



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	3
2	UA : Graphes	4
2.1	Graphes eulériens	4
2.1.1	Analyse mathématique	4
2.1.2	Méthode de résolution	4
2.1.3	Algorithmes	5
2.1.4	Tests	6
2.2	Graphes hamiltoniens	6
2.2.1	Analyse mathématique	6
2.2.2	Méthode de résolution	6
2.2.3	Algorithmes	7
2.2.4	Tests	7
2.3	Modélisation mathématique	8
2.4	Problème du voyageur de commerce	8
2.4.1	Analyse mathématique	8
2.4.2	Heuristiques	8
2.4.3	Recherche locale	9
2.4.4	Métaheuristiques	10
2.4.5	Conclusion	11
3	UA : Programmation linéaire	12
3.1	Problème du sac à dos	12
3.1.1	Présentation du problème	12
3.1.2	Résolution exacte	12
3.1.3	Résolution approchée	12
4	UA : Jeux	14
4.1	Shifumi	14
4.1.1	Stratégie développée	14
4.1.2	Variantes	14
5	UA : Processus stochastiques	15
6	UA : Ingénierie robotique	16
7	Bilan	17
7.1	Enseignement	17
7.2	Travail en groupe	17
7.3	Apprentissages	17
8	Références	18

1 Introduction

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

Ce rapport va 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.

2 UA : Graphes

2.1 Graphes eulériens

2.1.1 Analyse mathématique

Un graphe eulérien est un graphe contenant un cycle eulérien, c'est-à-dire une chaîne parcourant toutes les arêtes du graphe une et une seule fois, en revenant au sommet de départ ; ce problème est donc celui de la goudronneuse qui doit passer sur toutes les rues sans pouvoir repasser dessus une seconde fois. Un théorème fondamental garantit qu'un graphe connexe est eulérien si et seulement si chacun de ses sommets est associé à un nombre pair d'arêtes.

Un graphe semi-eulérien, quant à lui, contient une chaîne eulérienne : celle-ci passe également par toutes les arêtes du graphe une seule et unique fois, mais ne retourne pas au point 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 chaîne eulérienne aura pour départ l'un des deux sommets associés à un nombre impair d'arêtes et pour point d'arrivée le deuxième.

2.1.2 Méthode de résolution

Afin de trouver une chaîne ou un cycle eulérien dans un graphe, nous avons étudié deux méthodes : une méthode qui teste toutes les possibilités, et une autre plus intelligente et moins coûteuse.

Matrices latines La première méthode est inspirée des matrices latines. Chaque coefficient de la matrice sera un ensemble de chaînes, une chaîne 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 chaîne $[N_i, N_j]$ 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. Le produit de deux chaînes sera :

- nul si le dernier nœud de la première chaîne n'est pas le premier nœud du deuxième ;
- la concaténation des deux chaînes sinon.

Le produit de deux ensembles de chaînes 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 chaînes de longueur k reliant les nœuds i et j .

Puisque une chaîne 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 chaînes 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 chaîne 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 chaîne, 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.

2.1.3 Algorithmes

Méthode de la matrice latine

```
Entrée : un graphe
Sortie : la liste des cycles eulériens dans le cas d'un graphe eulérien
        la liste des chaînes eulériennes dans le cas d'un graphe semi-eulérien
        la liste vide sinon

Construire 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:
        pour chaque arête sortant de ce nœud:
            ajouter la liste [nœud de départ, nœud d'arrivée] à la case de la
            matrice correspondante

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:
    si le coefficient n'est pas nul:
        concaténer ce coefficient à la variable de retour
```

Produit matriciel

```
Entrée : A et B deux matrices latines
Sortie : le produit de ces deux matrices

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:
    pour k allant de 1 jusqu'à n:
        calculer les chaînes produits entre a(i,k) et b(k,j)
        ajouter au coefficient de la matrice ces chaînes
```

Produit entre listes de chaînes (coefficients de matrices latines)

```
Entrée : liste_1 et liste_2 deux listes de chaîne
Sortie : une liste de chaînes

créer une liste de chaîne vide (liste de retour)
pour i dans liste_1:
    pour j dans liste_2:
```

```

    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 :
        construire la chaine élémentaire menant du nœud k au nœud k+1
        si cette chaine n'est pas dans l'ensemble :
            ajouter cette chaine dans l'ensemble
        sinon :
            rendre la chaine nulle
            sortir de la boucle

    si le chaine n'est pas nulle :
        concaténer la chaine trouvée à la liste de retour
    retourner la liste de retour

```

2.1.4 Tests

La première solution étant très coûteuse en espace mémoire, elle lève une erreur mémoire dès que la taille du graphe devient trop importante. Afin de comparer nos deux algorithmes, nous avons lancé un test sur un graphe complet à 6 nœuds. Le premier algorithme met 138 secondes avant de donner son résultat, tandis que le deuxième met à peine plus d'un dixième de seconde.

On voit donc que la première méthode est inexploitable même sur de toutes petites matrices.

2.2 Graphes hamiltoniens

2.2.1 Analyse mathématique

Un graphe (semi-)hamiltonien est un graphe sur lequel on peut trouver un cycle (ou une chaîne) passant par tout les sommets une et une seule fois. Ce problème est donc celui de l'enfant qui souhaite 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 chaîne s'il y en a.

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

Par exemple un graphe complet est forcément hamiltonien (utile dans le cas du voyageur de commerce, voir section ??), il existe aussi des conditions sur les degrés des sommets (théorème de Dirac, d'Ore, etc.).

2.2.2 Méthode de résolution

Pour tester si un graphe est hamiltonien, nous avons utilisé les théorèmes de Dirac et Pósa qui donnent des conditions nécessaires, si ces conditions ne sont pas vérifiées, comme il n'y a aucun théorème qui permette d'affirmer qu'un graphe n'est pas semi-hamiltonien, on recherche une chaîne hamiltonienne dans ce graphe.

Pour rechercher une chaîne hamiltonienne dans un graphe, nous avons écrit un algorithme qui recherche parmi toutes les chaînes possibles. Sa complexité dans le pire des

cas est donc très mauvaise : $O(n!)$. Comme on peut s'arrêter dès qu'on a trouvé une chaîne sans devoir tester toutes les autres chaînes possibles, la complexité moyenne sera inférieure.

Nous avons écrit une version améliorée de cet algorithme qui essaye d'éviter les culs de sac.

2.2.3 Algorithmes

Tests de semi-hamiltoniannité

```
Entrée : un graphe
Sortie : un booléen indiquant si le graphe est semi-hamiltonien ou non

si le graphe suit les conditions du théorème de Dirac ou du théorème de Pósa :
    retourner Vrai
sinon :
    chercher une chaîne hamiltonienne
    retourner Vrai si on en a trouvé un, Faux sinon
```

Recherche de chaîne hamiltonienne

```
Entrée : un graphe graph
        un point de départ optionnel node_from
        un ensemble (éventuellement vide) de nœuds déjà parcouru nodes_done
Sortie : une chaîne hamiltonienne sous la forme d'une liste ordonnée de points,
        ou None s'il n'en existe pas

Si la fonction a été appelée sans node_from :
    node_from = "un nœud de graph"

ajouter node_from à nodes_dones

si cardinal(node_from) == ordre(graph) :
    retourner [node_from]

pour chaque arête dans le graphe :
    autre = "le point opposé à node_from par rapport à cette arête"
    si autre dans nodes_done :
        passer à la prochaine arête

    appeler la fonction récursivement avec graphe, node_from et nodes_dones comme
    paramètre
    si la liste retournée est non-vide :
        y ajouter node_from au début et la retourner

retourner None (si on arrive ici, aucune chaîne n'est bonne)
```

2.2.4 Tests

Nos algorithmes fonctionnent bien sur de petits graphes, mais ils sont beaucoup trop lents pour être utilisés sur de grands graphes pour lesquels il y a “peu” d'arêtes : jusqu'à 30 sommets et 46 arêtes, l'algorithme trouve une solution en moins d'une seconde. Pour 40 sommets et 63 arêtes, il faut déjà une minute. Pour 100 sommets et 150 arêtes, l'algorithme prend tellement de temps que l'avons arrêté après quelques heures.

Par contre, pour des graphes ayant beaucoup d'arêtes (graphes “presque complets”), l'algorithme reste rapide.

Nous avons pu constater que la deuxième version de notre algorithme ne sert à rien : il n'y a aucune amélioration des performances.

2.3 Modélisation mathématique

Nous avons choisi de représenter nos graphes comme 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 ayant un pointeur vers chaque sommet extrémité.

Dans la suite nous noterons n le nombre de sommets du graphe.

2.4 Problème du voyageur de commerce

2.4.1 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 coût qui correspond à la distance entre chaque point, on cherche alors le cycle hamiltonien de coût 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 chaînes hamiltoniennes, et à prendre celle la plus courte, mais le nombre de chaînes hamiltoniennes 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.

2.4.2 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 :

```

Entrée : g (Graphe complet)
Sortie : (coût, cycle) où cycle est un cycle hamiltonien construit selon la
méthode du plus proche voisin et coût son coût associé sous forme de liste de
points
coût = 0
cycle = ["un point de g au hasard"]

tant qu'il reste des points:
    # On ajoute au cycle le point suivant
    plus_proche = "point de g sur lequel on est pas encore passé le plus proche
du dernier point du cycle"

    coût += "coût de plus_proche au dernier point du cycle"
    cycle = cycle :: plus_proche

# On ferme le cycle
coût += "coût du dernier au premier point de cycle"
cycle = cycle :: "premier point de chaîne"

retourner (coût, cycle)

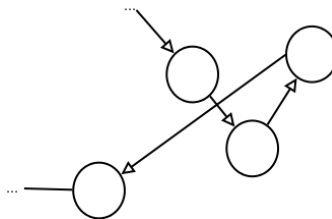
```

2.4.3 Recherche locale

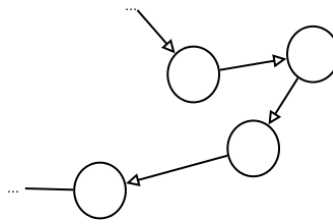
Les heuristiques nous donnant des solutions acceptables, choisies avec un minimum de « bon sens », il est ensuite possible de tenter d'améliorer ces solutions, via de la recherche locale. Partant d'une solution fournie, on va explorer les solutions voisines à cette dernière, afin de voir si on pourrait 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. Le principe du 2-opt consiste à tenter d'éliminer les « boucles » qui pourraient survenir dans le chemin, afin de le rendre plus court.

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 :



L'algorithme pour le 2-opt est le suivant :

```

Entrée : un cycle hamiltonien (liste de sommets) et son coût
Sortie : un cycle hamiltonien et son coût inférieur ou égal au coup d'entrée

pour chaque couple de points (a, b) dans le cycle :
    nouveau_coût = coût
    - "coût de a à son successeur dans le cycle"
    - "coût de b à son successeur dans le cycle"
    + "coût de a à b"
    + "coût du successeur de a et au successeur de b dans le
      cycle"

    si nouveau_coût < coût :
        coût = nouveau_coût
        cycle = cycle créée en échangeant a et b dans cycle

retourner (coût, cycle)

```

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

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.

2.4.4 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.

2.4.5 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 Problème du sac à dos

3.1.1 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 :

$$\begin{cases} \max v_i \\ i \in S \\ \sum_{j \in S} m_j \leq W \end{cases}$$

où S est l'ensemble des objets à choisir, v_i la valeur de l'objet i , 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.

3.1.2 Résolution exacte

Nous avons implémenté un algorithme de programmation dynamique, qui permet de résoudre le problème du sac à dos. Toutefois, il fonctionne uniquement si les poids des objets sont des entiers.

Sa complexité en temps est en $O(nW)$ et celle en mémoire en $O(W)$, avec n le nombre d'objets et W le poids maximum du sac.

Nous l'avons testé¹ sur plusieurs instances du problème, et l'algorithme est rapide.

3.1.3 Résolution approchée

Nous avons aussi implémenté l'algorithme glouton : celui-ci consiste simplement à choisir les « meilleurs » objets jusqu'à que la masse maximale soit dépassée. Le critère déterminant quels sont les meilleurs objets peut être la masse faible, le prix élevé, ou le rapport prix/masse élevé.

1. En utilisant le générateur de problèmes trouvé à l'adresse suivante : <http://www.diku.dk/~pisinger/codes.html>.

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).

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

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 Stratégie développée

Afin de démontrer qu'un adversaire ne jouant pas aléatoirement est facile à battre, nous avons développé une stratégie qui se base sur des chaînes 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 dés-équilibre, car un élément sera moins efficace contre les autres. L'équilibre de Nash du jeu va alors consister à ne jamais jouer cet élément.

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

5 UA : Processus stochastiques

6 UA : Ingénierie robotique

7 Bilan

7.1 Enseignement

Nous avons été 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 devons 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, 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 avons 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 certains rapports, nous avons aussi utilisé un « pad », qui permet d'éditer à plusieurs du texte en temps réel.

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 été 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

8 Références

Références

- [1] Fabrice Evrard, <http://evrard.perso.enseeiht.fr/Enseignement/2IN/>
- [2] Wikipédia, *Problème du sac à dos*,
http://fr.wikipedia.org/wiki/Probl%C3%A8me_du_sac_%C3%A0_dos
- [3] Wikipedia, *Knapsack problem*,
http://en.wikipedia.org/wiki/Knapsack_problem
- [4] David Pisinger's optimization codes,
<http://www.diku.dk/~pisinger/codes.html>