Feuille de travaux pratiques n° 3
Une méthode simple pour la résolution exacte du problème du sac à dos unidimensionnel en variables binaires

1 Définition du problème

Étant donné un ensemble d'objets $J = \{1, \ldots, n\}$, on associe à chaque objet $j \in J$ un profit $p_j \in \mathbb{N}^*$ et un poids $w_j \in \mathbb{N}^*$. Nous supposons de plus que chaque objet est disponible en un unique exemplaire. Le *problème de sac à dos unidimensionnel en variables binaires* consiste à ranger un sous-ensemble d'objets de J dans un conteneur (sac à dos) dont la capacité $\omega \in \mathbb{N}^*$ est limitée. Cela doit être réalisé en maximisant une fonction objectif définie par la somme des profits des objets sélectionnés. À chaque objet $j \in J$ est associé une variable de décision $x_j \in \{0, 1\}$ qui prend la valeur 1 si l'objet j est inséré dans le sac et 0 sinon. La modélisation du problème est donnée ci-dessous.

$$\max \sum_{j=1}^{n} p_j \, x_j$$
$$s.c. \sum_{j=1}^{n} w_j \, x_j \leq \omega$$
$$x_j \in \{0, 1\}, j = 1, \ldots, n$$

Ce problème est la variante la plus simple dans la famille des problèmes de sac à dos, qui inclut entre autres la variante multi-dimensionnelle (présentée en cours), ou la variante en variables entières permettant des objets en de multiples exemplaires. Le problème du sac à dos apparaît en dans les applications suivantes : découpe de matériaux, chargements, systèmes d'aide à la gestion de porte-feuille... Des problèmes d'optimisation combinatoire difficiles sont de plus souvent résolus en étant décomposés en un ensemble de sous-problèmes plus simples parmi lesquels les problèmes de sac à dos apparaissent. Résoudre le plus efficacement possible ces problèmes est donc très important.

Malgré son apparente simplicité, le problème du sac à dos unidimensionnel en variables binaires est $\mathcal{NP}$-hard. Cependant, il ne s'agit que de la classe de difficulté théorique. Ce problème a été très étudié et de nombreuses méthodes de résolution dédiées ont été proposées [1]. Dans le cadre de ce projet, nous partirons d'une méthode très basique de programmation dynamique, sur laquelle nous ajouteront incrémentalement des améliorations.

Dans ce problème, il existe des cas triviaux que nous supposerons exclus initialement. Tout d'abord, si $\sum_{j=1}^{n} w_j \leq \omega$, la solution optimale consiste tout simplement à insérer tous les objets dans le sac, c'est-à-dire à fixer toutes les variables à 1. Ensuite, si pour un objet j, on a un poids $w_j > \omega$, on a nécessairement $x_j = 0$ pour obtenir une solution admissible.

2 Modélisation et résolution en utilisant JuMP/GLPK

Face à un problème qui peut se modéliser linéairement, un premier réflexe est d'avoir recours à un solveur MIP. Il s'agira de la première étape de ce projet et cela donnera des résultats de référence permettant de valider les implémentations réalisées ensuite.

1. Les méthodes les plus rapides pour résoudre ce problème sont *Minknap* et *Combo*. Des liens sur des implémentations en C et des références à leur description peuvent être trouvés sur cette page : `http://hjemmesider.diku.dk/ pisinger/codes.html`.

3 Algorithmes de programmation dynamique

3.1 Définition d'une relation de récurrence

Nous considèrerons des sous-problèmes du problème du sac à dos initialement posé du type suivant,

$$\begin{aligned} &\max \sum_{j=k}^{n} p_j \, x_j \\ &s.t. \sum_{j=k}^{n} w_j \, x_j \leq \omega - d \\ &x_j \in \{0,1\}, j = k, \ldots, n \end{aligned}$$

où $k \in \{1, \ldots, n\}$ et $d \in \mathbb{N}$. Il s'agit d'un problème de sac à dos dans lequel les $k-1$ premiers objets sont retirés, et la capacité du sac est réduite.

On note $P_{k,d}$ le problème défini comme ci-dessus, c'est-à-dire un problème de sac à dos dans lequel les objets ne sont considérés qu'à partir du $k$ème et la capacité est $\omega - d$. Nous noterons $z_{k,d}$ la valeur optimale du problème $P_{k,d}$. Il est simple de définir une relation de récurrence entre le problème $P_{k,d}$ et des problèmes $P_{k+1,\times}$ où $\times$ est à préciser, liant ces problèmes et leurs solutions optimales. En effet, la différence essentielle se fait dans le choix effectué en ce qui concerne l'objet d'indice k :

- Si $d + w_k > \omega$, alors on a nécessairement $x_k = 0$ et les problèmes $P_{k,d}$ et $P_{k+1,d}$ sont donc équivalents. On a alors $z_{k,d} = z_{k+1,d}$.
- Si $d + w_k \leq \omega$, on a alors deux choix possibles pour la valeur de x_k : 0 ou 1.
  - Si $x_k = 0$, on considère à nouveau le problème $P_{k,d}$ dont la valeur optimale est $z_{k,d}$.
  - Si $x_k = 1$, la capacité du sac est réduite de w_k (alternativement, le poids des objets insérés dans le sac croît de w_k) et on considère le problème $P_{k+1,d+w_k}$ dont la valeur optimale est $z_{k+1,d+w_k}$.

  Comme les deux choix sont possibles et qu'on doit tenir compte du profit engendré par l'objet k, on a alors $z_{k,d} = \max\{z_{k+1,d}, p_k + z_{k+1,d+w_k}\}$.

En récapitulant, on obtient la relation de récurrence suivante,

$$z_{k,d} = \begin{cases} z_{k+1,d} & \text{si } w_k > \omega - d \\ \max\{z_{k+1,d}, p_k + z_{k+1,d+w_k}\} & \text{sinon} \end{cases}$$

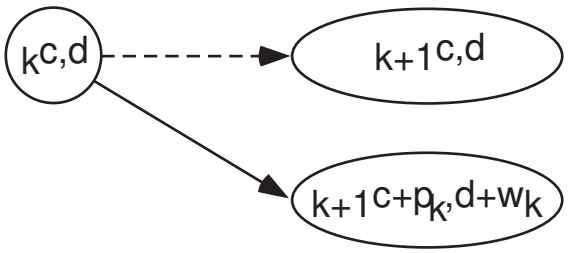
pour laquelle le cas de base est donné par $z_{n+1,d} = 0$ pour tout $d \in \mathbb{N}$.

Le problème que l'on souhaite initialement résoudre est $P_{1,0}$ et sa valeur optimale $z_{1,0}$ pourrait être simplement calculée en utilisant la relation de récurrence définie ci-dessus. Cela pourrait se faire très simplement en écrivant une fonction récursive. Cependant, un même problème $P_{k,d}$ pourrait être résolu de manière redondante dans des appels récursifs différents. Le principe de la programmation dynamique est de faire une implémentation itérative évitant ces redondances. Au passage, cela permettra de manière simple d'éviter des résolutions inutiles, car ne menant pas à des solutions optimales.

3.2 Algorithme de programmation dynamique basique

Afin d'éviter les redondances qui apparaîtraient dans une fonction récursive, il est nécessaire de stocker en mémoire les différents problèmes $P_{k,d}$ qui sont considérés. On construira d'abord le problème $P_{1,0}$, puis les problèmes $P_{2,0}$ et P_{2,w_1}, puis l'ensemble des problèmes $P_{3,d}$ pour des valeurs de $d \in \mathbb{N}$ qui sont utiles d'après la relation de récurrence,... Aucun problème n'apparaîtra deux fois, et nous stockerons également les profits engendrés par les objets déjà inclus dans le sac, ainsi que les liens entre les problèmes définis par la relation de récurrence. Il est courant et plus clair de représenter l'exécution d'un algorithme de programmation dynamique pour une résolution du problème de sac à dos en passant par une représentation par un graphe. Attention ! cette représentation ne correspond pas tout à fait aux meilleurs choix de structures de données pour une implémentation (il y a plus simple !).

On associe à chaque combinaison définie par un problème $P_{k,d}$ et le profit cumulé c ayant mené à ce problème (i.e. la somme des profits des objets ajoutés dans le sac), un sommet $k^{c,d}$ appelé *état* dans la terminologie de la programmation dynamique. Les liens entre combinaisons (problème, profit engendré) sont représentés par des arcs comme indiqué dans la figure ci-dessous.

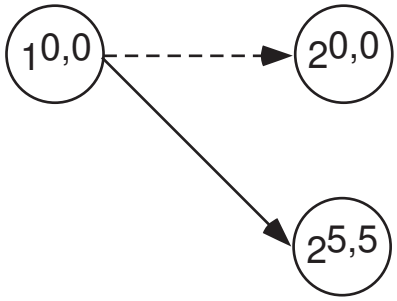


Les deux arcs indiquent que dans le cas où x_k peut être fixé à 0 ou à 1, la résolution du problème $P_{k,d}$ passe par la résolution des problèmes $P_{k+1,d}$ et $P_{k+1,d+w_k}$. Pour faciliter la visualisation, les arcs correspondant à un objet non-inclus seront représentés horizontalement et en pointillé, et les arcs correspondant à un objet inclus seront représentés de manière oblique. La résolution complète et l'illustration des tests de dominance, ayant pour but d'éviter des résolutions ne menant pas à une solution optimale sont ensuite présentés sur un exemple.

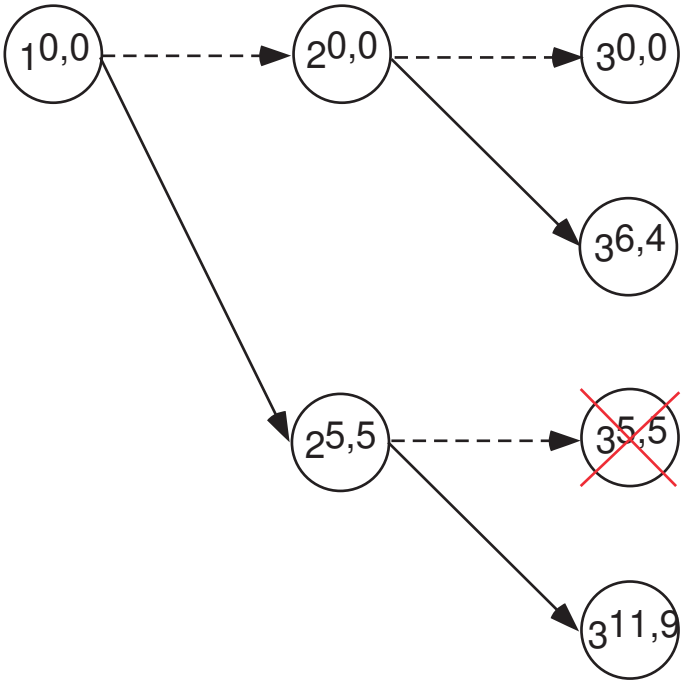
Exemple : On considère le problème de sac à dos suivant.

$$\begin{aligned} \max\ & 5x_1 + 6x_2 + 4x_3 + 9x_4 \\ s.c.\ & 5x_1 + 4x_2 + 6x_3 + 5x_4 \leq 10 \\ & x_j \in \{0,1\}, j = 1,\ldots,4 \end{aligned}$$

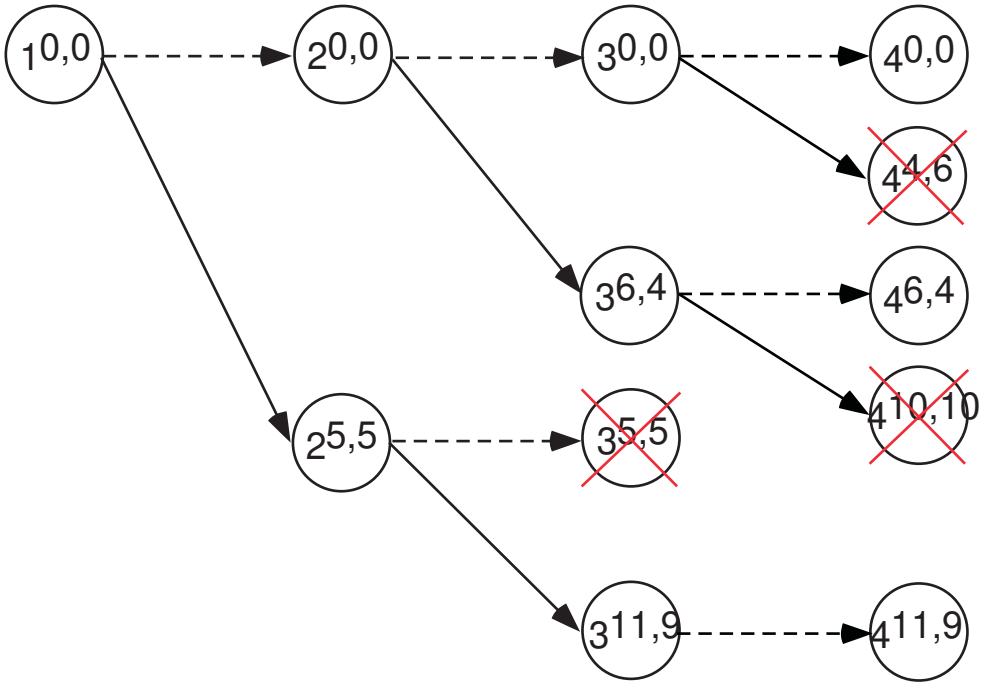
On construit d'abord l'état initial $1^{0,0}$, puis les deux états $2^{0,0}$ et $2^{5,5}$.



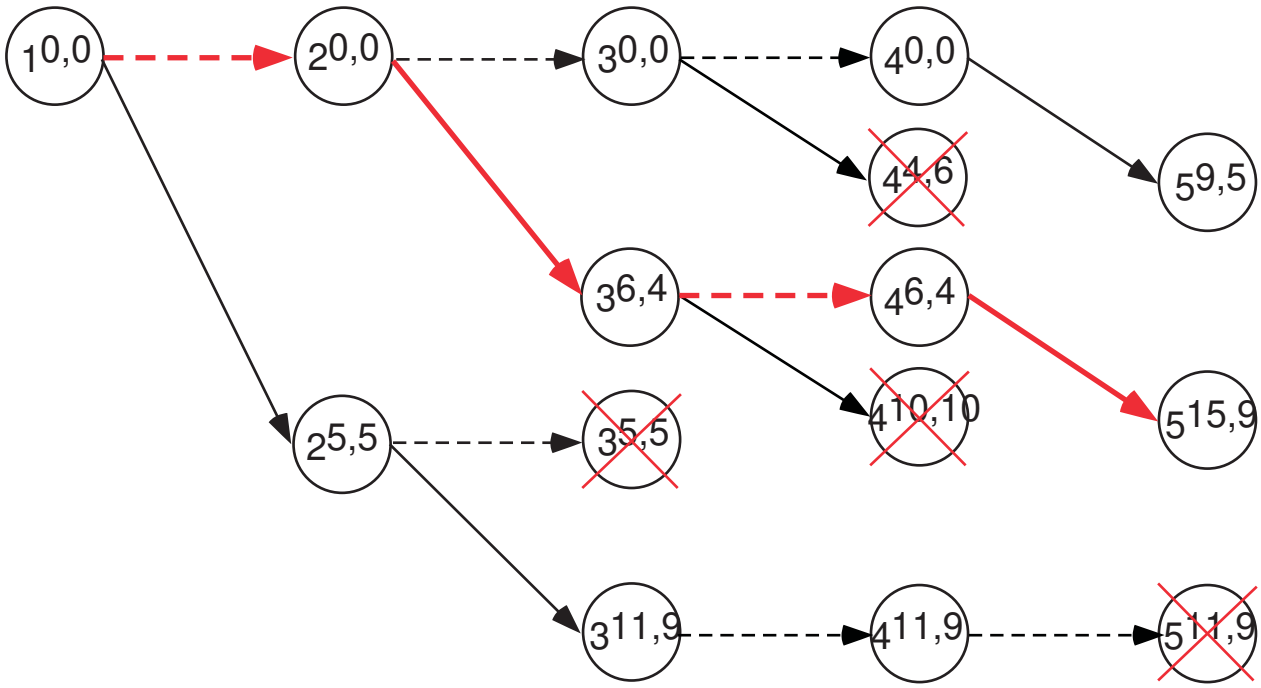
Nous pouvons ensuite construire les états qui découlent de l'état $2^{0,0}$ ($3^{0,0}$ et $3^{6,4}$), puis ceux qui découlent de l'état $2^{5,5}$ ($3^{5,5}$ et $3^{11,9}$).



On peut remarquer qu'il est impossible que l'état $3^{5,5}$ mène à une solution optimale. En effet, le profit des objets cumulés est de 5 pour un poids de 5. Il reste donc une capacité dite *résiduelle* de 5 dans le sac. L'état $3^{6,4}$ a un profit cumulé de 6 pour un poids de 4, la capacité résiduelle est donc de 6. L'état $3^{6,4}$ a donc un profit cumulé plus grand ($6 > 5$) et un poids cumulé plus faible ($4 < 5$) que l'état $3^{5,5}$. Comme les problèmes considérés dans ces deux états considèrent les objets à partir du troisième, l'état $3^{6,4}$ mènera nécessairement à une meilleure solution que l'état $3^{5,5}$. On dit que l'état $3^{5,5}$ est *dominé* par l'état $3^{6,4}$ et on peut donc le supprimer (ou plutôt ici éviter de l'inclure dans le graphe).
On peut ensuite passer à la construction successive des états découlant de $3^{0,0}$, $3^{6,4}$ et $3^{11,9}$.



Les premiers états instanciés sont donc $4^{0,0}$ et $4^{4,6}$ et ils ne sont pas dominés pour le moment. On instancie ensuite les états $4^{6,4}$ et $4^{10,10}$ qui ne sont pas dominés pour le moment. Par contre, l'état $4^{4,6}$ est maintenant dominé par l'état $4^{6,4}$ et doit donc être supprimé. Ensuite, un seul état découle de l'état $3^{11,9}$ car il n'est plus possible d'ajouter un objet de poids 6 (la capacité du sac est limitée à 10). L'état $4^{11,9}$ n'est pas dominé mais il domine l'état $4^{10,10}$ qui doit donc être supprimé du graphe.
On peut finalement passer à la construction successive des états découlant de $4^{0,0}$, $4^{6,4}$ et $4^{11,9}$.



Il n'y a cette fois qu'un seul état découlant des états $4^{0,0}$ et $4^{6,4}$. Cela est dû au fait que les problèmes $P_{4,0}$ et $P_{4,4}$ tombent dans le cas trivial pour lequel une solution optimale est obtenue en insérant tous les objets dans le sac (il ne reste que l'objet d'indice 4 ici). Par conséquent, seuls les états $5^{9,5}$ et $5^{15,9}$ sont générés. Il n'y a également qu'un seul état découlant de l'état $4^{11,9}$ qui est $5^{11,9}$, mais celui-ci est dominé par $5^{15,9}$.

La construction des états est maintenant achevée. Visuellement, le graphe a été construit colonne par colonne. On parcourt maintenant les états de la dernière colonne pour trouver l'état qui donne le profit cumulé le plus grand. Ici, c'est l'état $5^{15,9}$ avec un profit de 15. Pour connaître la solution associée, on parcourt le graphe à l'envers en partant de $5^{15,9}$ pour revenir à l'état $1^{0,0}$. Cela reconstruit la solution optimale associée à l'envers. Si on passe un par un arc oblique (si on a une différence de profit cumulé entre les deux états aux extrémités de l'arc) alors la variable associée prend la valeur 1, sinon elle prend la valeur 0. Ici, on obtient donc $x = (0, 1, 0, 1)$.

Suite à cet exemple, on peut faire quelques remarques. On a trois informations à stocker dans chaque état : son profit cumulé, son poids cumulé et son prédécessur (l'état dont il découle). Le prédecesseur sert à construire la solution optimale à la fin de la résolution. La construction des états colonne par colonne implique qu'il est possible de stocker chaque colonne sous la forme d'un tableau d'états. De plus, il est facile de voir que chaque colonne contient au plus deux fois plus d'états que la colonne précédente.

Le coût temporel principal dans la construction des colonnes du graphe vient des tests de dominance. En effet, chaque fois qu'un nouvel état est créé, on doit d'abord parcourir les états de la même colonne jusqu'à obtenir un état le dominant s'il y en a un. Ensuite, si aucun état existant dans la même colonne ne le domine, il faut à nouveau parcourir les états existants pour filtrer ceux qui sont dominés par ce nouvel état. Un état $k^{c,d}$ est dominé par un état $k^{c',d'}$ si $c \leq c'$ et $d \geq d'$. L'égalité est incluse ! Il est possible d'avoir deux états avec le même profit et le même poids cumulés. Dans ce cas, on ne conserve que le premier obtenu.

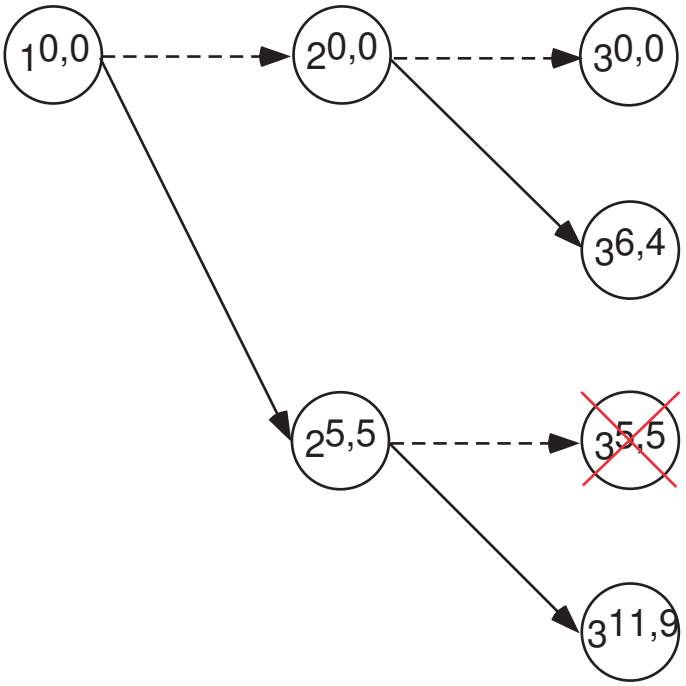
En organisant de manière ordonnée la construction des états, il est possible de réduire significativement les coûts de ces tests de dominance.

3.3 Algorithme de programmation dynamique un peu moins basique

Dans la sous-section précédente, pour construire la colonne $j + 1$ à partir de la colonne j, les états de la colonne j étaient parcourus dans l'ordre de stockage pour instancier immédiatement les états qui en découlaient dans la colonne $j + 1$. Il en résultait une construction désordonnée des états. Dans cette deuxième version de l'algorithme, nous allons construire les états de chaque colonne par valeur croissante du poids cumulé. Pour cela, il n'y aura plus un itérateur qui parcourra les états de la colonne j mais deux : un itérateur *avec* qui parcourt les états de la colonne j avec pour but d'instancier les états ajoutant l'objet considéré, un itérateur *sans* qui parcourt les

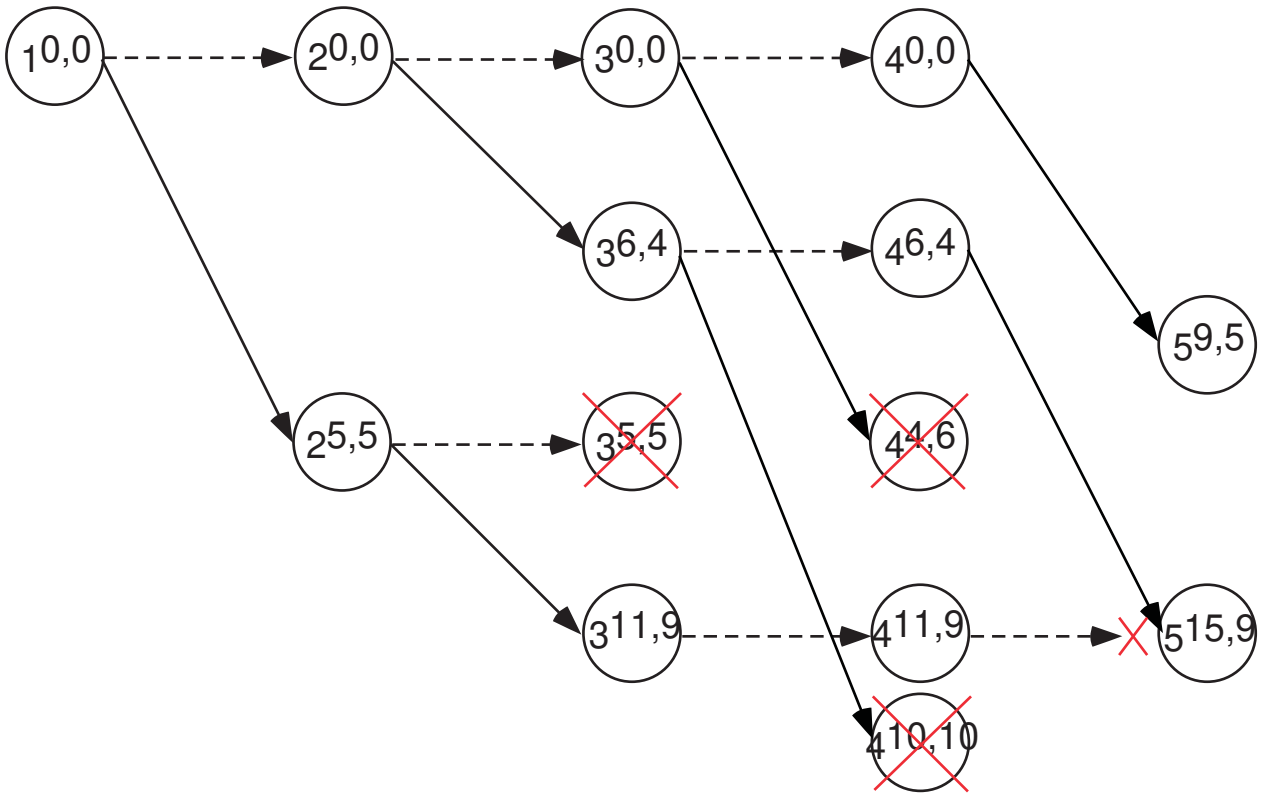
états de la colonne j avec pour but d'instancier les états n'ajoutant pas l'objet considéré. Pour illustrer simplement ce principe, nous reprenons l'exemple précédent.

Exemple : La construction des deux premières colonnes est identique à la version précédente. Les premières différences apparaissent dans la construction de la troisième colonne.



Les deux itérateurs pointent initialement sur l'état $2^{0,0}$. L'état au poids cumulé le plus petit sera donc obtenu sans ajout d'objet. On génère donc l'état $3^{0,0}$ et l'itérateur *sans* pointe ensuite sur l'état $2^{5,5}$. Comme l'objet 2 a un poids de 4 et comme $0 + 4 < 5$, l'état $3^{6,4}$ est maintenant généré à partir de l'état $2^{0,0}$. L'itérateur *avec* pointe alors également sur l'état $2^{5,5}$. Ensuite, l'état $3^{5,5}$ est créé à partir de l'état $2^{5,5}$. L'état $3^{5,5}$ est dominé par l'état $3^{6,4}$ car $5 \leq 6$. Nous n'avons pas besoin de regarder le poids cumulé dans ce test de dominance car les états d'une colonne sont générés par valeur croissante du poids cumulé. De plus, les états stockés dans une une colonne sont aussi générés par profit cumulé croissant (car dans le cas contraire, ils seraient dominés). La conséquence est que pour tester la dominance d'un nouvel état, on ne regarde plus que le dernier état ajouté dans la même colonne. Le parcours est maintenant terminé pour l'itérateur *sans*. Ensuite, l'état $3^{11,9}$ est généré à partir de l'état $3^{5,5}$ et le parcours est maintenant terminé également pour l'itérateur *avec*. L'état $3^{11,9}$ n'est pas dominé car $11 > 6$ (le profit du dernier état ajouté dans la colonne). Ensuite, il n'est pas nécessaire de vérifier si l'état $3^{11,9}$ domine d'autres états de la même colonne car c'est définitivement impossible ! En effet, leur poids cumulé est nécessairement inférieur à 9, du fait du caractère ordonné de l'énumération.

La construction du graphe se poursuit. Les états de chaque colonne sont générés de haut en bas sur la figure ci-dessous.



Seule la construction de la colonne 5 à partir de la colonne 4 est ici décrite, car elle présente de nouveaux cas intéressants. Tout d'abord, l'itérateur *sans* va directement pointer sur l'état $4^{11,9}$ car il n'y a aucun état à instancier à partir des états $4^{0,0}$ et $4^{6,4}$ sans insertion d'objet. L'itérateur *avec* pointe intialement sur l'état $4^{0,0}$. Comme $0 + 5 < 9$, nous instancions d'abord l'état $5^{9,5}$ à partir de l'état $4^{0,0}$. L'itérateur *avec* pointe ensuite sur l'état $4^{6,4}$. Nous avons cette fois $4 + 5 = 9$, i.e. les états générés à partir de $4^{6,4}$ (avec ajout de l'objet 4) et $4^{11,9}$ (sans ajout de l'objet 4) auront le même poids cumulé. Dans ce cas, pour faire une différence, il faut regarder le profit cumulé. Si un des états offre un profit cumulé plus grand, ce sera le seul état intéressant pour la poursuite de la résolution. En cas d'égalité, un choix arbitraire (d'implémentation) devra être effectué (donner la priorité à un ajout d'objet ou pas). Ici, les profits cumulés sont 11 et $6 + 9$ soit 15. Comme nous avons $15 > 11$, seul l'état $5^{15,9}$ est instancié à partir de $4^{6,4}$. Les deux itérateurs avancent alors vers l'état suivant (s'il y en a un). Le parcours de la colonne est achevé pour l'itérateur *sans*. Il en est finalement de même pour l'itérateur *avec* pointant maintenant sur l'état $4^{11,9}$ car il n'est pas possible d'ajouter l'objet 4 à partir de cet état.

La différence entre les deux variantes d'algorithmes de programmation dynamique semble minime. Les colonnes d'états ont exactement les mêmes contenus. Seul l'ordre de génération des états est différent. Le point essentiel pour générer les états d'une colonne par valeurs croissantes du poids cumulé est de comparer les poids cumulés des deux états qu'on peut générer à partir des états pointés respectivement par les itérateurs *avec* et *sans*. On privilégie toujours l'état au poids le plus faible et en cas d'égalité l'état au profit cumulé le plus important.

4 Prétraitement

Pour une résolution plus rapide, il sera intéressant de prétraiter le problème. Il est en effet possible à l'aide de bornes sur la valeur optimale du problème, de déduire que certaines variables devront forcément être égales à 1 et d'autres à 0 dans une solution optimale du problème. Nous présentons dans cette section des bornes spécifiques pour le problème de sac à doc unidimensionnel en variables binaires.

4.1 Bornes inférieures sur la valeur optimale

Pour obtenir une borne inférieure sur la valeur optimale du problème, il suffit de trouver une solution admissible. En effet, comme nous avons ici un problème en maximisation, une solution optimale a par définition une valeur supérieure ou égale à celle de toute solution admissible du problème.

Il existe une solution admissible évidente $x = (0, \ldots, 0)$ qui fournit une valeur de 0 qui est très peu satisfaisante. Il est possible de faire mieux de manière très simple dans le cas de ce problème. Tout d'abord, on peut commencer par essayer de définir un indicateur de qualité individuel aux objets du problème. Pour cela, il est naturel de considérer le ratio profit/poids. Les objets ayant un ratio élevé semblent individuellement plus intéressants. Nous commencerons donc par trier les objets par ordre décroissant par rapport au ratio profit/poids. Dans la suite de cette section, nous supposerons que les objets sont réindéxés par rapport à ce ratio.
Ensuite, il semble naturel d'insérer les objets dans le sac dans l'ordre des indices jusqu'à ce que l'un des objets ne puisse plus rentrer dans le sac. Nous obtenons ainsi la solution *dantzig* (du nom du chercheur célèbre qui l'a proposée) qui est admissible par construction et fournit une première borne inférieure sur la valeur optimale du problème. Dans la suite, nous appellerons *objet cassé* le premier objet ne rentrant pas dans le sac. L'indice de cet objet sera noté s et est formellement défini par $\min\{s \mid \sum_{j=1}^{s} w_j > \omega\}$. Cet objet cassé jouera ensuite un rôle crucial dans la détermination de borne supérieures sur la valeur optimale. La borne inférieure associée à la solution *dantzig* est donnée par $\sum_{j=1}^{s-1} p_j$.
Il est possible d'améliorer la solution *dantzig* en continuant le parcours des objets après l'objet cassé, toujours dans l'ordre des indices. Si un objet peut rentrer dans le sac, on l'ajoute. Sinon, on passe à l"objet suivant. Il s'agit ici tout simplement d'un algorithme glouton.

4.2 Bornes supérieures sur la valeur optimale

Dans des problèmes en variables entières ou binaires, la relaxation continue est souvent la manière la plus simple d'obtenir une borne supérieure sur la valeur optimale. Il suffit en général d'avoir recours à l'algorithme du simplexe pour l'obtenir. Ici, il est possible de l'obtenir pour un coût temporel beaucoup plus bas. Revenons à la solution *dantzig*. Les objets d'indice 1 à $s-1$ sont insérés dans le sac. La capacité résiduelle du sac est donc $\omega - \sum_{j=1}^{s-1} w_j$. Nous noterons cette capacité résiduelle du sac pour la solution *dantzig* $\bar{\omega}$. Nous avons par définition de l'objet cassé $w_s > \bar{\omega}$, c'est pour cela que l'objet d'indice s ne peut pas être inséré (en entier) dans le sac une fois les $s-1$ premiers objets insérés. Cependant, dans la relaxation continue, les variables $x_j \in \{0, 1\}$ sont remplacées par $x_j \in [0, 1]$. Il est donc possible d'insérer un fragment de l'objet cassé (d'où son nom). Cela revient à fixer la variable x_s à la valeur $\frac{\bar{\omega}}{w_s}$ de manière à utiliser toute la capacité du sac. Les variables x_j pour $j \in \{s+1, \ldots, n\}$ sont donc fixées à 0. La valeur optimale de la relaxation continue est donc donnée par $\sum_{j=1}^{s-1} p_j + \bar{\omega}\frac{p_s}{w_s}$. Comme le problème de sac à dos est en variables binaires et que nous faisons l'hypothèse que les coûts sont entiers, l'arrondi à l'entier inférieur de cette valeur optimale définit une borne supérieure sur la valeur optimale du problème de sac à dos. Finalement, une borne supérieure sur la valeur optimale du problème est donnée par

$$U_0 = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{\omega}\frac{p_s}{w_s} \right\rfloor.$$

Une borne supérieure un peu meilleure et à peine plus coûteuse que la relaxation continue à été proposée par Martello et Toth. Cette borne s'appuie sur une partition simple de l'ensemble des solutions admissibles du problème : soit $x_s = 0$ soit $x_s = 1$. Cela divise le problème initial en deux sous-problèmes P_{s_0} et P_{s_1}. On peut donc calculer borne supérieure sur la valeur optimale de chacun de ces deux problèmes, et la plus grande des deux nous donne une borne supérieure sur la valeur optimale du problème initial. Nous noterons ces deux bornes respectivement U^0 et U^1. En partant du raisonnement effectué dans le cas de la relaxation continue. On peut simplement poser

$$U^0 = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{\omega}\frac{p_{s+1}}{w_{s+1}} \right\rfloor,$$

$$U^1 = \sum_{j=1}^{s-1} p_j + \left\lfloor p_s - (w_s - \bar{\omega})\frac{p_{s-1}}{w_{s-1}} \right\rfloor.$$

Dans le cas de U^0, on ne peut pas ajouter un fragment de l'objet s car x_s est fixé à 0. Par conséquent, un "fragment" de l'objet suivant d'indice $s+1$ est ajouté. Des guillemets sont utilisés pour parler de fragment car on pourrait avoir $\frac{\bar{\omega}}{w_{s+1}} > 1$. La valeur U^0 obtenue n'en reste pas moins une borne supérieure sur la valeur optimale de P_{s_0}.
Dans le cas de U^1, après avoir ajouter les $s-1$ premiers objets puis l'objet s. Nous débordons de la capacité du sac (par définition de l'objet cassé). Il faut donc retirer un "fragment" de l'objet au ratio le moins intéressant soit l'objet d'indice $s-1$ pour respecter la contrainte de capacité. Une fois de plus, le mot fragment est mis entre

guillemets car on pourrait retirer plus d'une unité de l'objet d'indice $s-1$. La valeur U^1 obtenue n'en reste pas moins une borne supérieure sur la valeur optimale de P_{s_1}.
Finalement, la borne de Martello et Toth est définie par

$$U_1 = \max\{U^0, U^1\}.$$

4.3 Utilisation des bornes pour prétraiter le problème

Nous commencerons par déterminer une première solution admissible pour le problème du sac à dos en utilisant l'algorithme glouton posé à la fin de la section 4.1. Cette solution initiale fournit une une première borne inférieure LB sur la valeur optimale.

Ensuite, nous considèrerons $2n$ sous-problèmes fixant chacun une variable x_i soit à 0 soit à 1 pour $i \in \{1, \ldots, n\}$. Nous calculerons la borne supérieure de Martello et Toth sur la valeur optimale de chacun de ces sous-problèmes. Nous noterons $UB1(i)$ la borne supérieure obtenue pour le sous-problème fixant la variable x_i à 1, et $UB0(i)$ la borne supérieure obtenue pour le sous-problème fixant la variable x_i à 0. Il est intéressant de remarquer que dans le calcul de la borne de Martello et Toth, la solution *dantzig* (des sous-problèmes) est au passage calculée et que du fait de la variable x_i fixée à 0 ou à 1, celle-ci peut être différente de celle du problème initial. Cette solution *dantzig* pour un sous-problème reste une solution admissible pour le problème initial et il est possible d'obtenir une meilleure solution que celle initialement obtenue par un algorithme glouton. Comme il est intéressant d'avoir la meilleure borne inférieure possible, nous stockerons la meilleure solution admissible obtenue x_{heur} et sa valeur définissant une borne inférieure LB sur la valeur optimale du problème initial.

Pour chaque $i \in \{1, \ldots, n\}$:

— Si $UB0(i) \leq LB$, cela implique qu'en fixant x_i à 0, nous ne pourrons pas obtenir une meilleure solution que x_{heur}. Par conséquent, nous fixons x_i à 1 pour la suite de la résolution.

— Si $UB1(i) \leq LB$, cela implique qu'en fixant x_i à 1, nous ne pourrons pas obtenir une meilleure solution que x_{heur}. Par conséquent, nous fixons x_i à 0 pour la suite de la résolution.

Nous notons J_1 l'ensemble des indices des variables fixées à 1, et J_0 l'ensemble des indices des variables fixées à 0. Si J_1 est large, il est possible que pour certains objets $i \in \{1, \ldots, n\} \setminus (J_1 \cup J_0)$, on ait $w_i > \omega - \sum_{j \in J_1} w_j$. Dans ce cas, la variable x_i peut immédiatement être fixée à 0.
Plus rarement, il peut arriver que $\sum_{j \in J_1} w_j > \omega$. Dans ce cas, les variables déjà fixées à 1 impliquent que la capacité du sac n'est pas respectée. Cela signifie que pour trouver une meilleure solution que x_{heur}, il faut sortir de la région admissible. Conclusion : l'optimalité de la solution x_{heur} est alors prouvée par le prétraitement.
Cependant, dans la grande majorité des cas, il restera à résoudre un problème de taille réduite avec l'algorithme de programmation dynamique.

Il reste à préciser comment effectuer ces calculs de bornes. La manière la plus simple de le faire est de générer les données de chaque sous-problème et de calculer la borne de Martello et Toth directement en appliquant la formule donnée en fin de section 4.2. Cette façon de faire n'est pas particulièrement habile car elle implique de nombreuses redondances dans les calculs effectués. Cependant, il est recommandable de procéder ainsi dans un premier temps, car cette approche est simple et permettra de valider les résultats obtenus par une approche plus élaborée.

Supposons maintenant que nous calculons d'abord la borne de Martello et Toth UB sur la valeur optimale du problème initial. À priori, puisque nous avions déjà appliqué un algorithme glouton, nous connaissions déjà la solution *dantzig* (définissant la partie $\sum_{j=1}^{s-1} p_j$ des termes U^0 et U^1 de la borne de Martello et Toth) et également l'objet cassé d'indice s. Cette borne est donc calculée très rapidement. Une manière de calculer les bornes $UB0(i)$ est précisée ci-dessous. Ce sera à vous de préciser comment calculer efficacement les bornes $UB1(i)$.

Il y a 4 cas à considérer (Important : il est utile de travailler sur papier en effectuant la lecture) :

— Si $i \in \{s+2, \ldots, n\}$, il n'y a pas de changement en ce qui concerne la solution *dantzig* et l'objet cassé. De plus, les variables jouant un rôle dans la borne de Martello et Toth ont un indice inférieur ou égal à $s+1$. Par conséquent, nous pouvons conclure sans calcul que $UB0(i) = UB$.

— Si $i = s+1$, il n'y a une fois de plus aucun changement en ce qui concerne la solution *dantzig* et l'objet cassé. La variable x_{s+1} joue normalement un rôle dans le terme U^0 de la borne de Martello et Toth, mais elle est alors retirée du problème. L'indice suivant s est donc $s+2$, et le terme U^0 devient donc $\sum_{j=1}^{s-1} p_j + \left\lfloor \bar{\omega} \frac{p_{s+2}}{w_{s+2}} \right\rfloor$. Le terme U^1 n'est pas impacté.

— Si $i = s$, la solution *dantzig* est cette fois potentiellement modifiée et l'objet cassé change nécessairement, puisque l'objet s est retiré du problème. Il n'est cependant pas nécessaire de recalculer entièrement la

solution *dantzig*. En effet, nous savons déjà que les objets d'indice 1 à $n-1$ peuvent être insérés dans le sac. Nous continuons juste d'essayer d'insérer les objets tant que c'est possible en continuant à partir de l'objet d'indice $s+1$. Cela permet d'obtenir la solution *dantzig* éventuellement modifiée, et l'objet cassé du sous-problème dont nous notons l'indice s'. Nous avons nécessairement $s' \geq s+1$. Il y a ensuite deux sous-cas à considérer.

— Si $s' = s+1$, cela implique que la solution *dantzig* est inchangée. On a alors $U^0 = \sum_{j=1}^{s-1} p_j + \left\lfloor \bar{\omega} \frac{p_{s'+1}}{w_{s'+1}} \right\rfloor$ et $U^1 = \sum_{j=1}^{s-1} p_j + \left\lfloor p_{s'} - (w_{s'} - \bar{\omega}) \frac{p_{s-1}}{w_{s-1}} \right\rfloor$, car il ne faut pas oublier que l'objet s est retiré du problème.

— Si $s' > s+1$, la solution *dantzig* est alors modifiée. Sa valeur devient $\sum_{j=1}^{s-1} p_j + \sum_{j=s+1}^{s'-1} p_j$ et son poids $\sum_{j=1}^{s-1} w_j + \sum_{j=s+1}^{s'-1} w_j$. La capacité résiduelle du sac est donc $\bar{\omega}' = \bar{\omega} - \sum_{j=s+1}^{s'-1} w_j$. On a alors $U^0 = \left(\sum_{j=1}^{s-1} p_j + \sum_{j=s+1}^{s'-1} p_j\right) + \left\lfloor \bar{\omega}' \frac{p_{s'+1}}{w_{s'+1}} \right\rfloor$ et $U^1 = \left(\sum_{j=1}^{s-1} p_j + \sum_{j=s+1}^{s'-1} p_j\right) + p_{s'} - \left\lfloor (w_{s'} - \bar{\omega}') \frac{p_{s'-1}}{w_{s'-1}} \right\rfloor$.

— Si $i \leq s-1$, la solution *dantzig* est alors nécessairement impactée car un des $s-1$ premiers objets est retiré du problème. Pour compléter la nouvelle solution *dantzig*, on retire simplement l'objet d'indice i (on obtient alors une valeur $\sum_{j=1}^{s-1} p_j - p_i$ et un poids $\sum_{j=1}^{s-1} w_j - w_i$), avant de reprendre la complétion de la solution *dantzig* modifiée à partir de l'indice s. Nous notons s' l'indice de l'objet cassé obtenu. La solution *dantzig* obtenue a pour valeur $\sum_{j=1}^{s-1} p_j - p_i + \sum_{j=s}^{s'-1} p_j$ et pour poids $\sum_{j=1}^{s-1} w_j - w_i + \sum_{j=s}^{s'-1} w_j$. La capacité résiduelle du sac est donc $\bar{\omega}' = \bar{\omega} + w_i - \sum_{j=s}^{s'-1} w_j$. On a ensuite deux sous-cas à considérer :

  — Si $s' = i+1$ (cas uniquement possible si $i = s-1$ et $s' = s$), on a alors $U^0 = \left(\sum_{j=1}^{s-1} p_j - p_i + \sum_{j=s}^{s'-1} p_j\right) + \left\lfloor \bar{\omega}' \frac{p_{s'+1}}{w_{s'+1}} \right\rfloor$ et $U^1 = \left(\sum_{j=1}^{s-1} p_j - p_i + \sum_{j=s}^{s'-1} p_j\right) + p_{s'} - \left\lfloor (w_{s'} - \bar{\omega}') \frac{p_{i-1}}{w_{i-1}} \right\rfloor$.

  — Si $s' > i+1$, on a alors $U^0 = \left(\sum_{j=1}^{s-1} p_j - p_i + \sum_{j=s}^{s'-1} p_j\right) + \left\lfloor \bar{\omega}' \frac{p_{s'+1}}{w_{s'+1}} \right\rfloor$ et $U^1 = \left(\sum_{j=1}^{s-1} p_j - p_i + \sum_{j=s}^{s'-1} p_j\right) + p_{s'} - \left\lfloor (w_{s'} - \bar{\omega}') \frac{p_{s'-1}}{w_{s'-1}} \right\rfloor$.

Remarque importante : les formules pour U^0 et U^1 données dans chaque sous-cas n'ont pas pour but d'être appliquées ! L'important est surtout de bien remarquer quel terme est ajouté ou retiré !

5 Pour aller plus loin : utilisation des bornes dans la programmation dynamique

Il y a tous les ans des étudiants qui se plaignent que le projet est trop court et trop facile, mais il est encore possible d'apporter des améliorations à divers niveaux de l'algorithme, en utilisant simplement les éléments fournis dans ce sujet de projet. Il est par exemple possible d'utiliser les bornes introduites pour le prétraitement au coeur de l'algorithme de programmation dynamique. Par exemple, si un état n'est pas filtré par le test de dominance basique, on pourrait calculer une borne supérieure sur la valeur des solutions qui pourraient découler de cet état pour voir s'il est possible d'obtenir une meilleure solution admissible que la meilleure solution connue. Si ce n'est pas le cas, l'état pourrait être filtré... Ce principe se retrouve plus souvent dans les algorithmes de branch and bound mais peut aussi être appliqué dans un algorithme de programmation dynamique. Les enseignants pourront donner des pistes aux étudiants qui s'engageront dans cette voie.

6 Travail à effectuer

Une implémentation des méthodes présentées dans les sections 2, 3 et 4 est attendue. Cela inclut

— La résolution directe du problème du sac à dos par GLPK,

— Les deux variantes de la programmation dynamique,

— Le prétraitement avant un appel à l'algorithme de programmation dynamique.

La partie la plus technique de ce travail est tout de même le prétraitement, si votre implémentation se fait avec le but de calculer efficacement les bornes supérieures. C'est pourquoi il est recommandé de faire la version "simple et bourrine" dans un premier temps.

Les implémentations peuvent être réalisées au choix en langage C ou Julia. Plusieurs fichiers sont disponibles sur madoc dans l'archive `Projet 2020.zip` :

— Un dossier `Instances` contenant des instances numériques à résoudre,

— Le fichier `structKP.jl` fournit quelques structures de données (avec quelques explications sur les structures récursives en Julia) et un parseur pour les instances numériques.

Les fichiers de données suivent le format suivant :

— La première ligne contient le nombre d'objets du problème,
— La seconde ligne contient la capacité du sac,
— La troisième ligne contient les profits des objets,
— La quatrième ligne contient les poids des objets.

La date limite de remise des projets est fixée au dimanche 5 avril à 23h59. Le(s) code(s) source(s) ainsi que le rapport au format `.pdf` devront être remis sur madoc dans une archive (au format `.zip` ou `.tar.gz`). Voici quelques points qui devront impérativement apparaître dans votre rapport.

1. Une présentation et une justification de vos choix de structures de données.
2. Une description du calcul efficace des bornes supérieures si cela a été réalisé.
3. Une présentation éventuelle des difficultés rencontrées.
4. Une présentation des améliorations éventuellement proposées.
5. Une analyse des résultats (temps de résolution pour chacune des méthodes, nombre d'états stockés/filtrés, nombre de variables restantes à l'issue du prétraitement...) issus de la résolution des instances numériques par vos implémentations. Il sera en particulier intéressant de comparer les différentes méthodes implémentées.