

TD3 - Algorithmes d'approximation

OPTIMISATION ET RECHERCHE OPERATIONNELLE

M1 Info - semestre d'automne 2021-2022

UNIVERSITÉ CLAUDE BERNARD LYON 1

Christophe Crespelle

christophe.crespelle@univ-lyon1.fr

Florian Ingels

florian.ingels@ens-lyon.fr

Exercice 1.

Au suivant!

a. Appliquer l'algo *List Scheduling* vu en cours pour l'équilibrage de charge sur les listes suivantes :

— $L_1 = (5, 9, 4, 12, 4, 7, 5)$ et 3 machines

— $L_2 = (4, 4, 5, 5, 7, 9, 12)$ et 3 machines

— $L_3 = (3, 5, 2, 4)$ et 2 machines

— $L_4 = (1, 7, 8, 7, 2, 8)$ et 3 machines

Solution. Pour L_1 , on obtient : $M_1(14) : [5, 4, 5]$, $M_2(16) : [9, 7]$, $M_3(16) : [4, 12]$.

Pour L_2 , on obtient : $M_1(21) : [4, 5, 12]$, $M_2(11) : [4, 7]$, $M_3(14) : [5, 9]$.

Pour L_3 , on obtient : $M_1(5) : [3, 2]$, $M_2(9) : [5, 4]$.

Pour L_4 , on obtient : $M_1(16) : [1, 7, 8]$, $M_2(9) : [7, 2]$, $M_3(8) : [8]$.

b. Pour chacune des solutions obtenues, dites si elle est optimale ou non et prouvez le. Donnez le ratio d'approximation des solutions non optimales.

Solution. Pour L_1 , la solution obtenue est optimale. Dans ce cas, c'est facile à prouver car elle atteint une des bornes inférieures vues en cours. Aucune solution ne peut être meilleure que la moyenne des charges sur les trois machines. Ici, la moyenne est $46/3 > 15$. Donc aucune solution ne peut avoir un *makespan* inférieur à 16, ce qui est le cas de la solution obtenue, qui est donc optimale.

Pour L_2 , on est sûr que la solution n'est pas optimale car les durées sont exactement les mêmes que pour L_1 (L_2 est un réarrangement dans un ordre différent de L_1) et le nombre de machines est aussi le même. Or, le *makespan* de la solution obtenue avec L_2 est de 21, contre 16 avec L_1 : la solution obtenue avec L_2 n'est donc pas optimale et son ratio d'approximation est $21/16$. Remarque : si on nous avait posé la question pour L_2 sans nous donner L_1 , on peut se douter que la solution obtenue avec L_2 n'est pas optimale car la charge des machines n'est pas très équilibrée. On aurait alors essayé de construire une meilleure solution pour prouver que celle obtenue avec L_2 n'est pas optimale.

Pour L_3 , justement, on voit que dans la solution obtenue, la charge sur chacune des deux machines n'est pas très équilibrée : 5 et 9. On a envie d'essayer de construire une solution meilleure, ce qui est assez facile : $M_1(7) : [4, 3]$, $M_2(7) : [5, 2]$. Cette dernière solution est clairement optimale car toutes les machines ont la même charge (la borne inférieure de la charge moyenne des machines est donc atteinte). La solution obtenue avec L_3 n'est donc pas optimale et son ratio d'approximation est $9/7$.

Pour L_4 , on obtient un makespan de 16. On soupçonne qu'on pourrait faire mieux car les charges des machines ne sont pas très équilibrées : 16, 9 et 8. On peut déjà essayer l'algorithme *Longest Processing* (en triant les tâches de la liste par ordre décroissant de durée) pour voir le makespan obtenu. On obtient : $M_1(10) : [8, 2]$, $M_2(9) : [8, 1]$, $M_3(14) : [7, 7]$ et donc un makespan de 14. Cela est encore assez loin de la borne inférieure donnée par la charge moyenne $\frac{10+9+14}{3} = 11$ et de plus, les charges des machines, 10, 9, 14 paraissent assez pauvrement équilibrées. C'est pourtant le makespan optimum ! Pour le montrer, on peut utiliser le raisonnement suivant, qui fait intervenir le principe du pigeonnier ;) Il y a 4 tâches de durée supérieure ou égale à 7 et 3 machines : dans n'importe quelle affectation, il y a donc nécessairement au moins une machine qui reçoit au moins deux tâches de durée supérieure ou égale à 7. Ainsi, cette machine a une charge d'au moins 14 et le makespan de toute affectation est au moins 14. Comme on a trouvé une affectation de makespan 14, c'est le makespan minimum.

On note Π_1 le problème donné par la liste de durée des tâches $T = \{5, 9, 4, 12, 4, 7, 5\}$ et 3 machines ; on note Π_2 le problème donné par $T = \{3, 5, 2, 4\}$ et 2 machines.

c. Pour chacun des deux problèmes Π_1 et Π_2 donnez deux listes qui produisent respectivement le meilleur et le pire résultat pour l'algorithme *List Scheduling*. Donnez le ratio d'approximation de chacune de ces solutions.

d. Prouvez que vos réponses à la question précédente sont bien les meilleurs et les pires résultats pour l'algorithme *List Scheduling*.

Solution. On traite les deux questions ensemble. Remarquez que les deux problèmes proposés sont les mêmes que ceux étudiés aux questions précédentes.

Pour Π_1 on sait donc déjà que l'algorithme *List Scheduling* est capable de trouver une solution optimale, qui a un makespan de 16. Il nous reste donc à trouver le plus grand makespan qu'on peut obtenir par l'algorithme *List Scheduling* et à prouver que c'est bien le plus grand. Pour trouver un grand makespan, il faut essayer de faire une répartition des charges la plus déséquilibrée possible, avec par exemple la tâche la plus longue 12 qui arrive à la fin lorsque le reste était déjà assez équilibré. Par exemple, si on applique l'algorithme *List Scheduling* sur la liste (5, 7, 9, 5, 4, 4, 12) on obtient l'affectation suivante qui réalise un makespan de 22 : $M_1(22) : [5, 5, 12]$, $M_2(11) : [7, 4]$, $M_3(13) : [9, 4]$. Il y a plusieurs façons de montrer que 22 est le plus grand makespan qu'on puisse obtenir par l'algorithme *List Scheduling*, plus ou moins simples et plus ou moins concises. En voici une relativement simple et assez concise.

D'abord, remarquez que si t est la durée de la dernière tâche reçue par la machine de plus grande charge dans l'algorithme *List Scheduling* et S est la somme des durées de toutes les tâches, alors le makespan M obtenu vérifie $M \leq f(t)$, avec $f(t) = t + \frac{S-t}{m}$. En effet, avant de recevoir la tâche de durée t , la machine qui la reçoit était la moins chargée

et avait donc une charge d'au plus la moyenne des charges de toutes les machines a ce moment la. Ensuite, remarquez que la fonction $f(t) = \frac{m-1}{m}t + \frac{S}{m}$ est croissante avec t . Applique au probleme Π_1 , les deux plus grande valeur de $f(t)$ sont donc obtenues pour $t = 12$, $f(12) = 23 + \frac{1}{3}$, et pour $t = 9$, $f(9) = 21 + \frac{1}{3}$. Ainsi, comme toutes les taches sont de durees entieres, on obtient que le plus grand makespan qu'on peut obtenir par l'algo *List Scheduling* est soit 22 (le makespan de la solution qu'on a produite ci-dessus) soit 23 et qu'on ne peut l'obtenir que lorsque la tache de duree 12 est la derniere tache recue par la machine de plus grande charge.

Il reste a savoir si c'est possible d'obtenir un makespan de 23. On prouve que non par l'absurde : supposons qu'on obtienne un makespan de 23, on sait que la machine M_i de plus grande charge est alors celle qui a recu comme derniere tache celle de duree 12. Par consequent, elle avait juste avant une charge d'exactement 11. Il n'y a qu'une facon de faire 11 avec les taches restantes, qui sont de durees 9, 7, 5, 5, 4, 4 : avec 7 et 4. Donc, au moment ou M_i recoit la tache de duree 12, M_i a deja les taches de duree 7 et 4 qui lui sont affectees et les taches de duree 9, 5, 5, 4 (qui somment a 23) sont reparties sur les deux autres machines. Par consequent, l'une de ces 2 autres machines, notee M_j , a une charge d'au plus 10 (la seule solution pour eviter cela serait d'avoir une machine de charge 11 et une de charge 12 mais cela est impossible a realiser avec les durees 9, 5, 5, 4). Ainsi, M_j a une charge (au plus 10) strictement inferieure a celle de M_i (exactement 11) : c'est une contradiction, car dans ce cas c'est M_j qui aurait du recevoir la charge de duree $t = 12$ par l'algo *List Scheduling*. Cela montre qu'il n'existe pas de solution avec un makespan de 23, comme nous l'avions initialement suppose. Ainsi, la solution que nous avons exhibee realise le maximum du *makespan* qu'il est possible d'obtenir par l'algo *List Scheduling*.

Pour Π_2 , on sait que l'optimum est 7 et qu'on peut obtenir 9 par l'algo *List Scheduling*. Peut-on obtenir l'optimum par l'algo *List Scheduling* ? Oui, avec la liste $L = (5, 4, 3, 2)$ par exemple. 9 est-il le *makespan* maximum qu'on peut obtenir par l'algo *List Scheduling* ? On soupconne que oui, prouvons le. En utilisant le raisonnement precedent, c'est direct : la plus grande tache a une duree de 5 et $f(5) = 5 + \frac{9}{2} = 9 + \frac{1}{2}$. Le makespan obtenu par la'go de liste est donc toujours au plus $9 + \frac{1}{2}$ et comme toutes les taches sont de durees entieres, il est d'au plus 9.

Dans ce cas, on aurait pu le montrer sans utiliser $f(t)$ en faisant une disjonction de cas, qui n'est pas tres longue car il y a peu de taches. Si deux taches sont affectees a chacune des deux machines, alors 9 est clairement le maximum qu'on peut obtenir car les durees des deux plus longues taches (5 et 4) somment a 9. La seule facon d'obtenir eventuellement un plus grand *makespan* serait de mettre plus de deux taches sur une des machines. Il doit etre clair que par l'algo *List Scheduling* on ne peut pas avoir une machine qui reste sans tache (des lors qu'il y a au moins autant de taches que de machines, comme c'est le cas ici) car cette machine a la charge minimum a tout moment de l'algo *List Scheduling*. La seule chose qui pourrait arriver dans l'algo *List Scheduling* pour qu'une machine M_i receive au moins 3 taches serait qu'elle en receive exactement 3 et que l'autre machine M_j en receive exactement 1. Dans ce cas lorsque la machine M_i recoit sa troisieme tache, la somme des durees deux premieres taches qu'elle a recues n'excede pas la duree de la tache affectee a l'autre machine (qui a necessairement deja recu une tache). La seule solution avec les quatre taches fournies

est que les deux tâches déjà affectées à M_i soient celles de durée 2 et 3 et que celle affectée à M_j soit celle de durée 5. La dernière tâche, de durée 4, peut alors être affectée à M_i . De cette façon, on obtient bien une machine avec 3 tâches et une avec une seule tâche, mais le *makespan* est encore de $9 = 2 + 3 + 4$. Un *makespan* de 9 est donc le maximum (=le pire) qu'on peut obtenir par l'algorithme *List Scheduling*.

e. Pour chacun des deux problèmes Π_1 et Π_2 quelle est la solution obtenue en appliquant l'algorithme *List Scheduling* sur la liste triée par ordre décroissant de durée des tâches ? Cette solution est-elle optimale ?

Solution. Pour Π_1 , on obtient : $M_1(16) : [12, 4]$, $M_2(14) : [9, 5]$, $M_3(16) : [7, 5, 4]$. D'après ce qui précède, il s'agit bien de l'optimal.

Pour Π_2 , on obtient : $M_1(7) : [5, 2]$, $M_2(7) : [4, 3]$. Il s'agit encore de l'optimum.

f. Donnez un exemple de problème sur deux machines tel que l'algorithme *List Scheduling* qui reçoit en entrée la liste triée par ordre décroissant de durée des tâches ne fournit pas la solution optimale au problème.

Solution. Pour le problème Π donné par $T = \{6, 5, 4, 4, 3\}$ et 2 machines ; avec l'algorithme *List Scheduling* sur la liste par ordre décroissant, on obtient un *makespan* de 12 : $M_1(10) : [6, 4]$, $M_2(12) : [5, 4, 3]$. Alors que l'optimum est 11 : $M_1(11) : [6, 5]$, $M_2(11) : [4, 4, 3]$.

Exercice 2.

Le but de cet exercice est de montrer qu'il existe toujours une liste qui, fournie en entrée à l'algorithme *List Scheduling*, résulte en une affectation des tâches réalisant le *makespan* minimum. On note \mathcal{T} l'ensemble des n tâches et \mathcal{M} l'ensemble des m machines d'une instance du problème d'équilibrage de charges. Soit A une affectation des tâches de \mathcal{T} aux machines de \mathcal{M} , c'est à dire une application de \mathcal{T} dans \mathcal{M} , qui à chaque tâche associe une et une seule machine. Pour une machine $M \in \mathcal{M}$, on note $\mathcal{T}_A(M)$ l'ensemble des tâches affectées à M dans A .

De plus, on définit le profil d'une affectation A , note $p(A)$, comme la séquence décroissante des charges des m machines dans A . Un profil p est donc une séquence de m réels que l'on note $p = (p_1, p_2, \dots, p_m)$ avec $p_1 \geq p_2 \geq \dots \geq p_m$. L'ordre lexicographique définit un ordre total sur les profils. C'est à dire que pour deux profils $p = (p_1, p_2, \dots, p_m)$ et $p' = (p'_1, p'_2, \dots, p'_m)$, avec $p \neq p'$, en notant i le plus petit indice dans $\llbracket 1, m \rrbracket$ tel que $p_i \neq p'_i$, on a $p < p'$ ssi $p_i < p'_i$.

a. Soit A une affectation dont le profil est minimum pour l'ordre lexicographique. Montrez que A réalise le *makespan* minimum.

Solution. Soit A' une affectation qui réalise le *makespan* minimum. On note $p' = (p'_1, p'_2, \dots, p'_m)$ le profil de A' et $p = (p_1, p_2, \dots, p_m)$ le profil de A . Remarquez que le *makespan* d'une affectation n'est autre que le premier élément de son profil. On a donc $\text{makespan}(A') = p'_1$ et $\text{makespan}(A) = p_1$. Comme le profil de A est minimum pour l'ordre lexicographique, on a $p_1 \leq p'_1$, et donc $\text{makespan}(A) \leq \text{makespan}(A')$. Comme A' réalise le *makespan* minimum, cela implique que $\text{makespan}(A) = \text{makespan}(A')$ et que A réalise le *makespan* minimum.

b. Soit A une affectation dont le profil est minimum pour l'ordre lexicographique. Soit $S \subsetneq \mathcal{T}$ un sous-ensemble strict des tâches et soit $A[S]$ l'affectation obtenue en restreignant A aux tâches de S . Montrez que l'ensemble $R_{min} \subseteq \mathcal{M}$ des machines dont la charge est minimum dans $A[S]$ contient au moins une machine M telle que $\mathcal{T}_{A[S]}(M) \neq \mathcal{T}_A(M)$.

Solution. Supposons pour contradiction que ce n'est pas le cas : toutes les machines de R_{min} ont dans l'affectation $A[S]$ les mêmes tâches que dans l'affectation A . Comme S ne contient pas toutes les tâches de \mathcal{T} , il existe au moins une machine $M \in \mathcal{M}$ qui n'a pas reçu dans $A[S]$ toutes les tâches qui lui sont affectées dans A . Par notre supposition, cette machine M n'est donc pas de charge minimum dans $A[S]$. Soit $M' \in R_{min}$, c'est à dire M' de charge minimum dans $A[S]$. On peut former une nouvelle affectation A' en affectant à M' toutes les tâches qui sont affectées à M dans A et qui ne lui sont pas affectées dans $A[S]$. Pour obtenir une contradiction, montrons que $p(A') < p(A)$. Entre A et A' seules les charges de M et M' diffèrent. De plus, comme la charge de M' dans $A[S]$ est strictement supérieure à celle de M , on a $\max\{charge_{A'}(M), charge_{A'}(M')\} < charge_A(M) = \max\{charge_A(M), charge_A(M')\}$. On a donc (prouvez le!), $p(A') < p(A)$: ce qui contredit que A est une affectation dont le profil est minimum pour l'ordre lexicographique. En conséquence, contrairement à notre supposition, il existe une machine de R_{min} qui n'a pas les mêmes tâches dans l'affectation $A[S]$ que dans l'affectation A : $\mathcal{T}_{A[S]}(M) \neq \mathcal{T}_A(M)$.

c. Montrez que pour toute instance du problème d'équilibrage de charges, il existe toujours une liste qui, fournie en entrée à l'algorithme *List Scheduling*, donne la solution optimale au problème sur cette instance.

Indication. Considérez une affectation A dont le profil est minimum pour l'ordre lexicographique et construisez, tâche par tâche, une liste L telle que lorsque L est donnée en entrée à l'algorithme *List Scheduling*, il existe une exécution de l'algorithme qui donne exactement l'affectation A .

Solution. Suivons l'indication en formant pour tout $i \in \llbracket 1, n \rrbracket$ une liste $L_i = (T_1, T_2, \dots, T_i)$ de tâches telle que lorsque L_i est donnée en entrée à l'algorithme *List Scheduling*, il existe une exécution de l'algorithme qui produit l'affectation $A[L_i]$.

Pour $i = 1$, n'importe quelle tâche T_1 convient. En effet, l'algorithme *List Scheduling* appliqué sur une liste à une seule tâche peut placer cette tâche sur n'importe quelle machine, qui sont toutes de charge nulle, et donc il existe une exécution qui la place sur la machine que l'on souhaite, celle à laquelle elle est affectée dans A .

Lorsqu'on a construit la liste L_i , avec $i < n$, on choisit la tâche suivante T_{i+1} à mettre dans la liste de la façon suivante. On considère l'affectation $A[L_i]$ obtenue en restreignant A aux tâches de L_i . D'après la question 2, il existe une machine M de charge minimum dans $A[L_i]$ qui n'a pas reçu toutes les tâches qui lui sont affectées dans A . On prend pour T_{i+1} n'importe quelle tâche dans $\mathcal{T}_A(M) \setminus \mathcal{T}_{A[L_i]}(M)$. Lorsqu'on applique l'algorithme *List Scheduling* à la liste L_{i+1} ainsi formée, par notre hypothèse de récurrence, il existe une exécution qui affecte chaque tâche de L_i comme dans l'affectation $A[L_i]$. Et comme M est de charge minimum dans $A[L_i]$, il existe une exécution qui affecte T_{i+1} à M à l'étape suivante. Ainsi, dans cette exécution, toutes les tâches de L_{i+1} sont affectées comme dans l'affectation $A[L_{i+1}]$ obtenue en restreignant l'affectation A aux

taches de L_{i+1} .

En conclusion, pour $i = n$, on obtient une liste L_n pour laquelle il existe une execution de l'algo *List Scheduling* qui produit exactement l'affectation $A[L_n] = A$.

d. Si les n taches ont des durees deux a deux distinctes, combien y a-t-il de listes distinctes sur les durees des taches ? L'algorithme qui consiste a essayer toutes les listes possibles en entree de l'algorithme de liste est-il un algorithme efficace ? Comparez sa complexite a celle de l'algorithme *brute force* vu en cours.

Solution. Si les n taches ont des durees distinctes il y a exactement autant de listes distinctes sur les durees des taches que de listes sur les taches : $n!$.

Du coup, considerer toutes les listes possibles sur les durees des taches donnerait une complexite de $n!$ dans le pire des cas, ce qui n'est pas du tout efficace.

Neanmoins, cela constitue parfois une amelioration de l'algorithme brute force, qui essaye toutes les affectations et dont la complexite est $O(m^n)$. Si m est tres petit devant n (ce qui est le plus souvent le cas en pratique, ou m est souvent bornee par une constante), l'algorithme brute force est meilleur ! Mais si le nombre de machines disponibles est tres grand, c'est a dire que m est de l'ordre de n , par exemple $m = \frac{n}{K}$, avec K une constante, l'algorithme qui essaye toutes les listes en entree de l'algo *List Scheduling* devient plus rapide.

Exercice 3.

k-center selection

Dans ce probleme, on donne un ensemble S de n sites dans le plan euclidien (mais en toute generalite, n'importe quelle distance symetrique et qui verifie l'inegalite triangulaire permettrait d'obtenir les memes resultats) et on veut selectionner au plus k centres (pas necessairement sur les sites donnees en entree) ou placer des casernes de pompiers pour proteger au mieux les n sites. Le critere que l'on veut minimiser pour l'ensemble des k centres $C = \{c_1, c_2, \dots, c_k\}$ selectionnes est le rayon de couverture $r(C) = \max_{s \in S} \{d(s, C)\}$, ou $d(s, C) = \min_{c \in C} \{d(s, c)\}$. Le but de l'exercice est de concevoir un algorithme d'approximation du rayon de couverture minimum avec k centres, qui ait un ratio d'approximation de 2 sur le rayon de couverture et qui fournisse un ensemble de k centres realisant cet objectif.

Supposons pour commencer qu'on connaisse le rayon de couverture minimum r . On propose alors l'algorithme suivant.

Algorithme 1 : Algorithme de 2-approximation en connaissant le rayon de couverture minimum r .

```
1  $S' \leftarrow S$ ;  
2  $C \leftarrow \emptyset$ ;  
3 tant que  $S' \neq \emptyset$  faire  
4   | Selectionner un  $s \in S'$  quelconque;  
5   |  $C \leftarrow C \cup \{s\}$ ;  
6   | Retirer de  $S'$  tous les sites qui sont a distance au plus  $2r$  de  $s$ ;  
7 si  $|C| \leq k$  alors  
8   | retourner  $C$ ;  
9 sinon  
10  | retourner "le rayon de couverture minimum est  $> r$ ";
```

a. Que retourne l'algorithme? Qu'est-ce qui est approxime ici? Que retourne l'algorithme si le rayon r qui lui est fourni n'est pas le rayon de couverture minimum (contrairement a ce que nous supposons ici)?

Solution. L'algorithme retourne un ensemble de centres C . Ce qui est approxime est le rayon de couverture. Le rayon de couverture $r(C)$ de l'ensemble de centres C retourne par l'algorithme n'est pas le rayon optimal r (que l'on suppose connaitre). Remarquez que l'algorithme utilise effectivement le rayon optimal r , qui doit donc lui etre fourni. Dans le cas ou il n'existe pas d'ensemble de centres ayant un rayon de couverture r (c'est a dire que le rayon r fourni a l'algorithme n'est pas correct, il est inferieur au rayon de couverture minimum), l'algorithme s'en apercoit et retourne "le rayon de couverture minimum est $> r$ ". Par contre, dans le cas ou le rayon r fourni a l'algorithme est superieur au rayon minimum, l'algorithme ne s'en apercoit pas et retourne un ensemble C qui realise une 2-approximation du rayon r fourni en entree a l'algorithme. Mais dans ce cas, ce n'est pas necessairement une 2-approximation du rayon de couverture minimum.

b. Quel est la complexite de cet algorithme?

Solution. Le nombre d'iterations de la boucle tant que de la ligne 3 est au plus n . A chaque iteration, il faut scanner la liste de tous les sites restants dans S' pour en retirer ceux qui doivent l'etre, cela prend un temps $O(|S'|) = O(n)$ car le test de distance avec s (ligne 6) se fait en temps constant. La complexite de l'algorithme peut donc s'exprimer comme $O(n^2)$.

Le but des questions suivantes est de montrer que l'algorithme 1 est correct et realise un rapport d'approximation de 2 sur le rayon de couverture qui lui est fourni en entree. C'est a dire que s'il existe un ensemble d'au plus k centres qui a un rayon de couverture r , alors l'algorithme 1 termine avec $|S| \leq k$ et S realise un rayon de couverture d'au plus $2r$. Soit C^* une solution de rayon de couverture r . Soit C la solution retournee par l'algorithme.

c. Montrez que pour tout $c \in C$, il existe $c^* \in C^*$ tel que $d(c, c^*) \leq r$.

Solution. Une particularite de l'algorithme est que les centres qu'il selectionne appartiennent tous a S (ligne 4), on a donc $C \subseteq S$. Ainsi, comme $c \in C$ implique que $c \in S$,

il existe un centre c^* de la solution optimale C^* qui est a distance au plus r de c , car le rayon de couverture de la solution C^* est r . On a donc $d(c, c^*) \leq r$.

d. Soit $c^* \in C^*$ tel qu'il existe $c \in C$ tel que $d(c, c^*) \leq r$. Montrez que quel que soit $c' \in C \setminus \{c\}$, $d(c', c^*) > r$.

Solution. Considerons le premier $c \in C$ selectionne par l'algorithme tel que $d(c, c^*) \leq r$. Pour tous les centres $c' \in C$ qui ont ete selectionnes avant c dans l'algorithme, on a donc par definition $d(c', c^*) > r$. Soit maintenant un centre $c' \in C$ qui a ete selectionne apres c par l'algorithme. Comme au moment ou c a ete selectionne, l'algorithme retire de S' tous les sites s' tels que $d(s', c) \leq 2r$, on a donc $d(c', c) > 2r$ (car tous les centres selectionnes apres c le sont parmi les sites de S^* qui ne fait que diminuer, au sens de l'inclusion, au cours de l'algorithme). Par l'inegalite triangulaire, on a $d(c', c) \leq d(c, c^*) + d(c', c^*)$ et donc $d(c', c^*) \geq d(c', c) - d(c, c^*)$. Comme $d(c', c) > 2r$ et $d(c, c^*) \leq r$ cela donne $d(c', c^*) > 2r - r = r$.

e. Montrez par l'absurde que si l'algorithme termine avec $|C| > k$ alors il n'existe pas d'ensemble d'au plus k centres qui ait un rayon de couverture r .

Solution. Supposons pour contradiction que l'algorithme termine avec $|C| > k$ et qu'il existe un ensemble C^* , avec $|C^*| \leq k$, qui realise un rayon de couverture de r . D'apres la question c, on peut associer a chaque centre $c \in C$ un centre $assoc(c) \in C^*$. De plus, d'apres la question d, si $c, c' \in C$ et $c \neq c'$ alors necessairement $assoc(c) \neq assoc(c')$. Autrement dit, $assoc$ est une application injective de C dans C^* . Par consequence, on a $|C^*| > |C|$, ce qui est une contradiction. On en conclut que si l'algorithme termine avec $|C| > k$ alors il n'existe pas d'ensemble d'au plus k centres qui ait un rayon de couverture r .

f. En deduire que s'il existe un ensemble d'au plus k centres qui a un rayon de couverture r , alors l'algorithme trouve un ensemble d'au plus k centres qui a un rayon de couverture au plus $2r$.

Solution. Par contraposee de la propriete montree a la question precedente on a que s'il existe un ensemble d'au plus k centres qui ait un rayon de couverture r alors l'algorithme termine avec $|C| \leq k$. Or, lorsque l'algorithme termine S^* est vide. De plus, comme on retire un site de S^* seulement lorsqu'il est a une distance au plus $2r$ d'un centre $c \in C$ selectionne par l'algorithme (ligne 6), C realise toujours un rayon de couverture d'au plus $2r$.

On veut maintenant faire un algorithme de 2-approximation sur le rayon de couverture sans connaitre le rayon de couverture optimal. Pour cela, on remarque qu'a la ligne 4 du precedent algo, on peut choisir n'importe quel site s qui ne soit pas encore couvert par les centres selectionnes auparavant avec un rayon de $2r$.

g. Lorsqu'il existe un site non couvert par les centres selectionnes auparavant dans l'algorithme avec un rayon $2r$, comment peut-on choisir un site s pour etre sur qu'il soit non couvert sans meme connaitre r ?

Solution. S'il existe un site s qui est a une distance strictement superieure a $2r$ de tous les centres C selectionnes jusqu'a lors par l'algorithme, alors le site s dont la distance $d(s, C)$ aux centres de C est maximum satisfait cette condition. Ainsi, dans l'algorithme si on selectionne toujours le site le plus loin des centres deja selectionnes, on est sur de faire un choix valide, s'il en existe un.

h. En utilisant le resultat de la question precedente, ecrivez un algorithme pour le probleme k -center selection qui retourne toujours un ensemble C d'exactly k centres (on supposera $n > k$ sinon le rayon de couverture minimum est 0) et qui garantisse un facteur d'approximation de 2 sur le rayon de couverture minimum, sans supposer cette fois aucune connaissance sur le rayon de couverture minimum.

Solution. En suivant l'idee de la question precedente, on peut eviter le critere de la ligne 6 de l'algorithme 1, qui utilisait r , en selectionnant a la ligne 4 non pas un site quelconque mais le site le plus eloigne des centres deja selectionnes. On ne peut plus non plus gerer l'ensemble S' dont le maintient (ligne 6) supposait la connaissance de r . Ce qui nous oblige a changer la condition d'arret de la boucle. A la place, on peut executer la boucle exactement k fois pour selectionner exactement k sites comme centres, a chaque fois le site le plus eloigne des centres jusqu'alors selectionnes. Le nouvel algorithme ainsi obtenu, qui n'a pas connaissance du rayon minimum de couverture r , est decrit dans l'algorithme 2.

Algorithme 2 : Algorithme de 2-approximation sans connaitre le rayon de couverture minimum.

```

1  $C \leftarrow \emptyset$ ;
2 tant que  $|C| < k$  faire
3   Selectionner  $s \in S$  tel que  $d(s, C)$  est maximum;  $\backslash\backslash$  si  $C = \emptyset$ ,  $d(s, C) = +\infty$ 
4    $C \leftarrow C \cup \{s\}$ ;
5 retourner  $C$ ;
```

Il se peut que l'algorithme 2 selectionne des centres "inutiles" pour realiser un rayon de couverture d'au plus $2r$ (ou r est le rayon minimum) car il se peut que ce rayon de couverture de $2r$ soit realise avant d'avoir selectionne les k centres. Mais ce n'est pas un probleme : les k centres selectionnes, meme trop nombreux, realisent quand meme un rayon de couverture de $2r$. L'algorithme 2 n'a pas la possibilite de s'en apercevoir avant car il ne connait pas r .

i. Quel est la complexite de l'algorithme que vous avez propose a la question precedente?

Solution. La boucle s'execute toujours k fois. A l'interieur, pour chacun des n site s il faut calculer la distance de s a chacun des au plus k centres deja selectionnes, ce qui demande un temps $O(kn)$. Au total la complexite est donc $O(k^2n)$. On peut facilement ameliorer l'implementation pour obtenir une complexite totale de $O(kn)$. Pour cela, on peut maintenir dans un tableau $dist[]$ de taille n la distance $dist[s] = d(s, C)$ de chaque site s a l'ensemble C des centres deja selectionnes. Au depart, $dist[s] = +\infty$ pour tout s et a chaque fois qu'on selectionne un nouveau centre c on actualise $dist[s]$ pour tous les sites selon la formule $dist[s] \leftarrow \min\{dist[s], d(s, c)\}$. Cela prend un temps $O(n)$ a chaque iteration de la boucle et la selection du nouveau centre c prends aussi $O(n)$. Comme la boucle s'execute k fois on obtient une complexite totale de $O(kn)$.

j. Prouvez que le rayon de couverture \tilde{r} des k -centres retournees par votre algorithme est une 2-approximation du rayon de couverture minimum.

Solution. Pour prouver le rapport d'approximation, on peut se ramener au rapport qu'on a prouvé pour l'Algorithme 1. Exécutez l'algorithme 2 en maintenant en plus un ensemble S' comme dans l'algo 1 : on initialise S' à S au début de l'algo 2 et à chaque fois que l'algo 2 sélectionne un nouveau centre c , on retire de S' tous les sites qui sont à distance au plus $2r$ de c (comme dans l'algo 1), où r est le rayon minimum de couverture. Comme nous l'avons remarqué à la question g, le choix du nouveau centre qui est fait par l'algo 2 est aussi un choix valide pour l'algo 1, tant que S' n'est pas vide. Or, nous avons montré à la question f, que si r est bien le rayon de couverture minimum, l'algo 1 réalise un rayon de couverture au plus $2r$ avec $k' \leq k$ centres. Ainsi, après k' itérations dans l'algo 2, on a exactement les mêmes centres sélectionnés et ils ont donc un rayon de couverture d'au plus $2r$. Le fait que l'algo 2 sélectionne d'autres centres par la suite ne peut pas faire augmenter le rayon de couverture et à la fin de l'algo 2, le rayon de couverture de l'ensemble de centres C retourné est donc au plus $2r$.