous définissons la pénalité $\beta_{x,y}$ où

$$\beta_{x,y}(s) = \begin{cases} x - b(s) & \text{si } b(s) < x, \\ 0 & \text{si } b(s) \in [x, y], \\ b(s) - y & \text{si } b(s) > y. \end{cases} \tag{3.1}$$

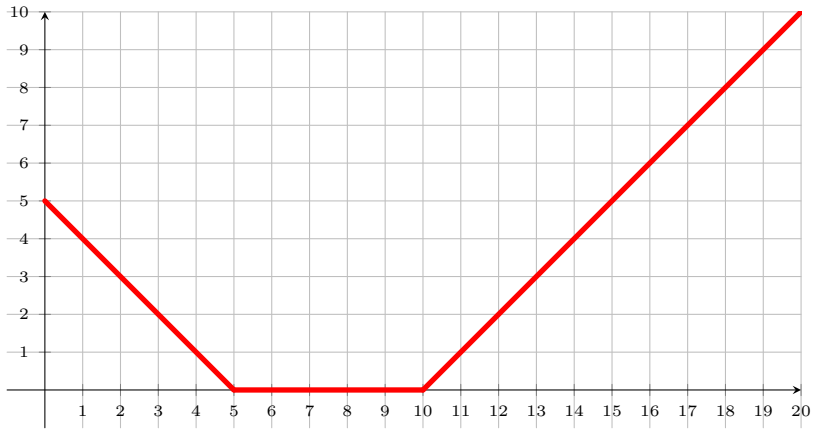


Figure 3.8 – Graphe de la pénalité $\beta_{5,10}$.

Avec cette contrainte, l'objectif à minimiser est maintenant la fonction g telle que $g(s) = f(s) + \beta_{x,y}(s)$ où f est la fonction objectif initiale. Les pénalités servent à guider la recherche locale vers des solutions respectant leurs critères. Comme illustré à la Figure 3.8, plus le nombre de bâtiments est proche de l'intervalle désiré, plus la pénalité est faible. Ainsi, LSCUBE va préférer naturellement les solutions satisfaisant la contrainte.

Remarques 3.2.3. Soit s une configuration de l'îlot et $f(s)$ la porosité de s. Pour tout s, $f(s) \in [0, 1]$. Si $b(s) \notin [x, y]$, nous avons $\beta_{x,y}(s) \geqslant 1$. Dès lors l'algorithme va prioriser la satisfaction de la contrainte à la minimisation de la porosité. Ainsi, afin d'accélérer la recherche, le programme a maintenant la possibilité de supprimer un bâtiment en entier en une seule opération de voisinage et non plus case par case en réalisant une combinaison de `ChangeColorMove`.

Ajouter uniquement cette contrainte à la recherche permet d'éviter des solutions avec beaucoup de petits bâtiments éparpillés sur l'îlot, comme dans le cas de la Figure 3.7. Cependant, cela ne limite que le nombre de bâtiments possible. Il est toujours possible d'obtenir un bâtiment immense accompagné de quelques bâtiments minuscules. Afin d'éviter cela, deux autres contraintes pénalisantes sont implémentées. La première exige que le périmètre de chaque bâtiment appartienne à un intervalle donné et la seconde que l'aire de chaque bâtiment appartienne à un intervalle donné. Pour chaque bâtiment, nous définissons une pénalité suivant la structure de l'équation (3.1) mais portant sur le périmètre ou l'aire. Les pénalités des deux contraintes sont la somme des pénalités individuelles des bâtiments.

3.2.5 Choix d'implémentation annexes

Avant de présenter quelques résultats obtenus par LSCUBE, nous détaillons certains choix d'implémentations et fonctionnalités notables.

Premièrement, l'utilisateur a la possibilité de verrouiller certaines cellules du pavage. Les cellules verrouillées ne peuvent pas changer de couleur pendant la recherche. Cette fonctionnalité permet de modéliser la présence dans l'îlot d'une zone naturelle protégée, d'un bâtiment historique, *etc.*

Deuxièmement, afin de garantir une évaluation de la fonction objectif efficace, un système d'invariants est implémenté. Un invariant [6] est un objet qui observe les solutions sélectionnées par l'algorithme et maintient diverses propriétés sur ces solutions. Ici, notre invariant maintient le nombre de bâtiments dans l'îlot, l'aire totale bâtie et le périmètre total bâti. Lorsque qu'une solution est transformée afin de générer un voisin, l'invariant est notifié et met à jour ses propriétés. Ensuite, la fonction objectif et les contraintes consultent l'invariant pour évaluer le voisin généré.

6. Ce qui ne varie pas dans un invariant, comme nous l'avons défini, est le fait que les propriétés sur la solution observée sont toujours maintenues et mises à jour.

3.2.6 Premiers résultats obtenus

Nous présentons ici certains résultats obtenus par LSCUBE. Notre implémentation se base sur la libraire JAMES [DDDF17], librairie JAVA fournissant une implémentation des différentes métaheuristiques ainsi que divers outils utiles à l'implémentation des solutions, des voisinages et des contraintes.

Les résultats présentés ont été obtenus par le *Parallel Tempering*, algorithme nous retournant les résultats les plus pertinents. Le *Parallel Tempering* est une variante du *Simulated Annealing* exploitant les processeurs multi-cœurs modernes. Sans entrer dans les détails, cet algorithme exécute plusieurs *Simulated Annealing* en parallèle. Chaque exécution a une température différente et à intervalles réguliers, elles échangent leurs solutions courantes. De plus amples informations sur le *Parallel Tempering* peuvent être trouvées dans la documentation de JAMES [DDDF17].

Les exemples suivants (Figures 3.9 à 3.12) ont été obtenus par notre programme sur un îlot de 102 m par 102 m pavé par des carrés de 2 m de côté. L'objectif est de minimiser la porosité. De plus, nous prenons en compte les contraintes suivantes : avoir entre 6 et 10 bâtiments, les bâtiments doivent avoir un périmètre compris entre 75 m et 500 m et il doit y avoir minimum 4 m (ou deux cellules vides) entre les façades de deux bâtiments.

Les couleurs des bâtiments des Figures 3.9 et 3.11 ne sont que visuelles, elles n'ont aucune signification particulière. Ces solutions ont été présentées aux membres du projet CoMod afin de débattre de leur qualité urbanistique. Il est ressorti de ces discussions que, bien que la forme des bâtiments soit peu conventionnelle (due à l'aléatoire utilisé par le *Parallel Tempering*), ces premiers résultats sont intéressants. Nous observons que, grâce à la contrainte de distance entre les façades le programme, crée naturellement des chemins dans l'îlot. Ces chemins, une fois aménagés, peuvent permettre aux piétons de facilement se déplacer à l'intérieur de l'îlot.

Les premiers résultats ne comprenant que des bâtiments étant prometteurs, l'étape suivante de modélisation a été d'ajouter un nouveau composant

de l'îlot : les espaces verts. Les travaux de De Smet [De 18] préconisent la présence d'au moins 20% d'espaces verts dans l'îlot. De plus, au moins la moitié de cette surface verte doit être composée d'au plus deux espaces verts. Imposer qu'un certain pourcentage de la surface de l'îlot soit recouvert d'espaces verts peut facilement être modélisé d'une part à l'aide d'une contrainte et d'autre part en permettant à l'opération de voisinage de colorer en vert les cellules du pavage.

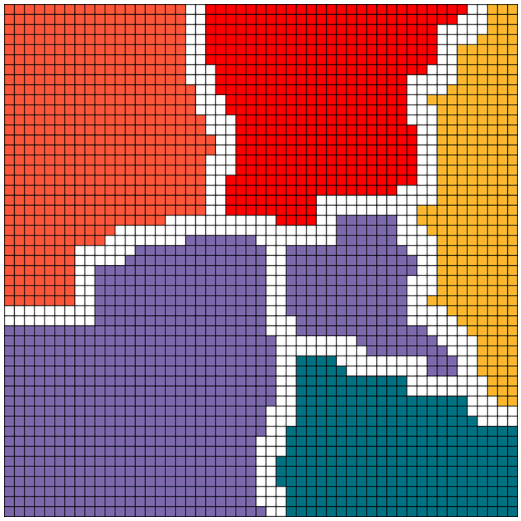


Figure 3.9 – Solution type retournée par LSCUBE.

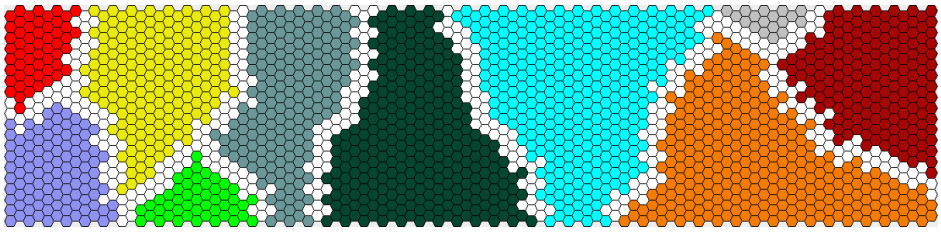


Figure 3.10 – Une solution obtenue sur un pavage hexagonal.

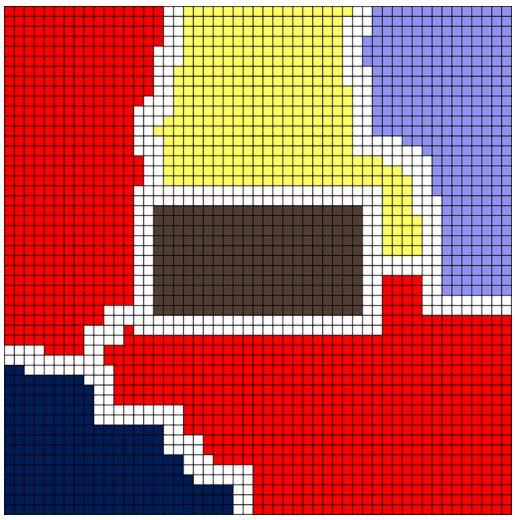


Figure 3.11 – Un résultat avec un bâtiment verrouillé par l'utilisateur (le bâtiment rectangulaire marron).

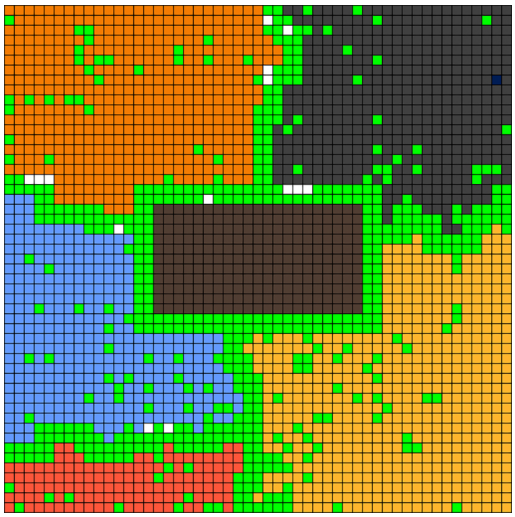


Figure 3.12 – Solution type retournée par LSCUBE avec des espace verts.

Cependant, l'ajout de cette contrainte met en évidence les limites du modèle actuel. Comme illustré à la Figure 3.12, pour atteindre son pourcentage d'espaces verts, LSCUBE se contente de remplir les chemins avec les espaces verts et ensuite de disperser des petits espaces verts dans tout l'îlot. Or la volonté derrière ces contraintes est la création de jardins ou de petits parcs au sein de l'îlot.

Ce comportement découle de la contradiction entre l'objectif principal de la recherche (minimiser la porosité) et recouvrir le terrain par des espaces verts. Une zone allouée à des espaces verts est une zone non-bâtie et donc augmente la porosité de l'îlot. C'est pour cela que le programme remplit dans un premier temps les «chemins», car ces cellules ne peuvent pas être bâties à cause des contraintes.

Le modèle actuel basé sur la recherche locale est un modèle mono-objectif. Comme déjà expliqué, pour les algorithmes de recherche locale, optimiser la fonction objectif et éviter les pénalités revient à un seul objectif principal. Ainsi, le programme est amené à hiérarchiser la fonction objectif et les contraintes. Plus la pénalité est grande, plus la contrainte est importante. Cela peut amener l'outil à totalement ignorer certaines contraintes.

Afin d'outrepasser ces problèmes, il est nécessaire que le deuxième module de CUBE se base sur un modèle multi-objectifs. Un tel modèle pourrait éviter la hiérarchisation des contraintes.

Dans le cadre de l'optimisation multi-objectifs, nous cherchons à trouver une solution qui optimise *une famille* de fonctions objectif. Cependant, dans l'optimisation multi-objectifs, il n'existe pas toujours de solution qui optimise *toutes* les fonctions objectifs *simultanément*. C'est le cas pour l'exemple de la Figure 3.13. Il n'y a aucune solution qui minimise les fonctions f_1 et f_2 en même temps. Ainsi, pour un problème d'optimisation multi-objectifs, l'attention est portée sur les solutions dites *Pareto optimales*. Dans la suite, nous supposons que nous voulons minimiser toutes les fonctions objectifs.

Définition 3.2.4 (Pareto dominance [EG09])**.** Un vecteur objectif $u = (u_1, \ldots, u_n)$ domine le vecteur $v = (v_1, \ldots, v_n)$ (noté $u \prec v$) si et seulement si aucune composante de v n'est plus petite que la composante correspondante dans u et qu'au moins une composante de u est strictement plus petite, *i.e.*,

$$\forall i \in \{1, \ldots, n\}, u_i \leqslant v_i \ \wedge \exists i \in \{1, \ldots, n\} \text{ tel que } u_i < v_i,$$

Définition 3.2.5 (Pareto optimalité [EG09])**.** Une solution $x^* \in S$ est Pareto optimal si pour tout $x \in S$, $f(x) \nprec f(x^*)$.

x	1	2	3	4
f_1	0	1	2	1
f_2	1	0	1	1

Figure 3.13 – Un ensemble de solutions pour un problème avec deux fonctions objectifs (f_1 et f_2). En gris, les solutions Pareto optimales (en admettant vouloir minimiser les deux fonctions).

Exemple 3.2.6. Dans le Figure 3.13, la solution 4 domine la solution 3. Les solutions 1 et 2 sont Pareto optimales.

Trouver un optimum de Pareto peut se faire à l'aide de métaheuristiques adaptées. Nous ne détaillons pas plus la notion d'optimum de Pareto ainsi que l'optimisation multi-critère dans le présent document. Le lecteur intéressé peut se référer à l'ouvrage d'El-Ghazali [EG09]. Au vue des contradictions et des conflits entre les critères, nous préférons étudier des notions issues de la *théorie des jeux*. En effet, la théorie des jeux est de par sa nature est particulièrement adaptée à la gestion de conflits. Les optima de Pareto restent, cependant, une notion intéressante à étudier dans un travail ultérieur.

CHAPITRE 4

CRITÈRES DE COMPACITÉ ET THÉORIE DES JEUX

Le principe fondamental de la *théorie des jeux* est de modéliser des interactions entre des entités ou des systèmes comme des jeux entre plusieurs joueurs. Les concepts de la théorie des jeux peuvent être appliqués à de nombreux domaines : informatique, économie, biologie, politique ou, dans notre cas, l'urbanisme. Chaque joueur peut avoir son propre objectif et chercher sa victoire individuelle. Ainsi, gérer des contradictions entre des objectifs potentiellement antagonistes est intrinsèque à la théorie des jeux.

La théorie des jeux peut ainsi aider à l'intégration des espaces verts dans la modélisation de l'îlot. Ce chapitre présente dans, un premier temps, les notions mathématiques théoriques utilisées dans la deuxième version de l'outil. Dans un second temps, nous décrivons comment nous modélisons le problème d'îlot compact comme un jeu multijoueur ainsi que certains choix d'implémentation.

4.1 Notions de théorie des jeux

4.1.1 Définition d'un jeu et Équilibre de Nash

Définition 4.1.1 (Jeu sous forme stratégique)**.** Un jeu sous forme stratégique est défini par $\mathcal{G} = (N, (S_i)_{i\in N}, (g_i)_{i\in N})$ où :

- $N = \{1, 2, \ldots, n\}$ est un ensemble de nombres naturels symbolisant l'ensemble des joueurs.
- Pour tout $i \in N$, S_i est l'ensemble des stratégies du joueur i.
  On note $\mathcal{S} = \times_{i=1}^{n} S_i$ et on appelle un élément de $\mathcal{S}$ un *profil de stratégies*.
- Pour tout $i \in N$, $g_i : \mathcal{S} \longrightarrow \mathbb{R}$ est la fonction de gain du joueur i.

Remarque 4.1.2. Dans le cas où un joueur i veut maximiser g_i, nous dirons que g_i est la fonction de gains du joueur i. Si le joueur veut minimiser g_i, cette fonction sera appelée fonction de coûts.

Remarque 4.1.3. Les stratégies des joueurs symbolisent les actions des différents joueurs dans le jeu et peuvent prendre différentes formes selon le contexte. Une stratégie peut donc être un nombre, le déplacement d'un pion aux échecs, le choix d'une position dans une grille de bataille navale, *etc.* Si pour tout $i \in N$, S_i est un ensemble *fini*, on dit que $\mathcal{G}$ est un jeu *fini*.

J_1 \ J_2	A	B	C
A	$(3, 3)$	$(-5, 9)$	$(-3, 8)$
B	$(9, -5)$	$(2, 2)$	$(2, 6)$
C	$(6, 3)$	$(7, 7)$	$(5, 5)$

Figure 4.1 – Un exemple de jeu sous forme stratégique.

Exemple 4.1.4. Considérons le jeu à deux joueurs de la Figure 4.1. Chaque joueur a trois stratégies A, B et C disposées verticalement pour le joueur 1

et horizontalement pour le joueur 2. Chaque couple du tableau représente les gains des joueurs en fonction des stratégies qu'ils ont choisies : la première composante pour le joueur 1 et la deuxième pour le joueur 2. Un tel jeu se joue à la façon d'un pierre-feuille-ciseaux, *i.e.* les joueurs choisissent une stratégie en même temps et reçoivent des gains en conséquence. Ainsi, si le joueur 1 choisit la stratégie B et le joueur 2 la stratégie C, ils ont respectivement un gain de 2 et de 6.

Remarque 4.1.5. Si, dans l'exemple précédent, les deux joueurs ont les mêmes ensembles de stratégies, ce n'est pas une obligation en général. Deux joueurs peuvent avoir des stratégies ou un nombre de stratégies différents.

Notations. Soit $s = (s_1, \dots s_n)$ un profil de stratégies. Nous notons s_{-i} le sous profil de stratégies de s ne contenant pas s_i. Nous notons également $\mathcal{S}_{-i} = \times_{k \in N \setminus \{i\}} S_k$.

Nous supposons que tous les joueurs sont *rationnels*, *égoïstes* et ont une connaissance parfaite des tenants et aboutissants du jeu. Par *rationnel*, nous entendons qu'un joueur choisit toujours une stratégie en adéquation avec son propre objectif. Un joueur *égoïste* est un joueur qui se concentre uniquement sur son objectif. Il joue sans se soucier d'avantager ou de désavantager les autres joueurs. Enfin, chaque joueur connaît toutes ses stratégies ainsi que celles des autres joueurs. Il a ainsi connaissance des gains qu'il peut engendrer.

Parmi les profils de stratégies, il y en a qui nous intéressent particulièrement : les *Équilibres de Nash*. Intuitivement, imaginons que les joueurs puissent se concerter avant de jouer et se mettre d'accord sur la stratégie qu'ils vont adopter. De par sa rationalité et son égoïsme, si un des joueurs se rend compte qu'il peut améliorer son gain en changeant de stratégie au dernier moment, il changera de stratégie quitte à léser les autres joueurs. Un *Équilibre de Nash* est un profil de stratégie pour lequel aucun des joueurs n'a intérêt à trahir l'accord préalable et changer de stratégie au dernier moment.

Définition 4.1.6 (Équilibre de Nash)**.**
Soient un jeu $\mathcal{G} = (N, (S_i)_{i \in N}), (g_i)_{i \in N})$ et $s^* = (s_1, \dots, s_n) \in \mathcal{S}$. Le profil

de stratégies s^* est un *Équilibre de Nash* (EN) si et seulement si

$$\forall i \in N, \forall t_i \in S_i, g_i(s_i, s_{-i}) > g_i(t_i, s_{-i})$$

en supposant que tous les joueurs veuillent maximiser leurs gains.

Exemple 4.1.7. Dans le jeu de la Figure 4.1, le profil (C, B) est un EN. En effet, pour ce profil, le joueur 1 à un gain de 7. S'il change sa stratégie pour B, il aura un gain de 2. S'il change sa stratégie pour A, il aura un gain de -5. Le même raisonnement peut être fait pour le joueur 2. S'il change de stratégie ses gains diminuent.

Dans un EN, il n'y a aucune garantie qu'un des joueurs optimise ses gains. Par exemple, dans le jeu de la Figure 4.1, les joueurs ont un gain de 7 dans l'EN (C, B). Cependant, le plus gros gain possible pour les deux joueurs est de 9. Un EN est une situation stable où les joueurs ne peuvent plus améliorer leurs gains. Pour tous les joueurs, un EN est un optimum si tous les autres joueurs fixent leur stratégie.

Trouver un EN peut permettre de définir la fin d'un jeu comme illustré ci-dessous. Les exemples suivants permettent de mieux comprendre et de mieux appréhender les nuances de la notion d'Équilibre de Nash.

Exemple 4.1.8 (Dilemme du prisonnier)**.** Considérons le jeu illustré à la Figure 4.2, jeu connu sous le nom du *dilemme du prisonnier*. Dans ce jeu, deux prisonniers sont interrogés séparément suite à un crime qu'ils ont commis. Les deux prisonniers ont deux stratégies possibles :

- parler (stratégie P) et dénoncer son complice ;
- se taire (stratégie T).

Si les deux prisonniers se taisent, ils obtiennent une peine minimale de 2 ans de prison faute de preuves. Si un seul des deux prisonniers dénonce l'autre, il est remis en liberté et l'autre obtient la peine maximale de 10 ans. S'ils se dénoncent entre eux, ils sont condamnés à une peine réduite de 5 ans de prison suite à leur coopération.
Intuitivement, il semble raisonnable de penser que les deux prisonniers vont

se taire par solidarité et afin d'obtenir une peine globalement minimale. Cependant, les deux prisonniers ont intérêt à parler. En effet, si un des deux prisonniers se tait, il n'a aucune garantie que son complice en fera autant. La menace d'être le seul à faire de la prison, pousse les deux joueurs à se dénoncer. Cet exemple met en lumière qu'un EN ne garantit pas l'optimalité des gains des joueurs. En effet, le seul EN de ce jeu est le profil (P, P) où les deux joueurs ont une peine de prison de 5 ans.

J_1 \ J_2	P	T
P	$(5, 5)$	$(0, 10)$
T	$(10, 0)$	$(2, 2)$

Figure 4.2 – Le dilemme du prisonnier.

Exemple 4.1.9 (Problème des marchands de glace)**.** Nous présentons ici un exemple de jeu un peu plus complexe. Comprendre cet exemple peut aider à comprendre certains résultats obtenus par CUBE. Tous les détails de cet exemple peuvent être trouvés dans l'ouvrage de Matsumoto et Szidarovszky [MS16].

Deux marchands de glaces MG_1 et MG_2 se font concurrence sur une plage et veulent trouver le meilleur emplacement pour leur échoppe. Pour simplifier le modèle, il est supposé que la plage est l'intervalle $[0, 1]$. Les deux marchands doivent donc choisir une position x_1 et x_2 dans $[0, 1]$. Les clients sont distribués uniformément le long de la plage. Chacun d'entre eux achète une crème glacée dans la boutique la plus proche. Un client se trouvant à la même distance des deux marchands choisit sa boutique aléatoirement. Le nombre de clients de chaque magasin est donc proportionnel à la longueur du sous-ensemble de $[0, 1]$ qui contient les points plus proches de lui que de son concurrent. Nous définissons donc un jeu où, les deux marchands de glaces sont les joueurs, les ensembles de stratégies $S_1 = S_2 = [0, 1]$, la fonction de gains de MG_1 est

définie par :

$$g_1(x_1, x_2) = \begin{cases} \frac{x_1+x_2}{2} & \text{if } x_1 < x_2, \\ \frac{1}{2} & \text{if } x_1 = x_2, \\ 1 - \frac{x_1+x_2}{2} & \text{if } x_1 > x_2, \end{cases}$$

et la fonction de gains de MG_2, $g_2(x_1, x_2) = g_1(x_2, x_1)$. La Figure 4.3 illustre une situation où les deux marchands se partagent équitablement la plage. En effet, nous avons $g_1(\frac{1}{4}, \frac{3}{4}) = g_2(\frac{1}{4}, \frac{3}{4}) = \frac{1}{2}$. Concentrons-nous sur MG_1. Si, comme à la Figure 4.3, $x_2 > \frac{1}{2}$, MG_1 peut augmenter ses gains en choisissant x_1 proche de x_2 par la gauche. Comme illustré à la Figure 4.4, en s'approchant de x_2, MG_1 augmente son nombre de clients potentiels. En effet, $g_1(\frac{2}{3}, \frac{3}{4}) = \frac{17}{24} \approx 0.71$. Symétriquement, si $x_2 < \frac{1}{2}$, MG_1 peut augmenter ses gains en s'approchant de x_2 par la droite. Dans les deux cas, MG_1 peut toujours augmenter ses gains en convergeant vers x_2. Il n'y a pas de maximum atteignable, et donc pas d'EN dans ces cas. Cependant, si $x_2 = \frac{1}{2}$, MG_1 peut maximiser ses gains en choisissant $x_1 = \frac{1}{2}$. Ces arguments sont aussi valables pour MG_2 par symétrie. Dès lors, si $x_1 = x_2 = \frac{1}{2}$, nous avons un EN.
Le point à retenir est que pour atteindre l'équilibre, les deux joueurs doivent être le plus proche possible du centre de la plage. Si S_1 et S_2 étaient finis, le même comportement serait observable : les joueurs doivent contrôler le centre pour atteindre un EN.

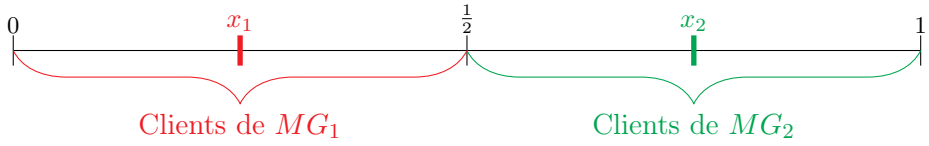


Figure 4.3 – Illustration du problème des marchands de glace où MG_1 place sa boutique en $\frac{1}{4}$ et MG_2 en $\frac{3}{4}$.

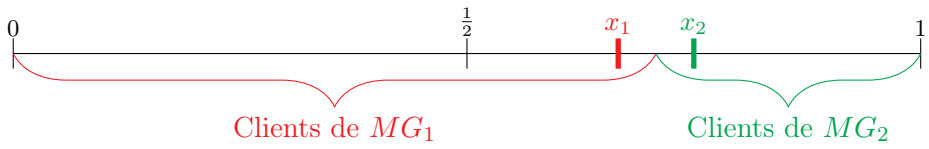


Figure 4.4 – Illustration du problème des marchands de glace où MG_1 place sa boutique en x_1 et MG_2 en x_2.

Étant donné un jeu, deux questions se posent. Existe-t-il un Équilibre de Nash dans ce jeu ? Si oui, comment le trouver ? Pour répondre à ces questions, nous utilisons des propriétés liées aux *jeux de potentiel*.

4.1.2 Jeux de potentiel

De façon brève, un jeu de potentiel peut être défini comme un jeu muni d'une fonction $\varphi : \mathcal{S} \longrightarrow \mathbb{R}$ qui associe à chaque profil de stratégie une valeur réelle. Selon le type de jeu de potentiel, cette fonction φ doit vérifier différentes propriétés. Cependant pour tout type de jeu, φ doit être définie de telle sorte que le changement de gain induit par un changement de stratégie d'un des joueurs transparaisse dans les valeurs de φ.

Un grand point d'intérêt des jeux de potentiel est qu'ils ont de nombreuses propriétés permettant de conclure à l'existence d'EN. Nous définissons ici les concepts nécessaires à l'obtention des théorèmes et propriétés qui nous intéressent. Nous ne prouvons pas les théorèmes qui suivent dans le présent document. Tous les détails omis ici sont trouvables dans l'ouvrage de Lã *et al.* [LCS16].

Définition 4.1.10 (Jeu de potentiel ordinal [LCS16])**.**
Le jeu $\mathcal{G} = (N, (S_i)_{i \in N}, (g_i)_{i \in N})$ est un jeu de potentiel ordinal si et seule-

ment si il existe une fonction $\varphi : \mathcal{S} \longrightarrow \mathbb{R}$ telle que, $\forall i \in N$:

$$g_i(t_i, s_{-i}) - g_i(s_i, s_{-i}) > 0 \Longleftrightarrow \varphi(t_i, s_{-i}) - \varphi(s_i, s_{-i}) > 0,$$

$\forall s_i, t_i \in S_i, \forall s_{-i} \in \mathcal{S}_{-i}$.

Un jeu de potentiel ordinal requiert uniquement que le changement de valeur de la fonction potentiel et le changement de gain induits par un changement de stratégie d'un des joueurs soient de même signe. En d'autres mots, si le joueur i augmente ses gains en changeant sa stratégie, la valeur de la fonction potentiel doit également augmenter et vice versa. S'il diminue ses gains, la valeur de la fonction potentiel doit également diminuer.

Remarque 4.1.11. La Définition 4.1.10 est la définition de base d'un jeu de potentiel ordinal. Cependant, la fonction de potentiel d'un jeu peut être très complexe à définir. Ainsi, dans la suite de cette section, nous énonçons une condition nécessaire et suffisante (Théorème 4.1.21) nous permettant de définir un jeu de potentiel ordinal sans définir de fonction de potentiel. Dans la suite de ce document, nous occultons donc la fonction de potentiel.

J_1 \ J_2	A	B
A	$(3,3)$	$(-5,9)$
B	$(9,-5)$	$(2,2)$

J_1 \ J_2	A	B
A	0	1
B	1	2

Figure 4.5 – À gauche, une matrice de gains d'un jeu. À droite une fonction potentielle pour ce jeu.

Exemple 4.1.12. La Figure 4.5 illustre un jeu de potentiel ordinal. Considérons le profil de stratégies (A, B). Si J_1 change de stratégie, il augmente ses gains. Ainsi, le potentiel de (B, B) est plus élevé que celui de (A, B). De façon analogue, si J_2 change de stratégie, il diminue ses gains. C'est pour cela que le potentiel de (A, A) est inférieur à celui de (A, B)

Avant de présenter les propriétés sur les jeux de potentiel ordinaux permettant de déduire l'existence d'EN dans de tels jeux, nous devons introduire quelques nouvelles notions.

Définition 4.1.13 (Chemin améliorant [LCS16])**.** Une suite de profils de stratégies $\rho = (\sigma_0, \sigma_1, \dots)$ telle que pour tout indice $k \geqslant 0$, σ_{k+1} est obtenu à partir de σ_k en permettant au joueur $i(k)$ (unique joueur changeant de stratégie à l'étape k) de changer sa stratégie est appelée un *chemin*. Le chemin ρ est un *chemin améliorant* si $\forall k \geqslant 0, g_{i(k)}(\sigma_{k+1}) > g_{i(k)}(\sigma_k)$. Si $\rho = (\sigma_0, \sigma_1, \dots, \sigma_K)$ est fini et que $\sigma_0 = \sigma_K$, alors ρ est dit être un *cycle*. Un chemin améliorant ne peut s'arrêter que s'il n'y a plus aucune amélioration de gain possible. Certains chemins peuvent ne pas avoir de fin, *i.e.* être infinis ou devenir des cycles.

Autrement dit, une suite de profils de stratégies est un chemin améliorant si, à chaque étape, le joueur qui change sa stratégie augmente son gain strictement (ou le diminue si tel est son objectif).

Définition 4.1.14 (Chemin non-détériorant [LCS16])**.** Un chemin $\rho = (\sigma_0, \sigma_1, \dots)$ est *non-détériorant* si pour tout $k \geqslant 0$, $g_{i(k)}(\sigma_k) \leqslant g_{i(k)}(\sigma_{k+1})$.

Remarque 4.1.15. Dans un chemin améliorant, un changement de stratégie induit une amélioration *stricte* des gains du joueur concerné. Dans un chemin non-détériorant, un joueur peut choisir une nouvelle stratégie qui ne change pas ses gains.

Définition 4.1.16 (Cycle améliorant faible [LCS16])**.**
Le cycle $\rho = (\sigma_0, \sigma_1, \dots, \sigma_K = \sigma 0)$ est appelé un *cycle améliorant faible* s'il est non-détériorant et qu'il existe $k \geqslant 0$ tel que $g_{i(k)}(\sigma_k) < g_{i(k)}(\sigma_{k+1})$.

Un cycle améliorant faible est un cycle non-détériorant tel qu'il existe au moins un joueur qui améliore strictement ses gains.

Exemple 4.1.17. Considérons le jeu à deux joueurs illustré à la Figure 4.6. Dans les différentes séquences qui suivent, le joueur 1 change de stratégie en

premier, puis le joueur 2 et ainsi de suite.
La séquence $\big((A,D),(B,D),(B,C),(C,C),(C,D)\big)$ est un chemin améliorant.
La séquence $\big((A,D),(B,D),(B,C)\big)$ *n'est pas* un chemin améliorant. En effet, le joueur 1 peut encore augmenter ses gains en changeant de stratégie. Or un chemin améliorant ne peut s'arrêter que s'il n'y a plus aucune amélioration de gain possible pour tous les joueurs.
La séquence $\big((B,A),(C,A),(C,B),(B,B),(B,A)\big)$ est un cycle non-détériorant.
La séquence $\big((B,A),(A,A),(A,B),(B,B),(B,A)\big)$ est un cycle améliorant faible. En effet, la transition du profil (A,A) au profil (A,B) augmente strictement les gains du joueur 2.

J_1 \ J_2	A	B	C	D
A	$(1,3)$	$(2,9)$	$(3,8)$	$(1,2)$
B	$(1,2)$	$(2,2)$	$(2,6)$	$(3,3)$
C	$(1,3)$	$(2,3)$	$(4,4)$	$(5,5)$

Figure 4.6 – Un exemple de jeu avec des chemins améliorants et non-détériorants.

Les théorèmes suivants énoncent des propriétés très intéressantes sur les chemins améliorants et les jeux de potentiel ordinaux.

Théorème 4.1.18 ([LCS16])**.** *Pour tout jeu sous forme stratégique, s'il existe un chemin améliorant fini, alors il termine en un Équilibre de Nash.*

Remarque 4.1.19. Le chemin améliorant de l'Exemple 4.1.17 se termine bien par un EN.

Théorème 4.1.20 ([MS96])**.** *Dans un jeu de potentiel ordinal fini, tous les chemins améliorants sont finis.*

Théorème 4.1.21 ([VN97]). *Soit un jeu* $\mathcal{G} = (N, (S_i)_{i \in N}, (g_i)_{i \in N})$. *Si* $\mathcal{S} = \prod_{i \in N} S_i$ *est fini ou dénombrable, alors* $\mathcal{G}$ *est un jeu de potentiel ordinal* si et seulement si $\mathcal{G}$ *ne possède* aucun *cycle améliorant faible.*

Grâce au Théorème 4.1.18, nous avons un moyen simple de trouver les EN via les chemins améliorants finis. Par le Théorème 4.1.20, tous les chemins améliorants sont finis dans un jeu de potentiel ordinal ; jeux que nous pouvons facilement définir grâce au Théorème 4.1.21. Le module théorie des jeux de CUBE se base sur ces trois théorèmes. L'idée est de modéliser le problème d'îlot compact comme un jeu fini sans cycle améliorant faible et de parcourir des chemins améliorants finis pour trouver un EN.

4.2 Modélisation d'îlots compacts à l'aide de la théorie des jeux

Nous expliquons, dans cette section, comment nous utilisons les notions théoriques définies précédemment dans le cadre du problème de l'îlot compact. Afin de faciliter la transition entre la théorie et la modélisation, nous devons introduire une nouvelle façon de représenter un jeu : la représentation sous forme de *graphe*.

Brièvement [7], un graphe est un ensemble de nœuds interconnectés par des arêtes. Les nœuds d'un graphe peuvent, par exemple, symboliser les états d'un système. Le système peut transiter d'un état à un autre si et seulement si ces deux états sont reliés par une arête dans le graphe.

Maintenant, un jeu est représenté comme un graphe dont les nœuds sont les différents profils de stratégies. Chaque joueur a son ensemble d'arêtes symbolisant les différents changements possibles de stratégies à partir d'un profil donné. Ainsi, par exemple, le jeu à deux joueurs J_1 et J_2 de la Figure 4.7 peut-être modélisé comme le graphe de la Figure 4.8. Les arcs continus représentent les changements de profil que J_1 peut effectuer. Les arcs en pointillés les changements que J_2 peut réaliser.

7. La notion de graphe est vue plus en détails à la section 5.1.

J_1 \ J_2	A	B	C
A	$(3,3)$	$(-5,9)$	$(-3,8)$
B	$(9,-5)$	$(2,2)$	$(2,6)$
C	$(6,3)$	$(7,7)$	$(5,5)$

Figure 4.7

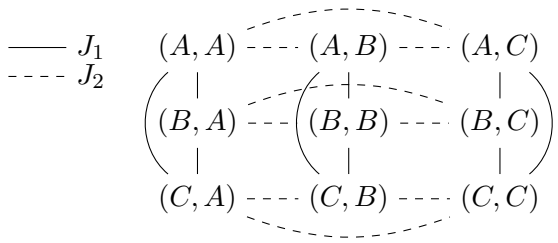


Figure 4.8 – Représentation sous forme de graphes du jeu de la Figure 4.7.

La première idée à la base du module théorie des jeux de CUBE est de remplacer les profils de stratégies du graphe par des configurations d'îlots urbains (voir Figure 4.9). La deuxième est d'attribuer à chaque joueur un critère de compacité. Ainsi, par exemple, dans la Figure 4.9, J_1 veut remplir l'îlot avec des bâtiments (cellules rouges) et J_2 avec des espaces verts (cellules vertes).

Afin de générer une configuration d'îlot urbain optimisant au mieux les différents critères, CUBE fonctionne comme suit. Au départ d'une solution donnée, les joueurs changent chacun à leur tour la solution courante pour une solution *voisine* améliorant leurs gains. Définir un voisinage d'une solution revient à définir les profils de stratégies connectés à un premier profil dans un graphe. Les joueurs changent à tour de rôle la solution courante jusqu'au moment où changer la solution n'améliore plus leurs gains. CUBE crée itérativement un chemin améliorant composé de configurations d'îlots. L'algorithme se termine quand une configuration équivalente à un EN est trouvée.

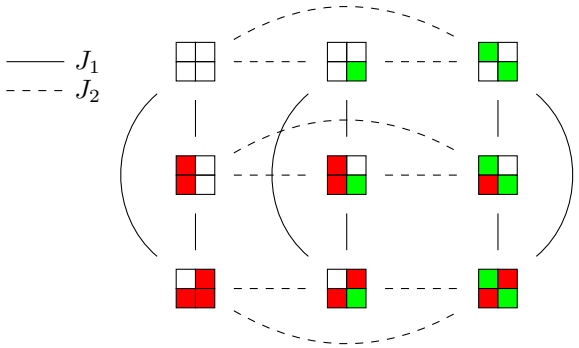


Figure 4.9 – Un jeu où les profils de stratégies sont des configurations d'îlots.

Les algorithmes de ce module de CUBE suivent la même philosophie que la recherche locale : partir d'une solution, évaluer un voisinage de cette solution et s'arrêter quand une solution meilleure que tous ses voisins est trouvée. Ici, au lieu de chercher à optimiser un unique objectif principal, les algorithmes de CUBE optimisent alternativement plusieurs objectifs. Le programme s'arrête quand tous les objectifs ne peuvent plus être améliorés.

Remarque 4.2.1. Avec cette approche, CUBE n'est plus amené à hiérarchiser lui-même les différents objectifs. Cependant l'utilisateur peut toujours donner plus ou moins d'importance à un critère. En permettant, par exemple, à un joueur de changer sa stratégie plusieurs tours de suite, il augmente ses chances d'optimiser ses gains.

Remarque 4.2.2. Pour trouver un EN, chaque joueur cherche à améliorer son objecitf *indépendament des autres joueurs*. Il est donc possible de dégrader des objectifs en faveur d'un seul. Trouver un optimum de Pareto se fait en exploitant la Paretor dominance des solutions entre elles. Trouver un optimum de Pareto ne peut donc se faire au détrimant des autres objectifs.

4.3 Implémentation

Afin d'implémenter la recherche d'EN, CUBE utilise plusieurs classes abstraites. Chacune de ces classes définit la structure et les méthodes nécessaires pour chaque composant d'un jeu (joueur, stratégie, *etc.*). Dans cette section, nous décrivons ces classes ainsi que les interactions qu'elles ont entre elles.

4.3.1 Jeux et invariants

La première classe définie est la classe `Game`. Cette classe modélise un jeu de façon générale. En considérant la représentation sous forme de graphe (voir section 4.2), une instance de `Game` est un profil de stratégies. Dans la suite de ce document, nous disons qu'un joueur applique une stratégie sur une instance de `Game` pour signifier que le joueur change sa stratégie dans un profil de stratégies. Les instances de cette classe et de ses sous-classes sont utilisées par les autres composants de CUBE.

Comme pour le module de recherche locale, un système d'invariant est implémenté. Un invariant observe une instance de `Game` et maintient diverses propriétés sur cette instance. Quand le jeu observé est modifié, l'invariant est notifié et met à jour ses propriétés.

4.3.2 Stratégies

```
class Strategy {
public:
  virtual void select(std::shared_ptr<Game> g) = 0;

  virtual void unselect(std::shared_ptr<Game> g) = 0;
}
```

Les stratégies sont implémentées via la classe `Strategy`. Cette classe définit deux méthodes : `select` et `unselect`.

La méthode `select` prend en entrée une instance de `Game`[8]. Cette méthode applique une transformation sur l'instance reçue en entrée. Chaque sous-classe de `Strategy` définit sa propre transformation. La méthode `unselect` est l'opération inverse de la méthode `select`. Elle permet de redonner à l'instance de `Game` l'état qu'elle avait avant l'application de la méthode `select`.

Ces deux méthodes sont utilisées lors de la génération d'un voisinage d'une solution. À partir d'une solution donnée, une première stratégie est appliquée via la méthode `select`. Une nouvelle solution est obtenue. Cette solution est ensuite évaluée par le joueur concerné. Une fois l'évaluation faite, on retourne à la solution de départ via la méthode `unselect` (voir Algorithme 6). Ainsi une autre stratégie peut être appliquée, évaluée et comparée à la première.

4.3.3 Joueurs

```
class Player {
public:
    Player(std::function<bool(double, double)> better);

    Player();

    virtual double payoff(std::shared_ptr<Game> game) =
       0;

    virtual std::vector<std::shared_ptr<Strategy>>
       getAllStrat(std::shared_ptr<Game> game) = 0;

    virtual std::shared_ptr<Strategy> getRandomStrat(std
       ::shared_ptr<Game> game) = 0;

    virtual bool isMaximizing() = 0;
}
```

8. Quand nécessaire, les instances de classe demandées en entrée sont passées via des pointeurs (*e.g.* des *shared pointer*). Cela permet d'éviter de copier à outrance des objets gourmands en mémoire.

La méthode `payoff` de la classe `Player` définit la fonction de gains ou de coûts du joueur. Elle prend en entrée une instance de `Game`. Elle peut utiliser les invariants observant cette instance pour réaliser ses calculs.

La méthode `isMaximizing` permet de déterminer si le joueur souhaite maximiser ou minimiser ses gains.

Une instance de cette classe peut optionnellement prendre en entrée une fonction `better`. Cette fonction est utilisée pour comparer les gains de deux solutions. Par défaut, l'inégalité stricte est utilisée. Grâce à la fonction `better`, l'inégalité large, par exemple, peut être utilisée à la place de la stricte.

La méthode `getAllStrat` prend en entrée une instance de `Game`. Soient p et g respectivement une instance de `Player` et une instance de `Game`. L'appel p.`getAllStrat`(g) retourne l'ensemble des stratégies que le joueur p peut appliquer sur g. Ces stratégies sont utilisées pour générer le voisinage de g pour le joueur p. La méthode `getRandomStrat` retourne une stratégie *aléatoire* que le joueur p peut appliquer sur g.

Un joueur peut également avoir accès à une mémoire finie. La mémoire finie est implémentée comme une file avec une taille maximale possible. Quand la mémoire est remplie, le premier élément de la file est supprimé. Cela laisse ainsi de la place pour un nouvel élément. La mémoire finie possède également une méthode permettant de compter le nombre de fois qu'un élément est présent dans la file.

Le joueur stocke dans sa mémoire les dernières stratégies qu'il a appliquées sur le jeu. Si le nombre d'occurrences d'une stratégie en mémoire est trop important, le joueur ne peut plus l'utiliser. Cela oblige le joueur soit à utiliser une autre stratégie soit à abandonner. Ainsi, en cas de conflit avec un autre joueur, les boucles infinies sont évitées.

4.3.4 Gameplay itératif

```
public:
    IterativeGameplay(std::vector<std::shared_ptr<Player
       >> players, std::shared_ptr<Game> game);

    virtual void play();

    virtual std::shared_ptr<Strategy> selectStrat(std::
       shared_ptr<Player> p) = 0;

    virtual std::vector<std::shared_ptr<Player>>
       orderPlayers() = 0;
```

Cette classe fournit la structure des algorithmes de recherche d'EN. Une instance de cette classe prend en entrée une liste de joueurs et une instance de `Game` servant de point de départ à la recherche.

Cette classe définit deux méthodes abstraites. La première de ces deux méthodes, `orderPlayers`, permet de définir l'ordre dans lequel les joueurs appliquent leur stratégie. Par défaut, l'ordre de la liste donnée en entrée est utilisé. Cependant, il est par exemple possible que l'ordre des joueurs soit choisi aléatoirement à chaque itération de la recherche.

La deuxième méthode abstraite, `selectStrat`, est la méthode qui définit la façon dont la stratégie d'un joueur est sélectionnée. Cette méthode, une fois implémentée par les sous-classes, permet de distinguer les différents algorithmes de recherche. Cette méthode peut permettre de définir des algorithmes déterministes ou stochastiques inspirés des algorithmes de recherche locale. Si le joueur, sur lequel la méthode `selectStrat` est appelée, ne trouve pas de stratégie améliorant ses gains, la méthode retourne un *Null Pointer*, noté `nullptr`[9].

9. Intuitivement, un `nullptr` peut être vu comme une variable qui ne contient rien. Utiliser un `nullptr` nous permet ici de signifier à l'algorithme qu'il n'existe pas de stratégie améliorante pour un joueur.

Algorithme 4 : Pseudo-code de la méthode `play`.

Résultat : La variable de classe `game` est modifiée jusqu'au moment où elle atteint un EN.

```
1  Soit une variable booléenne improvement
2  Répéter
3  |  ordered ← orderPlayers()
4  |  improvement ← false
5  |  pour chaque joueur p dans ordered faire
6  |  |  start ← selectStrat(p)
7  |  |  si strat ≠ nullptr alors
8  |  |  |  strat.select(game)
9  |  |  |  improvement ← true
10 tant que improvement
```

La méthode `play` (Algorithme 4) est la méthode qui recherche un EN pour les joueurs et le jeu donnés en entrée au constructeur d'`IterativeGameplay`. Cette méthode utilise une variable booléenne *improvement* afin de déterminer si au moins un des joueurs améliore son gain. Premièrement, l'algorithme détermine l'ordre des joueurs pour ce tour (*i.e.* cette itération) via la méthode `orderPlayers`. Suivant l'ordre établi des joueurs, la méthode `selectStrat` est appelée sur chaque joueur p. Si une stratégie améliorant le gain de p est trouvée, celle-ci est appliquée au jeu. Si aucun des joueurs ne trouve une stratégie améliorant son gain, l'algorithme s'arrête. Si le jeu atteint un état où aucun des joueurs ne peut améliorer son gain, cela signifie qu'un EN est trouvé.

Une première implémentation de la méthode `selectStrat` est présente dans la sous-classe `IterativeBest`. Cette classe cherche les EN dans un jeu en suivant la même philosophie que le *Hill Climbing*. Chaque joueur génère toutes les stratégies qu'il peut appliquer sur l'instance de `Game` donnée à la classe via sa méthode `getAllStrat`. Le joueur sélectionne la stratégie lui garantissant le plus de gains (en admettant que le joueur veuille maximiser ses gains). Si appliquer cette stratégie à l'état courant du jeu augmente les

gains du joueur, la méthode retourne ladite stratégie. Sinon, un `nullptr` est retourné. En combinant les Algorithmes 4 et 5, nous obtenons un *Hill Climbing* multijoueurs.

Algorithme 5 : Le pseudo-code de la méthode `selectStrat` pour le gameplay `IterativeBest`.

Entrée : Un joueur p.
Sortie : La stratégie améliorant le plus le gain de p si elle existe.

```
1  strats ← p.getAllStrat(game)
2  si strats est vide alors
3  |  retourner nullptr ;
4  sinon
5  |  s* ← arg max_{s∈strats} evaluate(p, game, s)
6  |  σ ← max_{s∈strats} evaluate(p, game, s)
7  |  si σ > p.payoff(game) alors
8  |  |  retourner s*
9  |  sinon
10 |  |  retourner nullptr
```

Algorithme 6 : Le pseudo-code de la méthode `evaluate` utilisée par l'Algorithme 5

Entrées : Un joueur p, un jeu g et une stratégie s.
Sortie : Les gains de p s'il applique la stratégie s sur g.

```
1 s.select(g)
2 x ← p.payoff(g)
3 s.unselect(g)
4 retourner x
```

Remarque 4.3.1. En tant que sous-classe de `IterativeGameplay`, le constructeur de `IterativeBest` prend également en entrée une instance de `Game` qu'il stocke dans la variable de classe `game`. L'Algorithme 5 fait appel à cette variable aux lignes 1, 5 et 6

Remarque 4.3.2. Un joueur peut suivre diverses contraintes, vues comme des règles du jeu. Ces contraintes restreignent les stratégies que le joueur peut appliquer sur le jeu. Ainsi, pour certains états du jeu, la méthode `getAllStrat` peut retourner une liste vide. Si c'est le cas, la méthode `getStrat` retourne simplement un `nullptr`.

L'Algorithme 4 présente le pseudo-code d'une version *déterministe* de la méthode `play`. La structure de la version *stochastique* est très similaire à celle de la version déterministe. La principale différence réside en la condition d'arrêt. La version stochastique de la méthode n'utilise plus une variable booléenne mais deux minuteurs. Le premier minuteur définit un temps maximal d'exécution de l'algorithme. Il est utilisé comme garde-fou contre les boucles infinies.

Le second minuteur mesure le temps écoulé sans qu'aucune stratégie améliorante ne soit trouvée. Pour la version stochastique, si la méthode `selectStrat` retourne un `nullptr`, cela ne signifie pas qu'il n'existe aucune stratégie améliorante. Cela signifie que la stratégie générée aléatoirement n'est pas améliorante. Permettre au joueur de générer une autre stratégie peut lui permettre d'améliorer son gain. Ce minuteur permet de distinguer le cas où le joueur ne génère pas une stratégie améliorante et le cas où il n'existe plus de stratégie améliorante.

4.4 Modélisation du problème de l'îlot compact

4.4.1 Définition du jeu

La modélisation sous forme de jeu est très similaire à la modélisation utilisée avec la recherche locale. Ici, une configuration d'îlot urbain est implémentée via l'objet `UrbanBlockGame`. Un tel objet est une sous-classe de `Game` contenant un pavage. Un pavage est implémenté à l'aide des outils de la librairie *OpenMesh* [BBS+02].

Les différents critères de compacité sont implémentés comme les fonctions de gain de différents joueurs. Les joueurs implémentés sont les suivants :

- un joueur voulant minimiser la porosité de l'îlot, noté J_P ;

- un joueur voulant avoir un nombre de bâtiments compris dans un intervalle donné, noté J_B ;
- un joueur voulant faire en sorte que le périmètre de chaque bâtiment soit compris dans un intervalle donné, noté J_π.

Les fonctions de coûts de ces différents joueurs sont calculées de la même façon que les pénalités des contraintes définies à la section 3.2.4. Ainsi, notons $b(s)$ le nombre de bâtiments présents dans une configuration d'îlot s. Soient $x, y \in \mathbb{N}$, les bornes de l'intervalle spécifié par l'utilisateur où $x \leqslant y$. Nous définissons g_B, la fonction de coûts de J_B comme suit :

$$g_B(s) = \begin{cases} x - b(s) & \text{si } b(s) < x, \\ 0 & \text{si } b(s) \in [x, y], \\ b(s) - y & \text{si } b(s) > y. \end{cases}$$

En plus de ces joueurs, deux joueurs ayant des objectifs liés aux espaces verts sont introduits dans le module théorie des jeux de CUBE. Le premier veut que le nombre d'espaces verts présents dans l'îlot soit compris dans un intervalle donné. Ce joueur est noté J_{GS}. La fonction de gains de J_{GS} est définie de façon analogue à la fonction de gains de J_B.

Le second joueur portant sur les espaces verts, noté J_α, désire que les espaces verts recouvrent au minimum un pourcentage α de la surface totale de l'îlot. Soient s une configuration d'îlot urbain et $\alpha \in [0, 1]$. Nous notons $A_{GS}(s)$ la surface occupée par des espaces verts dans s et $A(s)$ l'aire totale de l'îlot. Dès lors, la fonction de coûts de J_α, notée g_α, est définie comme suit :

$$g_\alpha(s) = \begin{cases} \alpha - \frac{A_{GS}(s)}{A(s)} & \text{si } \frac{A_{GS}(s)}{A(s)} < \alpha, \\ 0 & \text{sinon.} \end{cases}$$

Remarque 4.4.1. La plupart de ces joueurs sont des adaptations des contraintes pénalisantes définies dans le chapitre 3. Concernant la contrainte obligatoire portant sur la distance entre les bâtiments, celle-ci n'est pas adaptée en joueur. Cependant, elle reste une contrainte pour les différents joueurs. Les stratégies pouvant amener à une violation de cette contrainte ne sont pas retournées lors de la génération des stratégies d'un joueur.

Les stratégies de J_P, J_B et J_{GS} sont basées sur la même opération de base : changer la couleur d'une cellule blanche du pavage. Une telle stratégie est implémentée via l'objet `ChangeColorStrat`. Les joueurs J_P et J_B peuvent colorer une cellule en rouge, créant ainsi du bâti. Quant au joueur J_{GS}, il peut colorer une cellule en vert pour créer de l'espace vert. Notons que J_B peut également supprimer un bâtiment pour optimiser ses coûts.

Les stratégies de J_π et J_α se basent sur une opération plus complexe. Ils peuvent colorer toutes les cellules en périphérie d'un bâtiment ou d'un espace vert existant. Une telle stratégie est implémentée via l'objet `InflateStrat`. Le joueur J_π peut également supprimer toutes les cellules extérieures d'un bâtiment afin de réduire son périmètre grâce à une `DeflateStrat`. Permettre à ces deux joueurs de changer la couleur de plusieurs cellules à la fois, permet de faire gagner du temps d'exécution.

Remarque 4.4.2. Pouvoir colorer les cellules d'un pavage de plusieurs façons différentes en fonction des objectifs est grandement facilité par l'approche *théorie des jeux*. Vouloir le même genre de remplissage avec la recherche locale demanderait une opération de voisinage complexe.

Exemple 4.4.3. La Figure 4.10 illustre l'application d'une `InflateStrat` sur un bâtiment. La Figure 4.11 illustre également l'application d'une telle stratégie. Cependant, ici, la contrainte demandant au moins deux cellules vides entre deux bâtiments est prise en compte.

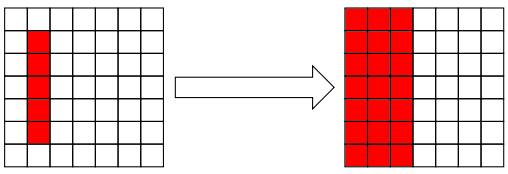


Figure 4.10 – Application d'une `InflateStrat`.

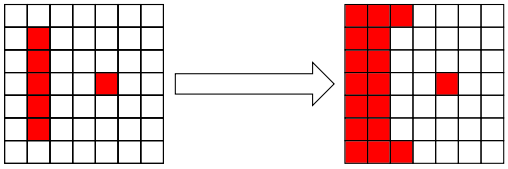


Figure 4.11 – Application d'une `InflateStrat` sur le bâtiment de gauche tout en laissant deux cellules blanches entre les deux bâtiments.

4.4.2 Résultats

Les Figures 4.12 et 4.13 illustrent des résultats obtenus via une recherche déterministe à partir d'un îlot vide. La solution de la Figure 4.12 est construite en optimisant les objectifs des joueurs J_P et J_B. Ces deux joueurs font attention à maintenir au minium deux cases vides entre chaque bâtiment. La Figure 4.13 est obtenue en cherchant à optimiser les objectifs de tous les joueurs définis précédemment. Pour cette configuration les joueurs doivent maintenir une distance de quatre cases vides entre les bâtiments et d'une case vide entre les espaces verts.

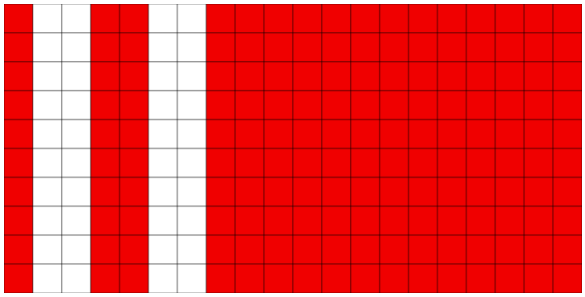


Figure 4.12 – Solution retournée si J_P et J_B jouent de façon déterministe.

Ces deux résultats, obtenus de façon totalement déterministe, illustrent les configurations types que CUBE peut retourner. L'ordre des joueurs fait que les bâtiments sont initialisés à partir d'une seule case. Chacune de ces cases est placée à intervalles réguliers afin de respecter la contrainte sur la distance entre

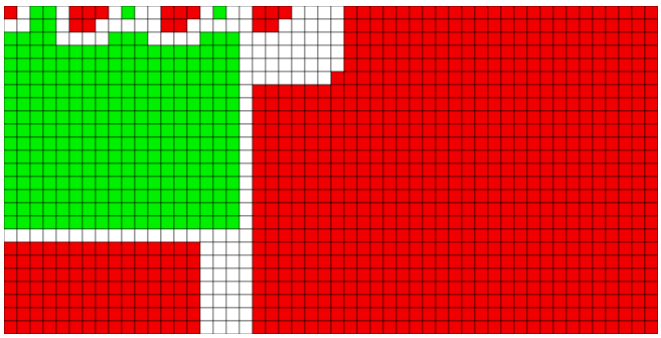


Figure 4.13 – Solution retournée si tous les joueurs jouent de façon déterministe.

les façades. Cela contraint ainsi les bâtiments à avoir une forme très longiligne comme à la Figure 4.12 ou très petite comme à la Figure 4.13.

Afin de varier les morphologies retournées par CUBE, la recherche peut démarrer d'une configuration *aléatoire* (voir Figure 4.14). Cependant, l'utilisation de l'aléatoire dans le processus de recherche d'EN met en évidence un conflit entre J_P et J_B. Ce conflit apparaît également quand les joueurs sélectionnent leurs stratégies aléatoirement.

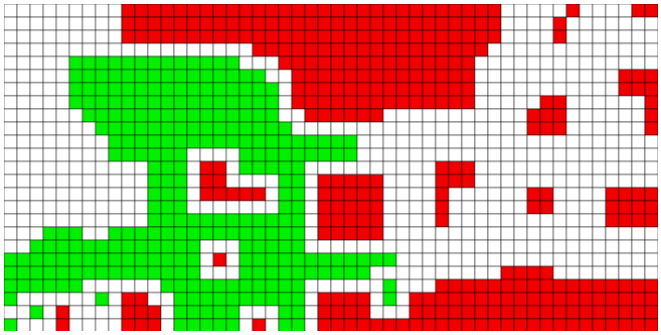


Figure 4.14 – Solution retournée à partir d'une solution aléatoire avec tous les joueurs.

Lors d'une recherche totalement déterministe, la sélection des stratégies est faite de telle sorte que J_P ne fait qu'agrandir des bâtiments déjà existants. Un potentiel conflit avec J_π peut également se produire. Cependant, il peut être évité si l'intervalle cible de J_π est suffisamment grand.

Avec des éléments stochastiques dans la construction de l'îlot, J_P est amené à créer de nouveaux bâtiments. À partir d'un certain nombre de bâtiments, J_B commence à en supprimer. L'algorithme entre ainsi dans une boucle où J_P crée un nouveau bâtiment et J_B en détruit un.

Augmenter le nombre de bâtiments tolérés par J_B n'est pas une façon raisonnable d'éviter ce conflit. En effet, le nombre de bâtiments doit rester cohérent avec les dimensions de l'îlot.

Cependant, minimiser la porosité est un objectif sous-jacent de J_π. En effet, plus le périmètre des bâtiments est élevé, plus la surface de l'îlot occupée par le bâti est élevée. Ainsi, il n'est peut-être pas nécessaire d'avoir un joueur dédié à la porosité de l'îlot.

Afin de vérifier cette assertion, nous avons réalisé une centaine d'expérimentations. Chacune de ces expériences consiste en trois exécutions de CUBE. La première est une exécution avec J_P avec une durée de maximum dix secondes. La deuxième est une exécution sans J_P avec une durée de maximum dix secondes. La dernière est une exécution sans J_P et sans limite de temps.

Les exécutions démarrent sur un pavage carré vide de 100 cellules de côté. Le joueur J_B vise un nombre de bâtiments entre 7 et 10. Le joueur J_π vise un périmètre entre 100 et 500. Le joueur J_{GS} vise un nombre d'espaces verts entre 3 et 5. Le joueur J_α veut que minimum 20% de la surface de l'îlot soit dédiée aux espaces verts. De plus, les joueurs doivent maintenir quatre cases blanches entre les bâtiments et une case vide entre les espaces verts. L'ordre des joueurs est le suivant : J_P, J_B, J_π, J_{GS} et J_α.

Remarque 4.4.4. Un ordinateur ne peut pas réellement générer aléatoirement un nombre. Pour simuler au mieux la génération de nombres aléatoires, les or-

dinateurs utilisent des algorithmes appelés *Pseudo Random Number Generator* (PRNG). Les PRNG génèrent des séquences de nombres dont les propriétés approximent les propriétés d'une séquence générée aléatoirement. Les séquences générées par un PRNG sont complètement déterminées par une valeur initiale appelée la *seed*. Connaître la *seed* d'une séquence permet de la générer à l'identique.
Pour les trois exécutions de CUBE composant une expérience, les joueurs utilisent la même seed. Ainsi, à une itération donnée, les joueurs vont à chaque fois choisir la même stratégie. Cela rend plus pertinente la comparaison de la valeur de la porosité entre les trois exécutions.

La Figure 4.15 reprend les différentes valeurs de porosité obtenues suite aux différentes expériences. Pour chacune des expériences, la porosité de l'îlot est plus faible sans joueur dont c'est l'objectif. Ainsi, les exemples qui suivent ont tous été obtenus sans joueur dédié à la porosité.

Remarque 4.4.5. En contraignant la distance minimale qu'il doit y avoir entre les bâtiments et les espaces verts, les joueurs J_B, J_π, J_{GS} et J_α n'ont pas la possibilité de se nuire directement. Par exemple, une cellule verte ne pas occuper la place d'une cellule bâtie et inversement. Un cycle améliorant faible ne peut donc pas apparaître. Grâce aux théorèmes 4.1.18, 4.1.20 et 4.1.21, CUBE trouvera toujours un EN sous ces conditions.

Les solutions illustrées par les Figures 4.16 à 4.21 ont été retournées par des recherches d'EN stochastiques de CUBE. Les méthodes où les joueurs choisissent aléatoirement une stratégie sont préférées aux méthodes déterministes car elles retournent des morphologies plus variées et demandent moins de temps de calculs.

Ces solutions ont été construites sur différents pavages. Les objectifs des différents joueurs sont les suivants :

- J_B vise un nombre de bâtiments entre 5 et 10 ;
- J_π veut que le périmètre de chaque bâtiment soit compris entre 75 et 500 ;
- J_{GS} vise un nombre d'espaces verts entre 3 et 5 ;

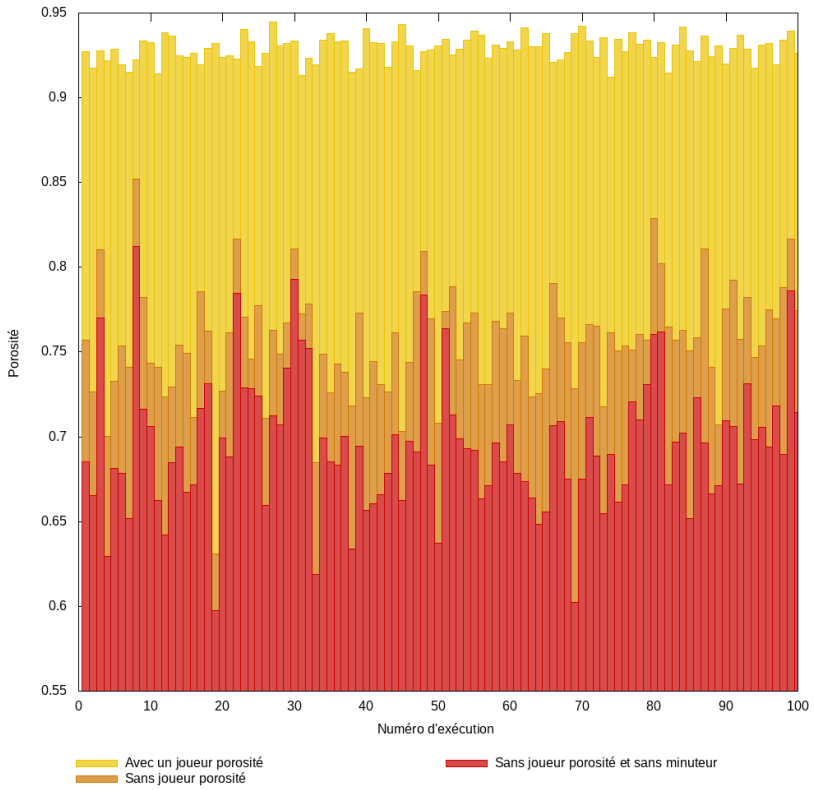


Figure 4.15 – Valeurs de la porosité obtenues suite à plusieurs exécutions de CUBE

- J_α veut qu'au moins 20% de la surface de l'îlot soit occupée par des espaces verts.

De plus, les joueurs doivent maintenir quatre cases blanches entre les bâtiments et une case vide entre les espaces verts. La recherche s'arrête quand deux secondes se sont écoulées sans qu'aucun joueur n'ait augmenté ses gains.

La solution de la Figure 4.16 est obtenue en démarrant la recherche à partir d'une solution aléatoire. Les configurations d'îlots obtenues à partir d'une solution aléatoire présentent des géométries très étalées et peu compactes. Les

configurations construites à partir de l'îlot vide contiennent des morphologies plus réalistes et sont donc préférées.

Remarque 4.4.6. La solution de la Figure 4.18 correspond à un EN où tous les joueurs ont optimisé leurs objectifs.

Remarque 4.4.7. Les formes brunes présentes dans les Figures 4.19 et 4.20 sont des bâtiments spécifiés et verrouillés par l'utilisateur. Comme pour LS-CUBE, un ensemble de cellules verrouillées ne peut être modifié lors de la recherche.

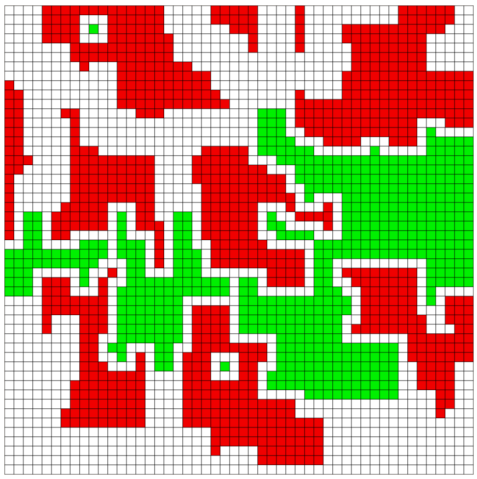


Figure 4.16 – Solution retournée par CUBE à partir d'une solution aléatoire.

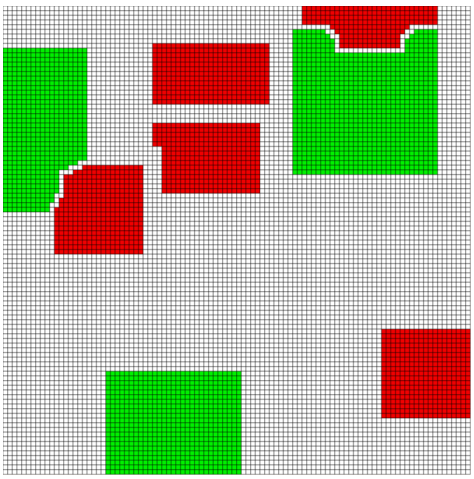


Figure 4.17 – Solution obtenue par CUBE à partir d'un pavage carré vide.

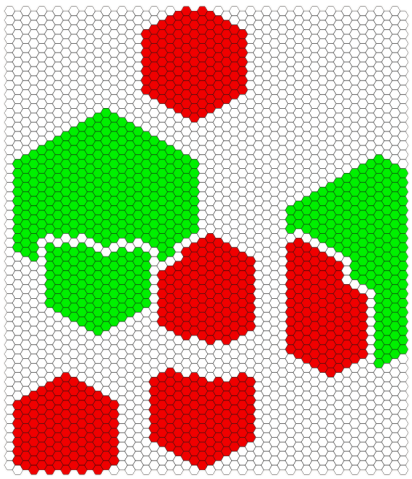


Figure 4.18 – Solution obtenue par CUBE à partir d'un pavage hexagonal vide.

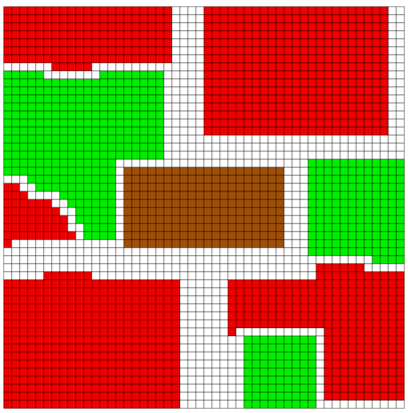


Figure 4.19 – Solution obtenue par CUBE sur un pavage carré. Le bâtiment marron est un bâtiment spécifié et verrouillé par l'utilisateur.

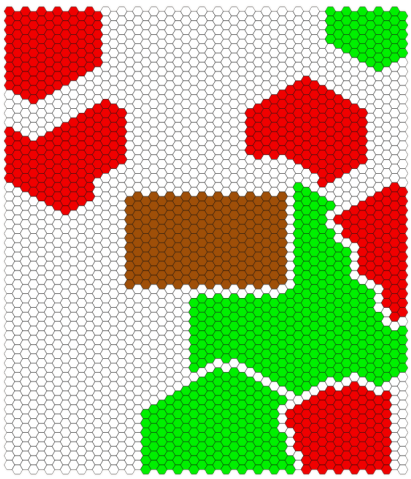


Figure 4.20 – Solution obtenue par CUBE sur un pavage hexagonal. Le bâtiment marron est un bâtiment spécifié et verrouillé par l'utilisateur.

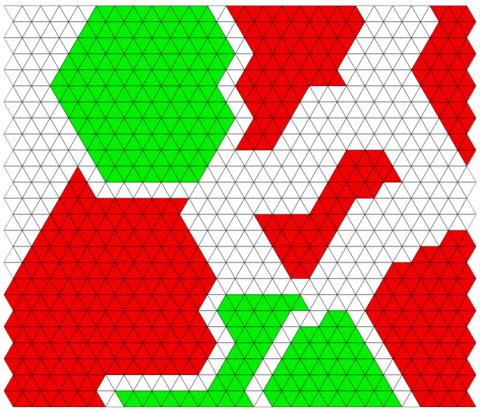


Figure 4.21 – Solution obtenue par CUBE à partir d'un pavage triangulaire vide.

4.5 Ajout d'une contrainte portant sur la luminosité

Le module théorie des jeux de CUBE permet de prendre en compte facilement la troisième dimension de l'îlot (traduisant l'altimétrie), via un joueur dédié. Créer un joueur dont l'objectif est d'augmenter la hauteur des bâtiments est aisé. Il suffit d'ajouter une variable de classe symbolisant la hauteur d'un bâtiment à la classe `GroupOfCells`. Les stratégies du joueur dédié à la hauteur des bâtiments, `HeightPlayer`, consistent à simplement augmenter la hauteur d'un bâtiment existant.

Considérer la hauteur des bâtiments lors de la conception d'un îlot compact ne se fait pas sans prendre en compte certaines contraintes. Les travaux de De Smet [De 18] définissent des contraintes en rapport avec la luminosité ambiante. À partir d'un point central sur une façade, situé à 1 m au-dessus du sol extérieur, un angle de 45° est projeté. Si la projection rencontre un obstacle, la façade est trop ombragée.

4.5.1 Présentation de la contrainte

La contrainte sur la luminosité a pour effet de limiter la hauteur maximale de deux bâtiments qui se font face. Ainsi, augmenter la hauteur d'un bâtiment ne peut se faire que si cela n'atténue pas de manière excessive la lumière reçue par les bâtiments voisins. La contrainte a également un effet sur les joueurs précédemment définis. Avant d'agrandir ou de créer un bâtiment, les joueurs doivent vérifier si les nouvelles façades ne sont pas à l'ombre d'un autre bâtiment. Inversement, les joueurs doivent également faire attention à ne pas gêner un autre bâtiment.

Remarque 4.5.1. Afin d'éviter le cas où deux bâtiments qui ne se font pas face obtiennent une hauteur techniquement irréaliste, `HeightPlayer` joue en considérant une borne sur les hauteurs. Une fois cette borne atteinte, augmenter la hauteur des bâtiments n'augmente plus les gains du joueur.

La contrainte sur la luminosité prend en entrée deux paramètres : un angle de projection θ et une hauteur x à partir de laquelle θ est projeté. La Figure 4.22 illustre comment ces deux arguments sont utilisés afin de vérifier si un bâtiment B_2 fait de l'ombre à un bâtiment B_1. Si la projection de θ ne croise pas B_2, la contrainte est satisfaite pour B_1.

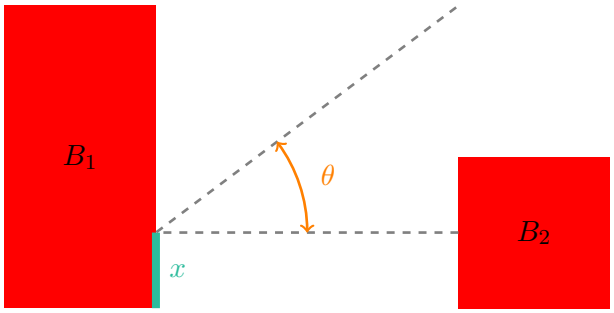


Figure 4.22 – Vérification de la contrainte portant sur la luminosité.

Outre la représentation visuelle de la contrainte, il est nécessaire de pouvoir la vérifier mathématiquement. Soient B_1 et B_2, deux bâtiments qui se font face. Supposons que ces deux bâtiments soient distants de d. Comme illustré

à la Figure 4.23, si la hauteur de B_2 n'excède pas $x + d\tan(\theta)$, B_2 n'obstrue pas la lumière reçue par B_1.

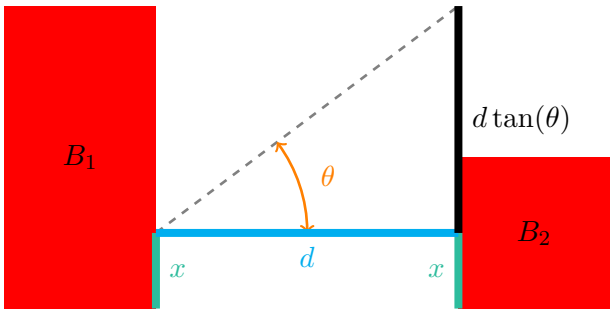


Figure 4.23 – Détails du calcul de la hauteur maximale de B_2.

Ainsi, il est possible de vérifier la contrainte tout en conservant la représentation en deux dimensions de l'îlot. Il suffit de déterminer la distance entre deux bâtiments face à face.

4.5.2 Implémentation des façades d'un bâtiment

Calculer la distance entre les bâtiments ne peut plus se faire en comptant simplement les cases vides entre eux. Utiliser cette technique nécessiterait de compter les cellules vides autour de chaque cellule en périphérie d'un bâtiment à chaque augmentation de sa hauteur.

Sur un pavage carré, les façades des bâtiments sont rectilignes et définies par des cellules adjacentes. Maintenir les coordonnées des extrémités des façades permet de réduire le nombre de vérifications à effectuer pour la contrainte. De plus, avec les coordonnées connues, le calcul des distances est plus précis.

Remarque 4.5.2. Pour des raisons de simplicité d'implémentation et de calcul, le maintien des coordonnées des façades n'est possible qu'avec un pavage carré.

La principale difficulté est de mettre à jour les façades d'un bâtiment lors de l'ajout d'une nouvelle cellule. Chaque arête de la cellule nouvellement insérée doit être analysée afin de déterminer si cette arête prolonge ou coupe une façade déjà existante ou si elle doit être considérée comme une nouvelle façade.

Les différents cas à considérer, à une symétrie près, sont illustrés à la Figure 4.24. Pour chaque arête, le premier test à effectuer est de vérifier si l'arête est incluse dans une façade. Si oui, deux cas sont possibles. Le premier, illustré à la Figure 4.24a, amène à réduire une façade à partir d'une de ses extrémités. Ainsi, après l'insertion de la nouvelle cellule, la façade AB laisse place à la façade CB. Le second cas, illustré à la Figure 4.24b, amène à une séparation d'une façade en deux. Ainsi, la façade AB se transforme en les deux façades AC et DB.

Si une arête n'appartient pas à une façade, elle peut quand même avoir des extrémités en commun avec une ou deux façades. Le cas le plus évident est illustré à la Figure 4.24c. L'arête relie deux façades pour n'en former qu'une seule. Cependant, il faut faire attention au cas de la Figure 4.24d. Ici, l'arête BC a des extrémités en commun avec les façades AB et CD. Néanmoins, elle ne prolonge uniquement que la façade AB. La façade CD n'a pas la même orientation que AB. Elle ne peut pas être une prolongation de AB. De façon analogue, l'arête BC ne prolonge aucune façade dans la Figure 4.24e.

Pour distinguer le cas 4.24c des cas 4.24d et 4.24e, il faut premièrement déterminer l'arête perpendiculaire à l'arête étudiée en l'extrémité en commun. Via cette arête perpendiculaire, nous pouvons identifier la cellule voisine à la cellule insérée et contenant l'extrémité de la façade potentiellement agrandie. Si cette cellule est rouge, la façade doit être prolongée. Sinon, elle ne doit pas être prolongée.

Les Figures 4.24f et 4.24g illustrent les cas où l'arête n'a qu'une seule extrémité en commun avec une façade. Dans le cas 4.24f, l'arête BC prolonge la façade AB. Dans le cas 4.24g, elle ne prolonge pas la façade AB.

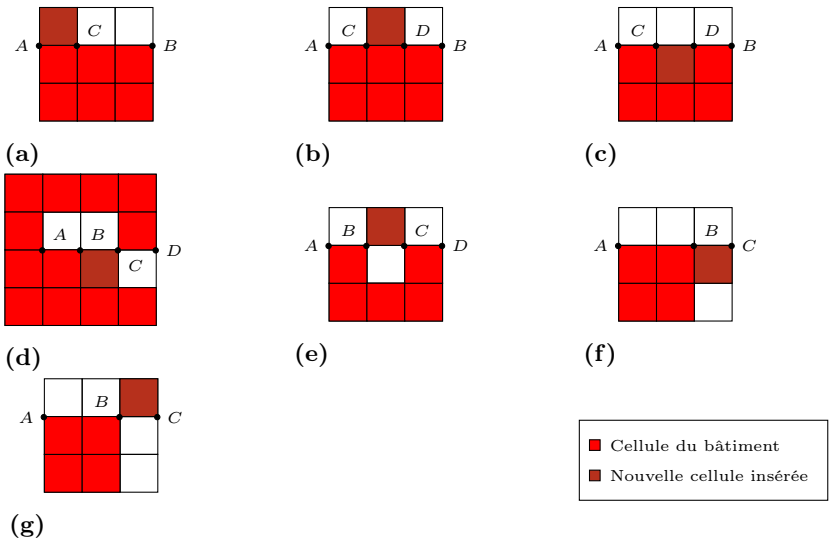


Figure 4.24 – Les différentes cas à considérer pour mettre à jour les façades d'un bâtiment.

Enfin, si l'arête n'est pas incluse à une façade et n'a pas d'extrémité en commun avec une façade, elle définit simplement une nouvelle façade.

Les façades sont implémentées via un objet `Segment`. Un `Segment` a connaissance des coordonnées de ses extrémités. Il peut être vertical ou horizontal. Via un ensemble de segments horizontaux et verticaux, il est possible de construire le polygone formé par la surface d'un bâtiment.

Maintenir l'ensemble des façades nécessite trois opérations de base : insérer une nouvelle façade, supprimer une façade et chercher une façade particulière. Il faut donc travailler avec des structures de données permettant d'effectuer ces trois opérations efficacement. Ainsi, CUBE utilise deux *Red-Black Trees* : un pour les façades horizontales et un pour les verticales.

Les *Red-Black Trees* sont des arbres binaires de recherche. Un arbre binaire est un ensemble de nœuds contenant des valeurs. Un nœud n peut être

connecté à deux autres nœuds appelés respectivement fils gauche et fils droit de n. Un arbre binaire de recherche ajoute les contraintes suivantes : la valeur d'un fils gauche doit être inférieure à celle de son père et la valeur d'un fils droit doit être supérieur à celle de son père. Un arbre binaire de recherche est une structure de données permettant de stocker et ordonner des données. Nous appelons la *racine* d'un arbre le nœud au sommet de l'arbre qui n'a pas de parents.

Dans un tel arbre, il est facile de tester si une valeur est stockée dans l'arbre en ayant connaissance de la racine de l'arbre. Par exemple, si nous voulons tester si un nœud à la valeur 6 dans l'arbre de la Figure 4.25, nous savons qu'il faut explorer les nœuds à gauche de la racine.

Un *Red-Black Tree* est un arbre binaire de recherche qui maintient un ensemble de propriétés lui permettant d'être balancé. Brièvement, un arbre est dit balancé si la différence entre le nombre de nœuds à gauche et à droite de la racine n'est pas trop élevée. La recherche d'une valeur dans un arbre balancé est plus efficace que dans un arbre qui ne l'est pas. Dans le présent document nous ne donnons que la définition d'un *Red-Black Tree*. Le lecteur intéressé par les détails du fonctionnement d'un tel arbre peut se référer à l'ouvrage de Cormen *et al.* [CLRS09].

Définition 4.5.3 (*Red-Black Tree* [CLRS09])**.** Un *Red-Black Tree* est un arbre binaire de recherche satisfaisant les propriétés suivantes :

1. Chaque nœud est rouge ou noir.
2. La racine (nœud au sommet de l'arbre) est noire.
3. Chaque feuille (nœud sans enfants noté NIL) est noire.
4. Si un nœud est rouge, alors ses deux enfants sont noirs.
5. Pour tout nœud, tous les chemins de ce nœud vers une feuille de sa descendance contiennent le même nombre de nœuds noirs.

Exemple 4.5.4. La Figure 4.25 illustre un exemple de *Red-Black Tree*. En effet cet arbre vérifie les cinq propriétés énoncées précédemment. Ainsi la racine

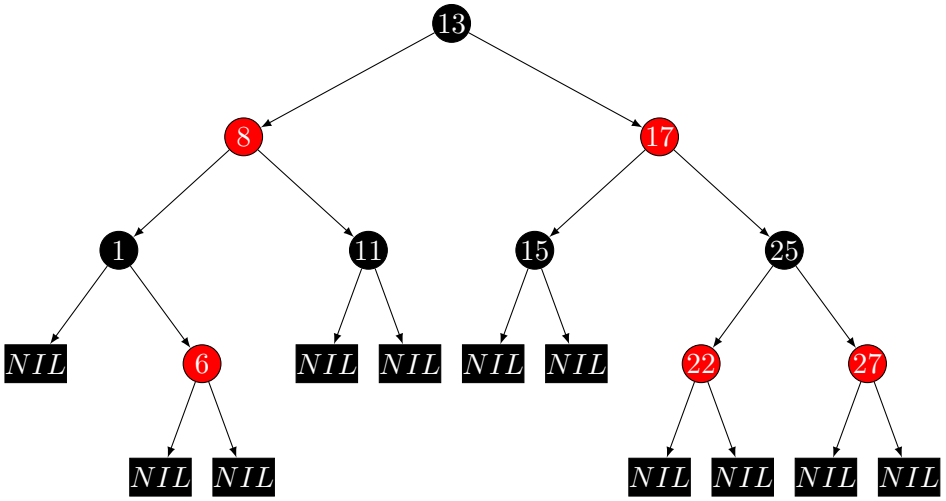


Figure 4.25 – Un exemple de *Red-Black Tree* [Nom19].

(nœud de valeur 13) et les feuilles (nœuds NIL) sont noires. Il est aisé de vérifier que les enfants d'un nœud rouge sont noirs. Par exemple, les enfants du nœud 8, les nœuds 1 et 11, sont bien noirs. Enfin, nous pouvons vérifier la cinquième propriété. Par exemple, à partir de la racine, les chemins vers les feuilles contiennent tous trois nœuds noirs.

Pour que les *Red-Black Trees* puissent travailler avec des segments, il faut définir une relation d'ordre sur ces derniers. Pour comparer deux segments horizontaux, CUBE compare d'abord les deux extrémités de gauche selon l'ordre lexicographique [10]. Si ces deux extrémités sont identiques, la comparaison se fait entre les extrémités de droite. Pour les segments verticaux, CUBE compare d'abord les extrémités du bas et si elles sont égales celles du haut [11].

Un fois une solution retournée par CUBE, il est possible d'exporter, dans

10. Pour comparer deux points du plan selon l'ordre lexicographique, nous comparons d'abord les abscisses des deux points. Si les abscisses sont égales, nous comparons ensuite les ordonnées.

11. Pour les segments verticaux, nous comparons d'abord les ordonnées puis les abscisses.

un fichier *JSON*, les coordonnées des sommets des surfaces inclues dans l'îlot et la hauteur des bâtiments. Via un script *Python*, ce fichier est utilisé par le logiciel *Blender* [Ble18] pour créer une représentation 3D de l'îlot. La Figure 4.26 contient le rendu 3D d'une configuration d'îlot retournée par CUBE.

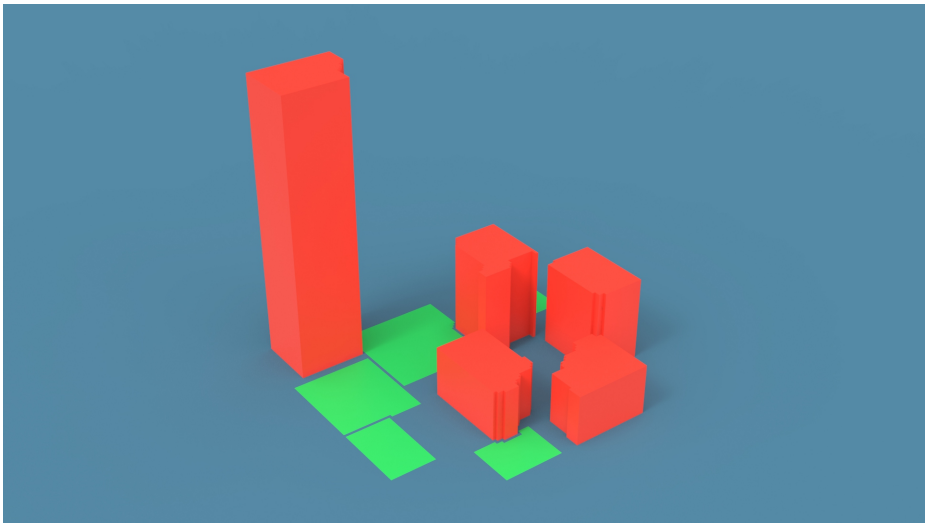


Figure 4.26 – Un rendu 3D de l'îlot basé sur une solution de CUBE.

Nous avons vu dans ce chapitre, comment la théorie des jeux peut nous aider à gérer des conflits entre différents composants de l'îlot, en particulier les conflits entre les parcelles bâties et les espaces verts. La théorie est également propice à l'ajout de nouveaux critères, via des nouveaux joueurs, ayant une façon de modifier l'îlot qui leur est propre.

Les critères de compacité pris en compte dans ce chapitre ont tous une influence sur la géométrie de l'îlot. Cependant ce n'est pas le type de critères identifiés dans les travaux de De Smet [De 18]. Dans le prochain chapitre, nous étudions les problématiques induites par ces autres types de critères à l'aide de notions de théories de jeux mais aussi de problèmes d'optimisation.

CHAPITRE 5

AFFECTATION DES BÂTIMENTS ET PROBLÈMES D'OPTIMISATION

Les critères considérés dans les chapitres précédents influent tous sur la géométrie de l'îlot. Grâce au module Théorie des Jeux de CUBE, il est aisé de considérer de nouveaux critères portant sur la forme des composants de l'îlot via un joueur dédié.

Cependant, ce n'est pas le seul type de critères mis en évidence par les travaux de De Smet [De 18]. Ces travaux préconisent, notamment, que les habitants aient une école maternelle et/ou primaire à au plus 600 m de leur logement. Travailler avec l'affectation des bâtiments demande une modélisation spécifique et adaptée. Il est maintenant nécessaire de travailler à une échelle urbanistique supérieure à celle de l'îlot. En effet, il ne serait pas pertinent de placer une école dans chaque îlot.

Le modèle basé sur des pavages est efficace à l'échelle d'îlots de l'ordre de 100 m sur 100 m. Néanmoins, travailler avec un pavage à une échelle supérieure amènerait à une augmentation de la taille du pavage. Démultiplier la taille du pavage aurait pour conséquence d'augmenter les coûts en mémoire et en temps de calcul, d'où la nécessité de changer de modélisation.

Pour déterminer l'affection d'un bâtiment, connaître sa géométrie précise n'est pas utile. Ainsi, nous pouvons ici abstraire la forme d'un bâtiment. Cette abstraction doit pouvoir conserver certaines données utiles d'un bâtiment telles que son aire, son périmètre, la position de son centre de gravité dans un système de coordonnées, *etc.*

Dans le module de CUBE décrit dans ce chapitre, cette abstraction est faite à l'aide d'un *graphe*.

5.1 Notions de théorie des graphes

Intuitivement, un graphe est un ensemble de nœuds reliés par des arêtes. Un graphe est un objet très versatile pouvant modéliser des interactions entre différentes entités. Un graphe peut, par exemple, modéliser un réseau social. Les nœuds représentent les utilisateurs du réseau social. Deux nœuds sont reliés par une arête si les utilisateurs sont amis sur le réseau.

Dans le cadre de CUBE, nous utilisons les graphes pour modéliser le réseau routier d'une ville. Les nœuds du graphe représentent les carrefours de la ville et les bâtiments et les arêtes symbolisent les routes reliant deux carrefours.

Définition 5.1.1 (Graphe non-orienté)**.** Un *graphe non-orienté* est défini par le couple (V, E) où

- V est un ensemble fini de sommets (aussi appelés nœuds) ;
- E est un ensemble fini de paires non ordonnées de sommets appelées arêtes. Une arête joignant un sommet u et un sommet v est notée $\{u, v\}$ (ou $\{v, u\}$).

Modéliser une ville à l'aide d'un graphe non-orienté implique que nous pouvons parcourir toutes les rues de notre ville dans les deux sens. Ceci est tout à fait cohérent si nous nous plaçons du point de vue d'un piéton mais pas du point de vue d'un conducteur. En effet, une ville peut posséder diverses rues à sens unique et ronds-points qu'un véhicule ne peut emprunter que dans un sens. Afin de modéliser une telle réalité, nous utilisons un *graphe orienté*.

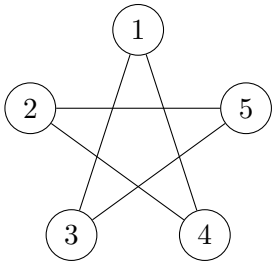


Figure 5.1 – Un graphe non-orienté.

Définition 5.1.2 (Graphe orienté)**.** Un *graphe orienté*, aussi nommé *graphe dirigé* est donné par le couple (V, E) où

- V est un ensemble fini de sommets (aussi appelés nœuds) ;
- E est un sous-ensemble de $V \times V$ dont les éléments sont appelés arcs. Un arc joignant un sommet u à un sommet v est noté (u, v).

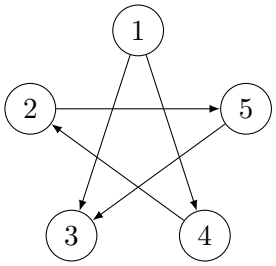


Figure 5.2 – Un graphe orienté.

La différence entre ces deux types de graphes réside dans le fait que, contrairement aux arcs, les arêtes n'ont pas de direction. Nous utilisons simplement la dénomination *graphe* quand le fait que ce dernier soit orienté ou non n'a pas d'importance dans le contexte.

Remarque 5.1.3. Nous pouvons facilement transformer un graphe non orienté en graphe orienté. Il suffit de transformer l'arête $\{u, v\}$ en la paire d'arcs (u, v)

et (v, u). Comme les définitions suivantes sont valables pour les deux types de graphes, nous utilisons ici la notation (u, v) aussi bien pour désigner une arête qu'un arc.

Au même titre que l'orientation des routes, il peut être intéressant que notre modèle prenne en compte certaines valeurs comme la distance entre deux carrefours voisins, le temps de parcours d'un segment de route, la vitesse maximale autorisée, *etc.* Pour ce faire, nous utilisons un *graphe pondéré*.

Définition 5.1.4 (Graphe pondéré)**.** Un *graphe pondéré* est donné par le triplet (V, E, c) où

- (V, E) est un graphe ;
- $c : E \longrightarrow \mathbb{R}$ est une fonction de poids aussi appelée fonction de coût.

Notation. Dans la suite, nous notons simplement $c(u, v)$ pour $c(\{u, v\})$ ou $c((u, v))$.

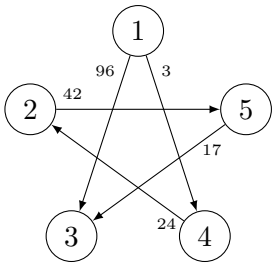


Figure 5.3 – Un graphe pondéré.

Remarque 5.1.5. Nous pouvons facilement convertir un graphe non pondéré en un graphe pondéré. Il suffit de considérer la fonction de coûts $c : E \longrightarrow \mathbb{R}$ telle que, $c(e) = 1$ pour tout $e \in E$.

Utiliser un graphe pour modéliser un réseau routier est d'autant plus pertinent qu'il est possible de simuler le déplacement d'un point A à un point B. Cela se fait via la notion de *chemin* dans un graphe.

Définition 5.1.6 (Chemin)**.** Soit $G = (V, E)$ un graphe. Un *chemin* de G est une suite de sommets $P = (s_1, \ldots, s_k)$ avec $k \geqslant 1$, tels que $(s_i, s_{i+1}) \in E$ pour tout $0 \leqslant i \leqslant k-1$. Un chemin est dit *élémentaire* si pour tout $i \neq j$, $s_i \neq s_j$. Nous notons $W(P)$ le poids ou la longueur de P défini comme suit

$$W(P) = \begin{cases} k-1 & \text{si } G \text{ n'est pas pondéré,} \\ \sum_{i=0}^{k-1} c(s_i, s_{i+1}) & \text{si } G = (V, E, c) \text{ est un graphe pondéré.} \end{cases}$$

Remarque 5.1.7. Un chemin passant par un seul sommet et n'empruntant aucun arc est accepté. La longueur d'un tel chemin vaut 0.

Notation. Soient $G = (V, E)$ et $s, t \in V$. Nous notons $\mathrm{Paths}(s, t)$, l'ensemble des chemins de s à t dans G.

Définition 5.1.8 (Cycle)**.** Un cycle d'un graphe G est un chemin $(s_1, \ldots, s_k)$ tel que $s_1 = s_k$. Nous définissons un *cycle élémentaire* de façon analogue à un chemin élémentaire. La seule différence est que nous acceptons que $s_1 = s_k$. Dans un graphe pondéré, un cycle C est dit *négatif* si $W(C) < 0$. Un graphe qui ne possède pas de cycles est dit *acyclique*. Sinon, il est dit *cyclique*.

Exemple 5.1.9. Considérons le graphe de la Figure 5.4. Nous avons :

- $(2, 3, 4, 1, 2, 3, 5)$ est un chemin de longueur 6 non élémentaire ;
- $(2, 3, 5)$ est un chemin élémentaire de longueur 2 ;
- $(2, 3, 4, 1, 2, 3, 4, 2)$ est un cycle de longueur 7 non élémentaire ;
- $(3, 4, 1, 2, 3)$ est un cycle élémentaire de longueur 4 ;
- G est cyclique.

Exemple 5.1.10. Le graphe de la Figure 5.3 est acyclique.

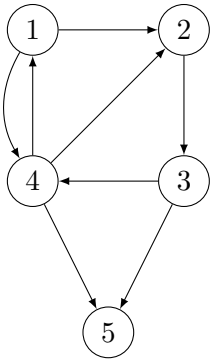


FIGURE 5.4

Définition 5.1.11 (Sous-graphe)**.** Soit $G = (V, E)$ un graphe. Le graphe $H = (W, F)$ est un sous-graphe de G si et seulement si $W \subseteq V$ et $F \subseteq \{(u, v) \in E : u, v \in W\}$.

Définition 5.1.12 (Sous-graphe induit)**.** Soit $G = (V, E)$ un graphe. Le graphe $H = (W, F)$ est un sous-graphe induit de G si et seulement si $W \subseteq V$ et $F = \{(u, v) \in E : u, v \in W\}$.

Remarque 5.1.13. Soient $G = (V, E)$ et $H = (W, F)$ tel quel $W \subseteq V$. Le graphe H est un sous-graphe de G induit par W si et seulement si F est l'ensemble de *tous* les arcs qu'il y a entre les nœuds de W dans G. Si F ne contient pas tous ces arcs, alors H est simplement un sous-graphe de G.

Exemple 5.1.14. Soit G le graphe de la Figure 5.4. Le graphe de la Figure 5.5a est un sous-graphe de G. Le graphe de la Figure 5.5b est un sous-graphe de G induit par les sommets 1, 2, 3 et 4.

Définition 5.1.15 (Graphe connexe)**.** Un graphe non orienté $G = (V, E)$ est dit *connexe* si pour tout $v, w \in V$, il existe un chemin de v à w. Un graphe dirigé $G = (V, E)$ est dit *fortement connexe* si pour tout $v, w \in V$,

(a) Un sous-graphe. **(b)** Un sous-graphe induit.

Figure 5.5

il existe un chemin de v à w. Un sous-graphe *maximal* connexe (respectivement fortement connexe) de G est appelé *composante connexe* (respectivement *fortement connexe*) de G.

Exemple 5.1.16. Tous les graphes illustrés précédemment sont connexes. Le graphe de la Figure 5.6 n'est pas connexe. Ses composantes connexes sont le sous-graphe induit par les sommets 1, 2 et 3 et le sous-graphe induit par les sommets 4 et 5.

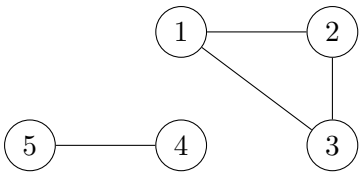


Figure 5.6

Nous observons à la Figure 5.3 qu'il y a deux chemins pour aller du sommet 1 au sommet 2, à savoir le chemin $(1, 3)$ de longueur 96 et le chemin $(1, 4, 2, 5, 3)$ de longueur 86. En situation réelle, quand nous avons plusieurs chemins pour aller d'un point A à un point B, nous sommes souvent intéressés par savoir lequel est *le plus court*.

Définition 5.1.17 (Plus court chemin)**.** Soient $G = (V, E)$ un graphe sans cycles négatifs et $u, v \in V$ tels que v est atteignable à partir de u. Le chemin P^* est un *plus court chemin* de u à v si

$$W(P^*) = \min_{Q \in \text{Paths}(u,v)} W(Q).$$

Remarque 5.1.18. Si un graphe a des cycles négatifs, il n'existe pas toujours un chemin dont la longueur est plus petite que celle des autres chemins. Par exemple, pour le graphe de la Figure 5.7, il n'existe pas de plus court chemin de 1 à 3 dans le sens où, plus le chemin boucle sur le nœud 2, plus le poids du chemin diminue.

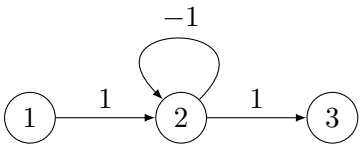


Figure 5.7

Remarque 5.1.19. Il ne faut pas confondre la non-existence d'un plus court chemin dans le cas d'un graphe non-connexe et dans le cas d'un graphe avec un cycle négatif. Pour les graphes non-connexes, un plus court chemin n'existe pas dans le sens où il n'y a *aucun* chemin entre certaines paires de sommets et donc *a fortiori* pas de plus court chemin. Pour les graphes avec un cycle négatif, un plus court chemin n'existe pas dans le sens où nous ne pouvons pas trouver un chemin plus court que tous les autres puisque chaque passage par un cycle de poids négatif diminue à chaque fois le poids d'un chemin.

En admettant que nous voulions toujours prendre le chemin le plus court entre deux points, nous définissons la distance entre deux sommets du graphe par la longueur d'un plus court chemin entre ces deux sommets.

Définition 5.1.20 (Distance dans un graphe)**.** Soit $G = (V, E)$ un graphe sans cycle de poids négatif et soient u, $v \in V$. Nous définissons la *distance entre u et v* par

$$d_G(u, v) = \begin{cases} \min_{P \in \mathrm{Paths}(u,v)} W(P) & \text{si } \mathrm{Paths}(u, v) \neq \varnothing, \\ +\infty & \text{sinon.} \end{cases}$$

Trouver la longueur d'un plus court chemin entre deux nœuds d'un graphe est un problème classique de théorie des graphes. Il existe plusieurs algorithmes permettant de résoudre ce problème. Dans l'implémentation de CUBE, nous utilisons principalement l'algorithme de Dijkstra (Algorithme 7) que nous présentons brièvement par la suite. Plus d'informations sur l'algorithme de Dijkstra et les autres algorithmes de plus courts chemins sont consultables dans l'ouvrage de Cormen *et al.* [CLRS09].

L'algorithme de Dijkstra nous fournit la longueur d'un plus court chemin entre un nœud source s et les autres nœuds du graphe. Pour ce faire, l'algorithme explore les sommets du graphe de proche en proche. Dans un premier temps, nous calculons la distance entre le nœud source s et ses successeurs. Ensuite, nous visitons le nœud *le plus proche* du nœud source et nous calculons la distance de ses successeurs non visités au nœud source. Et ainsi de suite jusqu'à avoir visité tous les sommets du graphe.

L'algorithme utilise une structure de données appelée *file de priorité*. Cette structure attribue à chacun de ses éléments une valeur spécifique appelée *priorité*. Une telle file est capable de supprimer et retourner son élément ayant la plus petite priorité via la méthode *extract_min*. Elle peut également modifier la priorité de ses éléments via la méthode *set_priority*.

Algorithme 7 : Algorithme de Dijkstra

Entrées : $G = (V, E, C)$ un graphe pondéré tel que $c : E \longrightarrow \mathbb{R}^+$ et un sommet $s \in V$.

Sortie : Une liste d telle que, pour tout $v \in V$, $d[v] = d_G(s, v)$.

```
1  soit d une liste vide
2  soit Q une file de priorité vide
3  d[s] ← 0
4  ajouter s à Q avec la priorité 0
5  pour tout v ∈ V\{s} faire
6  |  d[v] ← +∞
7  |  ajouter v à Q avec la priorité +∞
8  tant que Q ≠ ∅ faire
9  |  u ← Q.extract_min()
10 |  marquer u comme visité
11 |  pour tout v successeur non visité de u faire
12 |  |  alt ← d[u] + c(u, v)
13 |  |  si alt < d[v] alors
14 |  |  |  d[v] ← alt
15 |  |  |  Q.set_priority(v, d[v])
16 retourner d
```

Remarque 5.1.21. L'algorithme de Dijkstra suppose que le poids de tous les arcs est positif. Cette hypothèse est nécessaire car la présence de poids négatifs est incompatible avec l'unique exploration de chaque nœud du graphe. En effet, considérons le graphe de la Figure 5.8. En partant du nœud 1, nous explorons d'abord le nœud 2 puis le nœud 3. Le plus court chemin de 1 à 2 est le chemin $(1, 3, 2)$. Cependant, comme le nœud 2 a déjà été visité quand nous visitons le nœud 3, l'algorithme ne considère pas ce chemin et nous retourne donc une longueur de plus court chemin erronée.

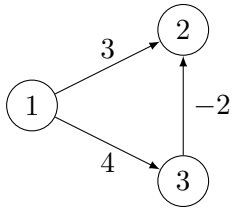


Figure 5.8

5.2 Graphes dynamiques et problème des chemins les plus rapides

Nous avons précédemment expliqué comment calculer la longueur du chemin le plus court en termes de distance. Il peut être également intéressant de déterminer le chemin le plus rapide entre deux points d'une ville. Une première approche simple consiste à simplement changer l'interprétation des poids des arcs du graphe. Le poids symbolise maintenant le temps de parcours moyen entre deux nœuds.

Cependant, en pratique, le temps de parcours moyen d'une route peut varier en fonction de l'heure. En effet, aux heures de pointe, la circulation est fortement ralentie et donc les trajets aussi. Afin de modéliser cette réalité, nous utilisons les graphes dynamiques présentés dans l'article de Chabini [Cha98].

5.2.1 Problème du chemin le plus rapide à partir d'un nœud source

Définition 5.2.1 (Graphe dynamique [Cha98])**.** Un *graphe dynamique* est donné par le quintuplet $(V, E, \alpha, \mathcal{C}, \mathcal{D})$ où

- (V, E) est un graphe orienté ;
- $\alpha \in \mathbb{N}$;
- $\mathcal{C} = (c_e)_{e \in E}$ est une famille de fonctions telles que, pour tout $e \in E$,

$c_e : \mathbb{N} \longrightarrow \mathbb{R}$ et pour tout $t \geqslant \alpha$, $c_e(t)$ est constante ; pour tout $t \in \mathbb{N}$, $c_e(t)$ dénote le coût pour traverser l'arc e en démarrant au temps t ;

- $\mathcal{D} = (d_e)_{e \in E}$ est une famille de fonctions telles que, pour tout $e \in E$, $d_e : \mathbb{N} \longrightarrow \mathbb{R}^+$ et pour tout $t \geqslant \alpha$, $d_e(t)$ est constante ; pour tout $t \in \mathbb{N}$, $d_e(t)$ dénote le temps nécessaire pour parcourir l'arc e en démarrant au temps t.

Remarque 5.2.2. Nous supposons que, en tout temps, notre graphe ne possède pas de cycles négatifs.

Avant d'énoncer formellement notre variante du problème des plus courts chemins, signalons que nous devons distinguer plusieurs cas pour réaliser nos calculs. Il faut premièrement discerner si lorsque nous atteignons un nœud, nous pouvons attendre ou non avant de redémarrer vers le prochain arc. Nous devons également distinguer le cas où notre graphe dynamique respecte la *condition FIFO* (First-In-First-Out), condition selon laquelle le premier à utiliser un arc est le premier à quitter cet arc.

Définition 5.2.3 (Condition FIFO [Cha98])**.** Soit $G = (V, E, \alpha, \mathcal{C}, \mathcal{D})$ un graphe dynamique. Le graphe G respecte la *condition FIFO* si pour tout $e \in E$ et pour $t \in \mathbb{N}$,

$$t + d_e(t) \leqslant (t+1) + d_e(t+1).$$

Remarque 5.2.4. Dans le cas où nous sommes intéressés par le chemin le plus rapide entre deux points, si notre graphe respecte la condition FIFO, il n'est jamais intéressant d'attendre dans un nœud avant de parcourir un arc.

Étant donné un nœud source s, nous voulons ici trouver le temps minimum pour accéder aux autres sommets du graphe. Nous supposons ici que d'attendre dans un nœud est interdit. Le cas, où l'attente est permise, est étudié dans l'article de Chabini [Cha98].

Notation. Soient $G = (V, E, \alpha, \mathcal{C}, \mathcal{D})$ un graphe dynamique et $s \in V$ le nœud source. Pour $v \in V$, nous notons τ_v le temps minimum pour atteindre v à partir de s en démarrant au temps 0. Nous supposons dans la suite que nous démarrons du temps 0 afin d'alléger les notations.

Soient $G = (V, E, \alpha, \mathcal{C}, \mathcal{D})$ un graphe dynamique et $s \in V$ notre nœud source. Intuitivement, pour trouver le chemin le plus rapide entre le nœud s et un nœud $w \in V$, nous avons envie de calculer les chemins les plus rapides vers les prédécesseurs de w et de compléter notre chemin vers w en partant du prédécesseur qui nous permet d'arriver le plus vite à w. Cependant, nous n'avons pas supposé que G respecte la condition FIFO.

Comme illustré à la Figure 5.9, il peut être ici plus intéressant de faire un détour vers les prédécesseurs de w. En effet, le chemin (s, v) est le plus rapide pour aller de s à v. Cependant, il nous fait arriver en v au temps 1. Comme nous ne pouvons pas attendre, si nous voulons aller vers w, il nous faudra 10 unités de temps supplémentaires pour y arriver. Alors que si nous avions pris le chemin (s, u, v), nous serions arrivés en v au temps 2 nous permettant ainsi de traverser l'arc (v, w) en une seule unité de temps.

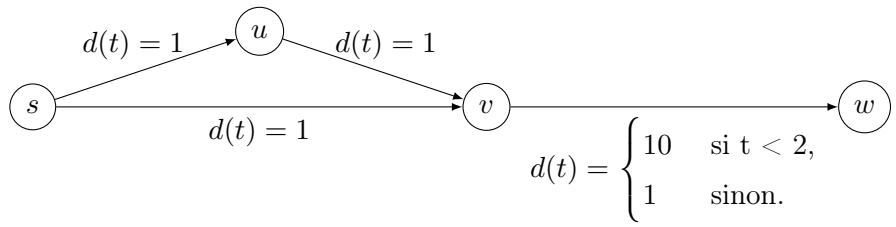


Figure 5.9 – Un graphe dynamique ne respectant la condition FIFO.

Ainsi, pour trouver la valeur τ_w, il nous faut considérer, pour chaque prédécesseur de w, tous les temps de départ possibles à partir de ces derniers. Nous avons donc la définition suivante :

Définition 5.2.5. Soient $G = (V, E, \alpha, \mathcal{C}, \mathcal{D})$ un graphe dynamique et $s \in V$. Pour tout $v \in V$, nous notons $T(v) = \{t$: il existe un chemin partant de s au temps 0 et arrivant en v au temps $t\}$ et $Pred(v)$ l'ensemble

des prédécesseurs de v dans le graphe (V, E). Nous avons

$$\tau_v = \begin{cases} \min_{u \in Pred(v)} \min_{t \in T(u)} (t + d_{(u,v)}(t)) & \text{si } v \neq s, \\ 0 & \text{sinon.} \end{cases} \tag{5.1}$$

Notons que si notre graphe dynamique satisfait la condition FIFO, notre première intuition est valide. En effet, sous cette condition, plus nous partons tôt d'un prédécesseur, plus nous arrivons tôt au nœud cible. Cela nous donne donc la propriété suivante :

Proposition 5.2.6.
Soient $G = (V, E, \alpha, \mathcal{C}, \mathcal{D})$ un graphe dynamique respectant la condition FIFO et $s \in V$. Pour tout $v \in V$, nous avons

$$\tau_v = \begin{cases} \min_{u \in Pred(v)} (\tau_u + d_{(u,v)}(\tau_u)) & \textit{si } v \neq s, \\ 0 & \textit{sinon.} \end{cases} \tag{5.2}$$

Revenons brièvement au problème des plus courts chemins classique. Soient $G = (V, E, c)$ un graphe pondéré et $s \in V$. Pour tout $v \in V$, calculer $d_G(s, v)$ peut se faire, comme le fait l'algorithme de Dijkstra, via l'équation suivante :

$$d_G(s, v) = \begin{cases} \min_{u \in Pred(v)} (d_G(s, u) + c(u, v)) & \text{si } v \neq s, \\ 0 & \text{sinon.} \end{cases} \tag{5.3}$$

Nous remarquons que les équations (5.2) et (5.3) sont très similaires pour ne pas dire identiques à notations près. En effet, pour les deux, nous avons besoin de connaître la distance (respectivement le temps de parcours) du nœud source aux prédécesseurs du nœud v et le coût des arcs. Le lemme suivant découle de cette observation.

Lemme 5.2.7.
Pour un graphe dynamique respectant la condition FIFO, le problème des chemins les plus rapides à partir d'un nœud source est équivalent au problème des plus courts chemins classique.

5.2.2 Problème du chemin le plus rapide vers un nœud cible

Trouver les plus courts chemins à partir de tous les nœuds du graphe vers un nœud cible est très aisé dans le cas classique. En effet, ce problème est équivalent au problème du plus court chemin à partir d'un nœud cible. Il suffit d'inverser le sens des arcs du graphe pour passer d'une version du problème à l'autre.

Cependant, nous ne pouvons pas réaliser une telle conversion pour les graphes dynamiques. En effet, nous devons ici tenir compte du temps qui s'écoule, celui-ci influençant le coût des arcs.

Nous avons donc besoin d'une méthode spécifique pour résoudre ce problème. L'article de Chabini [Cha98] propose un algorithme qui calcule le chemin le plus rapide entre chaque sommet du graphe et un nœud cible pour tous les temps de départ possibles pour le cas où l'attente est interdite.

Notation. Soit $G = (V, E, \alpha, \mathcal{C}, \mathcal{D})$ un graphe dynamique et $q \in V$ le nœud cible. Soit $v \in V$, nous notons $\chi_v(t)$ le temps minimum pour aller du sommet v au sommet q en démarrant au temps t.

Avant de décrire l'algorithme résolvant notre problème, intéressons-nous aux grandes idées exploitées par ce dernier. Pour $G = (V, E, \alpha, \mathcal{C}, \mathcal{D})$, nous savons que pour tout arc $e \in E$ et pour tout $t \geqslant \alpha$, $d_e(t)$ est constante. Dès lors, pour des temps de départ supérieurs à α, calculer $\chi_v(t)$ peut se faire une seule fois à l'aide des plus courts chemins classiques. Maintenant, supposons que nous démarrons du nœud v au temps $\alpha - 1$. Nous devons maintenant nous diriger vers le successeur de v, notons-le w, qui nous permet d'atteindre q le plus rapidement possible. Pour cela, nous devons tenir compte du temps de parcours de l'arc (v, w) et du temps pour aller de w à q. Notons que, si nous partons de v au temps $\alpha - 1$, nous arriverons en w en un temps supérieur à α. Nous avons donc déjà à notre disposition le temps minimal pour aller de w à q. Nous pouvons donc facilement calculer $\chi_v(\alpha - 1)$ pour tout $v \in V$. Ensuite, à l'aide de ces dernières valeurs, nous pouvons calculer $\chi_v(\alpha - 2)$ et ainsi de suite.

En bref, notre algorithme, nommé algorithme DOT pour Decreasing Order

of Time, calcule, pour tout $v \in V$ et pour tout $t \in \{0, \ldots, \alpha\}$, $\chi_v(t)$ à l'aide de l'équation

$$\chi_v(t) = \begin{cases} \min_{w \in Succ(v)} \left(d_{(v,w)}(t) + \chi_w(t + d_{(v,w)}(t)) \right) & \text{si } v \neq q, \\ 0 & \text{sinon,} \end{cases}$$

et ce, en commençant par $t = \alpha$ et en décroissant vers 0.

Notation. Soient $G = (V, E, \alpha, \mathcal{C}, \mathcal{D})$ un graphe dynamique et $t \in \mathbb{N}$. Nous notons $G(t)$ le graphe pondéré (V, E, c) où, pour tout $e \in E$, $c(e) = d_e(t)$.

Algorithme 8 : DOT [Cha98]

Entrées : Un graphe dynamique $G = (V, E, \alpha, \mathcal{C}, \mathcal{D})$ et $q \in V$ un nœud cible.

Sorties : Le temps minimal $\chi_v(t)$ pour aller du nœud v au nœud q en démarrant au temps t, pour tout $v \in V$ et pour tout $t \in \{0, \ldots, \alpha\}$

```
pour tout v ∈ V et t < α faire
    si v ≠ q alors
        χ_v(t) ← +∞
    sinon
        χ_q(t) ← 0
pour tout v ∈ V faire
    χ_v(α) ← d_{G(α)}(v, q)
pour t allant de α − 1 à 0 faire
    pour tout (v, w) ∈ E faire
        χ_v(t) ← min(χ_v(t), d_{(v,w)}(t) + χ_w(t + d_{(v,w)}(t)))
retourner χ_v(t) pour tout v ∈ V, pour tout t ∈ {0, ..., α}
```

Exemple 5.2.8. Considérons le graphe dynamique de la Figure 5.9. Nous supposons que le paramètre α associé à ce graphe vaille 3 et que notre cible est le nœud w. Exécutons l'algorithme DOT sur ce graphe.
Nous obtenons facilement que $\chi_s(3) = 2$, $\chi_u(3) = 2$ et $\chi_v(3) = 1$. Si nous démarrons au temps 2, nous obtenons les mêmes résultats. Si nous démarrons

au temps 1, nous avons cette fois-ci $\chi_v(1) = 10$. Comme en partant de s ou de u, nous arrivons en v au temps 2, nous avons $\chi_s(1) = 2$ et $\chi_u(1) = 2$. Si nous démarrons au temps 0, nous avons $\chi_v(0) = 10$ et $\chi_u(0) = 11$. En démarrant de s, si nous choisissons l'arc (s, v), nous arrivons en v au temps 1. Nous avons calculé précédemment qu'en démarrant de v au temps 1, il nous faut 10 unités de temps pour atteindre w. Il nous en faut donc 11 en partant de s au temps 0. Si nous choisissons l'arc (s, u), en arrivant au temps 1 au sommet u, il ne nous faudra que 2 unités de temps supplémentaires pour atteindre w. Nous en déduisons que $\chi_s(0) = 3$.

5.3 Affectation des bâtiments et jeu sur graphe

Afin de modéliser un quartier et les affectations des bâtiments qui le composent, CUBE utilise un graphe analogue à celui de la Figure 5.10.

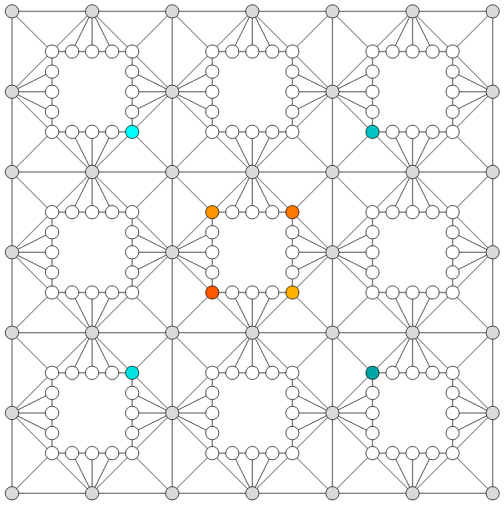


Figure 5.10

Les couleurs des différents nœuds du graphe ont une signification. Les nœuds gris sont des nœuds routiers, modélisant les carrefours, les passages piétons, *etc.* Les nœuds en nuances d'orange sont des commerces de proximité.

Les nœuds en nuances de bleu sont des écoles. Les nœuds blancs sont initialement des positions potentielles pour chacune des affectations (hors nœuds gris). Après avoir décidé quelles sont les positions des affectations, les nœuds blancs sont considérés comme des logements.

Remarque 5.3.1. Dans la Figure 5.10, un groupe de nœuds blancs entourés par des nœuds gris est donc un îlot.

Remarque 5.3.2. Nos graphes modélisant des quartiers ou des villes, nous supposons qu'ils sont connexes.

5.3.1 Trouver les positions optimales

Afin de déterminer les positions optimales, dans un graphe, des différentes affectations, CUBE utilise des notions de théorie des jeux. Intuitivement, nous avons pour ce jeu un nombre *fixe* de joueurs. Chaque joueur possède un jeton de couleur relatif à un type de bâtiment. Les joueurs peuvent placer leur jeton sur un nœud blanc du graphe.

Chaque joueur veut trouver la meilleure position possible pour son jeton. Pour cela, les joueurs vont, tour à tour, déplacer leur jeton vers un nœud blanc voisin. Le jeu s'arrête quand, pour tous les joueurs, bouger un jeton n'améliore plus les gains reçus.

Remarque 5.3.3. Trouver la position optimale d'une affectation d'un bâtiment en explorant les nœuds du graphe de proche en proche est analogue au *Hill Climbing* (voir section 3.1.1)

Dans le graphe de la Figure 5.10, se contenter d'explorer le graphe de nœud blanc en nœud blanc amène les joueurs à trouver la position optimale pour leur jeton dans un îlot. Or, l'objectif est de trouver la meilleure position dans tout le graphe. Pour pallier cela, nous explorons le graphe à la manière de la recherche par voisinage variable (VNS pour *Variable Neighborhood Search*) [HMBP19].

Le VNS est une métaheuristique qui utilise plusieurs opérations de voisinage. Le VNS consiste à exécuter premièrement un *Hill Climbing* classique. Une fois un optimum local trouvé pour la première opération de voisinage,

l'algorithme génère un nouveau voisinage avec la deuxième opération. Si une meilleure solution est trouvée avec ce voisinage, un *Hill Climbing* avec la première opération de voisinage est relancé. Si non, l'algorithme génère un voisinage avec une troisième opération et ainsi de suite. L'algorithme s'arrête quand une solution meilleure que ses voisins pour *toutes* les opérations de voisinage est trouvée.

Dans le cadre de notre jeu sur graphe, une fois la meilleure position trouvée dans un îlot, les joueurs peuvent bouger leur jeton vers un nœud blanc distant d'au plus deux arcs dans le graphe. Ce mouvement permet aux joueurs d'atteindre de nouveaux nœuds blancs sans être bloqués par les nœuds gris.

5.3.2 Les joueurs oranges

Les nœuds oranges symbolisent des commerces de proximité. Les joueurs associés à ces commerces veulent trouver la position qui leur apporte le plus de clients. Nous supposons que les clients sont uniformément distribués sur les nœuds blancs. Afin d'attirer le plus de clients possibles, les joueurs vont chercher les nœuds les plus facilement accessibles dans le graphe. Ainsi, l'objectif des joueurs est donc de trouver les nœuds du graphe avec la plus grande *centralité de proximité.*

Définition 5.3.4 (Centralité de proximité [Fre78])**.** Soient $G = (V, E)$ et $v \in V$. La *centralité de proximité* de v vaut :

$$C_C(v) = \frac{1}{\sum_{s \in V} d_G(s, v)}.$$

Remarque 5.3.5. Si G est un graphe non connexe, par convention, $C_C(v) = 0$ pour tout sommet v de G.

Définition 5.3.6. Soit un graphe G, nous notons $White(G)$ l'ensemble des nœuds blancs de G. Soit $N = \{1, \ldots, n\}$, l'ensemble des joueurs. Le problème d'affectation des commerces de proximité est modélisé comme le

jeu $\mathcal{G} = (N, (S_i)_{i \in N}, (g_i)_{i \in N})$, où pour tout $i \in N$, $S_i = White(G)$. Si nous notons $v_i \in White(G)$ le nœud de G choisi pas le joueur i, la fonction de gains de ce joueur est définie comme suit :

$$g_i(v_i) = \sum_{u \in White(G)} d_G(u, v_i),$$

où $d_G(u, v_i)$ est la longueur d'un plus court chemin du nœud u vers le nœud v. Les joueurs veulent minimiser leur fonction de coûts.

La Figure 5.11 illustre un EN pour une instance de ce jeu à quatre joueurs. Nous remarquons que, comme pour le problème des marchands de glaces (Exemple 4.1.9), les positions choisies par les joueurs sont proches du centre du graphe [12].

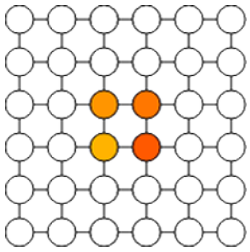


Figure 5.11 – Un EN pour quatre joueurs oranges.

Remarque 5.3.7. Une autre façon de déterminer la position optimale de k commerces de proximité est analogue au problème des marchands de glaces (Exemple 4.1.9) : les clients vont vers le commerce le plus proche et les joueurs veulent maximiser leur nombre de clients. Si pour tout $i \in N$, nous notons v_i le nœud sélectionné par le joueur i, nous pouvons définir la fonction de gains suivante :

$$g_i(v_i, v_{-i}) = |\{u \in White(G) : d_G(u, v_i) = min_{o \in \{v_i : i \in N\}} d_G(u, o)\}|.$$

12. La complexité algorithmique pour le problème d'affectation des commerces de proximité est étudiée dans l'annexe A

Par expériences, cette famille de fonctions de gains amène à l'apparition de plus d'EN dans le jeu. Cependant, en utilisant le VNS, les joueurs convergent vers le même EN que précédemment, *i.e.* les joueurs choisissent les nœuds proches du centre du graphe.

5.3.3 Les joueurs bleus

Les nœuds bleus sont des écoles. Comme pour les commerces, nous définissons un jeu multi-joueurs pour déterminer les meilleures positions. Les différences avec le précédent modèle sont les fonctions de gains.

Définition 5.3.8. Soit un graphe G, nous notons $White(G)$ l'ensemble des nœuds blancs de G et $Blue(G)$ l'ensemble de nœuds bleus de G. Soit $N = \{1, \dots, n\}$, l'ensemble des joueurs. Le problème d'affectation des écoles est modélisé comme le jeu $\mathcal{G} = (N, (S_i)_{i \in N}, (g_i)_{i \in N})$, où pour tout $i \in N$, $S_i = White(G)$. Si nous notons $v_i \in White(G)$ le nœud de G choisi par le joueur i, nous avons $Blue(G) = \{v_i : i \in N\}$la fonction de gains de ce joueur est définie comme suit :

$$g_i(v_i, v_{-i}) = \sum_{u \in White(G)} \min_{b \in Blue(G)} d_G(u, b),$$

où $d_G(u, b)$ est la longueur d'un plus court chemin du nœud u vers le nœud b. Les joueurs veulent minimiser leur fonction de gains.

Les fonctions de coûts sont les mêmes pour tous les joueurs bleus. Ils vont coopérer. Cela symbolise le fait que, dans notre modèle, les écoles ne se font pas directement concurrence. Leur objectif est que tous les habitants aient un accès rapide à l'éducation.

La Figure 5.12 illustre un EN pour une instance de ce jeu à quatre joueurs. La coopération des joueurs a pour conséquence que les positions choisies ne sont plus au centre du graphe [13]. Pour faire un parallèle avec le problème des marchands de glace (Exemple 4.1.9), c'est comme si les deux marchands

13. La complexité algorithmique pour le problème d'affectation des écoles est étudiée dans l'annexe A

travaillaient pour la même franchise. Ils ne sont donc plus en concurrence. Ils peuvent donc se partager la plage équitablement et se positionner de telle sorte à favoriser le confort des clients. Ainsi, par exemple, le profil de stratégies $(\frac{1}{4}, \frac{3}{4})$ est maintenant un EN.

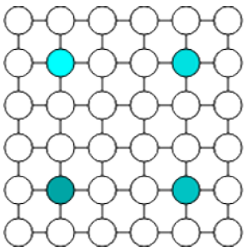


Figure 5.12 – Un EN pour quatre joueurs bleus.

Modéliser l'affectation des écoles via un jeu demande de travailler avec un nombre *fixe* d'écoles. Or, il est intéressant de déterminer le *nombre optimal* d'écoles à ouvrir pour satisfaire le plus grand nombre d'habitants. Pour cela, nous modélisons notre problème comme une instance du *Facility Location Problem*.

5.4 Nombre d'écoles et *Facility Location Problem*

5.4.1 Définition du Facility Location Problem

Intuitivement, le *Facility Location Problem* (FLP) est un problème d'optimisation modélisant la situation suivante. Nous avons un ensemble de positions où construire des entrepôts et un ensemble de clients. Ouvrir un entrepôt a un coût. Les clients doivent se faire livrer des marchandises par l'entrepôt le plus proche. L'objectif du FLP est de déterminer quels entrepôts ouvrir pour minimiser les coûts de transport vers les clients tout en considérant les coûts de construction.

Dans le cadre de CUBE, les entrepôts sont considérés comme des écoles et les clients comme des logements. Les parents inscrivent leurs enfants dans

l'école la plus proche.

Définition 5.4.1 (Facility Location Problem)**.** Soient $I = \{i_1, \ldots, i_m\}$ un ensemble de sites où des entrepôts peuvent être construits, $J = \{j_1, \ldots, j_n\}$ un ensemble de clients, le vecteur $f = (f_i)_{i \in I}$ où f_i est le coût de construction de l'entrepôt en i, et une matrice $C = (c_{ij})_{\substack{i \in I \\ j \in J}}$ où c_{ij} est le coût de transport de l'entrepôt i au client j.
L'objectif du FLP est de trouver $\varnothing \subset S \subseteq I$ tel que S minimise :

$$\varphi(S) = \sum_{i \in S} f_i + \sum_{j \in J} \min_{i \in S} c_{ij}.$$

Exemple 5.4.2. Une instance du FLP peut être modélisée comme une suite binaire. À chaque élément de la suite est attribué un entrepôt. Si à un indice donné, la valeur de la suite vaut 1, l'entrepôt est ouvert. Si la valeur vaut 0, l'entrepôt est fermé.

Pour l'instance de la Figure 5.13, les valeurs des solutions possibles sont les suivantes :

- $\varphi(000) = +\infty$ (par convention) ;
- $\varphi(100) = 7 + 2 + 5 + 4 = 18$;
- $\varphi(010) = 5 + 1 + 2 + 7 = 15$;
- $\varphi(001) = 5 + 5 + 4 + 1 = 15$;
- $\varphi(110) = 7 + 5 + 1 + 2 + 4 = 19$;
- $\varphi(101) = 7 + 5 + 2 + 4 + 1 = 19$;
- $\varphi(011) = 5 + 5 + 1 + 2 + 1 = 14$;
- $\varphi(111) = 7 + 5 + 5 + 1 + 2 + 1 = 21$;

Pour cette instance, la solution optimale consiste à ouvrir les entrepôts 2 et 3.

Remarque 5.4.3. La définition du FLP que nous utilisons est la définition la plus simple de ce problème, parfois appelée *Uncapacited Facility Location Problem*. Il existe des variantes plus complexes prenant en compte la capacité de stockage des entrepôts ou la demande en marchandises des clients. Dans le

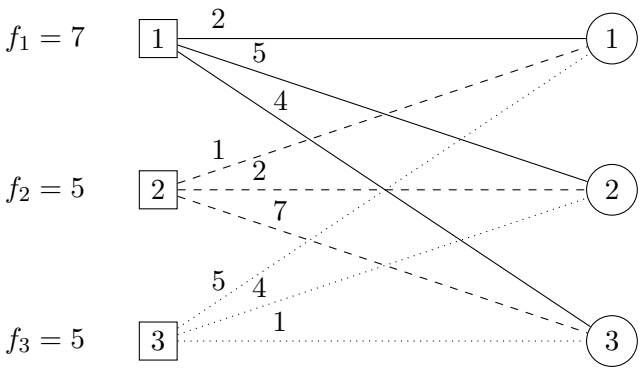


Figure 5.13

cadre de CUBE, nous travaillons uniquement avec l'*Uncapacited Facility Location Problem*. Cette variante est suffisante pour nos propos. Le lecteur intéressé peut trouver de plus amples informations sur ces variantes dans l'ouvrage de Farahani et Hekmatfar [FH09].

Comme beaucoup de problèmes d'optimisation, le FLP est NP-difficile. Comme expliqué dans le chapitre 3, trouver la solution optimale ne peut se faire en un temps raisonnable avec un algorithme exact. C'est pour cela qu'il est nécessaire d'utiliser des méthodes approchées comme les métaheuristiques pour résoudre ce genre de problèmes. Dans le cadre d'une bourse d'initiation à la recherche, Cardoen a réalisé un comparatif entre plusieurs métaheuristiques et algorithmes exacts pour le FLP [Car21]. Il résulte de ce travail que les *algorithmes génétiques* [EG09] ont un bon ratio entre la qualité des solutions retournées et le temps de calculs. Pour résoudre une instance du FLP, nous faisons donc appel à l'implémentation de ces algorithmes utilisée par Cardoen.

5.4.2 Algorithmes génétiques

Les algorithmes génétiques portent ce nom car leur fonctionnement est inspiré de la génétique dans le sens biologique du terme. Ces algorithmes travaillent sur une population de solutions. Ces solutions sont amenées à se croiser afin de générer de nouvelles solutions.

La première étape d'un algorithme génétique consiste à générer une population d'individus. Dans l'implémentation de CUBE, n suites binaires sont générées aléatoirement.

Remarque 5.4.4. Dans la suite, nous appelons «gène» un élément de la suite binaire modélisant un individu.

La deuxième étape est l'étape de croisement. Les individus de la génération courante forment aléatoirement des paires. Dans CUBE, chaque paire génère deux nouveaux individus via un *One Point Crossover*. Un *One Point Crossover* consiste à couper les deux suites binaires à un indice i choisi aléatoirement. Ensuite, les deux moitiés sont interchangées afin de créer deux nouveaux individus (voir Figure 5.14).

Les deux nouveaux individus ont ensuite une faible probabilité de subir une mutation. Ici, une mutation se traduit par changer aléatoirement une valeur de la suite. Un 0 devient un 1 et un 1 devient un 0. La présence de mutations permet d'avoir une plus grande diversité de solutions. Une plus grande diversité d'individus augmente les chances d'obtenir une solution de qualité.

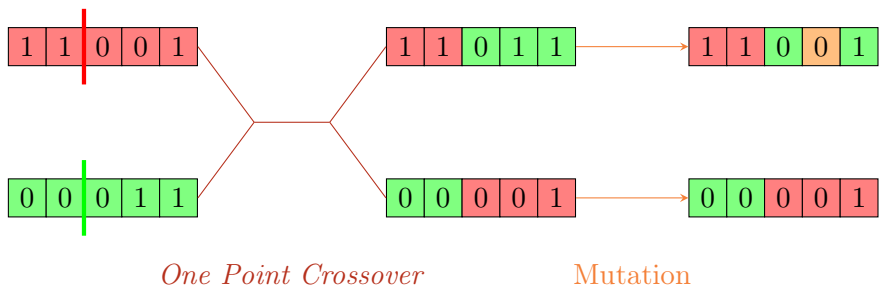


Figure 5.14

Remarque 5.4.5. La probabilité d'avoir une mutation est souvent comprise entre $0,1\%$ et 1%. Avoir une probabilité de mutation trop grande reviendrait à faire une simple recherche aléatoire. Nous perdrions donc l'intérêt des algorithmes génétiques.

La dernière étape est l'étape de sélection. Afin d'éviter une population qui grandit exponentiellement, seuls n individus sont conservés pour passer à la génération suivante. Pour réaliser la sélection, CUBE utilise «la technique de la roulette». La probabilité qu'un individu soit sélectionné est proportionnelle à son évaluation par la fonction objectif. Soit $\{s_1, \ldots, s_n\}$ la population courante. Soit φ, la fonction objectif du problème. La probabilité que l'individu s_i soit sélectionné est :

$$\frac{\varphi(s_i)}{\sum_{k=1}^{n} \varphi(s_k)},$$

si l'objectif est de maximiser φ. Si l'objectif est de minimiser φ, la probabilité d'être sélectionné est :

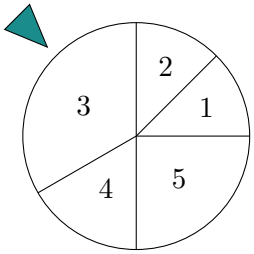
$$\frac{1}{n-1} - \frac{\varphi(s_i)}{(n-1) \cdot \sum_{k=1}^{n} \varphi(s_k)}.$$

La Figure 5.15 illustre une distrubution basée sur «la technique de la roulette». Une fois un individu sélectionné, une nouvelle roulette est créée avec les individus restants.

Lors de la phase de sélection, CUBE utilise une petite optimisation proposée par Kratica *et al.* [KFST96]. La meilleure solution de chaque génération passe directement à la génération suivante. Ainsi, un individu de qualité peut transmettre ses gènes à la génération suivante.

Après cette phase de sélection, une nouvelle phase de croisement a lieu avec les individus sélectionnés. L'algorithme s'arrête après avoir généré m générations d'individus. La meilleure solution rencontrée parmi toutes les générations est retournée.

Remarque 5.4.6. Dans la littérature, la phase de sélection a parfois lieu avant la phase de croisement. Seule la moitié des individus d'une génération peut se croiser. Nous avons testé les deux approches sur 1000 instances du FLP partant de la même population initiale. Effectuer la phase de croisement avant la phase de sélection retourne une meilleure solution dans 760 cas. Effectuer la phase de sélection retourne une meilleure solution dans 169 cas. Enfin, les deux approches retournent une solution de même qualité dans 71 cas.



Solution :	1	2	3	4	5
Évaluation :	45	45	120	60	90
Probabilité :	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{4}$

Figure 5.15 – Exemple de distribution en «roulette». Ici, plus l'évaluation d'une solution est élevée, plus la probabilité d'être sélectionné est élevée.

Réaliser la phase de croisement en premier amène à réaliser une sélection sur une population deux fois plus grande. Avoir une population plus grande permet d'avoir une plus grande diversité d'individus. Donc, un individu de grande qualité a plus de chance d'émerger.

Au vu de ces résultats, CUBE réalise la phase de croisement en premier.

5.4.3 Application et résultats

Dans notre implémentation du FLP, nous ne partitionnons pas les nœuds du graphe en nœuds clients et nœuds pouvant être des entrepôts. Ici, un nœud qui n'est pas un entrepôt est un client et inversement.

La Figure 5.16 illustre une solution retournée par CUBE pour une instance du FLP via un algorithme génétique. L'algorithme a travaillé avec des populations de 200 individus sur 2000 générations. La probabilité qu'un individu mute était de 1%. Ouvrir un entrepôt a un coût de 15. Les coûts de transport sont calculés comme étant la longueur d'un plus court chemin dans le graphe. Le poids de chaque arête du graphe est de 1.

L'utilisateur peut également spécifier des contraintes que CUBE doit considérer. Premièrement, l'utilisateur peut forcer des nœuds à être des clients ou des entrepôts. Les indices associés à ces nœuds dans les suites binaires modélisant une instance du FLP vont donc avoir une valeur fixe. Comme tous les individus d'une population ont le même gène, ce gène est toujours transmis aux nouveaux individus. Il faut juste vérifier que ces gènes fixés ne mutent pas.

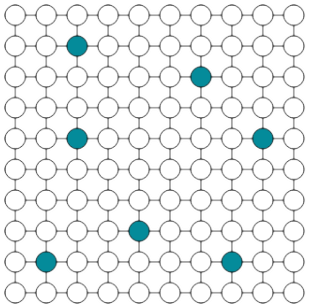


Figure 5.16 – En bleu canard, les entrepôts ouverts.

Deuxièmement, l'utilisateur peut spécifier une distance maximale à laquelle doit se trouver un entrepôt fournissant un client. Donner une pénalité plus grande que le coût d'ouverture d'un entrepôt encourage CUBE a ouvrir plus d'entrepôts. Cette contrainte permet de prendre en compte le critère «chaque habitant doit avoir accès à une école à au plus 600 m de chez lui» identifié dans les travaux de De Smet [De 18].

La Figure 5.17 illustre une solution retournée par CUBE avec les mêmes configurations que pour la Figure 5.16 mais avec des contraintes. Le nœud en haut à gauche est un client obligatoirement. Le nœud en bas à droite est un entrepôt obligatoirement. Une pénalité de 30 est donnée à une solution par client n'ayant pas un entrepôt à une distance d'au plus 2.

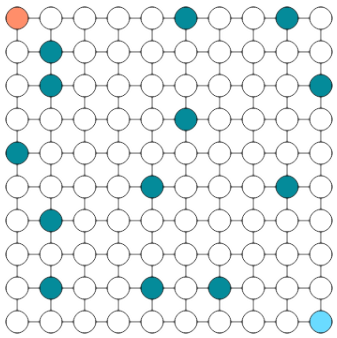


Figure 5.17 – En bleu canard, les entrepôts ouverts.

5.5 Évaluation d'affectation et *Knapsack Problem*

Le problème exposé dans cette section est basé sur un fait réel. L'École du Futur, située dans la ville de Mons en Belgique, a pour projet de déménager. De ce fait est né la question suivante : «Ce déménagement est-il une bonne idée ?»

Pour donner des pistes de réponses à ce genre de question, CUBE fournit des outils pour évaluer l'affectation existante des écoles dans une ville. Avec ces outils, il est possible de comparer l'évaluation de l'affectation avant et après un déménagement. Afin de réaliser ces évaluations, CUBE simule une assignation des élèves dans les différentes écoles. Ensuite, CUBE mesure le degré de satisfaction des parents à mettre leurs enfants dans telle ou telle école. Pour réaliser cela, nous utilisons des variantes du *Knapsack Problem*.

5.5.1 Le *Knapsack Problem*

Le *Knapsack Problem* (problème du sac à dos en français) peut être intuitivement expliqué via la situation suivante. Un cambrioleur veut dévaliser une salle contenant n trésors, tous numérotés de 1 à n. Le problème du cambrioleur est que son sac est trop petit pour contenir tous les trésors, ce dernier ne pouvant contenir que c kg. Sachant que chaque trésor i pèse w_i kg et a une

valeur de p_i €, l'objectif du cambrioleur est de remplir son sac le plus possible de façon à emporter les objets ayant la plus grande valeur totale.

Définition 5.5.1 (*Knapsack Problem* [KPP04])**.** Soit $N = \{0, 1, \ldots, n\}$, un ensemble de n objets ayant une capacité maximale c. Pour tout objet $i \in N$, $p_i \in \mathbb{R}$ symbolise la valeur de l'objet i et $w_i \in \mathbb{R}$ le poids de l'objet. L'objectif d'une instance du *Knapsack Problem* (KP) est de maximiser :

$$\sum_{i=1}^{n} p_i x_i,$$

sous la contrainte que

$$\sum_{i=1}^{n} w_i x_i \leqslant c,$$

où pour tout $i \in N$,

$$x_i = \begin{cases} 1 & \text{si l'objet } i \text{ est dans le sac,} \\ 0 & \text{sinon.} \end{cases}$$

Exemple 5.5.2. Considérons une instance du problème du sac à dos avec une capacité maximale de 9 et 7 objets avec les poids et valeurs suivants :

i	1	2	3	4	5	6	7
p_i	6	5	8	9	6	7	3
w_i	2	3	6	7	5	9	4

Une solution possible de cette instance est de choisir les objets 2 et 5. En effet, leur poids total vaut 8, nous pouvons donc mettre ces deux objets dans notre sac et nous avons une solution d'une valeur de 11. Une autre solution remplissant cette fois totalement le sac est de choisir les objets 1, 2 et 7 pour une valeur de 14. La solution optimale de cette instance, ayant une valeur de 15, est de choisir les objets 1 et 4.

Notations. À partir de maintenant, nous notons $KP(n, c)$ pour une instance du problème du sac à dos de capacité maximale c ayant n objets. Nous notons également $X^*(n, c)$ une solution optimale d'une telle instance. Enfin, nous notons $z(n, c)$ la valeur de la solution optimale.

Le KP est également un problème NP-difficile. Afin de résoudre une instance de ce problème, CUBE utilise le premier algorithme utilisant les concepts de programmation dynamique présenté dans l'ouvrage de Kellerer, Pferschy et Pisinger [KPP04]. La programmation dynamique consiste à résoudre un problème en le décomposant en sous-problèmes plus simples à résoudre. Un tel algorithme résout les sous-problèmes progressivement et garde en mémoire les résultats pour chacun d'entre eux. Les solutions mémorisées sont utilisées pour construire la solution d'un sous-problème plus complexe.

Dans le cadre du problème du sac à dos, la programmation dynamique repose sur le fait que nous pouvons construire la solution optimale d'une instance à n objets à partir d'une instance à $n-1$ objets. En effet, soit $KP(n, c)$ une instance du problème du sac à dos. Si $n \notin X^*(n, c)$, alors $X^*(n, c) = X^*(n-1, c)$. Et si $n \in X^*(n, c)$, alors $X^*(n-1, c-w_n) = X^*(n, c) \backslash \{n\}$. Cela signifie que si l'objet n n'est pas dans la solution optimale, alors la solution optimale est la même que celle de l'instance sans cet objet. Et si n est dans la solution optimale et si on le retire de cette solution, alors nous obtenons la solution optimale de l'instance à $n-1$ objets dont la capacité maximale a été diminuée de w_n. En partant de cette observation, nous pouvons donc calculer la valeur de la solution optimale comme suit :

$$z(n, c) = \begin{cases} z(n-1, c) & \text{si } w_n > c, \\ \max\{z(n-1, c), z(n-1, c-w_n) + p_n\} & \text{si } w_n \leqslant c. \end{cases}$$

Ainsi, l'Algorithme 9 garde en mémoire pour tout objet i les valeurs $z(i, d)$ où d est la capacité maximale du sac. Cette capacité d est augmentée progressivement jusqu'à valoir c. L'algorithme peut donc calculer la valeur de la solution optimale $z(n, c)$.

Exemple 5.5.3. Reconsidérons l'instance du problème du sac à dos de l'exemple 5.5.2. La Table 5.1 contient toutes les valeurs $z(i, d)$ calculées par l'algorithme. Sa construction se déroule comme suit : tout d'abord nous considérons que nous n'avons accès uniquement à l'objet 1. Nous ne pouvons le mettre dans le sac que si la capacité maximale de ce dernier est supérieure à 2.
Ensuite nous avons accès à l'objet 2. Si la capacité du sac vaut 3 ou 4, nous

Algorithme 9 : Programmation Dynamique pour KP

Entrées : Une instance du problème du sac à dos $KP(n, c)$ où n est le nombre total d'objets, c la capacité maximale du sac. De plus, tout objet i a un poids w_i et une valeur p_i.

Sortie : La valeur de la solution optimale pour cette instance.

```
pour d allant de 0 à c faire
    z(0, d) ← 0
pour i allant de 1 à n faire
    pour d allant de 0 à w_i − 1 faire
        z(i, d) ← z(i − 1, d)
    pour d allant de w_i à c faire
        si z(i − 1, c − w_i) + p_i > z(i − 1, d) alors
            z(i, d) ← z(i − 1, c − w_i) + p_i
        sinon
            z(i, d) ← z(i − 1, d)
retourner z(n, c)
```

pouvons uniquement mettre l'objet 2 dans le sac mais sa valeur est inférieure à celle de l'objet 1. Dès lors, pour l'instant, la solution optimale ne contient que 1. Une fois la capacité du sac dépassant 5, les deux objets peuvent être mis dans le sac.

Maintenant considérons l'objet 3 de poids 6 et de valeur 8. Si la capacité du sac vaut 6 ou 7, nous pouvons ajouter l'objet 3 à $X^*(2, 0)$ et $X^*(2, 1)$. Dans les deux cas, la solution $\{3\}$ de valeur 8 est construite. Cette solution est de moins bonne qualité que $X^*(2, 6)$ et $X^*(2, 7)$. Nous avons donc $z(3, 6) = z(3, 7) = z(2, 6) = z(2, 7) = 11$.

Ensuite, l'algorithme construit des solutions pour $KP(3, 8)$ et $KP(3, 9)$ en ajoutant l'objet 3 à $X^*(2, 2)$ et $X^*(2, 3)$. La solution $\{1, 3\}$ de valeur 14 est créée. Cette solution est meilleure que $X^*(2, 8)$ et $X^*(2, 9)$. Nous avons donc $z(3, 8) = z(3, 9) = 14$.

L'algorithme continue ses calculs de manière analogue jusqu'à obtenir $z(7, 9)$.

Remarque 5.5.4. L'objet 0 de la Table 5.1 est un objet fictif de poids et de valeur nuls. L'algorithme l'utilise pour démarrer ses calculs.

Table 5.1 – Tableau créé en mémoire par l'algorithme lors de son exécution sur l'instance de l'exemple 5.5.2.

d \ i	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
2	0	6	6	6	6	6	6	6
3	0	6	6	6	6	6	6	6
4	0	6	6	6	6	6	6	6
5	0	6	11	11	11	11	11	11
6	0	6	11	11	11	11	11	11
7	0	6	11	11	11	12	12	12
8	0	6	11	14	14	14	14	14
9	0	6	11	14	15	15	15	15

Remarque 5.5.5. De base, l'algorithme ne calcule pas la solution optimale mais uniquement sa valeur. Cependant, en utilisant le tableau créé en mémoire, il est possible de facilement construire cette solution. En effet, au vu de la construction du tableau, l'objet i est dans la solution optimale de l'instance $KP(i,d)$ si $z(i,d) = z(i-1, d-w_i)+p_i$. Dès lors, la construction de la solution optimale pour $KP(n,c)$ se fait de la manière suivante. Premièrement, nous vérifions si n est dans cette solution via le critère précédemment cité. Si oui, nous testons ensuite si $n-1$ est dans la solution optimale de $KP(n-1, c-w_n)$ et ainsi de suite. Sinon, nous vérifions si $n-1$ est dans la solution optimale de $KP(n-1, c)$. Si nous gardons une trace de chaque objet vérifiant le critère, une fois la vérification faite pour l'objet 1, la solution optimale est construite.

5.5.2 Le *Generalized Assignment Problem*

Le *Generalized Assignment Problem* (GAP) est un autre problème d'optimisation très semblable au problème du sac à dos. La principale différence entre ces deux problèmes est que pour le GAP nous disposons de plusieurs sacs pour ranger nos objets. Les sacs peuvent avoir des capacités différentes et la

valeur et le poids des objets peuvent varier en fonction du sac.

Un exemple pour comprendre intuitivement ce problème est le suivant : un patron cherche à répartir plusieurs tâches (les objets) entre ses employés (les sacs). Cependant, lors de la répartition, il doit tenir compte de la motivation de chacun de ses employés à réaliser telle ou telle tâche influant sur le temps que va mettre un employé à effectuer son travail (le poids des objets) et des domaines d'expertises de ses employés influant sur la qualité du travail rendu (la valeur des objets). L'objectif du patron est d'avoir à la fin de la journée le plus de tâches bien effectuées tout en tenant compte des horaires des employés (les capacités maximales des sacs).

Définition 5.5.6 (*Generalized Assignment Problem*)**.** Soient n objets et m sacs. Soit le vecteur $c = (c_1, \ldots, c_m)$ tel que, pour tout j, c_j soit la capacité maximale du sac j. Soient les matrices $W = (w_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant m}}$ et $P = (p_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant m}}$ telles que pour i et pour tout j, w_{ij} et p_{ij} sont respectivement le poids et la valeur qu'a l'objet i quand il est mis dans le sac j. L'objectif d'une instance du GAP est de maximiser :

$$\sum_{j=1}^{m} \sum_{i=1}^{n} p_{ij} x_{ij},$$

sous les contraintes que

$$\text{pour tout } 1 \leqslant j \leqslant m, \ \sum_{i=1}^{n} w_{ij} x_{ij} \leqslant c_j,$$

$$\sum_{j=1}^{m} x_{ij} \leqslant 1,$$

où pour tout $1 \leqslant i \leqslant n$ et pour tout $1 \leqslant j \leqslant m$,

$$x_{ij} = \begin{cases} 1 & \text{si l'objet } i \text{ est dans le sac } j, \\ 0 & \text{sinon.} \end{cases}$$

Exemple 5.5.7. Considérons l'instance du GAP à 4 objets et 3 sacs suivante :

$$W = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 3 & 3 \\ 2 & 3 & 4 \\ 1 & 2 & 3 \end{pmatrix} \qquad P = \begin{pmatrix} 3 & 1 & 5 \\ 1 & 1 & 1 \\ 5 & 15 & 25 \\ 25 & 15 & 5 \end{pmatrix} \qquad c = (2, 3, 4)$$

Pour cette instance, la solution optimale est $\big\{\{1, 4\}, \{2\}, \{3\}\big\}$.

Tout comme pour le KP, nous ne disposons pas d'algorithme permettant de calculer la solution optimale d'une instance du GAP en un temps raisonnable sans utiliser de mémoire. Il faut donc faire des concessions. Pour résoudre une instance du GAP, CUBE utilise un *algorithme d'approximation.*

Définition 5.5.8 (Algorithme d'approximation [WS11])**.** Un *algorithme d'α-approximation* pour un problème d'optimisation est un algorithme en temps polynomial qui retourne une solution dont la valeur vaut celle de la solution optimale à un facteur α près.

Comme, dans le cas du GAP, nous cherchons à maximiser un critère, nous supposons que α est compris entre 0 et 1. Ainsi, la solution retournée par un algorithme d'$\frac{1}{2}$-approximation vaut au moins la moitié de la solution optimale.

L'algorithme que nous utilisons (Algorithme 10) est le résultat des travaux de Cohen, Katzir et Raz [CKR06]. Nous notons $P = (p_{ik})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant k \leqslant m}}$ la matrice de valeurs. Cet algorithme fonctionne comme suit :

- appliquer un algorithme de résolution du problème du sac à dos sur le premier sac ;
- créer une matrice $Q = (q_{ik})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant k \leqslant m}}$ tel que pour tout $1 \leqslant i \leqslant n$, pour tout $1 \leqslant k \leqslant m$, $q_{ik} = p_{ik}$ si l'objet i a été mis dans le sac 1 ou si $k = 1$. Sinon, $q_{ik} = p_{ik}$;
- créer la matrice $R = (r_{ik})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant k \leqslant m}}$ tel que $R = P - Q$. Dès lors, pour tout $1 \leqslant i \leqslant n$, pour tout $1 \leqslant k \leqslant m$, si $r_{ik} > 0$, alors cela signifie que l'objet i à une plus grande valeur dans le sac k que dans le sac 1 ou qu'il n'a pas encore été sélectionné. En effet, la matrice Q contient les valeurs des

objets mis dans le sac 1. Donc si $r_{ik} > 0$, cela signifie que $p_{ik} > q_{ik}$. D'où l'objet i à une plus grande valeur dans le sac k.

- appliquer les étapes précédentes en utilisant la matrice R à place de P sur le deuxième sac. Une nouvelle matrice R va être crée, utilisée sur le troisième sac et ainsi de suite ;
- une fois cela fait pour chaque sac, combiner chaque solution en enlevant les doublons.

Lors de la dernière étape, lorsqu'un objet se trouve dans deux sacs différents, l'algorithme laisse l'objet dans le sac de plus grand indice. En effet, les modifications de la matrice des valeurs sont faites de sorte que si un objet est mis à nouveau dans un sac de plus grand indice, cela signifie que sa valeur dans ce sac est plus grande que celle pour les précédents sacs.

Exemple 5.5.9. Exécutons l'Algorithme 10 sur l'instance de l'exemple 5.5.7. La Table 5.2 reprend les valeurs des différentes variables utilisées par l'algorithme lors de son exécution. La première étape consiste à résoudre le problème du sac à dos en ne considérant que le sac 1. La solution optimale est $\{1, 4\}$. Grâce à cette solution, la matrice R est construite. Pour tout $1 \leqslant i \leqslant n$ et pour tout $1 \leqslant j \leqslant m$, si $r_{ij} > 0$, cela signifie que l'objet i a une valeur plus grande dans le sac j que dans le sac 1.
La matrice R est utilisée comme matrice de valeurs pour résoudre le KP sur le sac 2. La solution optimale est $\{3\}$. Avec cette solution, R est mise à jour. Elle est ensuite utilisée comme matrice de valeurs pour résoudre le KP sur le sac 3. La solution optimale est également $\{3\}$.
Avec ces trois solutions du KP, la solution pour le GAP est construite. L'objet 3 peut être dans le sac 2 ou le 3. Vu la construction de R, 3 a une plus grande valeur dans le troisième sac. Ainsi la solution retournée est $\{\{1, 4\}, \{\}, \{3\}\}$.

Remarque 5.5.10. La matrice Q n'est utilisée que pour construire la matrice R.

Algorithme 10 : Next-Bin [CKR06]

Entrées : Une matrice de valeurs P, une matrice de poids W et un vecteur de capacité c pour un GAP avec n objets et m sacs. Un algorithme A résolvant le problème du sac à dos. Un indice j

Sortie : Une solution approximée pour l'instance du GAP donnée en entrée.

Appliquer l'algorithme A sur le sac j en utilisant la colonne j des matrices P et W. Soit S^* la solution retournée par A.
Soient les matrices $Q = (q_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant m}}$ et $R = (r_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant m}}$
pour tout $1 \leqslant i \leqslant n$ *et* $1 \leqslant k \leqslant m$ **faire**
  si $i \in S^*$ *ou* $k = j$ **alors**
    $q_{ik} \leftarrow p_{ij}$
  sinon
    $q_{ik} \leftarrow 0$
  $r_{ik} \leftarrow p_{ik} - q_{ik}$
si $j \leqslant m$ **alors**
  Exécuter Next-Bin sur le sac $j+1$ avec R comme matrice de valeurs. Soit S_{j+1} la solution retournée par cette exécution.
  $S_j \leftarrow S_{j+1} \cup S^* \backslash \bigcup_{k=j+1}^{m} S_k$
  retourner S_j
sinon
  retourner S^*

Table 5.2 – Variables utilisées par l'Algorithme 10 pour résoudre l'instance de l'exemple 5.5.7.

j	P	S^*	Q	R
1	$\begin{pmatrix} 3 & 1 & 5 \\ 1 & 1 & 1 \\ 5 & 15 & 25 \\ 25 & 15 & 5 \end{pmatrix}$	$\{1, 4\}$	$\begin{pmatrix} 3 & 3 & 3 \\ 1 & 0 & 0 \\ 5 & 0 & 0 \\ 25 & 25 & 25 \end{pmatrix}$	$\begin{pmatrix} 0 & -2 & 2 \\ 0 & 1 & 1 \\ 0 & 15 & 25 \\ 0 & -10 & -20 \end{pmatrix}$

Suite à la page suivante

Table 5.2 – suite

j	P	S^*	Q	R
2	$\begin{pmatrix} -2 & 2 \\ 1 & 1 \\ 15 & 25 \\ -10 & -20 \end{pmatrix}$	$\{3\}$	$\begin{pmatrix} -2 & 0 \\ 1 & 0 \\ 15 & 25 \\ -10 & 0 \end{pmatrix}$	$\begin{pmatrix} 0 & 2 \\ 0 & 1 \\ 0 & 10 \\ 0 & -20 \end{pmatrix}$
3	$\begin{pmatrix} 2 \\ 1 \\ 10 \\ -20 \end{pmatrix}$	$\{3\}$	$\begin{pmatrix} 2 \\ 1 \\ 10 \\ -20 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

Supposons que Next-Bin (Algorithme 10) utilise un algorithme d'α-approximation pour résoudre les instances de KP. Cohen *et al.* prouvent dans leur article [CKR06] que cet algorithme est un algorithme d'$\frac{\alpha}{1+\alpha}$-approximation. Dans son implémentation, CUBE utilise un algorithme qui, grâce à de la mémoire, retourne la solution optimale d'une instance de KP. Donc, l'implémentation de Next-Bin de CUBE est un algorithme d'$\frac{1}{2}$-approximation.

5.5.3 Application et résultats

Afin de simuler l'assignation des élèves aux différentes écoles, nous définissons le problème d'optimisation suivant :

Définition 5.5.11 (Problème d'Assignation des Écoles)**.** Soient n foyers et m écoles. Soit le vecteur $c = (c_1, \ldots, c_m)$ tel que pour tout j, c_j soit le nombre maximum d'élèves que l'école j peut accueillir. Soit le vecteur $(w_1, \ldots, w_n)$ où pour tout $1 \leqslant i \leqslant n$, w_i est le nombre d'enfants à scolariser dans le foyer i. Soit $P = (p_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant m}}$ tel que pour i et pour tout j, p_{ij} est la satisfaction du foyer i de mettre leurs enfants à l'école j. L'objectif d'une instance de ce problème est de maximiser :

$$\sum_{j=1}^{m} \sum_{i=1}^{n} p_{ij} x_{ij},$$

sous les contraintes que pour tout $1 \leqslant j \leqslant m$

$$\text{pour tout } 1 \leqslant j \leqslant m,\ \sum_{i=1}^{n} w_i x_{ij} \leqslant c_j,$$

$$\sum_{j=1}^{m} x_{ij} \leqslant 1,$$

où pour tout $1 \leqslant i \leqslant n$ et pour tout $1 \leqslant j \leqslant m$

$$x_{ij} = \begin{cases} 1 & \text{si le foyer } i \text{ met ses enfants à l'école } j, \\ 0 & \text{sinon.} \end{cases}$$

Le Problème d'Assignation des Écoles [14] est un cas particulier du GAP où le poids des objets ne varie pas d'un sac à un autre. Il est donc possible d'utiliser un algorithme résolvant le GAP pour notre problème.

Comme dit en introduction de cette section, nous cherchons à évaluer le bienfait pour les parents d'élèves du déménagement d'une école de la ville de Mons. La première étape est de créer un graphe qui va modéliser le réseau routier de cette ville. En effet, une fois le graphe connu, nous pouvons déterminer le nombre de foyers, le nombre d'écoles ainsi que les distances entre ces différents bâtiments.

Pour créer un tel graphe, nous utilisons les cartes extraites d'OpenStreetMap [Ope17], alternative libre d'accès à Google Maps. Nous manipulons ces cartes à l'aide du logiciel QGIS [QGI09]. Après diverses manipulations, nous obtenons un graphe dont un fragment (le graphe en entier ayant 10394 nœuds et 22088 arcs) est présenté à la Figure 5.18. Le nœud bleu est une école, les rouges des autres bâtiments qui serviront de foyers et les gris servent à faire les jonctions entre les bâtiments et le réseau routier et les routes entre elles.

Afin de construire la matrice de satisfaction des parents P, CUBE peut prendre en compte 3 critères. Le premier est basé sur la distance (exprimée

14. La complexité algorithmique du Problème d'Assignation des Écoles est étudiée dans l'annexe A

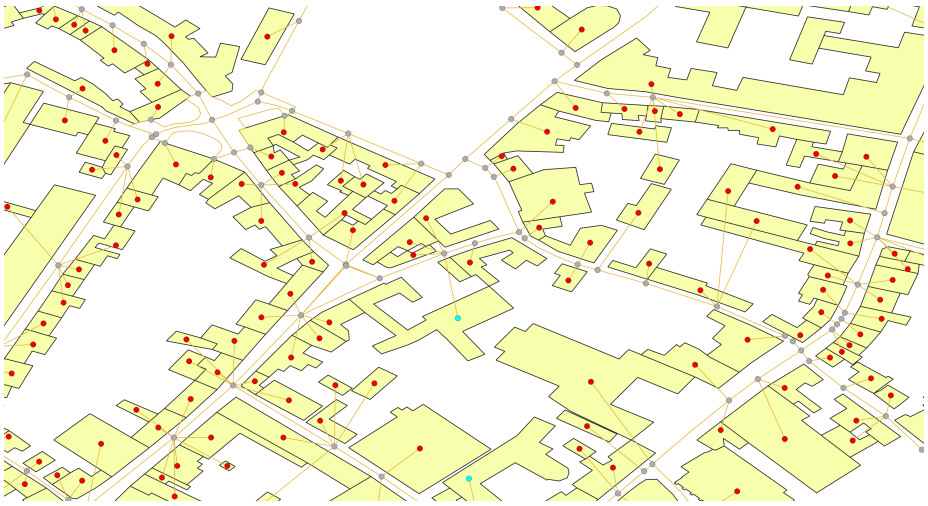


Figure 5.18 – Un fragment du graphe de Mons.

en mètres) entre les foyers et les écoles. Ces distances sont calculées comme les longueurs des plus courts chemins dans le graphe modélisant la ville. Si pour tout foyer i et toute école j, $d(i,j)$ est la distance entre i et j, alors la satisfaction des parents vaut :

- 5 si $0 \leqslant d(i,j) < 500$;
- 4 si $500 \leqslant d(i,j) < 1000$;
- 3 si $1000 \leqslant d(i,j) < 1500$;
- 2 si $1500 \leqslant d(i,j) < 2000$;
- 1 si $2000 \leqslant d(i,j)$.

Le fait de diminuer la valeur de p_{ij} tous les 500 mètres jusqu'à 2000 mètres est totalement arbitraire. Nous pourrions totalement travailler avec une autre valeur maximale ou une autre subdivision.

Le deuxième critère est basé sur le temps de trajet entre les foyers et les écoles. Nous avons utilisé l'outil TomTom Traffic Stats - Area Analysis [Tom20]. Cet outil nous a permis d'extraire des données sur l'occupation du réseau routier de la ville de Mons en avril 2019. Grâce à ces données, nous pouvons

estimer le temps de trajet entre deux nœuds (exprimé en secondes) en fonction de l'heure de la journée, via l'algortihme DOT (Algorithme 8). Si pour tout foyer i et toute école j, $t(i,j)$ est le temps de trajet nécessaire pour aller de i à j, la satisfaction des parents vaut :

- 4 si $0 \leqslant t(i,j) < 600$;
- 3 si $600 \leqslant t(i,j) < 1200$;
- 2 si $1200 \leqslant t(i,j) < 1800$;
- 1 si $1800 \leqslant t(i,j)$.

Encore une fois, une autre répartition des valeurs peut être définie par l'utilisateur.

Enfin, le dernier critère est une préférence aléatoire pour telle ou telle école. Cela permet de modéliser les préférences arbitraires des parents pour une école en particulier.

Pour rappel, l'objectif de base de cette modélisation est d'évaluer si déménager l'École du Futur (bâtiment bleu dans la Figure 5.19) dans le zoning des Grands Prés (zone aux alentours du bâtiment rouge dans la Figure 5.19) est avantageux pour les parents d'élèves. Pour réaliser cela, CUBE résout plusieurs instances du problème d'assignation des écoles. Grâce aux évaluations des solutions retournées, nous pouvons comparer la qualité des différentes affectations.

La Figure 5.20 reprend le comparatif des valeurs obtenues pour trois types d'instances du problème d'assignation des écoles. Le premier type répartit les élèves dans les différentes écoles avec l'École du Futur en centre-ville. Le deuxième considère le déménagement mais ne change pas la répartition des élèves. Le dernier type répartit tous les élèves en considérant le nouvel emplacement de l'École du Futur.

La matrice de satisfaction des parents est construite de telle sorte à prendre en compte la distance entre les foyers et les écoles, le temps nécessaire pour se rendre à l'école en démarrant à 8 h et une préférence arbitraire pour telle ou telle école. Le nombre d'enfants à scolariser par foyer est décidé aléatoirement.

Pour les exemples qui suivent, nous utilisons les données du recensement de 2011 [Dir14] sur le nombre d'habitants par logement et sur l'âge moyen des habitants dans l'arrondissement de Mons pour créer notre distribution afin de nous rapprocher de la vie réelle. Nous avons également fixé la capacité de chaque école à 360 élèves.

Avec cette configuration, les résultats tendent à dire que le déménagement est profitable aux parents. Ces résultats peuvent être expliqués par la présence de routes plus adaptées à une forte circulation dans le zoning des Grands Près. Le fait de ne pas devoir passer par le centre de Mons, zone fortement embouteillée en heures de pointe, peut également avoir son influence sur les résultats.

Figure 5.19

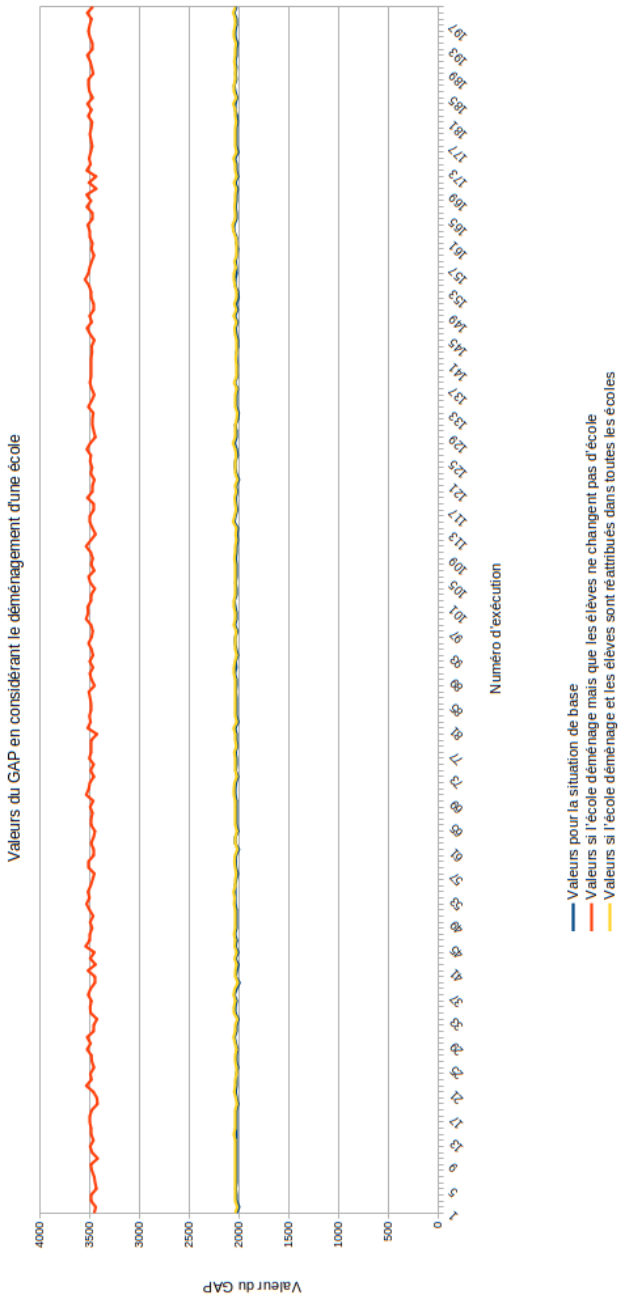

Valeurs du GAP en considérant le déménagement d'une école
Valeur du GAP
Numéro d'exécution
Valeurs pour la situation de base
Valeurs si l'école déménage mais que les élèves ne changent pas d'école
Valeurs si l'école déménage et les élèves sont réattribués dans toutes les écoles

Figure 5.20

CHAPITRE 6

MODE D'EMPLOI DE CUBE

Les modèles présentés dans les chapitres précédents ont été implémentés et regroupés dans l'outil CUBE, qui nous permet ainsi d'illustrer les modèles utilisés. Grâce aux solutions retournées par CUBE, il nous est possible de valider l'efficacité des modèles dans le cadre de problématiques liées à la compacité urbaine.

Ce chapitre est un guide d'utilisation des différents modules de CUBE. Les consignes d'installation et le code de CUBE sont consultables à l'adresse suivante : `https://github.com/QuentinMeurisse/CUBE`. Le module LS-CUBE est implémenté en Java 8 et est utilisable sur tous systèmes d'exploitation. Les autres modules de CUBE sont implémentés en C++ 17. CUBE a uniquement été testé et compilé sous système Linux. Une vidéo de démonstration de CUBE est disponible à l'adresse suivante : `https://youtu.be/uT6pCcfWI3c`

6.1 LSCUBE

Le premier outil implémenté, LSCUBE, se base sur les modèles présentés dans le chapitre 3. LSCUBE pose un ensemble de questions à l'utilisateur afin de définir le problème à résoudre ainsi que la métaheuristique à employer.

Comme illustré à la Figure 6.1, les premières questions ont pour objectif de définir le pavage. L'utilisateur doit donc définir le type de pavage (carré, triangulaire ou hexagonal), le nombre de cellules en largeur et en longueur, et enfin la longueur d'un côté d'une cellule.

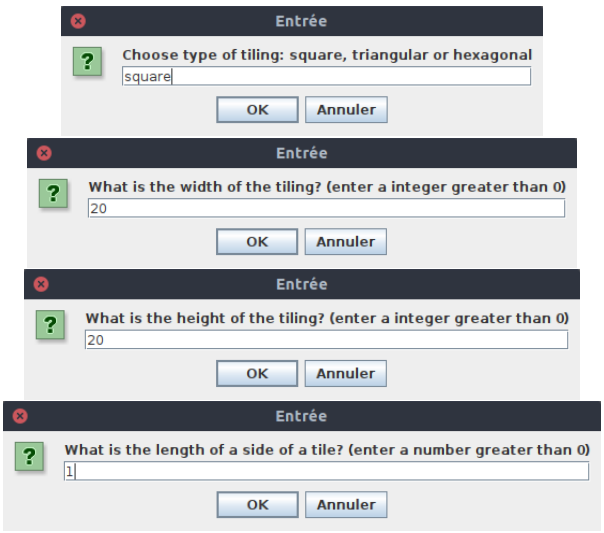


Figure 6.1 – Configuration du pavage dans LSCUBE.

L'utilisateur a ensuite la possibilité de verrouiller des cellules. Verrouiller des cellules a pour effet de créer des zones dans l'îlot non modifiables par l'algorithme. Comme illustré à la Figure 6.2, pour verrouiller une cellule, l'utilisateur doit entrer le numéro associé à cette dernière. Les cellules d'un pavage sont numérotées de gauche à droite puis de haut en bas. La cellule en haut à gauche d'un pavage porte le numéro 0.

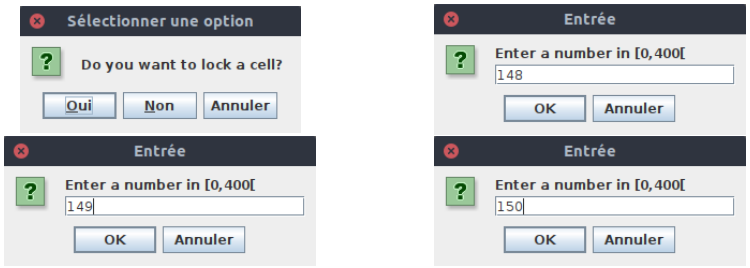


Figure 6.2 – Verouillage de cellules dans LSCUBE.

La Figure 6.3 contient les différentes questions posées à l'utilisateur pour définir les contraintes à prendre en compte lors de la recherche.

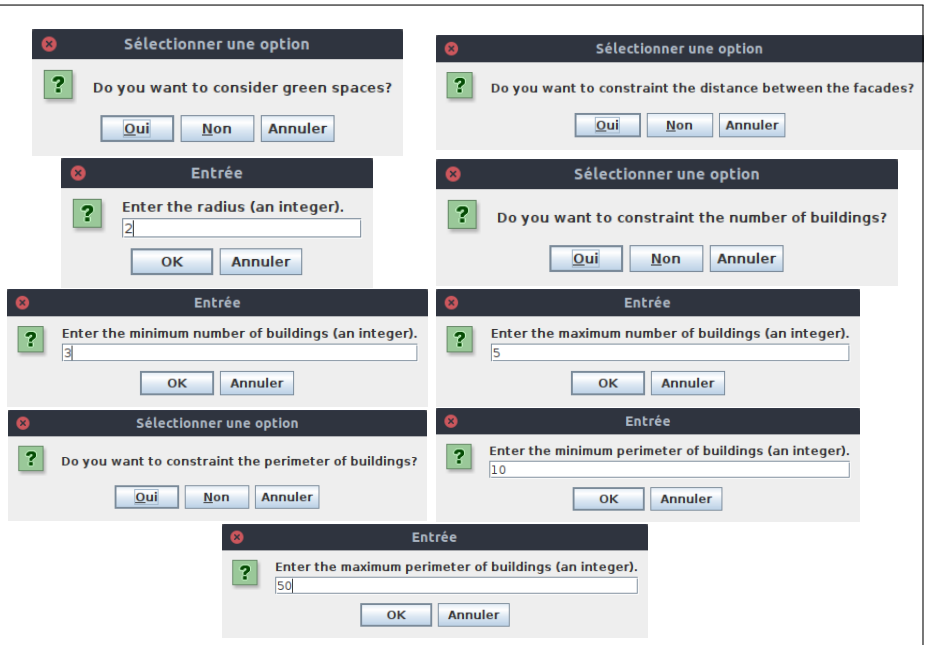


Figure 6.3 – Définition des contraintes dans LSCUBE.

Enfin, l'utilisateur est invité à choisir et configurer une métaheuristique (voir Figure 6.4). S'il choisit le *Random Descent* ou le *Parallel Tempering*, il

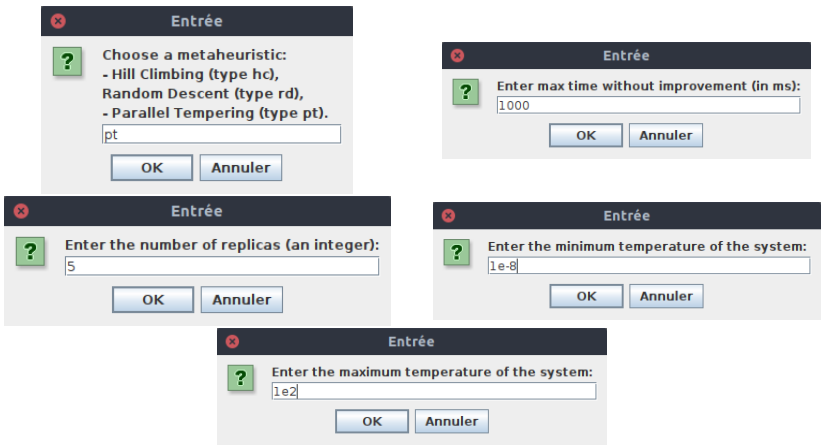


Figure 6.4 – Choix et configuration de la métaheuristique dans LSCUBE.

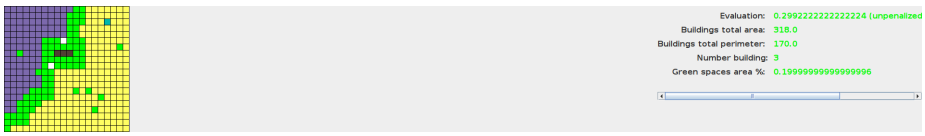


Figure 6.5 – Affiche d'une solution dans LSCUBE.

doit définir le temps maximal d'exécution de ces deux algorithmes sans que la solution soit améliorée. Si le *Parallel Tempering* est choisi, l'utilisateur doit également configurer le nombre d'exécutions que l'algorithme effectue en parallèle et les températures minimales et maximales du système.

Une fois que l'utilisateur a répondu à toutes ces questions, une fenêtre montrant la construction de l'îlot s'ouvre (voir Figure 6.5).

6.2 Module Théorie des Jeux de CUBE

Nous présentons, dans cette section, le fonctionnement du module de CUBE basé sur les modèles du chapitre 4. Ce module est dans son fonctionnement très similaire au module précédent. L'utilisateur doit fournir un ensemble de paramètres afin de définir la forme du pavage et les joueurs. Cependant, l'utilisateur a maintenant accès à une interface graphique (voir Figure 6.6).

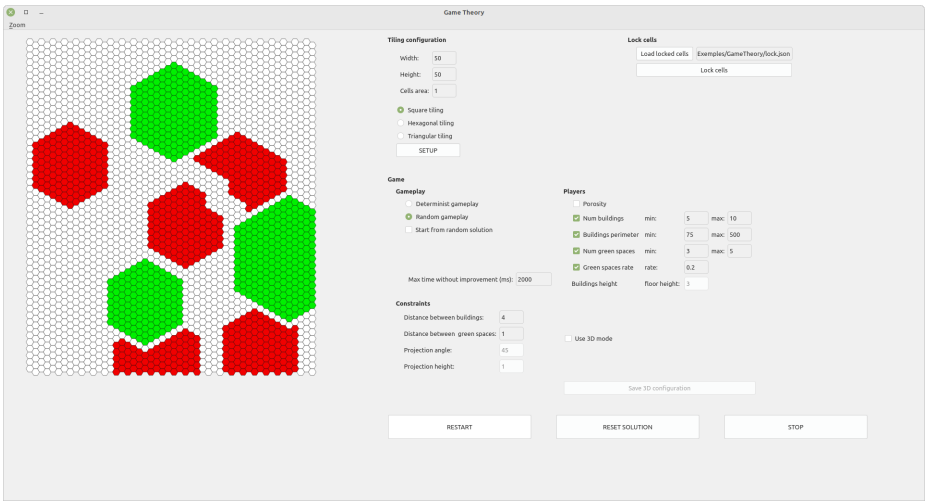


Figure 6.6 – Module Théorie des Jeux de CUBE.

La Figure 6.7 illustre les différents champs à remplir pour définir le pavage : le nombre de cellules en largeur, en longueur et l'aire d'une cellule. L'utilisateur peut également choisir le type de pavage. Grâce au bouton `SETUP`, il est possible de visualiser le pavage. L'utilisateur peut déplacer le pavage et zoomer grâce au menu zoom. L'utilisateur peut également réaliser ces actions via les raccourcis clavier suivants : `ctrl` + +, `ctrl` + −, `ctrl` + ←, `ctrl` + →, `ctrl` + ↑ et `ctrl` + ↓.

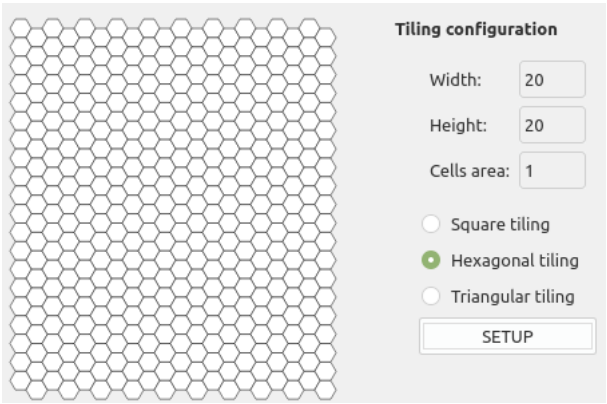


Figure 6.7 – Panneau de configuration du pavage.

L'utilisateur peut maintenant verrouiller des cellules en chargeant un fichier JSON (voir Figure 6.8). Ce fichier contient une liste de numéros de cellules identifiée par la clé `"lock"`. Le bouton `Lock cells` permet de verrouiller cette liste de cellules et de les afficher sur le pavage précédemment défini.

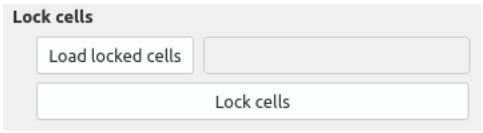


Figure 6.8 – Boutons permettant le verrouillage de certaines cellules.

Le formulaire de la Figure 6.9 permet de définir le jeu utilisé pour construire un îlot. L'utilisateur doit choisir si les joueurs sélectionnent leur stratégie de façon déterministe (la meilleure stratégie parmi toutes les stratégies possibles) ou de façon aléatoire. Il est possible aussi de spécifier si la recherche doit démarrer d'une solution aléatoire ou non. L'utilisateur peut également sélectionner les joueurs qui vont influer sur la construction de l'îlot et spécifier leurs paramètres. Enfin, l'utilisateur doit spécifier le nombre de cellules blanches entre les bâtiments et les espaces verts.

Il est également possible de prendre en compte la troisième dimension et

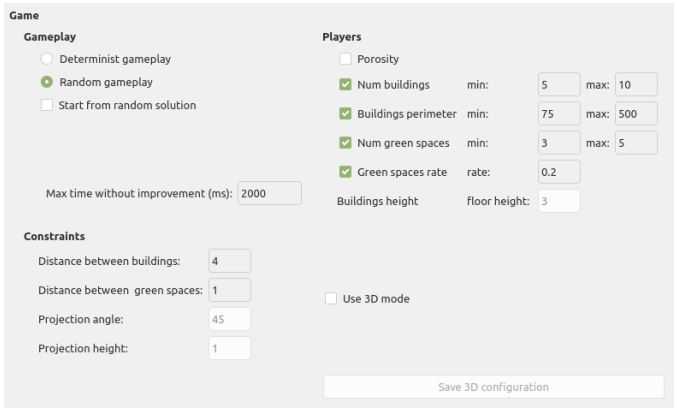


Figure 6.9 – Panneau de configuration du jeu.

la contrainte présentée dans la section 4.5 en cochant la case `Use 3D mode`. Cocher cette case donne accès à un nouveau joueur modifiant la hauteur des bâtiments. L'utilisateur doit aussi choisir l'angle et la hauteur de la projection utilisée pour la contrainte de la luminosité. Rappelons que la gestion de la troisième dimension n'est disponible que pour un pavage carré. Ainsi, il n'est plus possible de choisir un pavage hexagonal ou triangulaire. Si un pavage a déjà été configuré, il est automatiquement converti en pavage carré.

Si le mode 3D est activé, l'utilisateur peut exporter la solution obtenue par CUBE grâce au bouton `Save 3D configuration`. Grâce au script Python `generate_polyhedron.py`, il est possible d'afficher la configuration sauvegardée dans le logiciel Blender [Ble18]. Une fois le script ouvert dans blender, l'utilisateur doit spécifier le nom du fichier à la ligne 61 du script comme illustré à la figure 6.10. Il est nécessaire d'indiquer le chemin d'accès au fichier complet, encadré par des guillemets.

Figure 6.10 – Ligne où spécifier le fichier contenant la configuration d'îlot dans le script Blender.

Le bouton `START` (voir Figure 6.11) permet de démarrer la recherche une fois le pavage, les joueurs et les contraintes configurés. Une fois une solution obtenue, le bouton `RESET SOLUTION` permet d'effacer la solution pour avoir un pavage vide. Enfin, le bouton `STOP` permet de stopper une recherche en cours.

Figure 6.11 – Boutons de contrôles de la recherche d'EN.

6.3 Module Jeu sur Graphe de CUBE

Cette section est consacrée à la présentation du module de CUBE basé sur le modèle de la section 5.3. L'objectif de ce module est de déterminer les meilleures positions dans un graphe pour des commerces de proximité et des écoles en plaçant des jetons de couleur sur les nœuds du graphe. La Figure 6.12 illustre le panneau de contrôle de ce module.

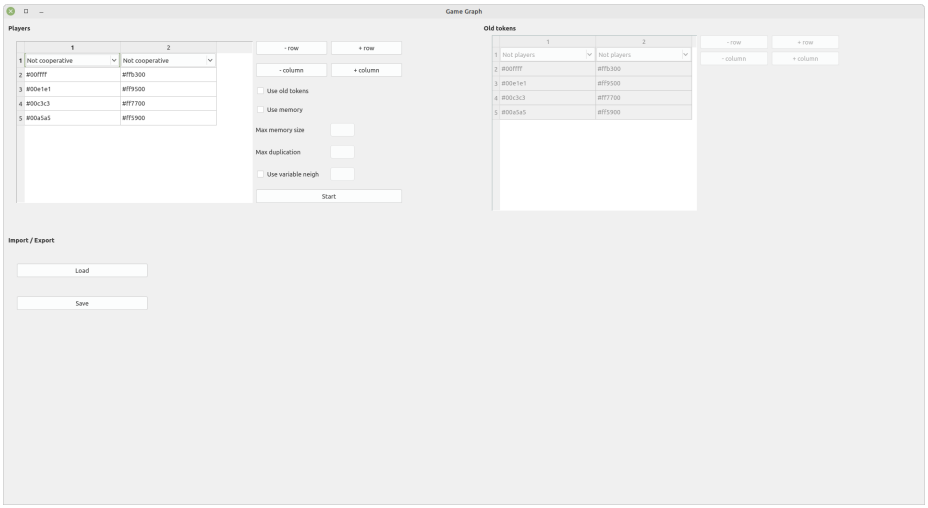


Figure 6.12 – Module Jeu sur Graphe de CUBE.

L'utilisateur peut charger un graphe via le bouton `Load` et sauvegarder la solution obtenue par CUBE via le bouton `Save`. Les graphes sont stockés dans des fichiers JSON. Un fichier contenant un graphe possède deux clés d'accès : `"nodes"` pour les nœuds du graphe et `"links"` pour les arcs. Chaque nœud possède deux attributs :

- `"id"`, un entier représentant le numéro du nœud ;
- `"data"`, une liste de trois éléments contenant les coordonnées du nœud dans le plan et la couleur du nœud sous forme d'une chaîne de caractères HTML.

Chaque arc possède trois attributs :

- `"source"`, le nœud source de l'arc,
- `"target"`, le nœud cible de l'arc ;
- `"weight"`, le poids de l'arc.

L'utilisateur peut définir les joueurs via le tableau illustré à la Figure 6.13. Chaque colonne définit un groupe de joueurs. Chaque groupe de joueurs symbolise un type de facilité. Les joueurs d'un groupe peuvent être coopératifs (comme pour les écoles) ou non (comme pour les commerces de proximité). Définir un joueur se fait en donnant la couleur, via son code HTML, du jeton associé à chaque joueur dans le jeu. L'utilisateur peut ajouter ou supprimer des joueurs en utilisant respectivement les boutons `+row` et `−row`. Il est également possible d'ajouter et supprimer des groupes de joueurs via les boutons `+column` et `−column`.

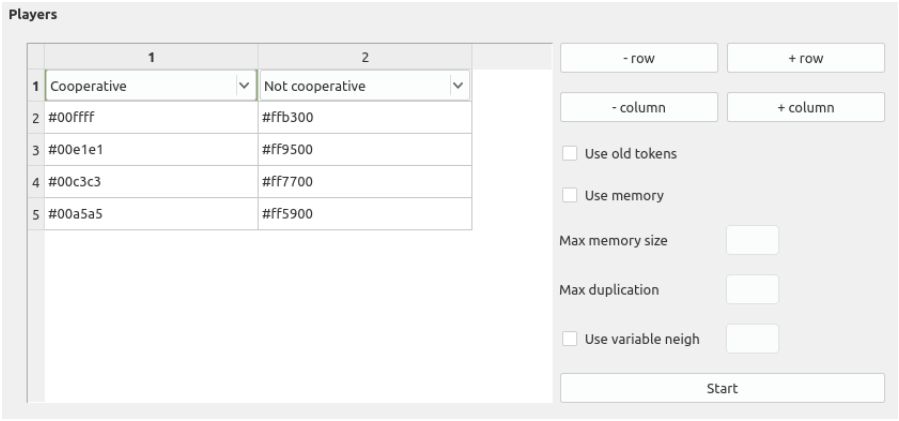


Figure 6.13 – Panneau de configuration des jetons et des joueurs.

Dans ce module, les joueurs sont implémentés de sorte à pouvoir bouger leur jeton vers un nœud de même valeur que leur nœud courant. Si deux nœuds voisins sont de même valeur, il est possible que le joueur bouge infiniment son jeton d'un de ces nœuds à l'autre. Pour éviter cela, il est possible d'attribuer à chaque joueur une mémoire finie via la case `Use memory`. La mémoire finie est implémentée via une liste ayant une taille maximum (définie via le champ

`Max memory size`). Cette mémoire contient les nœuds précédemment choisis par le joueur. Si un nœud a été choisi récemment un trop grand nombre de fois (nombre défini via le champ `Max duplication`), le joueur ne peut plus choisir ce nœud. Cela l'oblige à choisir un autre nœud ou stopper la partie.

Comme expliqué dans la section 5.3, il est parfois intéressant de permettre au joueur de choisir un nœud distant de n arcs par rapport à son nœud courant. Cocher la case `Use variable neigh` permet d'activer la recherche par voisinage variable et de définir ce paramètre n.

Si une solution a déjà été construite et que l'utilisateur veut modifier cette solution en y ajoutant des nouveaux joueurs, il est possible de prendre en compte les jetons déjà placés. En cochant la case `Use old tokens` de la Figure 6.13, l'utilisateur a accès à la table de la Figure 6.14.

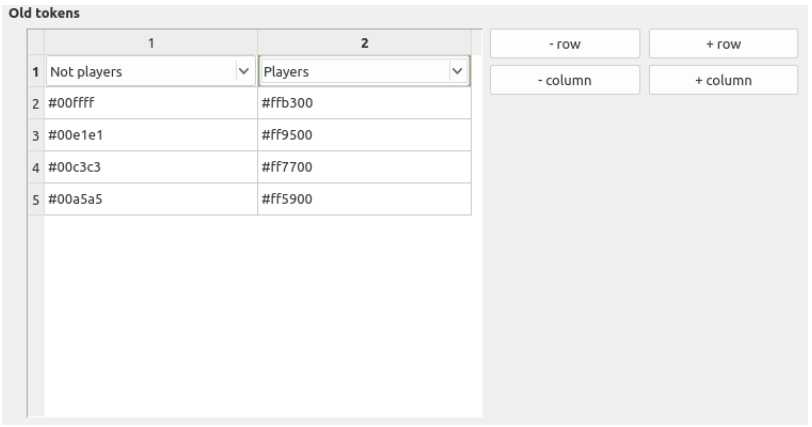


Figure 6.14 – Panneau permettant d'utiliser des jetons déjà placés sur le graphe.

La table de la Figure 6.14 contient les couleurs des jetons déjà placés, regroupés par type de facilité. Ces jetons peuvent être considérés comme des joueurs ou non. S'ils sont considérés comme des joueurs, le programme peut les déplacer. Sinon, ils restent fixes. Chaque colonne de la Figure 6.14 est associée à la même colonne de la Figure 6.13. Ainsi, un jeton de la colonne 1 est

considéré comme un jeton du premier groupe de joueurs.

Enfin, le bouton `Start` de la Figure 6.13 permet d'ouvrir une nouvelle fenêtre affichant le graphe ainsi que la position initiale des joueurs s'ils ont été définis (voir Figure 6.15).

Figure 6.15 – Fenêtre contenant le déroulement du jeu.

L'utilisateur peut zoomer ou déplacer le graphe grâce au menu `Zoom` ou via les mêmes raccourcis clavier que pour le pavage du module théorie des jeux. Le bouton `START` permet de lancer la recherche des positions optimales pour chaque jeton. Le bouton `STOP` permet d'arrêter la recherche.

Remarque 6.3.1. Un troisième type de joueurs est implémenté. Ces joueurs ont une fonction de coûts basée sur la fonction objectif du problème des k-centres. Soit un graphe $G = (V, E)$. L'objectif de ces joueurs est de trouver un ensemble de nœuds $S \subseteq V$ de taille k qui minimise :

$$\max_{s \in S} \min_{v \in V} d_G(v, u).$$

Cette fonction est très similaire à celle des joueurs coopératifs symbolisant les écoles. Ces joueurs cherchent à minimiser la somme des distances entre les foyers et l'école la plus proche. Les joueurs basés sur le k-centre cherchent à minimiser la plus grande distance entre les foyers et l'école la plus proche. Cette nuance donne des comportements différents aux joueurs. Plus de détails sur le problème des k-centres sont consultables dans l'ouvrage de Farahani et Hekmatfar [FH09].

6.4 Module FLP de CUBE

Cette section présente le module résolvant des instances du FLP grâce à des algorithmes génétiques, comme expliqué dans la section 5.4. La Figure 6.16 illustre la fenêtre principale de ce module. Dans cette fenêtre, l'utilisateur a accès au nombre d'entrepôts/écoles ouvert(e)s et à l'évaluation de la solution affichée.

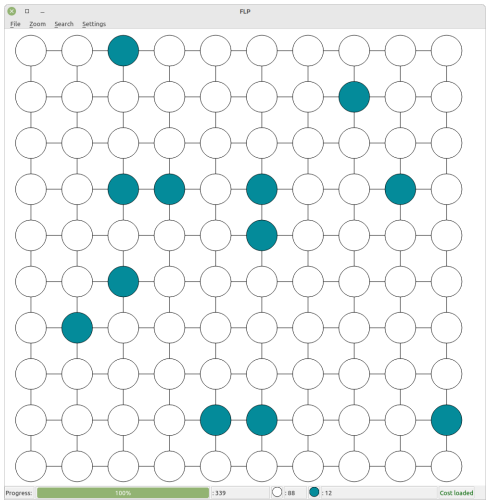


Figure 6.16 – Module FLP de CUBE.

L'utilisateur peut charger et sauvegarder des graphes via des fichiers JSON (grâce au menu `File`). Ces fichiers suivent les mêmes conventions que pour le module précédent. Il est également possible de charger et de sauvegarder les coûts d'ouverture des entrepôts et les coûts de transport via des fichiers JSON. Un tel fichier possède deux clés :

- `"facilities"` qui renvoie vers une liste contenant les coûts de construction sur chaque nœud du graphe ;
- `"transport"` qui renvoie vers une matrice contenant les coûts de transport entre chaque paire de nœuds du graphe.

En sélectionnant l'option `Settings` du menu `Search`, l'utilisateur a accès à la fenêtre de la Figure 6.17. Cette fenêtre permet de configurer l'algorithme génétique en spécifiant le nombre de générations, la taille de la population et la probabilité qu'un individu subisse une mutation.

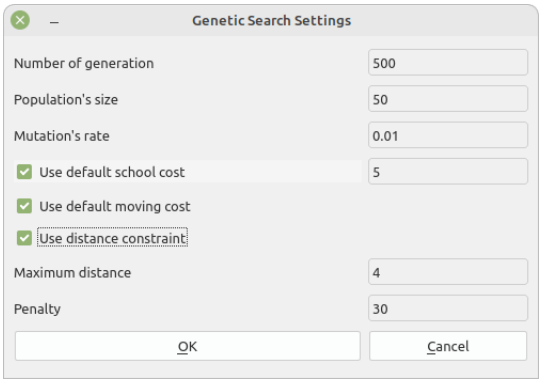


Figure 6.17 – Fenêtre de configuration de l'algorithme génétique et de l'instance du FLP.

Si l'utilisateur ne possède pas de fichier JSON contenant les coûts relatifs au FLP, il peut les définir dans cette fenêtre. Le champ `Use default facilities cost` attribue, à tous les nœuds du graphe, la valeur donnée en entrée comme coût d'ouverture. Si l'utilisateur coche la case `Use default transport cost`, les coûts de transport sont calculés comme étant les longueurs des plus courts chemins du graphe.

Dans la fenêtre de la Figure 6.17, l'utilisateur peut également définir une contrainte. Cette contrainte porte sur la distance maximale qu'il doit y avoir entre un foyer et l'école la plus proche. Il donc possible de spécifier cette distance ainsi que la pénalité à ajouter à la fonction objectif pour chaque client étant trop éloigné des entrepôts.

Dans le menu `Settings` de la fenêtre principale (Figure 6.16), l'utilisateur a accès aux options `Add schools` et `Add homes`. En sélectionnant l'option `Add schools`, l'utilisateur peut forcer certains nœuds à être des écoles. Une

fois l'option activée, il suffit de cliquer sur un nœud pour le transformer en école. Cliquer une seconde fois sur ce nœud annule ce choix. L'option `Add homes` fonctionne de façon analogue et permet d'ajouter des foyers.

Une fois toutes les configurations réalisées, l'utilisateur peut démarrer l'algorithme grâce à l'option `Start` du menu `Search` ou la touche `F1` de son clavier. L'utilisateur peut réinitialiser la solution à l'écran grâce à l'option `Refresh` du menu `File` ou la touche `F5`.

6.5 Module GAP de CUBE

Cette section présente le module de CUBE basé sur les modèles détaillés à la section 5.5. Ce module est une interface permettant de définir les critères à prendre en compte pour évaluer l'affectation des écoles dans une ville.

Ce module ne fournit pas les outils pour créer et afficher le graphe associé à une ville. Pour créer et exporter de tels graphes, nous utilisons la version 3.16 du logiciel QGIS [QGI09] muni des plugins Networks et SAGA. À l'aide du script `qgis_buildgraph.py`, il est possible de générer un tel graphe à partir des fichiers `buildings.shp` et `roads.shp` obtenus à l'aide d'OpenStreetMap [Ope17]. Le script `qgis_exportgraph.py` permet d'exporter un tel graphe sous format JSON.

La Figure 6.18 illustre les différentes options que l'utilisateur doit remplir pour évaluer une affectation des écoles à l'aide du GAP. Il doit dans un premier temps charger le fichier contenant le graphe sur lequel travailler. L'utilisateur doit également spécifier le dossier dans lequel sauvegarder la solution trouvée pour le GAP ainsi que les matrices de poids et de profits.

L'utilisateur doit spécifier la clé du fichier JSON donnant accès aux longueurs des plus courts chemins entre les foyers. Comme expliqué dans la section 5.5, ces longueurs sont catégorisées. À chaque catégorie est associée une valeur symbolisant le profit des parents à mettre leurs enfants dans une école proche. Plus la distance est courte, plus le profit est grand. L'utilisateur est donc invité à spécifier le nombre de catégories ainsi que la plus grande distance tolérable. Toutes les distances supérieures à cette valeur maximale appartiennent à la même catégorie. De manière analogue, l'utilisateur peut prendre en compte le temps de parcours et une préférence aléatoire pour définir la matrice de profits.

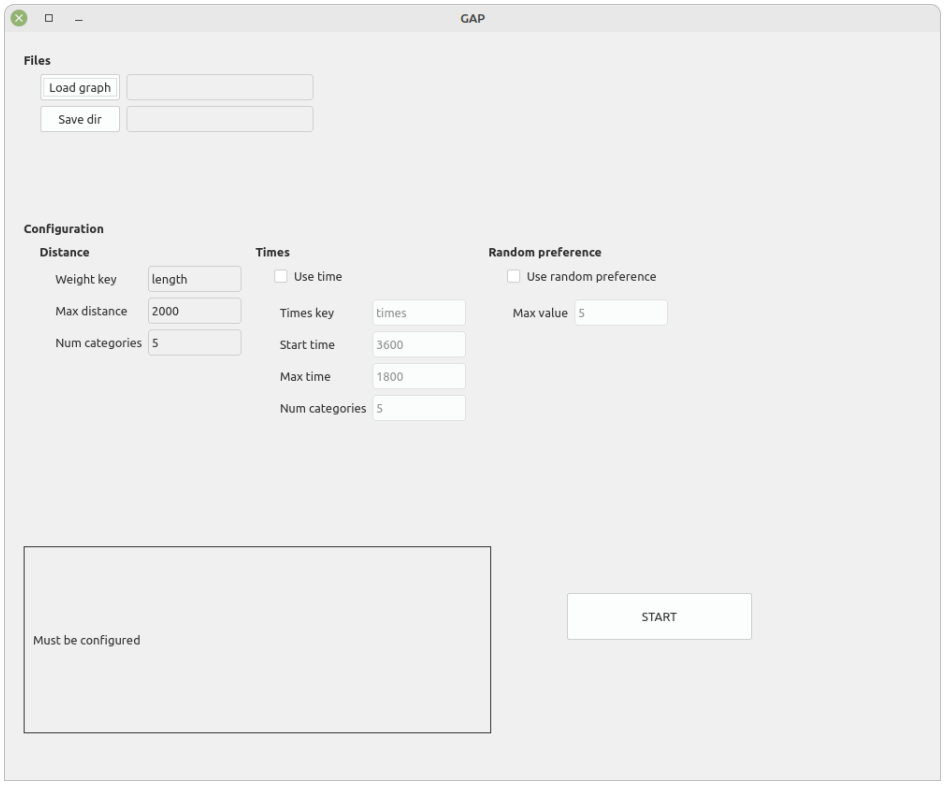


Figure 6.18 – Module GAP de CUBE.

CHAPITRE 7

CONCLUSION ET PERSPECTIVES

7.1 Conclusion

Nous avons, tout au long de cette thèse, présenté divers modèles mathématiques que nous appliquons à la problématique de la compacité urbaine. Nous avons également implémenté ces modèles afin de tester leur efficacité dans le cadre des problèmes étudiés.

Dans le chapitre 3, nous avons présenté une modélisation de l'îlot en deux dimensions. Ce modèle est adapté à l'utilisation d'algorithmes de recherche locale et de métaheuristiques. Un premier ensemble de contraintes a été implémenté afin de guider CUBE vers des solutions plus pertinentes. Grâce à la collaboration entre les différents membres de CoMod, il est ressorti que les premières solutions retournées par CUBE était prometteuses. Ainsi, le modèle semblait assez mûr pour considérer un composant supplémentaire dans l'îlot : les espaces verts. Toutefois, les solutions retournées étaient de qualité seulement si l'on considère que des bâtiments. Une fois les espaces non-bâtis introduits, *i.e.* un composant de l'îlot en opposition avec le bâti, cette première approche a montré ses limites.

Au vu des limites rencontrées par le modèle du chapitre 3, nous avons, dans

le chapitre 4, introduit des notions de théorie des jeux. Grâce à ces notions, les conflits entre le bâti et les espaces verts ont pu être gérés. De plus, grâce à la théorie des jeux, il s'avère plus facile de définir un comportement influant sur la construction de l'îlot spécifique à chaque critère. Il est également aisé de considérer de nouveaux critères, via l'ajout de joueurs. Nous avons donc pu traiter la hauteur des bâtiments et ajouter la troisième dimension de l'îlot.

Dans le chapitre 5, nous avons travaillé sur des problématiques liées à l'affectation des bâtiments. Ces problématiques nous ont amené à changer d'échelle urbaine. En effet, gérer l'affectation des bâtiments a plus de sens à l'échelle du quartier ou de la ville. Disposant désormais d'un outil générant des îlots compacts, nous avons pu abstraire la forme des bâtiments et utiliser un graphe pour modéliser un quartier ou une ville. À l'aide des graphes, nous avons pu étudier l'affectation des commerces de proximité et des écoles via la théorie des jeux et les problèmes d'optimisation que sont le *Facility Location Problem* et le *Knapsack Problem*.

7.2 Perspectives

Pour conclure, nous discutons ici de quelques perspectives liées à notre travail. Nous avons fait le choix, dans cette thèse, de générer des configurations d'îlot via la recherche d'Équilibres de Nash dans un jeu. Cependant, implémenter un module basé sur la recherche d'optima de Pareto reste une perspective pertinente.

Ces deux notions sont conceptuellement différentes. Lors de la recherche d'un Équilibre de Nash, améliorer un objectif se fait indépendamment des autres critères. Il est donc possible de dégrader les tous les autre objectifs en faveur d'un seul. Atteindre un optimum de Pareto ne peut pas se faire au détriment des autres critères. Cette différence fondamentale peut amener à des solutions très différentes. Ainsi, avoir un module supplémentaire dédié aux optima de Pareto nous permettrait de comparer les deux approches et d'étudier leurs avantages et inconvénients pour notre problème.

Nous avons étudié plusieurs problématiques liées à la compacité. Cela nous a permis de présenter une plus grande variété de modèles et d'algorithmes destinés aux urbanistes. Ainsi, bien que le module Théorie des Jeux de CUBE soit suffisamment flexible pour considérer un grand nombre de critères de compacité, il ne prend en compte que quelques critères simples. Un travail futur pourrait consister en l'implémentation de nouveaux critères de compacité. L'ajout de critères pourrait augmenter la qualité urbanistique des solutions proposées par CUBE.

Nous pourrions, notamment, ajouter une contrainte pour encourager CUBE à construire des bâtiments en bordure de l'îlot plutôt que proche du centre. L'ajout d'un joueur qui observe le rapport entre l'aire et le périmètre des bâtiments est également intéressant. Un tel joueur pourrait aider à éviter les formes en escaliers que peuvent avoir certains bâtiments.

Un travail sur les pavages peut également améliorer les configurations obtenues. Utiliser un pavage composé de plusieurs type de cellules permettrait d'une part de varier la forme de l'îlot et d'autre part d'avoir des formes intra-îlot avec des propriétés différentes. Une autre possibilité est de faire varier l'échelle des cellules. Avoir des zones de cellules de plus petite échelle nous donne la possibilité d'être plus précis dans la conception de la forme des bâtiments.

Bien que nous ayons traité des problématiques à l'échelle du quartier et de la ville dans le chapitre 5, nous avons surtout travaillé à l'échelle de l'îlot. Or, ces deux autres échelles apportent leur lot de questions. Ainsi, déterminer si une ville compacte est simplement un agencement d'îlots compacts est une première question pouvant être posée aussi bien aux urbanistes qu'aux mathématiciens et informaticiens.

La ville est un système beaucoup plus complexe que l'îlot. Il sera donc nécessaire de concevoir des outils plus complexes pour modéliser une ville. En outre, la ville et le quartier ont des spécificités que l'îlot n'a pas, notamment la gestion du réseau routier.

Un façon d'aborder ce changement d'échelle pourrait se faire en changeant l'interprétation des cellules d'un pavage via un effet de «zoom et dézoom». Ici, une cellule ne serait plus une unité de $1\,\mathrm{m}^2$ mais un îlot complet. Nous pourrions ainsi utiliser un pavage pour déterminer les emplacements des îlots dans une ville ou un quartier. Il serait ensuite possible de retravailler à l'échelle de l'îlot pour ajuster les configurations en fonction de leur emplacement et des nouveaux critères identifiés.

Dans leur article de 2018 [EFCA18], Erin *et al.* retracent l'évolution des méthodes quantitatives utilisées pour étudier les morphologies urbaines. Les méthodes les plus récentes utilisent, à différents degrés, des outils d'aide informatiques. Cependant, ces outils informatiques peuvent être uniquement dédié à la problématique étudiée. Une prochaine étape dans l'étude des morphologies urbaines pourrait être d'y intégrer des modèles issus des sciences mathématiques. Comme stipulé précédemment, ces modèles et algorithmes sont très généraux et flexibles. Ainsi, un même modèle ou algorithme peut ne nécessiter que quelques adaptations pour résoudre d'autre problèmes, urbanistiques ou non.

Par exemple, de tels modèles peuvent aider à l'aménagement de ville afin de faire face à divers vulnérabilités. En effet, la théorie des jeux modélise des interactions entre des entités ou des systèmes comme des jeux multijoueurs. Ces entités peuvent être de diverses natures et symboliser des phénomènes naturels. Il est donc possible de créer un jeu où un des joueurs est un cours d'eau qui déborde et les autre joueurs des entités devant faire face aux inondations. Un tel jeu pourrait aider à déterminer une distance de sécurité entre les habitations et le cours d'eau et les lieux où il est impératif de placer des digues, par exemple.

Une autre utilisation des modèles présentés dans cette thèse peut être faite en changeant leur interprétation. Il est naturel d'utiliser le FLP pour des problèmes liés à l'ouverture d'entrepôts pour fournir des clients. Nous avons utilisé ce problème d'optimisation afin de déterminer la position et le nombre opti-

mal d'écoles à ouvrir afin que les foyers aient un accès rapide à l'éducation. Le FLP pourrait également servir à gérer certains aspects d'une pandémie. Au lieu d'ouvrir des entrepôts ou des écoles, il peut être utilisé afin de déterminer les meilleurs emplacements pour des centres de dépistage.

BIBLIOGRAPHIE

[Alh07] Alhzeiia : Barcelone et le plan cerdà, avec ses îlots réguliers. `https://commons.wikimedia.org/w/index.php?curid=7363603`, 2007. Consulté le 16/12/2021.

[ASL+20] Ahmed Khairadeen Ali, Hayub Song, One Jae Lee, Eun Seok Kim et Haneen Hashim Mohammed Ali : Multi-agent-based urban vegetation design. *International Journal of Environmental Research and Public Health*, 17(9):3075, 2020.

[BBS+02] Botsch Steinberg Bischoff, M Botsch, S Steinberg, S Bischoff, L Kobbelt et Rwth Aachen : Openmesh–a generic and efficient polygon mesh data structure. *In OpenSG Symposium*, 2002.

[Ble18] Blender Online Community : *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[Car21] Guillaume Cardoen : Le problème de placementd'entrepôts, Bourse d'initiation à la recherche (UMONS), 2021.

[Cha98] Ismail Chabini : Discrete dynamic shortest path problems in transportation applications : Complexity and algorithms with optimal run time. *Transportation research record*, 1645(1):170–175, 1998.

[CKR06] Reuven Cohen, Liran Katzir et Danny Raz : An efficient approximation for the generalized assignment problem. *Information Processing Letters*, 100(4):162–166, 2006.

[CLRS09] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest et Clifford Stein : *Introduction to algorithms*. MIT press, 2009.

[CPRML20] Jose Carpio-Pinedo, Guillermo Ramírez, Salas Montes et Patxi J. Lamiquiz : New urban forms, diversity, and computational design : Exploring the open block. *Journal of Urban Planning and Development*, 146(2):04020002, 2020.

[CR21] Julien Charlier et Isabelle Reginster : *Les polarités de base – Des balises pour identifier des centralités urbaines et rurales en Wallonie*. L'Institut wallon de l'évaluation, de la prospective et de la statistique, 2021.

[DDDF17] Herman De Beukelaer, Guy F. Davenport, Geert De Meyer et Veerle Fack : James : An object-oriented java framework for discrete optimization using local search metaheuristics. *Software : Practice and Experience*, 47(6):921–938, 2017.

[De 18] Isabelle De Smet : *Elaboration et expérimentation d'un outil d'évaluation et d'aide à la conception compacts à dominante d'habitat suivant une densité de population cible.* Thèse de doctorat, UMONS, 2018.

[Des11] Xavier Desjardins : Pour l'atténuation du changement climatique, quelle est la contribution possible de l'aménagement du territoire ? *Cybergeo : European Journal of Geography*, 2011.

[Dir14] Direction générale Statistique - Statistics Belgium : Census 2011 belgique, 2014.

[DSL19] Isabelle De Smet et David Laplume : Design of compact residential blocks for sustainable urban regeneration : Determination of consistent qualitative criteria. *Journal of Urban Regeneration & Renewal*, 12(3):248–257, 2019.

[DSMB+22] Isabelle De Smet, Quentin Meurisse, Vincent Becue, Thomas Brihaye, Jérémy Cenci, David Laplume, Hadrien Mélot et

Cédric Rivière : Compacts typo-morphologies by use of local search methods. *In Cities as Assemblages. Proceedings of the XXVI International Seminar on Urban Form 2019 | 2-6 July 2019, Nicosia, Cyprus*, volume 3, pages 329–337. tab edizioni, Rome, 2022.

[EFCA18] Irem Erin, Giovanni Fusco, Ebru Cubukcu et Alessandro Araldi : Quantitative methods of urban morphology in urban design and environmental psychology. *In 24th ISUF International Conference. Book of Papers*, pages 1391–1400. Editorial Universitat Politècnica de València, 2018.

[EG09] Talbi El-Ghazali : *Metaheuristics : from design to implementation*, volume 74. John Wiley & Sons, 2009.

[FH09] Reza Zanjirani Farahani et Masoud Hekmatfar : *Facility location : concepts, models, algorithms and case studies.* Springer Science & Business Media, 2009.

[Fre78] Linton C Freeman : Centrality in social networks conceptual clarification. *Social networks*, 1(3):215–239, 1978.

[Gé19] Géoconfluence (ENS Lyon) : Définition de centralité. `http://geoconfluences.ens-lyon.fr/glossaire/centralite`, 2019.

[HMBP19] Pierre Hansen, Nenad Mladenović, Jack Brimberg et José A Moreno Pérez : Variable neighborhood search. *In Handbook of metaheuristics*, pages 57–97. Springer International Publishing, 2019.

[HSO+16] Ernest I Hennig, Tomas Soukup, Erika Orlitova, Christian Schwick, Felix Kienast et Jochen AG Jaeger : *Urban Sprawl in Europe. Joint EEA-FOEN report. No 11/2016.* Publications Office of the European Union, 2016.

[Ing98] Gregory K Ingram : Patterns of metropolitan development : what have we learned ? *Urban studies*, 35(7):1019–1035, 1998.

[KFST96] Jozef Kratica, V Filipovic, V Sesum et D Tosic : Solving the uncapacitated warehouse location problem using a simple genetic

algorithm. *In Proceedings of the XIV International Conference on Material handling and warehousing*, pages 3–33, 1996.

[KPP04] Hans Kellerer, Ulrich Pferschy et David Pisinger : *Knapsack Problems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[Kö08] Kay Körner : The terrain of dresden. `https://commons.wikimedia.org/wiki/File:Dresden_%C3%9Cberblick_9.jpg`, 2008. Consulté le 16/12/2021.

[LCS16] Quang Duy LÃ, Yong Huat Chew et Boon-Hee Soong : *An Introduction to Game Theory*, pages 3–22. Springer International Publishing, Cham, 2016.

[MDSM+20] Quentin Meurisse, Isabelle De Smet, Hadrien Mélot, David Laplume, Thomas Brihaye, Cédric Rivière, Emeline Coszach, Jérémy Cenci, Sesil Koutra et Vincent Becue : Recherche locale et théorie des jeux appliqués à la création de typomorphologies compactes. *In SHS Web of Conferences*, volume 82, page 03004. EDP Sciences, 2020.

[Mou15] Ana Clara Mourão Moura : Geodesign in parametric modeling of urban landscape. *Cartography and Geographic Information Science*, 42(4):323–332, 2015.

[MS96] Dov Monderer et Lloyd S Shapley : Potential games. *Games and economic behavior*, 14(1):124–143, 1996.

[MS16] Akio Matsumoto et Ferenc Szidarovszky : Continuous static games. *In Game Theory and Its Applications*, pages 21–47. Springer Japan, Tokyo, 2016.

[MT90] Silvano Martello et Paolo Toth : *Knapsack problems : algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[Neu05] Michael Neuman : The compact city fallacy. *Journal of planning education and research*, 25(1):11–26, 2005.

[Nom19] Nomen4Omen : Example red-black tree with nil-leaves. `https://commons.wikimedia.org/wiki/File:`

Red-black_tree_example_with_NIL.svg, 2019. Consulté le 15/04/2022.

[Non09a] R. A. Nonenmacher : Regular tiling 3-6 (triangular). https://commons.wikimedia.org/wiki/File:Tiling_Regular_3-6_Triangular.svg, 2009. Consulté le 17/01/2022.

[Non09b] R. A. Nonenmacher : Regular tiling 6-3 (hexagonal). https://commons.wikimedia.org/wiki/File:Tiling_Regular_6-3_Hexagonal.svg, 2009. Consulté le 17/01/2022.

[Non09c] R. A. Nonenmacher : Semiregular tiling 4-8-8 (truncated square). https://commons.wikimedia.org/wiki/File:Tiling_Semiregular_4-8-8_Truncated_Square.svg, 2009. Consulté le 17/01/2022.

[Ope17] OpenStreetMap contributors : Planet dump retrieved from https ://planet.osm.org . https://www.openstreetmap.org, 2017.

[Pou04] Guillaume Pouyanne : Des avantages comparatifs de la ville compacte à l'interaction mobilité-forme urbaine. méthodologie et premiers résultats. *Cahiers scientifiques du transport*, 45(1):49–82, 2004.

[QGI09] QGIS Development Team : *QGIS Geographic Information System*. Open Source Geospatial Foundation, 2009.

[Sha02] Vadim Shapiro : Chapter 20 - solid modeling. *In* Gerald Farin, Josef Hoschek et Myung-Soo Kim, éditeurs : *Handbook of Computer Aided Geometric Design*, pages 473–518. North-Holland, Amsterdam, 2002.

[SO06] Roberto Solis-Oba : Approximation algorithms for the k-median problem. *In* Evripidis Bampis, Klaus Jansen et Claire Kenyon, éditeurs : *Efficient Approximation and Online Algorithms : Recent Progress on Classical Combinatorial Optimization Problems and New Applications*, pages 292–320. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[SPM+97] Lena Sanders, Denise Pumain, Hélène Mathian, France Guérin-Pace et Stéphane Bura : SIMPOP : a multi-agent system for the study of urbanism. *Environment and Planning B : Planning and Design*, 24(2):287–305, 1997.

[Tom20] TomTom N.V. : Tomtom traffic stats. `https://www.tomtom.com/products/traffic-stats/`, 2020.

[Tor06] Paul M Torrens : Simulating sprawl. *Annals of the Association of American Geographers*, 96(2):248–275, 2006.

[VN97] Mark Voorneveld et Henk Norde : A characterization of ordinal potential games. *Games and Economic Behavior*, 19(2): 235–242, 1997.

[WS11] David P Williamson et David B Shmoys : *The design of approximation algorithms*. Cambridge university press, 2011.

[YGY19] Chunxia Yang, Zhuoxing Gu et Ziying Yao : Adaptive urban design research based on multi-agent system-taking the urban renewal design of shanghai hongkou port area as an example. 2019.

ANNEXE A

COMPLEXITÉ DES PROBLÈMES PRÉCÉDEMMENT DÉFINIS

Dans cette annexe, nous étudions la *complexité algorithmique* des différents problèmes définis précédemment. Nous supposons que le lecteur est familier avec les classes de complexité et la notation grand O. La cas échéant, le lecteur peut se référer à l'ouvrage de Cormen *et al.* [CLRS09] pour de plus amples informations.

A.1 Affectation des commerces via un jeu

Nous avons défini, dans la section 5.3, un jeu dont l'objectif est de placer k commerces dans une ville. Pour rappel, la définition de ce jeu est la suivante :

Définition A.1.1. Soit un graphe G, nous notons $White(G)$ l'ensemble des nœuds blancs de G. Soit $N = \{1, \ldots, n\}$, l'ensemble des joueurs. Le problème d'affectation des commerces de proximité est modélisé comme le jeu $\mathcal{G} = (N, (S_i)_{i \in N}, (g_i)_{i \in N})$, où pour tout $i \in N$, $S_i = White(G)$. Si nous notons $v_i \in White(G)$ le nœud de G choisi pas le joueur i, la fonction de

gains de ce joueur est définie comme suit :

$$g_i(v_i) = \sum_{u \in White(G)} d_G(u, v_i), \tag{A.1}$$

où $d_G(u, v_i)$ est la longueur d'un plus court chemin du nœud u vers le nœud v. Les joueurs veulent minimiser leur fonction de coûts.

Ici, les gains d'un joueur ne dépendent pas des nœuds choisis par les autres joueurs. En ayant connaissance des longueurs des plus courts chemins entre chaque paire de nœuds de G, il est donc possible de déterminer la valeur de l'équation A.1 pour chaque sommet de G en $O(n^2)$, où n est le nombre de sommets de G.

Une fois chaque nœud évalué, il suffit de sélectionner les k nœuds engendrant les plus petits gains. L'objectif des k joueurs étant de minimiser leurs gains, sélectionner ces k nœuds retourne la solution optimale.

Bien que résoudre ce jeu peut se faire naïvement en temps polynomial, l'approche proposée dans la section 5.3 reste pertinente. En effet, nous travaillons avec des graphes modélisant des villes. Ces graphes peuvent posséder un grand nombre de nœuds. Ainsi, malgré des calculs en $O(n^2)$, évaluer tous les nœuds du graphe peut quand même nécessiter un temps de calcul conséquent. Appliquer une recherche de proche en proche, pour trouver les meilleurs nœuds, réduit le nombre de nœuds à évaluer et donc le temps de calcul.

A.2 Affection des écoles via un jeu

Nous avons défini, dans la section 5.3, un jeu dont l'objectif est de placer k écoles dans une ville. Pour rappel, la définition de ce jeu est la suivante :

Définition A.2.1. Soit un graphe G, nous notons $White(G)$ l'ensemble des nœuds blancs de G et $Blue(G)$ l'ensemble de nœuds bleus de G. Soit $N = \{1, \ldots, n\}$, l'ensemble des joueurs. Le problème d'affectation

des écoles est modélisé comme le jeu $\mathcal{G} = (N, (S_i)_{i\in N}, (g_i)_{i\in N})$, où pour tout $i \in N$, $S_i = White(G)$. Si nous notons $v_i \in White(G)$ le nœud de G choisi par le joueur i, nous avons $Blue(G) = \{v_i : i \in N\}$la fonction de gains de ce joueur est définie comme suit :

$$g_i(v_i, v_{-i}) = \sum_{u\in White(G)} \min_{b\in Blue(G)} d_G(u, b),$$

où $d_G(u, b)$ est la longueur d'un plus court chemin du nœud u vers le nœud b. Les joueurs veulent minimiser leur fonction de gains.

À notations près, la fonction de gains de ce jeu est identique à la fonction objectif du *k-Median Problem*.

Définition A.2.2 (*k-Median Problem* [SO06])**.** Soient $I = \{i_1, \ldots, i_m\}$ un ensemble de sites où des entrepôts peuvent être construits, $J = \{j_1, \ldots, j_n\}$ un ensemble de clients et une matrice $C = (c_{ij})_{\substack{i\in I\\ j\in J}}$ où c_{ij} est le coût de transport de l'entrepôt i au client j.
L'objectif du *k-Median Problem* est de trouver $\varnothing \subset S \subseteq I$ tel que S minimise :

$$\varphi(S) = \sum_{j\in J} \min_{i\in S} c_{ij},$$

et $|S| \leqslant k$

Le *k-Median Problem* est très similaire au FLP. Ici, ouvrir un entrepôt n'a pas de coût mais on ne peut en ouvrir qu'au plus k.

Dans son article, Solis-Oba prouve que le *k-Median Problem* est NP-difficile [SO06]. Nous en déduisons que le jeu d'affectation de k écoles est aussi NP-difficile.

A.3 Problème d'Assignation des Écoles

Dans la section 5.5 nous avons défini le Problème d'Assignation des Écoles comme suit :

Définition A.3.1 (Problème d'Assignation des Écoles)**.** Soient n foyers et m écoles. Soit le vecteur $c = (c_1, \ldots, c_m)$ tel que pour tout j, c_j soit le nombre maximum d'élèves que l'école j peut accueillir. Soit le vecteur $(w_1, \ldots, w_n)$ où pour tout $1 \leqslant i \leqslant n$, w_i est le nombre d'enfants à scolariser dans le foyer i. Soit $P = (p_{ij})_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant m}}$ tel que pour i et pour tout j, p_{ij} est la satisfaction du foyer i de mettre leurs enfants à l'école j. L'objectif d'une instance de ce problème est de maximiser :

$$\sum_{j=1}^{m} \sum_{i=1}^{n} p_{ij} x_{ij},$$

sous les contraintes que

$$\text{pour tout } 1 \leqslant j \leqslant m \sum_{i=1}^{n} w_i x_{ij} \leqslant c_j,$$

$$\sum_{j=1}^{m} x_{ij} \leqslant 1,$$

où pour tout $1 \leqslant i \leqslant n$ et pour tout $1 \leqslant j \leqslant m$

$$x_{ij} = \begin{cases} 1 & \text{si le foyer } i \text{ met ses enfants à l'école } j, \\ 0 & \text{sinon.} \end{cases}$$

Afin de déterminer la complexité algorithmique de ce problème, nous devons définir le *Multiple Knapsack Problem* (MKP). Ce problème est une variante du *Knapsack Problem* utilisant plusieurs sacs à dos.

Définition A.3.2 (*Multiple Knapsack Problem*)**.** Soient n objets et m sacs. Soient le vecteur $c = (c_1, \ldots, c_m)$tel que pour j, c_j soit la capacité maximale du sac j et les vecteurs $(w_1, \ldots, w_n)$ et $(p_1, \ldots, p_n)$ tels que

pour tout $1 \leqslant i \leqslant n$, w_i et p_i sont respectivement le poids et la valeur qu'a l'objet i. L'objectif d'une instance du MKP est de maximiser :

$$\sum_{j=1}^{m} \sum_{i=1}^{n} p_i x_{ij},$$

sous les contraintes que pour tout $1 \leqslant j \leqslant m$

$$\text{pour tout } 1 \leqslant j \leqslant m \ \sum_{i=1}^{n} w_i x_{ij} \leqslant c_j,$$

$$\sum_{j=1}^{m} x_{ij} \leqslant 1,$$

où pour tout $1 \leqslant i \leqslant n$ et pour tout $1 \leqslant j \leqslant m$

$$x_{ij} = \begin{cases} 1 & \text{si le foyer } i \text{ met ses enfants à l'école } j, \\ 0 & \text{sinon.} \end{cases}$$

Lemme A.3.3 ([MT90]). *MKP est NP-difficile*

Lemme A.3.4. *Le Problème d'Assignation des Écoles est NP-difficile*

Preuve. Trivialement, MKP est équivalent au cas particulier du Problème d'Assignation des Écoles où la satisfaction des foyers ne change pas d'une école à une autre *i.e.*, le cas où la matrice P est de la forme

$$\begin{pmatrix} p_1 & \cdots & p_1 \\ p_2 & \cdots & p_2 \\ \vdots & \vdots & \vdots \\ p_n & \cdots & p_n \end{pmatrix}$$

D'où le Problème d'Assignation des Écoles est NP-difficile □