INP ENSEEIHT

Graphes et Recherche opérationnelle

Rapport final

Chef de projet :
Martin Carton
Responsable qualité :
Maxime Arthaud

Maxence Ahlouche
Korantin Auguste
Thomas Forgione
Thomas Wagner

20 décembre 2013

Table des matières

1 Introduction **4**

2 UA : Graphes **5**
2.1 Introduction 5
2.2 Définitions 5
2.3 Modélisation mathématique 6
2.4 Graphes eulériens 6
2.4.1 Analyse mathématique 6
2.4.2 Méthode de résolution 7
2.5 Graphes hamiltoniens 10
2.5.1 Analyse mathématique 10
2.5.2 Méthode de résolution 10
2.6 Problème du postier chinois 12
2.6.1 Analyse mathématique 12
2.6.2 Méthode de résolution 12
2.6.3 Exemple 12
2.7 Problème du voyageur de commerce 15
2.7.1 Analyse mathématique 15
2.7.2 Heuristiques 16
2.7.3 Recherche locale 16
2.7.4 Métaheuristiques 18
2.7.5 Conclusion 19

3 UA : Programmation linéaire **20**
3.1 Introduction 20
3.2 Problème du sac à dos 20
3.2.1 Présentation du problème 20
3.2.2 Résolution exacte 20
3.2.3 Résolution approchée 21
3.2.4 Relation avec le voyageur de commerce 22
3.3 Problème d'optimisation linéaire 22
3.3.1 Point de vu mathématique 22
3.3.2 Algorithme du simplexe 23

4 UA : Jeux **25**
4.1 Shifumi 25
4.1.1 Chaines de Markov 25
4.1.2 Variantes 25
4.2 Jeu de la somme magique 26
4.2.1 Représentation sous forme de morpion 26
4.2.2 Algorithme du minimax appliqué au jeu du morpion 26

4.2.3 Élagage alpha-bêta 27
4.3 Duopôles 28
4.3.1 Stratégies possibles 28

5 UA : Processus stochastiques **29**
5.1 Introduction – Chaines de Markov 29
5.2 Colonie de scarabées 29
5.2.1 Chaines de Markov 29
5.2.2 Temps moyen entre rencontres 30
5.2.3 Exemple 30
5.3 Théorie des files d'attente 31
5.3.1 Cas M/M/1 32
5.3.2 Cas M/M/S 33

6 UA : Ingénierie robotique **34**
6.1 Suivi de murs 34
6.2 Labyrinthe 34
6.2.1 Suivi de mur 34
6.2.2 Algorithme de Dijkstra 34
6.2.3 Algorithme A* 36

7 Bilan **37**
7.1 Enseignement 37
7.2 Travail en groupe 37
7.3 Apprentissages 37

1 Introduction

Dans ce rapport, nous allons présenter les différents algorithmes, modèles, théorèmes, etc. que nous avons pu découvrir ou approfondir lors de l'UE de graphes et recherche opérationnelle.

Ce rapport ne nous servira pas à présenter les travaux accomplis lors des différentes unités d'acquisitions, ni à présenter nos programmes : pour cela, se référer aux rapports des UAs.

Il se terminera par un bilan de ce que nous avons acquis lors des différentes UAs, mais aussi des remarques sur notre travail en groupe.

2 UA : Graphes

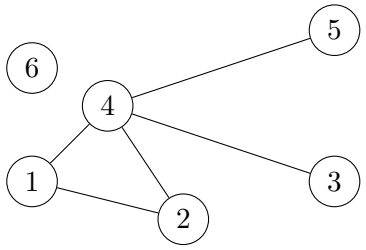
2.1 Introduction

La théorie des graphes est un domaine entre les mathématiques et l'informatique très utilisé dans la résolution de nombreux problèmes : cela peut aller du routage sur internet au calcul d'itinéraire par des GPS...

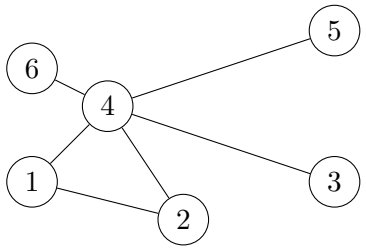
Nous verrons dans cette section quelques applications des graphes. Une dernière application sera vu dans la section 6.2.

2.2 Définitions

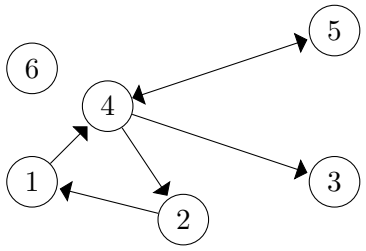
Graphe Un graphe est constitué d'un ensemble de sommets, et d'un ensemble d'arêtes (c'est-à-dire des couples de sommets) qui relient ces sommets.



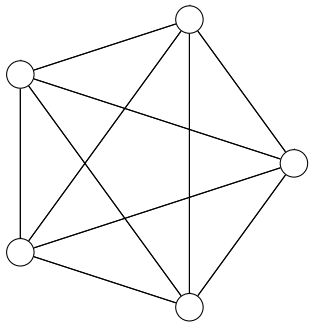
Graphe connexe Un graphe est dit connexe quand on peut aller de n'importe quel sommet vers n'importe quel autre sommet en suivant des arêtes.



Graphe orienté Un graphe orienté n'est pas constitué d'arêtes mais d'arcs (des paires de sommets).



Graphe complet Un graphe est dit complet quand tout ses sommets sont reliés deux à deux par une arrête. Le nombre de d'arêtes d'un tel graphe est alors $\frac{n(n-1)}{2}$.



Degré d'un sommet Le degré d'un sommet est le nombre d'arêtes incidentes à ce sommet.

2.3 Modélisation mathématique

Il existe plusieurs moyens de représenter des graphes. Parmi ceux-ci, le plus simple est la matrice d'adjacence, où l'on stocke une matrice de taille $n \times n$ (n étant le nombre de sommets), dont chaque colonne et chaque rangée représente un sommet. La case i, j de la matrice contient un 1 si les sommets i et j sont reliés par une arête (ou un arc dans le bon sens, dans le cas orienté). Évidemment, cette représentation est loin d'être efficace, la mémoire utilisée étant exponentielle quand le nombre de sommets du graphe augmente. Toutefois, elle peut servir pour certains algorithmes. Notamment, un algorithme de recherche de chemin peut multiplier la matrice d'adjacence par elle-même m fois : alors, il existera un chemin de taille m entre deux sommets si la case correspondante contient un 1. On notera toutefois que les matrices d'adjacences étant souvent creuses (remplies de 0), il existe des moyens de les représenter autrement que par un tableau.

On peut également représenter les graphes par une liste de sommets, chacun ayant une liste d'arêtes. En mémoire, cette structure est donc constituée d'une liste de pointeurs vers des sommets. Les sommets contenant une liste de pointeurs vers des arêtes. Chaque arête dispose d'un pointeur vers chaque sommet extrémité. Cette structure est évidemment plus efficace, car elle ne stocke que les informations nécessaires.

2.4 Graphes eulériens

2.4.1 Analyse mathématique

Un graphe eulérien est un graphe contenant un cycle eulérien, c'est-à-dire une chaine parcourant toutes les arêtes du graphe une et une seule fois, en revenant au sommet de départ.

Théorème d'Euler Un graphe connexe est eulérien si et seulement si chacun de ses sommets est de degré pair.

Un graphe semi-eulérien, quant à lui, contient une chaine eulérienne : celle-ci passe également par toutes les arêtes du graphe une seule et unique fois, mais ne retourne

pas au sommet de départ. Le théorème précédent se généralise alors aux graphes semi-eulériens : un graphe connexe est semi-eulérien si et seulement tous ses sommets sauf deux sont associés à un nombre pair d'arêtes. Dans ce cas, la chaine eulérienne aura pour départ l'un des deux sommets associés à un nombre impair d'arêtes et pour sommet d'arrivée le deuxième.

2.4.2 Méthode de résolution

Afin de trouver une chaine ou un cycle eulérien dans un graphe, on peut utiliser deux méthodes : une méthode qui teste toutes les possibilités, et une autre plus intelligente et moins couteuse.

Matrices latines La première méthode (voir algorithme 1) est inspirée des matrices latines. Chaque coefficient de la matrice sera un ensemble de chaines, une chaine étant elle-même une liste de sommets. La matrice latine de notre graphe sera la matrice M dont chaque coefficient $m_{i,j}$ vaudra :

— l'ensemble vide si le nœud i n'est pas relié au nœud j dans le graphe ;
— un ensemble contenant pour unique élément la chaine $[N_i, N_j]$ si les nœuds i et j sont reliés (où N_k représente le nœud k).

Nous définissons ensuite un produit sur les coefficients d'une telle matrice (voir algorithmes 2 et 3). Le produit de deux chaines sera :

— nul si le dernier nœud de la première chaine n'est pas le premier nœud du deuxième ;
— la concaténation des deux chaines sinon.

Le produit de deux ensembles de chaines sera l'ensemble contenant les produits de chaque couple de nœuds.

Pour tout k entier naturel, le coefficient (i, j) de la matrice M^k représentera l'ensemble des chaines de longueur k reliant les nœuds i et j.

Puisque une chaine eulérienne passe une unique fois par chaque arête, il suffira de calculer la matrice latine élevée à cette puissance pour trouver sur sa diagonale l'ensemble des cycles possibles. En éliminant à chaque produit les chaines qui passent plusieurs fois par la même arête, on trouve l'ensemble des cycles eulériens.

La complexité de cet algorithme est exponentielle : calculer la puissance de la matrice latine revient en fait à calculer chaque chaine possible dans le graphe, et tester si elle est un cycle eulérien ou non.

Algorithme d'Euler La deuxième méthode, basée sur l'algorithme d'Euler est nettement plus efficace. Une fonction récursive cherche un cycle eulérien d'un sous-graphe de notre graphe de départ, puis s'appelle récursivement sur chacun des sommets parcourus par cette chaine, dans le graphe où l'on a supprimé les arêtes déjà parcourues. En reconstruisant ces cycles astucieusement, on parvient à trouver un cycle eulérien de complexité linéaire en le nombre d'arêtes du graphe.

Entrées : un graphe
Sorties : la liste des cycles ou chaines eulériens si le graphe est eulérien
semi-eulérien ou la liste vide sinon
début
  `/* Construction de la matrice latine du graphe */`
  construire une matrice à n lignes et n colonnes;
  remplir la matrice de listes vides;
  pour *chaque nœud du graphe* **faire**
    pour *chaque arête sortant de ce nœud* **faire**
      ajouter la liste [noeud de départ, noeud d'arrivée] à la case de la matrice correspondante;
    fin
  fin
  n ← le nombre d'arêtes total du graphe;
  calculer la puissance $(n-1)$ième de la matrice
  pour *chaque coefficient de la matrice ainsi calculée* **faire**
    si *le coefficient n'est pas nul* **alors**
      concaténer ce coefficient à la variable de retour;
    fin
  fin
fin

Algorithme 1 : Méthode de la matrice latine

Entrées : A et B deux matrices latines
Sorties : le produit de ces deux matrices
début
  construire la matrice de retour à n lignes et n colonnes;
  initialiser chaque coefficient de cette matrice à la liste vide;
  pour *chaque coefficient de la matrice de retour* **faire**
    pour *k allant de* 1 *jusqu'à n* **faire**
      calculer les chaines produits entre a(i,k) et b(k,j);
      ajouter au coefficient de la matrice ces chaines;
    fin
  fin
fin

Algorithme 2 : Produit matriciel

```
Entrées : liste_1 et liste_2 deux listes de chaine
Sorties : une liste de chaines
début
    créer une liste de chaine vide (liste de retour);
    pour i dans liste_1 faire
        pour j dans liste_2 faire
            construire la chaine résultante de la concaténation de i et j (en enlevant
            le nœud présent deux fois);
            construire un ensemble de chaine vide;
            pour k allant de 1 à la longueur de la chaine construit faire
                construire la chaine élémentaire menant du nœud k au nœud k + 1;
                si cette chaine n'est pas dans l'ensemble alors
                    ajouter cette chaine dans l'ensemble;
                fin
                sinon
                    rendre la chaine nulle;
                    sortir de la boucle;
                fin
            fin
        fin
        si le chaine n'est pas nulle alors
            concaténer la chaine trouvée à la liste de retour
        fin
    fin
fin
```

Algorithme 3 : Produit entre listes de chaines (coefficients de matrices latines)

2.5 Graphes hamiltoniens

2.5.1 Analyse mathématique

Un graphe hamiltonien (resp. semi-hamiltonien) est un graphe sur lequel on peut trouver un cycle (resp. une chaine) passant par tout les sommets une et une seule fois. Ce problème est donc celui d'un enfant qui souhaiterait visiter de manière unique toutes les salles d'un musée.

Le problème de savoir si un graphe est (semi-)hamiltonien est NP-complet, de même que de trouver un cycle ou une chaine s'il y en a.

Il existe cependant des conditions suffisantes pour lesquelles on peut affirmer qu'un graphe est hamiltonien.

Théorème Un graphe complet est hamiltonien. C'est une conséquence du théorème de Dirac.

Théorème de Dirac Un graphe simple à n sommets ($n \geq 3$) dont chaque sommet est au moins de degré $\frac{n}{2}$ est hamiltonien.

Théorème de Ore Un graphe simple à n sommets ($n \geq 3$) tel que la somme des degrés de toute paire de sommets non adjacents vaut au moins n est hamiltonien.

Théorème de Pósa Un graphe simple à n sommets ($n \geq 3$) est hamiltonien si :

- pour tout entier k tel que $1 \leq k < \frac{n-1}{2}$ le nombre de sommets de degré inférieur ou égal à k est inférieur à k ;
- le nombre de sommets de degré inférieur ou égal à $\frac{n-1}{2}$ est inférieur ou égal à $\frac{n-1}{2}$.

Fermeture d'un graphe La fermeture d'un graphe est le graphe construit à partir de celui en rajoutant des arrêtes entre chaque sommets a et b tel que $\deg(a)+\deg(b) > n$ tant qu'il en existe.

Théorème de Bondy et Chvátal Un graphe est hamiltonien si et seulement si sa fermeture est hamiltonienne.

Ce théorème n'est utile que si l'on peut utiliser l'un des théorèmes précédents sur la fermeture.

2.5.2 Méthode de résolution

Pour tester si un graphe est hamiltonien on peut utiliser les théorèmes précédents. Si le graphe ne respecte les conditions d'aucun de ces théorèmes, on recherche une chaine hamiltonienne dans ce graphe.

Pour rechercher une chaine hamiltonienne dans un graphe, on pourrait chercher parmi toutes les chaines possibles. La complexité d'un tel algorithme (voir algoritme 4) dans le pire des cas est donc très mauvaise : $O(n!)$. Comme on peut s'arrêter dès qu'on a trouvé une chaine sans devoir tester toutes les autres chaines possibles, la complexité moyenne sera inférieure.

```
Entrées : un graphe
Entrées : un sommet de départ optionnel node_from
Entrées : un ensemble (éventuellement vide) de nœuds déjà parcouru nodes_done
Sorties : une chaine hamiltonienne sous la forme d'une liste ordonnée de sommets,
          ou None s'il n'en existe pas
début
    si node_from n'a pas été fourni alors
        node_from ← un nœud du graphe;
    fin
    ajouter node_from à nodes_dones;
    si cardinal(node_from) == ordre(graphe) alors
        retourner [node_from]
    fin
    pour chaque arête dans le graphe faire
        autre ← le sommet opposé à node_from par rapport à cette arête;
        si autre dans nodes_done alors
            passer à la prochaine arête
        fin
        appeler la fonction récursivement avec le graphe, node_from et nodes_dones
        comme paramètres;
        si la liste retournée est non-vide alors
            y ajouter node_from au début et la retourner;
        fin
    fin
fin
```

Algorithme 4 : Recherche de chaine hamiltonienne

2.6 Problème du postier chinois

2.6.1 Analyse mathématique

Le problème du postier chinois consiste à trouver la chaine la plus courte dans un graphe connexe non-orienté passant au moins une fois par chaque arête, et revenant à son point de départ.

Ce problème est donc celui du facteur qui souhaite réaliser une tournée la plus rapidement possible en passant par toutes les rues et retournant à la poste.

Ce problème peut être réduit à la recherche d'un couplage parfait de cout minimum, qui peut être résolu en temps polynomial dans le cas général.

Couplage Un couplage d'un graphe est un ensemble d'arêtes de ce graphe qui n'ont pas de sommets en commun.

Couplage Parfait Un couplage parfait est un couplage tel que tout sommet du graphe est présent une fois et une seule dans le couplage.

Clique Une clique est un ensemble de sommets deux à deux adjacents.

2.6.2 Méthode de résolution

On remarque que dans le cas d'un graphe eulérien, il suffit d'appliquer l'algorithme d'Euler (voir algorithmes 5 et 6 pour avoir la chaine voulue.

Dans les autres cas, la méthode de résolution consiste à transformer le graphe en graphe eulérien, en ajoutant des arêtes. Une méthode possible est la suivante :

1. on crée d'abord le graphe partiel contenant uniquement les sommets de degré impair ;
2. on transforme ensuite ce graphe en clique : pour chaque couple de sommets non reliés entre eux, on crée une arête les rejoignant, de poids égal au cout le plus faible possible pour rejoindre ces sommets dans le graphe initial (ceci se calcule facilement avec l'algorithme de Dijkstra) ;
3. on cherche le couplage parfait de cout minimum en utilisant des algorithmes comme celui d'Edmonds. On peut aussi tester simplement toutes les possibilités ;
4. pour chaque arête de cet ensemble, on double la chaine la plus courte reliant les sommets reliés par cette arête dans le graphe initial ;
5. on obtient alors un graphe eulérien, sur lequel on applique l'algorithme d'Euler.

2.6.3 Exemple

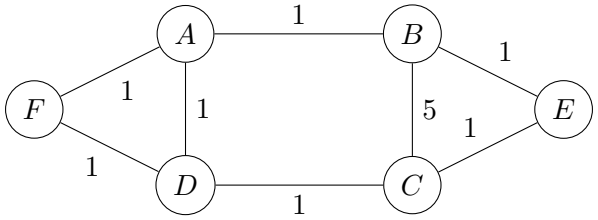
Prenons comme exemple le graphe ci-dessous, et essayons de trouver un parcours partant de A :

Entrées : Un graphe non-orienté et connexe g
Sorties : le cycle le plus court permettant de visiter toutes les arêtes de g
début
    `/* Création du graphe partiel */`
    pour *chaque sommet de g* **faire**
        si *le sommet est de degré impair* **alors**
            créer le sommet dans graphe_partiel;
        fin
    fin
    pour *chaque arête de g* **faire**
        si *ses 2 sommets sont dans graphe_partiel* **alors**
            créer la même arête dans graphe_partiel;
        fin
    fin
    `/* Transformation en clique */`
    pour *chaque couple de sommet (s1, s2) dans graphe_partiel* **faire**
        si *il n'y a pas d'arête reliant s1 et s2* **alors**
            $(cout, chaine) = dijkstra(s1, s2)$;
            créer l'arête reliant s1 et s2 dans graphe_partiel, de cout cout;
        fin
    fin
    $couplage, cout = aux(ensemblevide, ensemblevide, 0)$;
    pour *chaque arête dans couplage* **faire**
        $(s1, s2)$ = sommets reliés par arête dans g;
        $(cout, chaine) = dijkstra(s1, s2)$;
        pour *chaque arête dans chaine* **faire**
            doubler cette arête dans g;
        fin
    fin
    retourner *le cycle eulérien de g*
fin

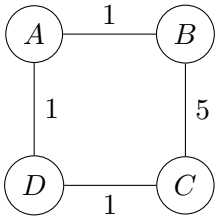
Algorithme 5 : Algorithme du postier chinois

```
Entrées : arêtes, sommets_visités, cout
début
    si sommets_visités contient tous les sommets de graphe_partiel alors
        retourner (arêtes, cout)
    fin
    sinon
        meilleur_couplage = Vide;
        meilleur_cout = 0;
        pour chaque arête de graphe_partiel faire
            si les 2 sommets de l'arête ne sont pas dans sommets_visités alors
                arêtes_copie = copie de arêtes;
                sommets_visités_copie = copie de sommets_visités;
                ajouter arête dans arêtes_copie;
                couplage, cout = aux(arêtes_copie, sommets_visités_copie, cout +
                cout de arête);
                si meilleur_couplage = Vide ou meilleur_cout > cout alors
                    meilleur_couplage = couplage;
                    meilleur_cout = cout;
                fin
            fin
        fin
        retourner (meilleur_couplage, meilleur_cout)
    fin
fin
```

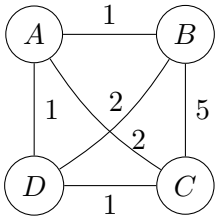
Algorithme 6 : Recherche du couplage parfait de cout minimum par bruteforce



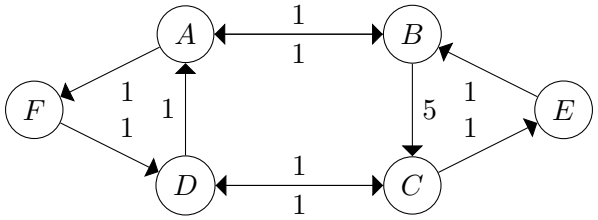
On créé le graphe partiel :



On le transforme en clique, avec les bons poids :



Nos couplages parfaits sont donc :
— $\{AB, DC\}$ de poids 2
— $\{AD, BC\}$ de poids 6
— $\{AC, BD\}$ de poids 4
On duplique donc les arêtes AB et DC, et on applique l'algorithme d'Euler :



2.7 Problème du voyageur de commerce

2.7.1 Analyse mathématique

Le problème du voyageur de commerce consiste à chercher un chemin passant par tous les sommets du graphe, de longueur minimale. Ce problème peut s'illustrer par l'exemple

d'une fraiseuse qui doit percer des trous dans une plaque le plus rapidement possible, ou encore par un car de touristes qui souhaiterait trouver l'itinéraire le plus rapide pour visiter un certain nombre de lieux.

On peut modéliser ce problème par un graphe complet, dont les arêtes ont un cout qui correspond à la distance entre chaque sommet, on cherche alors le cycle hamiltonien de cout minimal. On sait qu'un tel cycle existe car le graphe est complet.

Cependant trouver un tel cycle est un problème NP-complet : il n'existe donc pas d'algorithme efficace pour trouver ce cycle, à part une recherche exhaustive. En effet, la seule méthode exacte consisterait à tester toutes les chaines hamiltoniennes, et à prendre celle la plus courte, mais le nombre de chaines hamiltonniennes croît exponentiellement en fonction du nombre de sommets dans le graphe.

Nous allons donc nous concentrer sur les méthodes approchées de résolution, qui peuvent donner de très bons résultats tout en étant rapides. Toutefois, le résultat n'est donc pas forcément le plus court.

2.7.2 Heuristiques

Les heuristiques vont nous permettre de construire un chemin court (par rapport au plus court possible), de manière rapide, avec le moins de calcul possible. Étant donné qu'on est confronté à énormément de possibilités pendant la recherche, elles vont permettre d'orienter cette dernière, en faisant des choix les plus judicieux possibles sur les possibilités à explorer.

Exemple Une heuristique simple consiste à partir d'un sommet au hasard du graphe et d'aller au sommet le plus proche sur lequel on n'est pas encore passé (puis à retourner au sommet de départ pour boucler le cycle). Cet algorithme est en $O(n)$ et donc rapide. Mais il n'offre cependant aucune garantie de résultat, il existe même des graphes pour lesquels il donne le pire cycle.

Plus généralement, chercher parmi les p sommets les plus proches s'avère être une solution relativement efficace, avec une complexité en $O(p^n)$ (donc toujours exponentielle si $p \neq 1$).

Une méthode purement basée sur cette heuristique consisterait donc à parcourir tout le graphe, en allant sur le voisin le plus proche du sommet courant :

2.7.3 Recherche locale

Les heuristiques nous donnant des solutions acceptables, choisies avec un minimum de « bon sens », il est ensuite possible de tenter d'améliorer ces solutions, via de la recherche locale. Partant d'une solution fournie, on va explorer les solutions voisines à cette dernière, afin de voire si on pourrait pas trouver des solutions encore meilleures parmi leur voisinage.

Exemple Un algorithme de recherche locale adapté au problème du voyageur de commerce est le 2-opt (voir algorithme 8). Le principe du 2-opt consiste à tenter d'éliminer

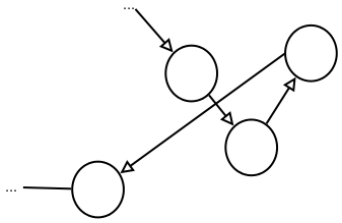
Entrées : un graphe complet g
Sorties : (cout, cycle) où cycle est un cycle hamiltonien construit selon la méthode du plus proche voisin et cout son cout associé sous forme de liste de sommets

```
début
    cout = 0;
    cycle = une liste composée d'un sommet de g au hasard;
    tant que il reste des sommets faire
        /* On ajoute au cycle le sommet suivant                    */
        plus_proche = sommet de g sur lequel on est pas encore passé le plus proche
        du dernier sommet du cycle;
        cout += cout de plus_proche au dernier sommet du cycle construit;
        ajouter plus_proche au cycle;
    fin
    /* On ferme le cycle                                           */
    cout += cout du dernier au premier sommet de cycle;
    ajouter le premier sommet de la chaine à cycle;
    retourner (cout, cycle)
fin
```

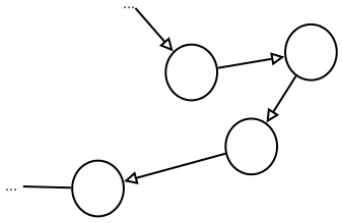
Algorithme 7 : Plus proche voisin

les « boucles » qui pourraient survenir dans le chemin, afin de le rendre plus court. [6]

Ainsi, partant du chemin suivant (qu'on obtiendrait logiquement en suivant l'heuristique consistant à aller sur le sommet le plus près) :



On obtiendrait ceci, en éliminant le croisement :



Il a donc une complexité quadratique du nombre de sommets du cycles.

Entrées : un cycle hamiltonien (liste de sommets) et son cout
Sorties : un cycle hamiltonien et son cout inférieur ou égal au coup d'entrée
début
  pour *chaque couple de sommets (a, b) dans le cycle* **faire**
    nouveau_cout ← cout
      − cout de a à son successeur dans le cycle
      − cout de b à son successeur dans le cycle ;
      + cout de a à b
      + cout du successeur de a au successeur de b dans le cycle
    si *nouveau_cout < cout* **alors**
      cout = nouveau_cout cycle = cycle crée en échangeant a et b dans cycle;
    fin
  fin
  retourner *(cout, cycle)*
fin

Algorithme 8 : 2-opt

L'application du 2-opt sur le chemin obtenu via une heuristique simple peut donner des résultats plus proches de la solution optimale qu'on pourrait le penser, et la combinaison des deux est donc une bonne méthode.

2.7.4 Métaheuristiques

Plutôt que d'utiliser une simple heuristique pour trouver une solution à priori plutôt bonne, puis d'y appliquer une recherche locale pour tenter de l'améliorer encore, il est possible d'utiliser des « métaheuristiques ». Ces algorithmes vont avoir besoin d'heuristiques et de recherche locale, mais vont s'en servir en boucle, pour tenter sans cesse de trouver une solution meilleure.

Ils vont partir explorer différentes parties de l'espace, souvent en guidant leur exploration grâce à l'heuristique, et en essayant de retomber sur des parties de l'espace les plus intéressantes possibles grâce aux algorithmes de recherche locale.

Il existe énormément de métaheuristiques. En voici quelques uns :

Recherche locale itérée métaheuristique très simple consistant à utiliser une heuristique puis appliquer de la recherche locale pour améliorer son résultat. Ensuite, on perturbe légèrement ce résultat, on applique à nouveau une recherche locale et on recommence.

Recherche tabou amélioration de la recherche locale itérée, qui va utiliser une « liste taboue » bannissant toute recherche autour des zones de l'espace déjà explorées.

Recuit simulé explore d'abord l'espace sans se restreindre aux parties donnant des solutions efficaces, puis se restreint de plus en plus au voisinage de celles-ci. Converge donc vers les solutions les plus efficaces trouvées, puis relâche les contraintes et

explore autour de ces dernières, quitte à trouver des solutions vraiment moins efficaces. Recommence à se contraindre aux plus efficaces, etc...

Algorithmes génétiques imitent la sélection naturelle, avec une population de solutions qui évoluent en mutant et en s'échangeant leurs caractéristiques entre elles. On peut même faire évoluer des populations séparément avec les modèles en îles, pour avoir plusieurs populations très différentes.

Colonies de fourmis imitent là encore la nature en simulant des phéromones déposées par des fourmis virtuelles, qui orientent la recherche au fil du temps.

2.7.5 Conclusion

Il est intéressant de constater que les heuristiques, les méthodes de recherche locales et les métaheuristiques sont des choses extrêmement générales, utilisées pour résoudre énormément de problèmes demandant d'explorer un espace extrêmement grand.

Elles ne sont donc pas propres au voyageur du commerce, même si on a vu comment, dans ce cas précis, on pouvait obtenir des résultats corrects en se passant de métaheuristiques. On pourrait donc améliorer ces résultats en en utilisant.

3 UA : Programmation linéaire

3.1 Introduction

La programmation linéaire, ou optimisation linéaire, consiste à maximiser (ou de manière équivalente minimiser) une fonction linéaire sur un polyèdre convexe (dont un cas particulier courant est sous des contraintes linéaires).

3.2 Problème du sac à dos

3.2.1 Présentation du problème

Ce problème paraît simple en apparence : nous avons un ensemble d'objets, chaque objet pouvant avoir une masse différente et ayant une certaine valeur, et nous voulons remplir un sac à dos de manière à maximiser la valeur totale, sans dépasser une certaine masse maximale.

Résoudre ce genre de problème est utile par exemple en gestion de portefeuilles pour trouver le meilleur rapport entre rendement et risque, ou en découpe de matériaux, pour minimiser les chutes.

Ce problème est un problème d'optimisation linéaire, en effet, cela revient à résoudre le problème :

$$\max_{i \in S' \subset S} v_i \quad \Bigg| \quad \sum_{i \in S'} m_i \leq W$$

où S est l'ensemble des objets, S' est un ensemble de n objets choisis, v_i la valeur de l'objet $i \in S$, m_i sa masse et W la masse maximale autorisée dans le sac.

Cependant la résolution de ce problème n'est pas simple : déterminer s'il est possible de dépasser une valeur minimale sans dépasser le poids maximal est un problème NP-complet.

3.2.2 Résolution exacte

Une exploration exhaustive de l'ensemble des parties de S n'est pas très réaliste, car celui-ci est de cardinal $2^{\text{card}S}$.

Mais, ce problème peut être résolu en utilisant la programmation dynamique [1]. En

1. Qui consiste à résoudre un problème de taille n à partir de la résolution d'un problème de taille $n-1$

effet, on peut déterminer si un objet i fait partie de l'ensemble des objets à choisir en considérant le problème sur l'ensemble $S\backslash\{i\}$ et la masse maximale $W - m_i$.

Toutefois, un tel algorithme (voir algorithme 9) fonctionne uniquement si les poids des objets sont des entiers. De plus sa complexité en temps est en $O(nW)$ et celle en mémoire en $O(W)$ [2].

```
Entrées : une liste d'objets
Entrées : une masse maximale autorisée
Sorties : la valeur maximale qu'il est possible d'atteindre
début
    ligne_courante = liste composée de (masse_max+1) zéros;
    ligne_prec = liste composée de (masse_max+1) zéros;
    pour chaque objet obj de la liste d'entrée faire
        pour m variant de 0 à masse_max faire
            si masse(obj) ≤ m alors
                ligne_courante[m] ← max(ligne_prec[m],
                ligne_prec_line[m-masse(obj)] + prix(obj));
            fin
        fin
        ligne_prec ← ligne_courante
    fin
    retourner ligne_courante[masse_max]
fin
```

Algorithme 9 : Algorithme de résolution exacte du problème du sac à dos

3.2.3 Résolution approchée

Un autre algorithme pour résoudre ce problème, dit algorithme glouton, consiste simplement à choisir les « meilleurs » objets jusqu'à que la masse maximale soit dépassée. Le critère déterminant quels sont les meilleurs objets pourrait être la masse faible, le prix élevé, ou le rapport prix/masse élevé.

Cet algorithme est beaucoup plus rapide que le précédent (il a une complexité en temps de $O(n \log n)$ (pour le tri des objets)) et ne nécessite en mémoire que la liste des objets, mais ce n'est qu'un algorithme approché. Les résultats obtenus sont cependant très satisfaisant, en effet en considérant le ratio prix/masse, on obtient des résultats très proches de l'optimum (quelques pourcents d'erreur relative en moyenne, mais aucune garantie n'est fournie : il peut même fournir la pire solution).

De plus il peut être utilisé quand les masses ne sont pas entières.

2. En pratique on pourrait l'utiliser sur des masses non-entières en les multipliant, ce qui augmenterait la complexité du même facteur. De plus on peut réduire la complexité temporelle en $O(nW')$ avec $W' = \frac{W}{\text{ppcm(toutes les masses)}}$.

Un autre algorithme pour résoudre ce problème est inspiré du comportement des fourmis : une "fourmi" se promène plus ou moins au hasard dans l'ensemble des possibilités en marquant les objets choisis comme intéressants (comme les fourmis le font avec les phéromones). Ces fourmis vont essayer de choisir de plus en plus souvent les objets ayant souvent été marqués intéressants. On s'arrête après un certain nombre de tours ou quand les nouvelles itérations ne sont plus considérés suffisamment intéressantes.

3.2.4 Relation avec le voyageur de commerce

Il est intéressant ici de faire le lien avec le problème du voyageur du commerce : les solutions utilisées ici sont *les mêmes*.

En effet, on a vu que ce problème (avec des poids non entiers, donc impossible à résoudre par programmation dynamique) se résout seulement par une recherche exhaustive.

On a aussi vu comment le résoudre de manière approchée via une heuristique simple, et l'algorithme des fourmis est une métaheuristique qu'on a mentionnée précédemment.

Plus intéressant encore, on peut même construire une « bijection » entre un problème du sac à dos et un problème du voyageur de commerce sur un graphe ! [7] Ceci démontre que ces deux algorithmes sont équivalents, et appartiennent donc à la même classe d'algorithmes.

3.3 Problème d'optimisation linéaire

Le but du problème est de maximiser une fonction linéaire sous certaines contraintes, linéaires elles aussi.

3.3.1 Point de vu mathématique

Considérons le problème suivant :

$$(P) \quad \max_{x \in C \subset \mathbb{R}^n} f(x)$$

Nous nous placerons dans le cas où f est linéaire, où $x \geqslant 0$, et où C est décrit par des contraintes d'inégalités linéaires, c'est-à-dire qu'il existe une matrice A et un vecteur b tels que $Ax \leqslant b$.

Existence de solutions Pour un tel problème, trois possibilités s'offrent à nous :
- les contraintes sont incompatibles ;
- la fonction est non majorée sur C ;
- le problème admet un maximum sur C.

Nous savons de plus que C est un polyèdre convexe. Un théorème garantit alors que si ce problème à une solution, alors il a une solution en un de ses sommets. Nous allons donc chercher les solutions parmi les sommets de C.

3.3.2 Algorithme du simplexe

Le principe de cet algorithme est de considérer un des sommets du polyèdre, puis de se déplacer en suivant les arêtes de ce polyèdre en augmentant à chaque itération le gain. L'algorithme se terminera lorsque nous nous trouverons sur un sommet, dont tous les sommets adjacents présentent un gain plus faible. La convexité du polyèdre nous garantit que le résultat est optimal.

L'algorithme du simplexe a une complexité dans le pire des cas exponentielle, mais en pratique, cet algorithme est efficace.

Cet algorithme ne permet pas de maximiser une fonction pour des variables entières (par exemple pour connaitre un nombre de produits à produire, donc un nombre entier) à produire pour maximiser un gain (bien qu'on pourrait en pratique l'utiliser en considérant que la solution optimale entière est suffisamment proche de la solution optimale réelle).

Forme standard et tableau canonique Pour résoudre le problème, la première étape est le mettre sous forme standard. Pour cela on ajoute à chaque inéquation j de la forme $\sum a_{j,i}x_i \leq 0$ une variable dite d'écart pour la transformer en égalité : $\sum a_{j,i}x_i + s_j = 0$ où $s_j \geq 0$.

Les inéquations de la forme $\sum a_i x_i \geq 0$ sont d'abord multipliées par -1 avant cette étape.

On construit ensuite un tableau dit canonique représentant le problème comme suit :

- la première ligne de la matrice est $[m_0, m_1, \cdots, m_n, 0, \cdots, 0]$ où les (m_i) sont les coefficients du problème $\min \sum m_i x_i$ et à laquelle on ajoute autant de 0 qu'on a ajouté de variables d'écart ;
- les autres lignes de la matrice sont $[a_{j,0}, \cdots, a_{j,n}, 0, \cdots, 0, 1, 0, \cdots, 0]$ où les 1 sont placés de manière à former une matrice identité (ils correspondent aux variables d'écart ajoutées).

```
Entrée : matrice (un tableau canonique)
Sortie : le résultat optimum
         les quantités de chaque produit à produire
         les quantités de chaque ressource restante

base = indices des variables de base de la matrice

tant qu'il reste des nombres strictement positifs sur la première ligne:
    à_ajouter = indice de la colonne dont le premier élément est maximal
    à_retirer = indice de la ligne (>1) telle que :
                matrice[à_retirer, dernière colonne]/matrice[à_retirer, à_ajouter]
                est minimum

    remplacer le (à_retirer)ième élément de base par à_ajouter

    diviser la ligne à_retirer de matrice par matrice[à_retirer, à_ajouter]
    et soustraire aux autres lignes y le vecteur :
        matrice[y,à_ajouter] * matrice[à_retirer,] / matrice[à_retirer, à_ajouter]

pour chaque variable d'origine n (n dans [0, nombre de variables hors base[):
  à_produire[base[n]] = matrice[n+1, dernière colonne]
```

```
pour chaque variable d'écart n (n dans [nombre de variables hors base, nombre de
    variables total[):
  restes[base[n]] = matrice[n+1, dernière colonne]

retourner (-matrice[0, dernière colonne], à_produire, restes)
```

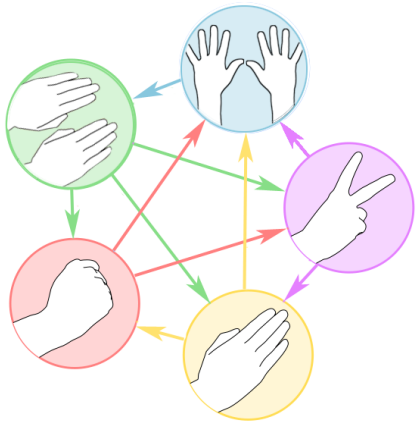
Dégénérescence Un problème du simplexe est dit dégénéré si plus de deux contraintes vont devoir être nulles en un sommet. Graphiquement, cela veut dire qu'au moins 3 droites vont se rencontrer en un sommet du polyèdre.

Ceci va empêcher l'algorithme du simplexe de progresser entre deux itérations : il va simplement changer de base. Le problème étant que sur des cas particuliers, il pourra changer de base sans progresser, puis boucler à l'infini en faisant un cycle sur des bases qui n'améliorent pas la solution.

Pour éviter cela, on pourrait utiliser des règles d'anti-cyclage, dont la règle de Bland, qui consiste à choisir judicieusement les variables qu'on fera entrer et sortir de la base, dans le cas où il y aurait plusieurs possibilités aussi intéressantes les unes que les autres.

4 UA : Jeux

4.1 Shifumi



Une stratégie simple et efficace à laquelle on pourrait penser pour gagner au Shifumi serait de jouer de manière aléatoire.

Et en effet, il s'avère que si les deux joueurs jouent de manière équiprobable, on a affaire à un équilibre de Nash : aucun changement de stratégie de la part d'un joueur ne pourra lui permettre d'augmenter ses chances de gagner.

De plus, si un adversaire ne joue pas de manière aléatoire (ou augmente la probabilité de jouer un certain élément), alors on pourra prévoir ce qu'il va jouer et donc trouver une stratégie qui pourra le battre. Les humains étant très mauvais pour jouer de manière aléatoire, il est assez facile d'écrire une stratégie permettant de les battre.

4.1.1 Chaines de Markov

Une autre stratégie se base sur des chaines de Markov : en se basant sur les derniers éléments joués, elle regarde dans l'historique pour voir l'élément qui était joué le plus souvent par l'adversaire après les derniers coups joués.

Cette stratégie s'avère vraiment efficace contre un joueur humain. Toutefois, elle est prévisible : si on sait qu'on a affaire à une telle stratégie, on peut jouer de manière à la battre.

C'est pour cela qu'une stratégie aléatoire est la seule pouvant maximiser nos gains dans le pire des cas.

4.1.2 Variantes

Toutes les variantes du Shifumi qui consistent à rajouter des éléments pour obtenir un nombre d'éléments pair (par exemple pierre/papier/ciseaux/puits) vont créer un déséquilibre, car un élément sera moins efficace contre les autres. L'équilibre de Nash du jeu va alors consister à ne jamais jouer cet élément (et jouer aléatoirement parmi les autres).

Si le nombre d'éléments est impair, alors le jeu pourra être équilibré, comme un Shifumi classique.

4.2 Jeu de la somme magique

Ce jeu consiste à choisir, à tour de rôle, n nombres parmi n^2 afin que leur somme soit égale à $\frac{n(n^2+1)}{2}$.

4.2.1 Représentation sous forme de morpion

Une représentation possible de ce jeu est le carré magique : les joueurs doivent choisir, l'un après l'autre une case dans un carré magique, leur but étant de contrôler une ligne, une colonne ou une diagonale entière du carré magique ; alors, les nombres qu'ils auront choisis totaliseront le score voulu. De même, ce problème correspond exactement au jeu du morpion, étendu à des grilles $n \times n$.

Ainsi, une des stratégies possibles pour un joueur du jeu de la somme magique est de construire un carré magique, et de représenter les nombres choisis par l'adversaire par un rond dans la case correspondante. Afin de choisir un nombre, il suffit d'appliquer la stratégie de morpion de son choix sur le carré magique, et de jouer le nombre correspondant.

Le choix du carré magique n'importe pas. En effet, dans un carré magique sont présentes toutes les possibilités de combinaison de nombres pour obtenir la somme voulue. Par conséquent, peu importe le carré magique d'ordre n que l'on choisit, les représentations sous forme de morpion seront toutes équivalentes.

Le morpion étant un jeu où l'on essaie de minimiser la perte maximum, on peut s'intéresser à l'algorithme du minimax, pour déterminer une stratégie non perdante.

4.2.2 Algorithme du minimax appliqué au jeu du morpion

L'algorithme du minimax consiste à évaluer toutes les positions de jeu atteignables depuis la position courante, sur une certaine profondeur (autrement dit, un certain nombre de tours de jeu), et à jouer de manière à atteindre la position la plus avantageuse, en supposant que l'adversaire joue toujours le meilleur coup pour lui-même (ce coup étant évalué avec notre propre fonction d'évaluation, qui n'est pas forcément la même que celle de l'adversaire).

Par conséquent, afin d'implémenter l'algorithme du minimax, il faut commencer par déterminer une fonction d'évaluation.

Fonction d'évaluation La fonction d'évaluation que nous avons choisie est très simple : une ligne, colonne ou diagonale (que nous appelleront désormais simplement "ligne") complétée avec notre symbole (ce qui signifie qu'on a gagné) vaut $+\infty$; si, au contraire, l'adversaire a complété une ligne, alors cette ligne vaut $-\infty$. Une ligne contenant uniquement notre symbole rapporte le nombre d'occurrences de notre symbole dans cette

ligne ; à l'inverse, une ligne contenant uniquement le symbole de l'adversaire rapportera négativement le nombre. Toutes les autres lignes ne rapportent aucun point. Ainsi, l'évaluation d'une position de jeu est la somme des points rapportés par chacune de ses lignes, colonnes et diagonales.

Minimax L'algorithme du minimax va construire (implicitement) l'arbre des coups possibles à partir de la position courante. L'évaluation d'un nœud de cet arbre sera :

— si c'est à notre tour de jouer, le maximum de l'évaluation de nos fils ;
— si c'est à l'adversaire de jouer, le minimum de l'évaluation de ses fils (i.e. on suppose que l'adversaire joue le meilleur coup à sa disposition, selon la fonction d'évaluation du joueur courant).

L'évaluation d'une feuille de l'arbre se fera par la fonction d'évaluation définie précédemment.

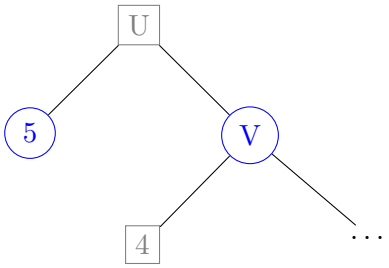
Ainsi, l'algorithme se dirigera naturellement vers la position la plus avantageuse pour lui.

4.2.3 Élagage alpha-bêta

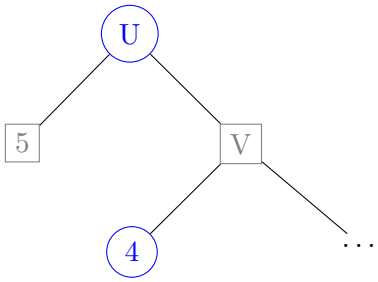
L'élagage alpha-bêta permet de réduire le nombre de nœuds à parcourir durant l'algorithme du minimax.

Cet algorithme arrête le parcours des fils d'un nœud quand il se rend compte qu'il ne pourra pas faire mieux.

Dans l'exemple suivant, où les nœuds en bleus sont ceux où l'on doit prendre le minimum, et ceux en gris le maximum :



Coupure Alpha



Coupure Béta

On se rend bien compte qu'on n'a pas besoin de parcourir les nœuds suivant, car on prend le maximum des minimums, ou l'inverse.

Par exemple, pour la coupure alpha : si on trouve des fils de V plus petit que 4, on va prendre le maximum, donc c'est 4 qui sera utilisé, et si on trouve plus grand ou égal à 5, on devra prendre le minimum au niveau de V, donc on prendra 5 dans tous les cas.

Au final, cette amélioration permet de gagner un temps considérable pour obtenir le même résultat.

4.3 Duopôles

Les jeux précédents sont des jeux non coopératifs à somme nulle (on peut soit gagner, soit perdre).

Il peut aussi être intéressant de voir les jeux à somme non nulle, c'est à dire qu'il n'y a pas forcément un gagnant (1 point) et un perdant (-1 point), mais il est possible de faire des situations plus ou moins gagnant-gagnant, voire même perdant-perdant.

Le duopôle de Cournot est un bon exemple de ce genre de jeu : il met en jeu deux entreprises qui produisent des biens sur le même marché. Elles se font donc concurrence, mais plus le marché est inondé de produits, plus leur marge est faible (elles peuvent même faire des pertes !).

On modélise le cout de production par une fonction affine :

$$c_E(x) = sigma + gamma * x$$
$$c_F(y) = sigma + gamma * y$$

Ainsi, il y a un cout fixe qui modélise l'acquisition de matériel, et un cout directement proportionnel à la quantité de produits fabriqués.

Et le prix de vente est une fonction affine décroissante : $p(z) = \alpha - \beta * (x + y)$. Il est bien évidemment commun aux deux entreprises.

4.3.1 Stratégies possibles

Coopération Il est possible de coopérer complètement avec l'adversaire, en partant du principe que s'il produit autant que nous, on maximisera le gain commun de chaque entreprise. Cette stratégie permet de très bons gains avec toute autre stratégie qui coopère, mais elle se fera souvent écraser par des stratégies qui vont « profiter d'elle ».

Stackelberg TODO

Œil pour œil, dent pour dent Le principe de cette stratégie est d'être coopérative tant que l'adversaire l'est, et de devenir plus agressive quand il ne l'est plus : à chaque fois que l'autre joueur n'est pas coopératif, on joue comme le ferait la stratégie Stackelberg.

Une variante de cette stratégie consiste à le pénaliser de plus en plus : la première fois on le pénalise une fois, puis deux, puis trois, etc.

Ces deux stratégies sont efficaces à la fois quand l'autre joueur est coopératif (on est alors coopératif) et contre un joueur non-coopératif (on devient alors agressif).

5 UA : Processus stochastiques

5.1 Introduction – Chaines de Markov

Un processus stochastique représente l'évolution au cours du temps d'une variable aléatoire. Dans cette section, nous étudierons particulièrement les processus de Markov, qui sont des processus stochastiques dont l'état futur ne dépend pas des états passés.

Enfin, une chaine de Markov est un processus de Markov à temps discret. On peut la représenter sous forme d'un graphe, dont les sommets représentent des états et les arêtes des transitions d'un état à l'autre.

5.2 Colonie de scarabées

Le problème consiste, à partir d'un graphe dont les nœuds représentent des positions et les arêtes contiennent la probabilité pour un scarabée de passer d'une position à l'autre, à calculer les probabilités de présence de chaque scarabée au bout de N itérations, selon sa position de départ.

Pour cela, on représente le graphe sous la forme de matrice d'adjacence, où chaque élément de la matrice représente la probabilité de passer d'un nœud à un autre.

Si on met cette matrice à la puissance N, elle contiendra les probabilités de passer d'un nœud à un autre en exactement N itérations.

En multipliant cette matrice par un vecteur contenant uniquement des zéros, sauf au nœud de départ (on mettra un 1), on peut obtenir la probabilité pour le scarabée de se trouver à chaque point, au bout d'exactement N tours. Si cette probabilité est nulle, il est impossible qu'il s'y trouve.

Pour savoir la probabilité que les deux scarabées se rencontrent en un point au bout de N itérations, il suffit de multiplier les probabilités de présence de chaque scarabée en ce point. Pour connaitre leur probabilité de se rencontrer en n'importe quel point, on peut tout simplement sommer les probabilités de rencontre sur chaque point.

De plus, si on calcule $\lim_{N\to\infty} A^N$ (une telle limite n'existera que si la chaine est irréductible, récurrente positive et apériodique, et ça ne sera pas toujours le cas !), le produit du vecteur avec la position de départ par cette matrice nous donnera les probabilités pour le scarabée d'être en chaque position une fois qu'il aura suffisamment voyagé et que sa position de départ n'aura plus d'importance.

5.2.1 Chaines de Markov

Le problème des scarabées peut aisément être modélisé par des chaines de Markov. En effet, nos coléoptères se promènent en temps discret dans leur graphe de promenade ; de plus, l'état d'un scarabée (autrement dit, sa position dans sa promenade) ne dépend pas du passé, mais uniquement de l'état présent.

La matrice de transition de notre promenade représente les probabilités de passer d'un état aux autres. Par conséquent, la matrice de transition à la puissance n représente la probabilité de passer d'un état à un autre par un chemin de longueur n.

On constate qu'on se rapproche beaucoup de la matrice décrite précédemment : la matrice d'adjacence du graphe de promenade et la matrice de transition de notre chaine de Markov sont équivalentes !

5.2.2 Temps moyen entre rencontres

À partir d'un certain point Considérons que les deux scarabées sont sur le même point. On peut donc calculer la probabilité qu'après N tours, ils finissent à nouveau sur la même case (en sommant le carré des probabilités que chacun se retrouve sur une case).

Notons $U_k = pA^k(pA^k)^\intercal$ la probabilité que les deux scarabées se retrouvent sur la même case après k tours.

Notons V_k la probabilité que les deux scarabées se retrouvent sur la même case après k tours, sans s'être déjà rencontrés avant. On a :

$$V_k = (1 - U_1) \times (1 - U_2) \times \cdots \times (1 - U_{k-1}) \times U_k$$

V_k peut donc se réécrire par récurrence :

$$\begin{aligned} V_1 &= U_1 \\ V_{k+1} &= V_k \times \frac{1 - U_k}{U_k} U_{k+1} \end{aligned}$$

Pour avoir le temps moyen au bout duquel ils vont se rencontrer en partant de ce même point, il suffit de prendre l'espérance de cette suite V_k.

À partir de deux points différents Dans le cas où les points de départs des scarabées sont différents, on peut appliquer le même principe.

Par contre, on ne peut pas supposer $U_k \neq 0$. On peut introduire une suite W_k :

$$\begin{aligned} W_0 &= 1 \\ W_k &= W_{k-1} \times (1 - U_k) \end{aligned}$$

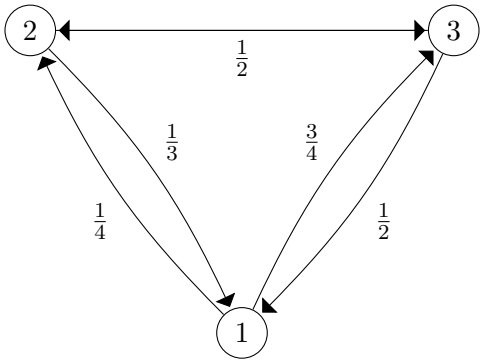
On a donc :

$$\begin{aligned} V_1 &= U_1 \\ V_k &= W_{k-1} \times U_k \end{aligned}$$

Pour avoir un temps moyen de rencontre global, on fait la moyenne du temps de rencontre pour chaque position initiale possible.

5.2.3 Exemple

Partons du graphes suivant :



La matrice d'adjacence de ce graphe est :

$$A = \begin{pmatrix} 0 & \frac{1}{4} & \frac{3}{4} \\ \frac{1}{3} & \frac{1}{6} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}$$

Si on veut connaitre la probabilité qu'un scarabée se trouve au sommet 2 en partant du sommet 1 sur 10 tours, on calcule :

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \times \begin{pmatrix} 0 & \frac{1}{4} & \frac{3}{4} \\ \frac{1}{3} & \frac{1}{6} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \end{pmatrix}^{10} \approx \begin{pmatrix} 0.30188 \\ 0.32244 \\ 0.37568 \end{pmatrix}$$

La probabilité voulue est donc 0.32244.

De plus, on remarque que A^n converge vers

$$\begin{pmatrix} 0.29787 & 0.31915 & 0.38298 \\ 0.29787 & 0.31915 & 0.38298 \\ 0.29787 & 0.31915 & 0.38298 \end{pmatrix}$$

lorsque $n \to +\infty$, donc quelque soit la position de départ du scarabée, après un nombre suffisant de tours, la probabilité qu'il soit sur le sommet 2 est 0.31915.

5.3 Théorie des files d'attente

Une file d'attente est un système ayant une entrée par laquelle arrivent des tâches, ou client, qui attendent leur tour avant d'être traitée une pour en sortir.

Elles sont utilisées pour l'étude de beaucoup de systèmes : attentes de clients à des guichets, trafic routier, traitement des tâches par un serveur, etc.

Les files sont étudiées en fonction de la loi qui gère l'entrée, la loi qui gère la sortie et le nombre de "serveurs" qui traitent les clients.

La loi de Little est cependant un résultat général :

$$N = \lambda T_s$$

où N est le nombre moyen de clients dans le système, λ la fréquence moyenne d'entrée et T_s le temps moyen passé dans le système.

5.3.1 Cas M/M/1

On considère une file d'attente où la loi d'entrée suit une loi de Poisson de paramètre λ (λ est dont le nombre moyen d'arrivées par unité de temps) et que le temps de service suit une loi exponentielle de paramètre μ ($\frac{1}{\mu}$ est donc le temps moyen de traitement, sans compter l'attente dans la file).

On dit que ce genre de file d'attente est de type $M/M/1$ selon les notations de Kendall.

Dans ce cas, le nombre moyen d'arrivées pendant le temps de service, appelé trafic offert, est $\rho = \frac{\lambda}{\mu}$. Le système converge alors si et seulement si $\rho < 1$.

La file peut être représentée comme une chaine de Markov où chaque état i représente la probabilité qu'il y ait i voitures dans le système (voir figure 1).

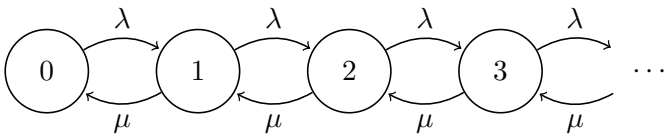


FIGURE 1 – Représentation d'une file M/M/1 comme une chaine de Markov [5]

Ces états sont liés en régime permanent par les équations :

$$\begin{cases} \lambda p_0 = \mu p_1 \\ \lambda p_1 = \mu p_2 \\ \quad \dots \\ \lambda p_n = \mu p_{n+1} \\ \quad \dots \end{cases}$$

et

$$\sum_{i=0}^{+\infty} p_n = 1$$

D'où $p_n = \left(\frac{\lambda}{\mu}\right)^n p_0 = \left(\frac{\lambda}{\mu}\right)^n \left(1 - \frac{\lambda}{\mu}\right) = \rho^n(1-\rho)$ (d'où la condition $\rho < 1$).

Le nombre moyen de clients dans le système est donc :

$$N = \sum_{i=0}^{n} n p_n = \frac{\rho}{1-\rho}$$

D'après la loi de Little, le temps moyen passé dans le système est donc :

$$T_s = \frac{1}{\lambda} \frac{\rho}{1-\rho} = \rho \frac{\lambda}{\mu - \lambda}$$

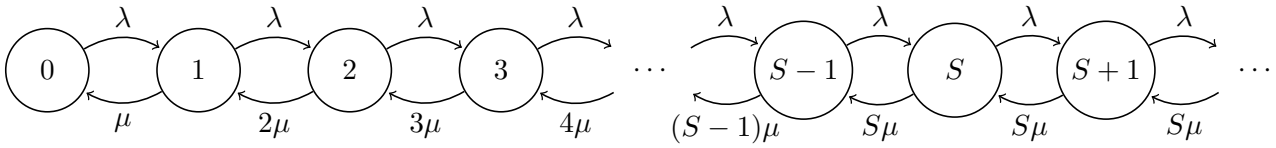


FIGURE 2 – Représentation d'une file M/M/S comme une chaine de Markov [5]

5.3.2 Cas M/M/S

Une généralisation du cas précédent sont les files de type $M/M/S$, où S dénote le nombre de serveurs. Les lois d'entrée et sortie restent les mêmes.

Ces files peuvent aussi être représentées avec les chaines de Markov (voir figure 2).

L'étude de ces chaines est plus compliquée car les états ne sont plus liés par la même équation :

$$\begin{cases} \lambda p_0 = \quad \mu p_1 \\ \lambda p_1 = 2 \times \mu p_2 \\ \quad \dots \\ \lambda p_{S-2} = (S-1) \times \mu p_{S-1} \\ \lambda p_{S-1} = S \times \mu p_S \\ \lambda p_S = S \times \mu p_{S+1} \\ \lambda p_{S+1} = S \times \mu p_{S+2} \\ \quad \dots \end{cases}$$

On peut montrer que le système est en équilibre si et seulement si $\rho = \frac{\lambda}{S\mu} < 1$. Le nombre moyen de clients et temps moyen dans le système sont par contre beaucoup moins intuitifs :

$$N = \rho \left(1 + \frac{P_a}{S - \rho}\right)$$

$$T_s = \frac{1}{\mu} \left(1 + \frac{P_a}{S - \rho}\right)$$

où $P_a = P_0 \dfrac{\rho^S}{(S-1)!(S-\rho)}$ est la probabilité d'attendre et $P_0 = \dfrac{1}{\sum_{k=0}^{S-1} \frac{\rho^k}{k!} + \frac{\rho^S}{S!} \frac{1}{1-\rho/S}}$ est la probabilité que le système soit vide.

On remarque qu'on retrouve bien les mêmes résultats que précédement pour $S = 1$.

6 UA : Ingénierie robotique

6.1 Suivi de murs

On s'intèresse ici à la programmation d'un robot [3] afin qu'il puisse longer un mur.

Afin de longer un mur, on peut par exemple poser un capteur à ultrasons sur le côté gauche de notre robot. Lorsque nous détectons que nous sommes trop éloignés du mur, nous tournons légèrement à gauche.

Cette méthode donnant des résultats peu probants si la direction initiale du robot n'est pas exactement parallèle au mur, on peut ajouter un autre capteur à ultrasons à l'avant du robot. Ainsi, on peut tourner désormais à gauche, tout en avançant légèrement, dès qu'on détecte que le mur est trop loin ; et si on se retrouve face à notre mur, alors on tourne vers la droite jusqu'à ne plus l'être.

Ainsi, on peut suivre un mur droit de manière assez régulière.

6.2 Labyrinthe

6.2.1 Suivi de mur

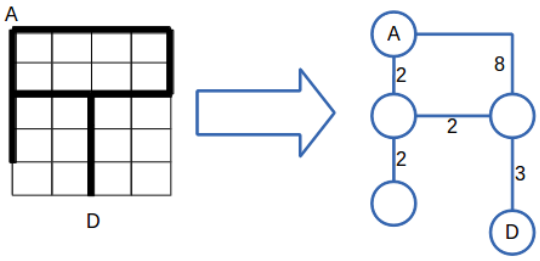
Le suivi de mur est déjà un algorithme admissible pour la résolution de labyrinthe, il trouvera une solution s'il en existe une. L'algorithme de Dijkstra et l'algorithme A* (qui est considéré comme une extension de premier) sont utilisés pour trouver le chemin le plus court.

6.2.2 Algorithme de Dijkstra

L'algorithme de Dijkstra est un algorithme permettant de trouver le plus court chemin entre deux sommets d'un graphe (orienté ou non). Sa complexité est polynomiale.

Son principe est simple : il parcourt le graphe en gardant en mémoire la longueur du chemin le plus court vers chaque sommet.

Cet algorithme est appliquable une fois qu'on a obtenu un graphe a partir du plan du labyrinthe.



3. Un robot en Lego Mindstorms NXT, nous ne disposons de plus que de deux capteurs à ultrasons et deux boutons poussoirs.

```
Entrées : deux sommets s1 et s2
Sorties : le plus court chemin entre s1 et s2
début
    pour chaque sommet du graphe faire
        sommet.parcouru ← ∞;
        sommet.precedent ← 0;
    fin
    s1.parcouru ← 0;
    non_visités ← ensemble des sommets du graphe;
    tant que non_visités est non vide faire
        s ← le sommet de non_visités avec parcouru minimum;
        supprimer s de non_visités;
        pour chaque sommet f fils de s faire
            si f.parcouru > s.parcouru + poids de l'arc entre s et f alors
                f.parcouru ← s.parcouru + poids de l'arc entre s et f;
                f.precedent ← s;
                ajouter f dans non_visités;
            fin
        fin
    fin
    chaine ← liste vide;
    s = s2;
    tant que s ≠ s1 faire
        chaine.ajouter(s);
        s ← s.precedent;
    fin
    chaine.ajouter(s1);
    retourner chaine
fin
```

Algorithme 10 : Algorithme de Dijkstra

On peu appliquer l'algorithme de Dijkstra tel qu'il a été présenté dans la première partie "Graphes" à ce graphe et trouver le chemin le plus court.

6.2.3 Algorithme A*

L'algorithme A* est une amélioration de Dijkstra, qui utilise une heuristique pour orienter la recherche. Si l'heuristique est bien choisi, il est même possible de s'assurer qu'A* donnera bien le résultat optimal (mais plus rapidement que Dijkstra).

Cet algorithme utilise deux listes de nœuds : une liste, dite ouverte, qui contient les nœuds explorables et une liste, dite fermée, qui contient les nœuds déjà explorés.

— On crée un nœud en lui attribuant un cout heuristique qui correspond au cout du nœud plus une estimation de la distance de ce nœud à l'arrivée.
  Remarque : l'utilisation de ce cout heuristique est une différence notable avec l'algorithme de Dijkstra, elle permet de s'assurer que l'on va toujours plus ou moins dans la bonne direction.
  Par exemple dans un graphe en étoile avec des branches de même taille et plusieurs nœuds par branches ou l'on part du centre pour rejoindre l'extrémité d'une branche l'algorithme de Djkstra explore toutes les branches simultanément alors qu'avec l'utilisation du cout heuristique on explore directement et uniquement la bonne branche.
— On ajoute ce nœud à la liste ouverte.
— On prend le nœud qui a le meilleur cout heuristique dans la liste ouverte et on l'ajoute à la liste fermée.
— On crée les nœuds adjacents et pour chacun d'eux :
  — Leur cout est égal à la somme des couts de leurs prédécesseurs et du cout entre les deux.
  — Si l'un d'eux est présent dans la liste ouverte on vérifie si ce nouveau chemin trouvé est plus rapide. Si c'est le cas on remplace celui qui est dans la liste ouverte par le nouveau sinon on oublie le nouveau.
  — S'il est déjà dans la liste fermée c'est qu'il a déjà été traité ou qu'il est en train d'être traité donc on l'oublie.
  — Et si il n'est ni dans la liste ouverte ni dans la liste fermée on l'ajoute à la liste ouverte.

À la fin, en remontant tous les prédécesseurs on remonte le chemin le plus court.

7 Bilan

7.1 Enseignement

Nous avons étés surpris de la façon dont fonctionnent les séances de TD, et avons eu du mal à nous y faire : lors des séances, nous étions censés être lâchés sur les différents problèmes et nous renseigner par nous-mêmes. Toutefois, le professeur présent se mettait souvent à présenter des choses au tableau, et nous ne savions plus trop si nous devions avancer, ou stopper tout travail et l'écouter.

Il a aussi été assez frustrant de se répartir le travail : comme nous étions dans un groupe de 6 et avions quand même pas mal de choses à faire, il fallait forcément se concentrer sur un seul problème. Du coup, chacun a simplement abordé quelques points précis sur chaque UA, et n'a pas vraiment acquis de connaissances poussées sur tout le reste des sujets abordés.

Le fait de devoir faire des rapports à chaque séance était aussi très lourd à gérer, et assez peu pratique : nous avions l'impression de passer énormément de temps à écrire des rapports de séances, rapports d'UA... Ajouté au rapport final, cela donne un travail de rédaction très important.

7.2 Travail en groupe

L'utilisation d'un gestionnaire de version nous a permis de nous faciliter énormément le travail en centralisant tout le code et les rapports. Pour les rapports de fin de séance, nous avons aussi utilisé un « pad », qui permet d'éditer à plusieurs du texte en temps réel (alternative libre et simpliste à Google Docs).

Concernant le travail en groupe, il est clair que travailler en groupe de 6 en faisant en sorte que tout le monde travaille efficacement n'est pas évident.

Nous avons donc essayé de nous répartir le travail le plus efficacement possible mais ça n'a pas non plus évident : à de nombreuses reprises, il fallait se recentrer et essayer de se re-répartir les tâches clairement, et ce n'est pas évident. Il y a donc eu certaines séances très peu productives (ou alors seulement sur certains points) avec aussi des pertes de motivation lors de certaines UAs...

Toutefois, les tâches étaient quand même généralement assignées à des binômes ou trinômes, et le travail a, dans l'ensemble, été plutôt efficace.

7.3 Apprentissages

Une chose reste certaine : nous avons tous appris de nouvelles choses pendant la plupart des UAs, même si elles ne reflètent qu'une partie de ce qui était demandé.

Le fait de se répartir le travail nous a aussi permis de travailler sur ce qui nous intéressait le plus, ce qui est plutôt motivant.

Références

[1] Fabrice Evrard, `http://evrard.perso.enseeiht.fr/Enseignement/2IN/`

[2] David Pisinger's optimization codes,
`http://www.diku.dk/~pisinger/codes.html`

[3] Frédéric Sur, *Les files d'attentes*,
`www.loria.fr/~sur/enseignement/RO/Files1_FSur.pdf`

[4] Nils Berglund, *Processus aléatoires et applications*,
`www.univ-orleans.fr/mapmo/membres/berglund/procal.pdf`

[5] Les figures 1 et 2 sont basées sur le travail de Gareth Jones, sous licence Creative Commons.
`http://en.wikipedia.org/wiki/File:Mmc-statespace.svg`

[6] Heuristics for the Traveling Salesman Problem `http://web.tuke.sk/fei-cit/butka/hop/htsp.pdf`

[7] Reducing Knapsack to TSP `http://cs.joensuu.fi/pages/franti/asa/DAA-Knapsack-to-TSP.ppt`