République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique
ECOLE SUPÉRIEURE EN INFORMATIQUE
8 Mai 1945 - Sidi-Bel-Abbès

الجمهورية الجزائرية الديمقراطية الشعبية
وزارة التعليم العالي والبحث العلمي
المدرسة العليا للإعلام الآلي
8 ماي 1945 - سيدي بلعباس

Département : Cycle Supérieur

Laboratoire : LABRI

Polycopié pédagogique

Auteur

BEKKOUCHE Mohammed

Titre

Complexité et Résolution de Problèmes : Manuel d'exercices corrigés

Cours destiné aux étudiants de

Second Cycle : 2ieme année Cycle Supérieur, spécialité : Intelligence Artificielle et Sciences de Données (IASD)

Année : 2022-2023

Table des matières

Résumé

L'étude de la complexité des problèmes permet de savoir si un problème peut être résolu par un algorithme et combien de ressources (en termes de temps et d'espace) faudra-t-il pour résoudre ce problème par un algorithme. La résolution des problèmes peut être effectuée selon différentes méthodes. Certaines stratégies vont résoudre un problème plus efficacement que d'autres. Ce polycopié présente des exercices corrigés sur la complexité algorithmiques et des problèmes ainsi que sur des méthodes de résolution de problèmes en intelligence artificielle.

Introduction

Ce polycopié est destiné aux étudiants de deuxième année de cycle supérieur, spécialité : Intelligence Artificielle et Sciences de Données (IASD), de l'école supérieure en informatique de Sidi Bel Abbès. Il sert de manuel d'exercices corrigés pour le module Complexité et Résolution de Problèmes enseigné au premier semestre. Les lecteurs doivent avoir suivi les cours d'algorithmique et de structures de données statiques, ainsi que d'algorithmique et de structures de données dynamiques. Des connaissances en théorie des graphes et en langages sont également nécessaires.

Ce document est une compilation d'exercices corrigés conçus pour les étudiants ayant déjà suivi le cours sur la complexité et la résolution de problèmes. Il met l'accent sur les aspects pratiques de ces concepts et vise à permettre aux étudiants de tester leurs connaissances et compétences acquises.

Les objectifs de ce manuel d'exercices corrigés sont les suivants :

- Mettre en pratique les notions de complexités algorithmiques.
- Estimer le temps d'exécution pour pouvoir comparer plusieurs algorithmes traitant du même problème.
- Approfondir la compréhension des différentes notions permettant de classer les problèmes.
- Reconnaître les problèmes qui ne peuvent pas être résolus ou pour lesquels il est difficile de concevoir une solution efficace.
- Consolider les connaissances théoriques et pratiques liées à la complexité et à la résolution de problèmes.
- Appliquer les concepts à des problèmes concrets dans le domaine de l'intelligence artificielle.
- Familiariser avec la démarche de résolution et de construction de solutions.
- S'entraîner à utiliser différentes méthodes pour la résolution de problèmes.

Ce manuel constitue donc une ressource complémentaire qui permettra aux lecteurs d'évaluer leurs compétences et de renforcer leur compréhension pratique de la complexité et de la résolution de problèmes.

Ces notions seront présentées à travers des problèmes provenant de différents domaines de l'informatique.

Par conséquent, ce document fournit également un support pédagogique aux étudiants pour leur permettre d'acquérir et d'améliorer leurs compétences en programmation.

Ce manuel d'exercices corrigés est structuré en trois chapitres comme suit :

Le premier chapitre est dédié à l'étude de la complexité des algorithmes. Il présente des exercices qui permettent de comprendre la notion de complexité temporelle, son utilité et les méthodes pour la calculer.

Le deuxième chapitre traite de la complexité des problèmes. Il aborde les différentes classes de complexité auxquelles un problème peut appartenir, en fonction de la complexité des algorithmes connus pour le résoudre.

Le troisième chapitre est consacré à la résolution de problèmes en intelligence artificielle. Il propose des exercices portant sur les méthodes de recherche de solutions dans l'espace d'états, ainsi que sur les techniques de résolution par décomposition de problèmes.

Ce manuel contient de nombreux exercices accompagnés de leurs corrigés. En outre, une liste de références bibliographiques est fournie à la fin du livre, sur lesquelles nous nous sommes appuyés pour rédiger cet ouvrage.

Chapitre 1

Complexité des algorithmes

1.1 Introduction

La question de performance est au cœur de l'informatique. Les utilisateurs sont satisfaits si les problèmes sont traités dans un délai raisonnable. Souvent, peu d'attention est portée à la mesure précise du temps d'exécution, mais seulement à une approximation de ce temps. Si le temps obtenu est perçu comme relativement long, se posera la question d'une solution plus efficace. La machine étant fixée, l'amélioration portera alors sur la façon dont sont conçus les algorithmes en fonction du volume n de données à traiter. Par exemple, si, en doublant la quantité de données, un algorithme renvoie un résultat en multipliant le temps d'exécution par 2, et qu'un autre algorithme renvoie un résultat en multipliant le temps d'exécution par $n > 2$, on choisira évidemment le premier : on dit que sa complexité temporelle est meilleure. Ajoutons que l'on peut également définir sa complexité spatiale pour tenir compte de l'espace mémoire occupé lors de l'exécution de l'algorithme. Plus sa complexité est grande, plus le programme qui implémente l'algorithme a besoin de zones mémoire pour stocker des données. Dans ce polycopié, on s'intéresse à l'étude de la complexité temporelle pour les raisons suivantes :

— Les machines actuelles sont très robustes; avec des capacités matérielles extraordinaires.

— On ne s'intéressera donc seulement qu'à la complexité temporelle car la complexité spatiale étant en général moins critique.

Les grands points de ce chapitre sont : mesure asymptotique et grandeurs des fonctions, notations de Landau, optimalité d'un algorithme, equation de récurrences et technique de résolution. Il va permettre aux étudiants d'acquérir les compétences nécessaires afin d'analyser la complexité d'algorithmes, ce qui permettra de concevoir des algorithmes corrects et efficaces pour la résolution d'un problème donné.

1.2 Temps d'exécution et taille des données

Exercice 1 (inspiré de (Oussous & Wegrzynowski, 2008) et de (Cohen, s. d.))
Soit un ordinateur exécutant 40 mille milliards (40×10^{12}) instructions par seconde. On exécute un algorithme qui utilise, pour une donnée de taille, n, $f(n)$ instructions, $f(n)$ étant n, n^2, n^3, $\log n$, $n \log(n)$ et 2^n.

1. Remplissez un tableau qui donne, en fonction de la taille $n = 10, 20, 60$ et 100, et de la fonction $f(n)$, la durée d'exécution de l'algorithme.
2. Reclassez le tableau en fonction de l'ordre croissant des $f(n)$.

Solution
40×10^{12} instructions par seconde signifie que toute instruction possède une durée d'exécution de $1/(40 \times 10^{12}) = 25 \times 10^{-15}$

$f(n)$ \ n	10	20	60	100
$\log n$	$\log(10) \times 25 \times 10^{-15}$ $= 57,56 \times 10^{-15}$	$\log(20) \times 25 \times 10^{-15}$ $= 74,89 \times 10^{-15}$	$\log(60) \times 25 \times 10^{-15}$ $= 10,24 \times 10^{-14}$	$\log(100) \times 25 \times 10^{-15}$ $= 11,51 \times 10^{-14}$
n	$10 \times 25 \times 10^{-15}$ $= 25 \times 10^{-14}$	$20 \times 25 \times 10^{-15}$ $= 50 \times 10^{-14}$	$60 \times 25 \times 10^{-15}$ $= 15 \times 10^{-13}$	$100 \times 25 \times 10^{-15}$ $= 25 \times 10^{-13}$
$n \log(n)$	$10 \log(10) \times 25 \times 10^{-15}$ $= 57,56 \times 10^{-14}$	$20 \log(20) \times 25 \times 10^{-15}$ $= 14,98 \times 10^{-13}$	$60 \log(60) \times 25 \times 10^{-15}$ $= 61,41 \times 10^{-13}$	$100 \log(100) \times 25 \times 10^{-15}$ $= 11,51 \times 10^{-12}$
n^2	$10^2 \times 25 \times 10^{-15}$ $= 25 \times 10^{-13}$	$20^2 \times 25 \times 10^{-15}$ $= 10 \times 10^{-12}$	$60^2 \times 25 \times 10^{-15}$ $= 90 \times 10^{-12}$	$100^2 \times 25 \times 10^{-15}$ $= 25 \times 10^{-11}$
n^3	$10^3 \times 25 \times 10^{-15}$ $= 25 \times 10^{-12}$	$20^3 \times 25 \times 10^{-15}$ $= 20 \times 10^{-11}$	$60^3 \times 25 \times 10^{-15}$ $= 54 \times 10^{-10}$	$100^3 \times 25 \times 10^{-15}$ $= 25 \times 10^{-9}$
2^n	$2^{10} \times 25 \times 10^{-15}$ $= 25,6 \times 10^{-12}$	$2^{20} \times 25 \times 10^{-15}$ $= 26,21 \times 10^{-9}$	$2^{60} \times 25 \times 10^{-15}$ $= 28,82 \times 10^{3}$	$2^{100} \times 25 \times 10^{-15}$ $= 31,69 \times 10^{15}$

Remarque

— Le tableau ci-dessus est calculé en utilisant un logarithme népérien.

— Il est intéressant de remarquer que $31,69 \times 10^{15}$ secondes $\approx$ 1 004 883 307,96 ans.

1.3 Algorithme : correction et complexité

Exercice 2

Soit un algorithme Tri-Selection pour trier n nombres stockés dans un tableau A. Tri-Selection commence par trouver le plus petit élément de A et l'échange avec la première case de A. Puis, l'algorithme trouve le second plus petit élément et l'échange avec la deuxième case. Et ainsi de suite avec les autres éléments.

1. Écrivez l'algorithme en pseudo-code.
2. Exprimez l'invariant de la boucle de l'algorithme et prouver sa correction.
3. Trouvez le coût dans le meilleur des cas, et le pire des cas.
4. Calculez la complexité de l'algorithme à la base des opérations fondamentales.

Solution

1. Le pseudo-code de l'algorithme de tri par sélection :

Algorithme 1 : Tri-Selection(A)

Data : Une séquence de n nombres $A = \langle a_1, a_2, ..., a_n \rangle$
Result : Une permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ des permutations de la suite d'entrée telle que $a'_1 \leq a'_2 \leq ... \leq a'_n$

```
1  for i ← 1 to longueur(A) − 1 do
2  |  m ← i ;
3  |  for j ← i + 1 to longueur(A) do
4  |  |  if A[j] < A[m] then
5  |  |  |  m ← j ;
6  |  |  end
7  |  end
8  |  temps ← A[i];
9  |  A[i] ← A[m];
10 |  A[m] ← temps;
11 end
```

2. L'invariant de la boucle 2 de l'algorithme : Au début de chaque itération de la boucle for, A[m] est le plus petit élément de A[i..j-1]
   On exploitera cette propriété (l'invariant) pour démontrer que la boucle 2 de l'algorithme Tri-Selection est correcte, solution inspirée de (Cormen, Leiserson, Rivest, & Stein, 2022) :
   — **INITIALISATION :** Avant la première itération $i = m$ et $j = i + 1$. A[m] est le seul élément de A[i..i] qui est trivialement le plus petit élément.
   — **CONSERVATION :** Ensuite, nous abordons la deuxième partie : montrer que chaque itération maintient l'invariant de la boucle. Nous supposons que pour j l'invariant de boucle est vrai, c'est-à-dire que A[m] est le plus petit de A[i..j-1]. Pour la prochaine itération de la boucle où $j = j + 1$, si $A[j + 1] < A[m]$ nous mettons à jour m à j + 1. Maintenant A[m] est le nouveau minimum dans A[i..j]. Sinon on ne fait rien,

dans ce cas A[m] reste le plus petit de A[i..j]. En fin d'itération juste après le "j ← j+1" le tableau A[i..j] précédemment évoqué devient donc le tableau A[i..j-1]. Nous pouvons donc affirmer que si l'invariant est vrai avant une itération de la boucle 2, il le reste avant l'itération suivante.

— **TERMINAISON :** La condition qui provoque la fin de la boucle est j = n+1 (pour un tableau comportant n élément). Maintenant, par la condition de terminaison et l'invariant de boucle, nous pouvons conclure que A[m] est le plus petit de A[i..n], ce qui signifie que nous avons trouvé l'élément minimum A[m] dans le sous-tableau A[i..n] (l'objectif de la boucle 2).

L'invariant de la boucle 1 de l'algorithme : Au début de chaque itération de boucle, A[1..i-1] contient les i-1 plus petits éléments de A[1...n] mais dans un ordre trié.
On exploitera cette propriété (l'invariant) pour démontrer que l'algorithme TRI-SELECTION est correct.
Preuve de la correction de TRI-SELECTION, solution inspirée de (Cormen et al., 2022) :

— **INITIALISATION :** Avant la 1re itération de la boucle 1, le tableau A[1..i-1] contient aucun élément (i=1) et notre invariant est vrai.

— **CONSERVATION :** Nous devons montrer que chaque itération maintient la propriété invariante. Nous supposons que pour i le prédicat est vrai, A[1..i-1] contient i-1 plus petits éléments du tableau d'origine dans l'ordre trié. La boucle 2 permet de trouver l'indice du minimum dans le sous-tableau A[i..n] (pour un tableau comportant n élément). À la fin de cette boucle, ce minimum est échangé avec la case A[i]. Donc, A[1..i] contient les i plus petits éléments du tableau d'origine dans l'ordre trié. En fin d'itération juste après le "i ← i+1" le tableau t[1..i] précédemment évoqué devient donc le tableau t[1..i-1]. Nous pouvons donc affirmer que si l'invariant est vrai avant une itération de la boucle 1, il le reste avant l'itération suivante.

— **TERMINAISON :** La condition de terminaison qui est $i = n$ avec l'invariant implique que A[1..n-1] contient $n - 1$ plus petits éléments du tableau d'origine dans l'ordre trié. Puisque A[1..n-1] contient les n-1 plus petits éléments du tableau A[1..n], automatiquement le dernier élément qui est A[n] est supérieur ou égal à tous les éléments de A[1 ..n-1] et se trouve donc au bon endroit dans le tableau trié. D'où la correction de l'algorithme.

3. Le coût dans le meilleur des cas, et le pire des cas (solution inspirée de (Lebbah, 2021)) :
Une instruction i qui s'exécute en un temps c_i et qui est exécutée n fois interviendra pour $c_i n$. Pour calculer $T(n)$, le temps d'exécution de TRI-SELECTION, on additionne les produits des coûts par le nombre de fois.

— Si la suite est déjà ordonnée (meilleur des cas), on obtient :
$T_1(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=1}^{n-1}(n-i+1) + c_4 \sum_{i=1}^{n-1}(n-i) + c_8(n-1) + c_9(n-1) + c_{10}(n-1)$

— Si la suite est dans un ordre inverse, on se trouve dans le pire des cas :
$T_2(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=1}^{n-1}(n-i+1) + c_4 \sum_{i=1}^{n-1}(n-i) + c_5 \sum_{\mathbf{i=1}}^{\mathbf{\frac{n}{2}}}(\mathbf{2i-1}) + c_8(n-1) + c_9(n-1) + c_{10}(n-1)$

La seule différence entre $T_1(n)$ et $T_2(n)$ est que le coût de l'instruction 5 multiplié par le nombre de fois où est exécutée cette instruction [1] n'est pas inclus dans $T_1(n)$ alors qu'il est dans $T_2(n)$, cela parceque l'instruction 5 n'est jamais exécuté dans le meilleur des cas (tableau initial déjà trié).
Sachant que :
$\sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1}(i) = \frac{(n-1)n}{2}$, $\sum_{i=1}^{n-1}(n-i+1) = \sum_{i=2}^{n}(i) = \frac{(n-1)(2+n)}{2}$ et $\sum_{i=1}^{\frac{n}{2}}(2i-1) = \frac{n^2}{4}$

On obtient :
$T_1(n) = c_1 n + c_2(n-1) + c_3(\frac{(n-1)(2+n)}{2}) + c_4(\frac{(n-1)n}{2}) + c_8(n-1) + c_9(n-1) + c_{10}(n-1)$

$T_1(n) = (c_3/2 + c_4/2)n^2 + (c_1 + c_2 + c_3/2 - c_4/2 + c_8 + c_9 + c_{10})n + (-c_2 - c_3 - c_8 - c_9 - c_{10})$

$T_2(n) = (c_3/2 + c_4/2 + c_5/4)n^2 + (c_1 + c_2 + c_3/2 - c_4/2 + c_8 + c_9 + c_{10})n + (-c_2 - c_3 - c_8 - c_9 - c_{10})$

$T_1(n)$ et $T2(n)$ sont de la forme $an^2 + bn + c$ où a, b et c sont des constantes, donc l'algorithme est $O(n^2)$ dans le meilleur et le pire des cas.

4. L'opération élémentaire la plus importante (fondamentale) dans l'algorithme TRI-SELECTION est la condition $A[j] < A[m]$ (ligne 4). Soit $Min(n)$, $Max(n)$ et $Moy(n)$ représentent respectivement la complexité de l'algorithme dans le meilleur, le pire et en moyenne.

$$Min(n) = Max(n) = Moy(n) = \sum_{i=1}^{n-1}(n-i) = \frac{(n-1)n}{2} = n^2/2 - n/2 = O(n^2)$$

1. On suppose que n (la taille du tableau) est pair. Cette hypothèse n'affecte pas l'ordre de grandeur asymptotique de la complexité de l'algorithme.

1.4 Récursivité et l'approche diviser pour régner

Exercice 3

La méthode dite **diviser pour régner** se décompose en trois phases (Gaudel, Soria, & Froidevaux, 1987 ; Lebbah, 2021) :

Diviser : On divise les données initiales en plusieurs **sous-parties**.

Régner : On résout **récursivement** chacun des sous-problèmes associés (ou on les résout directement si leur taille est assez petite).

Combiner : On **combine** les différents résultats obtenus pour obtenir une solution au **problème initial**.

— Donnez l'algorithme qui calcul le maximum d'un tableau de nombres à l'aide de cette méthode.

— Traduisez la complexité de l'algorithme obtenu par une relation de récurrence.

Solution

— Calcul du maximum d'un tableau de nombres :
En adoptant le paradigme "diviser pour régner", l'idée pour résoudre cette question est de calculer récursivement le maximum de la première moitié du tableau et celui de la seconde, puis de les comparer. Le plus grand des deux sera le maximum de tout le tableau. La condition d'arrêt à la récursivité sera l'obtention d'un tableau à deux ou à un seul élément(s), son maximum étant bien sûr le maximum des deux éléments et la valeur de l'élément respectivement.
Voici donc les trois étapes de la résolution de ce problème via cette méthode :

Diviser le tableau en deux sous-tableaux en le "coupant" par la moitié.

Calculer récursivement le maximum de chacun des sous-tableaux. Arrêter la récursion lorsque les tableaux n'ont plus que deux ou un seul élément(s).

Retourner le plus grand des deux maximums précédents.

Voici donc l'algorithme :

Algorithme 2 : Calcul du maximum d'un tableau de nombre

Data : TAB un tableau de nombres, *debut* et fin sont l'indice recpectivement du début et de la fin de TAB.
Result : Le nombre maximum de TAB.

```
1  function MAXIMUM(TAB, debut, fin) returns un élément:
2      if debut = fin then
3          return TAB[debut] ;
4      else if fin − debut = 1 then
5          if TAB[debut] > TAB[fin] then
6              return TAB[debut] ;
7          else
8              return TAB[fin] ;
9          end
10     milieu ← ⌊(debut+fin)/2⌋ ;
11     x ← MAXIMUM(TAB, debut, milieu) ;
12     y ← MAXIMUM(TAB, milieu + 1, fin) ;
13     if x > y then
14         return x ;
15     else
16         return y ;
17     end
```

— La relation de récurrence :

$$T(n) = \begin{cases} \Theta(1) & \text{si } n \leq 2 \\ 2T(\frac{n}{2}) + \Theta(1) & \text{, pour } n > 2 \end{cases}$$

On démontrera plus loin que $T(n) = \Theta(n)$.

1.5 Types d'analyse (meilleur et pire des cas et cas moyenne)

Exercice 4

Soit l'algorithme MATMULT de multiplication de deux matrices (Lebbah, 2021).

Algorithme 3 : MatMults(A, B)

Data : $A = a_{ij}$, $B = b_{ij}$ deux matrices $n \times n$ à coefficients dans $\mathbb{R}$
Result : $C = A \times B$

```
1 for i ← 1 to n do
2 |   for j ← 1 to n do
3 |   |   C[i, j] ← 0 ;
4 |   |   for k ← 1 to n do
5 |   |   |   C[i, j] ← C[i, j] + A[i, k] ∗ B[k, j] ;
6 |   |   end
7 |   end
8 end
```

— Calculez la complexité de MatMult.

Solution

Les opérations fondamentales dans MatMults multiplication et addition de deux nombres (de réels) (ligne 5).

$$Min(n) = Max(n) = Moy(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} 2 = 2n^3 = O(n^3)$$

Exercice 5

Soit l'algorithme RechSeq de recherche séquentielle d'un élément dans une liste.

Algorithme 4 : RechSeq(L, X)

Data : L une liste à n éléments et X un élément quelconque
Result : Recherche la place j de l'élément X dans L. Si X n'est pas dans la liste, j est mis à zéro

```
1 j ← 1 ;
2 while (j ≤ n) ∧ (L[j] ≠ X) do
3 |   j ← j + 1 ;
4 end
5 if (j > n) then
6 |   j ← 0 ;
7 end
```

— Déterminez sa complexité dans le meilleur, pire et en moyenne des cas.

Solution (inspirée de (Gaudel et al., 1987) et de (Zampieri, Riviere, & Amerein-Soltner, 2018))

L'opération fondamentale dans RechSeq est la comparaison entre cet élément et les entrées de la liste ($L[j] \neq X$) (ligne 2).

Dans le meilleur des cas, l'élément X se trouve dans la première case de la liste, d'où $Min(n) = 1 = O(1)$.

Dans le pire des cas, X se trouve dans la dernière cas ou n'est pas présent dans la liste, d'où $Max(n) = n = O(n)$.

Pour calculer $Moy(n)$, on doit se donner des probabilités sur L et X ; on suppose que tous les éléments sont distincts :

— On note q la probablité que X soit dans L ;

— On suppose que si X est dans L, toutes les places sont équiprobables.

On note D_n^i pour $1 \leq i \leq n$, l'ensemble des données où x est à la i-ème place et D_n^0 l'ensemble des données où x est absent. D'après les conventions ci-dessus, on a :

$$p(D_n^i) = \frac{q}{n} \qquad \text{et} \qquad p(D_n^0) = 1 - q$$

D'après l'analyse de l'algorithme, on a aussi :

$$c(D_n^i) = i \qquad \text{et} \qquad c(D_n^0) = n$$

Par conséquent :

$$\begin{aligned} Moy(n) &= \sum_{i=0}^{n} p(D_n^i).c(D_n^i) \\ &= (1-q).n + \sum_{i=1}^{n} \frac{q}{n}.i \\ &= (1-q).n + \frac{q}{2n}.n.(n+1) \\ &= (1-q).n + \frac{q}{2}.(n+1) \end{aligned}$$

Si on sait que X est dans la liste, on a $q = 1$ d'où :

$$Moy(n) = \frac{n+1}{2} = O(n)$$

Si X a une chance sur deux d'être dans la liste, on a $q = \frac{1}{2}$ d'où :

$$Moy(n) = \frac{n}{2} + \frac{1}{4}(n+1) = \frac{3n+1}{4} = O(n)$$

Exercice 6

Donnez la complexité temporelle des algorithmes suivants ("Complexité d'un algorithme", 2021/2022) :

Algorithme 5 : $F_1(n)$

```
1 x ← 0 ;
2 i ← n ;
3 while i > 1 do
4 |   x ← x + 1 ;
5 |   i ← ⌊i/2⌋ ;
6 end
7 return x ;
```

Algorithme 6 : $F_2(n)$

```
1 x ← 0 ;
2 i ← n ;
3 while i > 1 do
4 |   for j ← 1 to n do
5 |   |   x ← x + 1 ;
6 |   end
7 |   i ← ⌊i/2⌋ ;
8 end
9 return x ;
```

Algorithme 7 : $F_3(n)$

```
1 x ← 0 ;
2 i ← n ;
3 while i > 1 do
4 |   for j ← 1 to i do
5 |   |   x ← x + 1 ;
6 |   end
7 |   i ← ⌊i/2⌋ ;
8 end
9 return x ;
```

Solution

1. On peut prendre comme opération fondamentale dans l'algorithme F_1 l'addition $x+1$ à la ligne 4.
   Le nombre n est une mesure de la taille du problème considéré.
   Pour chaque boucle sur i, on réalise une fois l'opération fondamentale. Ainsi, si p est le nombre d'itérations sur i, la complexité temporelle est en $O(p)$.
   Déterminons une majoration de p.
   - Itération 1 : $i \leq \frac{n}{2}$
   - Itération 2 : $i \leq \frac{n}{4}$
   - Itération 3 : $i \leq \frac{n}{8}$
     ...
   - Itération $p-1$: $i \leq \frac{n}{2^{p-1}}$

   Après $p-1$ itéraitons, $i=1$, d'où :
   $1 \leq \frac{n}{2^{p-1}} \iff 2^{p-1} \leq n \iff p-1 \leq \log n \iff p \leq \log n + 1$. Ainsi, $p = O(\log n)$.
   La complexité tomporelle de l'algorithme F_1 : $Min_{F_1}(n) = Max_{F_1}(n) = Moy_{F_1}(n) = O(\log n)$.
2. Comme l'algorithme F_1, on prend comme opération fondamentale pour F_2 l'addition $x+1$ à la ligne 5.
   Le nombre n est une mesure de la taille du problème considéré.
   Pour chaque valeur i, on réalise n boucles sur j.
   Pour chaque itération de la boucle sur j (**for**), on réalise une seule fois l'opération fondamentale. Ainsi, si p est le nombre de boucle sur i, la complexité temporelle des boucles imbriquées est $O(n \times p)$.
   Dans la question présécente, on a montré que $p = O(\log n)$.
   Finalement, la complexité temporelle, $Min_{F_2}(n) = Max_{F_2}(n) = Moy_{F_2}(n)$, est en $O(n \log n)$.
3. Pareil, on prend l'opération $x+1$ (ligne 5) comme fondamentale pour l'algorithme F_3.
   Le nombre n est une mesure de la taille du problème considéré.
   Pour chaque valeur $i \in [1, n]$, on réalise i boucle sur j. Quand pour une valeur de i donnée, toutes les boucles sur j sont exécutées, l'instruction $\lfloor \frac{i}{2} \rfloor$ divise i par 2, et donc le nombre de boucles sur j par 2. Ainsi,
   - Pour $i == n$, on a n boucles sur j.
   - Pour $i == \lfloor \frac{n}{2} \rfloor$, on a $\lfloor \frac{n}{2} \rfloor$ boucles sur j, nombre majoré par $\frac{n}{2}$.
   - Pour $i == \lfloor \frac{\lfloor \frac{n}{2} \rfloor}{2} \rfloor == \lfloor \frac{\frac{n}{2}}{2} \rfloor$ [2], on a $\lfloor \frac{n}{2^2} \rfloor$ boucles sur j, nombre majoré par $\frac{n}{2^2}$.
     ...
   - Pour $i == \lfloor \frac{n}{2^{p-1}} \rfloor == 1$, le nombre de boucle sur j est majoré par $\frac{n}{2^{p-1}}$

   Par conséquent, le nombre totale S de boucles sur (i, j) est majoré par :

   $$n + \frac{n}{2} + \frac{n}{2^2} + \frac{n}{2^3} + ... + \frac{n}{2^{p-1}}$$

   C-à-d, $S \leq n(1 + \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + ... + \frac{1}{2^{p-1}})$
   C-à-d, $S \leq n(\frac{1-\frac{1}{2^p}}{1-\frac{1}{2}}) \iff S \leq 2n(1-\frac{1}{2^p}) \iff S \leq 2n - \frac{n}{2^{p-1}}$
   On sait que $\lfloor \frac{n}{2^{p-1}} \rfloor == 1$ donc $(1 \leq \frac{n}{2^{p-1}} < 2)$ [3]
   C-à-d, $S \leq 2n$
   Pour chaque itération de la boucle sur j (**for**), on réalise une seule fois l'opération fondamentale. Donc, la complexité temporelle des boucles imbriquées sur i et j est en $O(n)$.
   Finalement, la complexité temporelle, $Min_{F_3}(n) = Max_{F_3}(n) = Moy_{F_3}(n)$, est en $O(n)$.

Exercice 7 (inspiré de (Lebbah, 2021))
On considère un tableau $A[1..n]$ trié dans l'ordre croissant et un nombre X. On veut profiter de l'ordre de A afin d'accélérer la recherche de X dans A. Le principe de base est simple : on compare la valeur cherchée x avec celle au milieu du tableau ; si celle de x est plus petite, on cherche à gauche du milieu du tableau, sinon on cherche à sa droite ; ensuite, on recommence.

1. Écrivez une version itérative de l'algorithme RechDicho de recherche dichotomique en pseudo-code.
2. Quelle est sa complexité ?

2. Si $n > 0$ alors $\lfloor \frac{\lfloor x \rfloor}{n} \rfloor = \lfloor \frac{x}{n} \rfloor$
3. $\lfloor x \rfloor = m$ si et seulement si $m \leq x < m+1$

Solution

1. Le pseudo-code de l'algorithme de recherche dichotomique :

Algorithme 8 : RechDicho(A, X)

Data : $A[1..n]$ un tableau à n éléments trié dans l'ordre croissant et X un élément quelconque.
Result : Retourner l'indice de l'élément X dans A. Si X n'est pas dans la liste, retourner -1.

```
1  L ← 1 ;
2  R ← n ;
3  while L ≤ R do
4  |  m ← ⌊((L+R)/2)⌋ [a] ;
5  |  if A[m] < X then
6  |  |  L ← m + 1 ;
7  |  else if A[m] > X then
8  |  |  R ← m − 1 ;
9  |  else
10 |  |  return m ;
11 |  end
12 end
13 return −1 ;
```

a. Le nombre $\frac{(L+R)}{2}$ est arrondi à l'unité inférieure

2. L'opération fondamentale dans l'algorithme RechDicho est la comparaison ($A[m] < X$) à la ligne 5.
Dans le meilleur des cas, X se trouve au milieu du tableau, d'où $Min(n) = 1 = O(1)$.
Dans dans le cas le plus défavorable ou le pire des cas, la valeur recherchée (X) n'est pas dans le tableau !
Le nombre de fois où l'opération fondamentale est exécutée dans le pire des cas est égale au nombre maximal p d'itérations de la boucle **while**. L'algorithme a donc une complexité temporelle dans le pire des cas en $O(p)$.
Déterminons un majorant du nombre maximal d'itérations p en fonction de la longueur n du tableau A.
Considérons la valeur $R - L + 1$ qui représente la langueur du tableau après une dichotomie :
   - Au début : $R - L + 1 \leq n$
   - Après une itération : $R - L + 1 \leq \frac{n}{2}$
   - Après deux itération : $R - L + 1 \leq \frac{n}{4}$
   - ...
   - après $p - 2$ itérations : $R - L + 1 \leq \frac{n}{2^{p-2}}$

   Après $p - 2$ itérations, $R - L = 0$ (la langueur du tableau après une dichotomie est 1), d'où :
   $1 \leq \frac{n}{2^{p-2}} \iff 2^{p-2} \leq n \iff p - 2 \leq \log n \iff p \leq \log n + 2$, c-à-d, $p = O(\log n)$.
   La complexité temporelle dans le pire des cas de RechDicho est donc en $O(\log n)$.

Exercice 8

La structure de tas (binaire) est un object tabulé qui peut être vu comme un arbre binaire quasi-parfait (voir la Figure 1.1).

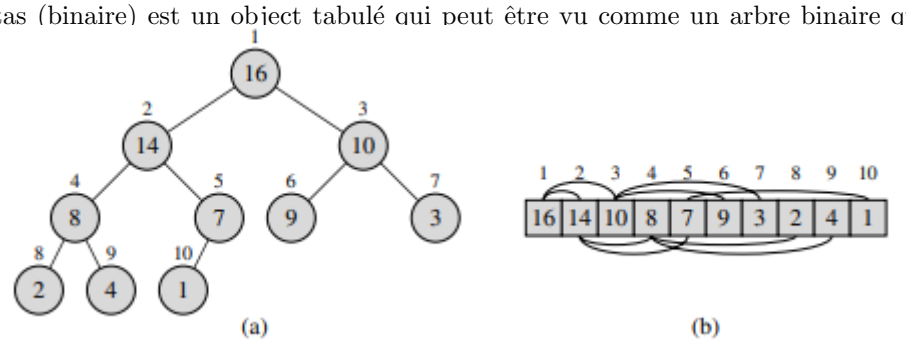


Figure 1.1 – Un exemple d'un tas (Cormen et al., 2022).

Pour passer d'un arbre à un tableau, on utilise les fonctins suivantes :

$$\text{Pere}(i) = \lfloor \tfrac{i}{2} \rfloor \qquad \text{Gauche}(i) = 2i \qquad \text{Droit}(i) = 2i + 1$$

Etant donné un tableau A, on a la propriété basique des tas : $A[\text{Pere(i)}] \geq A[i]$
La structure de donnée comporte trois opérations :

Entasser permet de conserver la structure de tas

ConstruireTas permet de construire un tas

TriTas permet de faire un tri d'un tableau en place

La fonction Entasser Son objectif est de conserver la propriété de tas. Si cette propriété est violée, on fait alors descendre la clé violant la propriété dans le tas. Pour cela, on fait l'hypothèse que le tas est presque correct (seul notre noeud courant peut violer la propriété) et on rétabli la structure de données. Dans ce cas, la valeur maximale du tas enracinée en notre noeud courant est soit celui-ci soit dans un de ces deux fils.
La fonction ConsruireTas Cettes fonction consiste à parcourir les noeuds qui ont des fils et à leur appliquer l'opération ENTASSER, en commençant par les noeuds qui sont à la plus grande profondeur dans l'arbre.
La fonction TriTas On construit un tas. On utilise le fait que le premier élément du tas est le plus grand : autant de fois qu'il y a d'éléments, on extrait le premier du tas et on reconstruit le tas avec les éléments restants.

1. Écrivez l'algorithme en pseudo-code de chacune des fonctions : ENTASSER, CONSRUIRETAS et TRI.
2. Donnez la complexité de chaque algorithme.

Solution (inspirée de (Cormen et al., 2022))

1. Les algorithmes des fonctions : ENTASSER, CONSRUIRETAS et TRI.

Algorithme 9 : Fonction permettant d'entasser un tas.

Data : Un tableau A et un indice i dans le tableau.
Result : Faire descendre la valeur de A[i] dans le tas en respectant la propriété des tas.

```
1  function ENTASSER(A,i):
2      l ← GAUCHE(i) ;
3      r ← DROIT(i) ;
4      max ← i ;
5      if (l ≤ taille[A]) ∧ (A[l] > A[i]) then
6          max ← l ;
7      end
8      if (r ≤ taille[A]) ∧ (A[r] > A[max]) then
9          max ← r ;
10     end
11     if (max ≠ i) then
12         ÉCHANGER A[i] et A[max] ;
13         ENTASSER(A,max)
14     end
```

Explication : A chaque étape, on détermine le plus grand des éléments $A[i]$, $A[\text{GAUCHE}(i)]$, et $A[\text{DROIT}(i)]$, et son indice est rangé dans max. Si $A[i]$ est le plus grand, alors le sous-arbre enraciné au noeud i est un tas. Sinon, l'un des deux fils contient l'élément le plus grand, et $A[i]$ est échangé avec $A[max]$, ce qui permet au noeud i et à ses fils de satisfaire la propriété des tas. Toutefois le noeud max contient la valeur initial de $A[i]$, et donc le sous-arbre enraciné en max viole peut-être la propriété de tas. ENTASSER doit être appelée récursivement sur ce sous-arbre. La Figure 1.2 illustre ce procédé.

Algorithme 10 : Fonction permettant de construire un tas : on transforme un tableau en un tas.

Data : Un tableau A.
Result : Le tas de A.

```
1  function CONSTRUIRETAS(A):
2      for i = ⌊longueur(A)/2⌋ à 1 do
3          ENTASSER(A,i) ;
4      end
```

Explication : On parcourt les noeuds dans l'ordre décroissant des indices et leur applique l'opération ENTASSER, en partant du dernier noeuds qui a des fils, le noeud d'indice $\left\lfloor \frac{longueur(A)}{2} \right\rfloor$. L'ordre dans lequel les noeuds sont traités garantit que les sous-arbres sont des tas. La Figure 1.3 illustre le procédé de CONSTRUIRETAS.

Algorithme 11 : Fonction permettant de trier un tableau à l'aide d'un tas.

Data : Un tableau A.
Result : Le tableau A trié.

```
1  function TRITAS(A):
2      CONSTRUIRETAS(A) ;
3      for i = longueur(A) à 2 do
4          ÉCHANGER A[1] et A[i] ;
5          taille[A] ← taille[A] − 1 ;
6          ENTASSER(A,1) ;
7      end
```

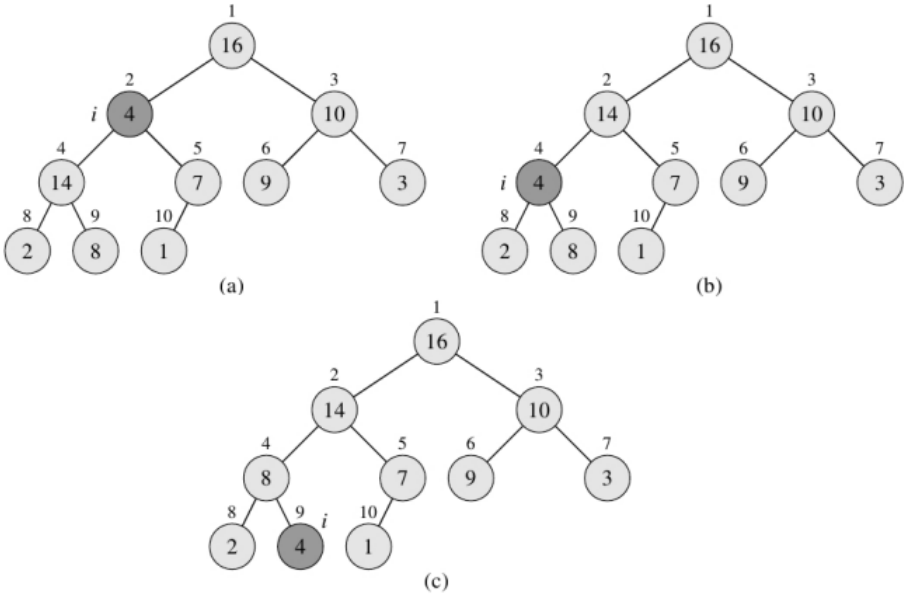


Figure 1.2 – Illustration de ENTASSER (Cormen et al., 2022).

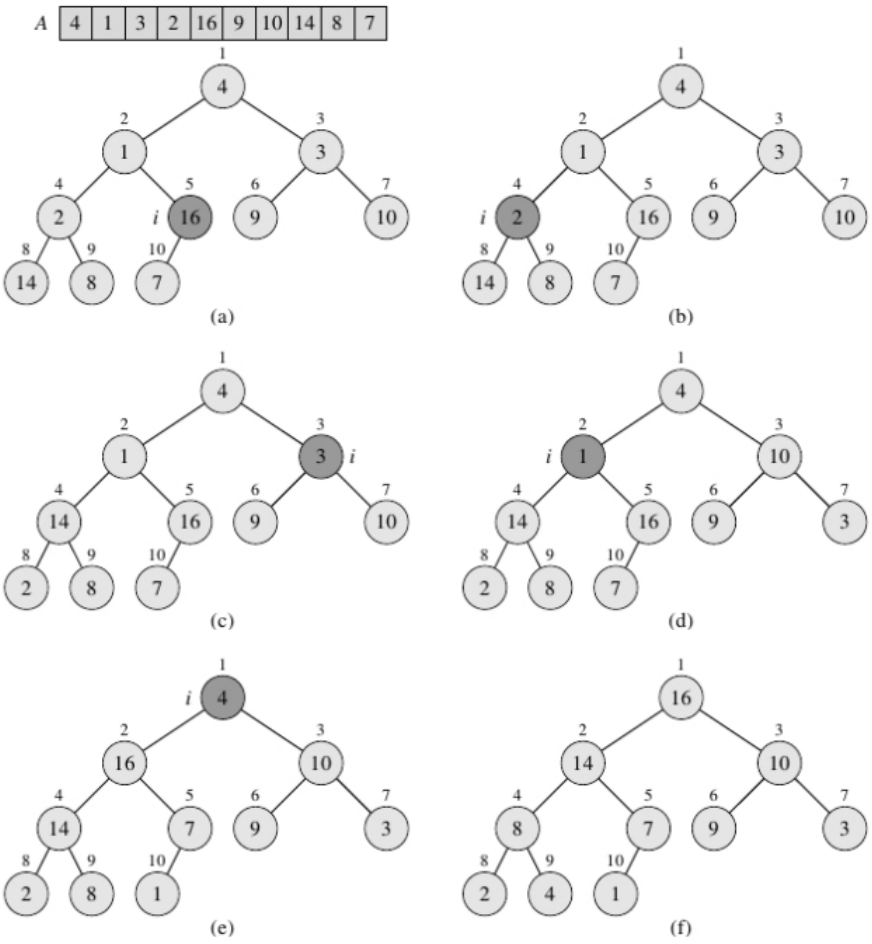


Figure 1.3 – Illustration de CONSTRUIRETAS (Cormen et al., 2022).

Explication : L'algorithme 11 du tri par tas commence par se servir de CONSTRUIRETAS pour construire un tas sur le tableau d'entrée $A[1..n]$, où $n = longueur(A)$. Comme l'élément maximum du tableau est stocké à la racine $A[1]$, on peut le placer dans sa position finale correcte en l'échangeant avec $A[n]$. Si on sort le noeud n du tas (en décrémentant $taille[A]$), on observe que $A[1..(n-1)]$ peut facilement être transformé en tas, en relancant l'entassement de $A[1]$. La procédure continue ainsi jusqu'à ce que le tas se ramène à une seule case.

Une illustration est donnée dans la Figure 1.4.

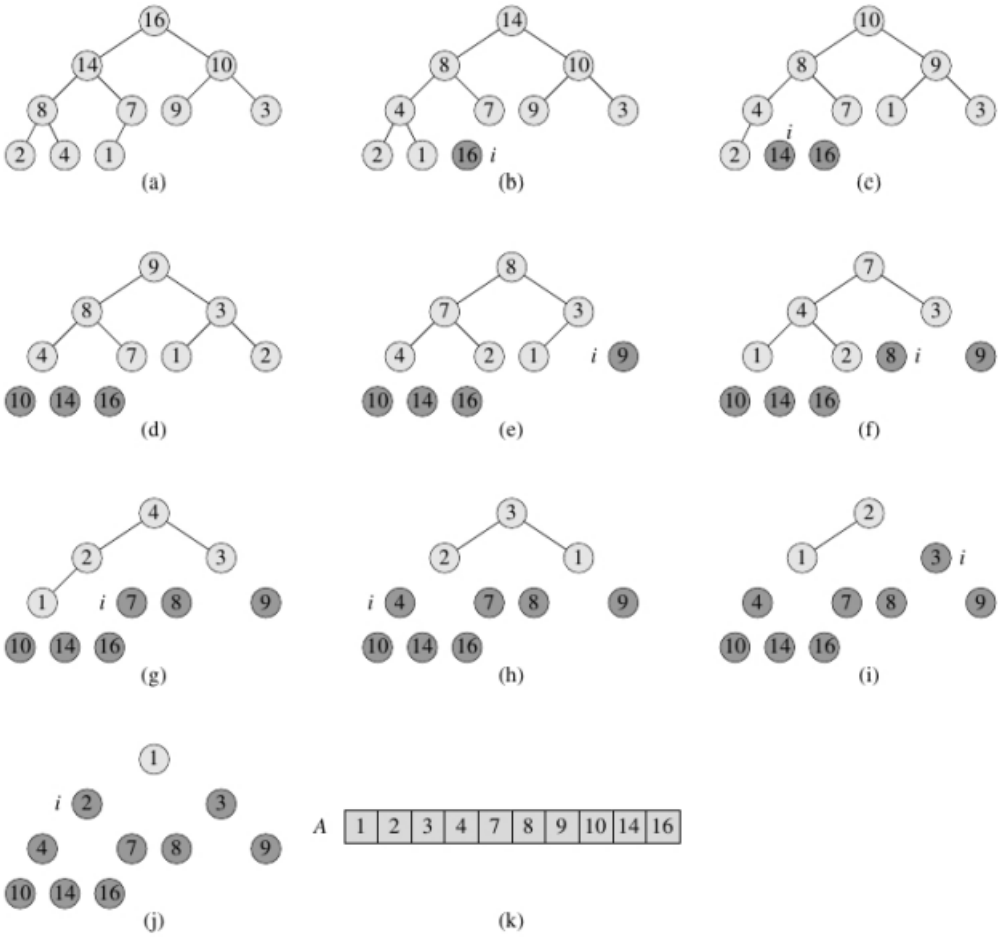


Figure 1.4 – Illustration de TriTas (Cormen et al., 2022).

2. La complexité de chaque algorithme.

Théorème 1. *L'algorithme d'entassement du tas (Algorithme 9) s'exécute en* $O(\log n)$.

Démonstration. Les opérations fondamentales ici peuvent être les comparaisons ($A[l] > A[i]$) (ligne 5) et (($A[r] > A[max]$)) (ligne 8). On effectue une unique descente dans l'arbre binaire du tas. À chaque étape, on effectue les deux comparaisons. Donc, on a une complexité dans le pire des cas en $O(h)$ où h est la hauteur.
Le nombre de noeuds au niveau zéro d'un tas est 2^0, puis 2^1 au niveau 1, ..., 2^{h-1} au niveau $h-1$. Un tas de n noeuds de hauteur h, a tous ses niveaux pleins, sauf le dernier niveau h qui contient au moins un élément, et au plus 2^h éléments. On obtient alors :

$$\begin{aligned} n &\geq 2^0 + 2^1 + ... + 2^{h-1} + 1 \\ &\geq \sum_{k=0}^{h-1} 2^k + 1 \frac{1-2^h}{1-2} + 1 = 2^h - 1 + 1 = 2^h \end{aligned}$$

En passant au log :

$$\begin{aligned} n &\geq 2^h \\ h &\leq \log n \end{aligned}$$

□

Théorème 2. *L'algorithme de construction d'un tas (Algorithme* 10*) s'exécute en* $O(n)$.

Démonstration. Comme la procédure Entasser est en $O(\log n)$, et il existe $O(n)$ appels, alors le temps d'exécution de ConstruireTas est au plus $O(n \log n)$. Cette borne supérieure, quoique correcte, n'est pas approchée asymptotiquement. En fait le coût de la construction est $O(n)$. En effet, la fonction effectue un grand nombre d'entassements sur des arbres de petites hauteur (pour les noeuds les plus profonds), et très peu d'entassements sur la hauteur du tas.

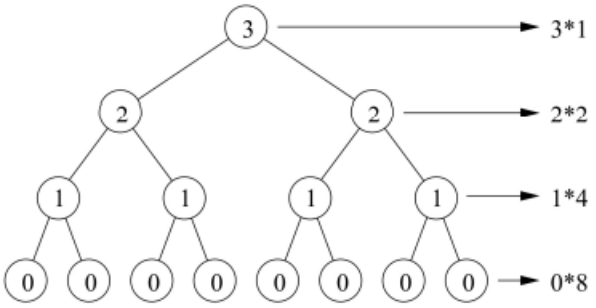


Figure 1.5 – Nombre de parcours par hauteur dans ConstruireTas.

La Figure 1.5 illustre le coût des entassements par hauteurs dans le pire des cas. La complexité $T(n)$ de l'algorithme ConstruitTas vérifie donc :

$$\begin{aligned} T(n) &\leq \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j\frac{2^h}{2^j} \\ &\leq 2^h \sum_{j=0}^{h} \frac{j}{2^j} \end{aligned}$$

La série $\sum_{j=0}^{\infty} \frac{j}{2^j}$ converge vers 2. En effet,

$$S = \sum_{j=0}^{\infty} \frac{j}{2^j} = \sum_{j=1}^{\infty} \frac{j-1+1}{2^j} = \frac{1}{2}\sum_{j=1}^{\infty} \frac{j-1}{2^{j-1}} + \sum_{j=1}^{\infty} \frac{1}{2^j} = \frac{S}{2} + 1$$

En somme :

$$T(n) \leq 2^h \sum_{j=0}^{h} \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \leq 2^h 2$$

Comme $2^h \leq n$, on en déduit que $T(n) \leq 2n$. L'algorithme ConstruireTas est linéaire ($O(n)$) dans le pire des cas. □

Théorème 3. *L'algorithme de tri par tas (Algorithme 11) s'exécute en $O(n \log n)$.*

Démonstration. La construction du tas coûte O(n). On effectue ensuite n fois un échange et l'opération Entasser sur un tas de hauteur au plus $\log n$. La complexité du tri par tas est donc $O(n \log n)$. □

1.6 Notations de Landau

Exercice 9

Soit les complexité suivantes :
$T_1(n) = 3n \log n + \log n$
$T_2(n) = 2^n + n^3 + 25$
$T_3(n, k) = k + n$ où $k \leq n$
$T_4(n) = n^2 + 3^{\log_4 n}$
Démontrez que $T_1(n) = O(n \log n)$, $T_2(n) = O(2^n)$, $T_3(n) = O(n)$ et $T_4(n) = O(n^2)$.

Solution

Étant donné deux fonctions f et g de $\mathbb{N}$ dans $\mathbb{R}^+$,

$$f = O(g) \Leftrightarrow \exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N} \text{ tel que } \forall n > n_0, f(n) \leq c.g(n)$$

1. $T_1(n) = O(n \log n)$
   Selon la définition de la notations de Landau O, on a :

$$0 \leq 3n \log n + \log n \leq c.n \log n$$

$$0 \leq \frac{3n \log n + \log n}{n \log n} \leq \frac{c.n \log n}{n \log n}$$

$$0 \leq 3 + \frac{1}{n} \leq c$$

$$\forall n > 0, \frac{1}{n} \leq 1 \Longrightarrow c = 4 \text{ et } n_0 = 0$$

2. $T_2(n) = O(2^n)$
   Selon la définition de la notations de Landau O, on a :

$$0 \leq 2^n + n^3 + 25 \leq c.2^n$$

$$0 \leq \frac{2^n + n^3 + 25}{2^n} \leq \frac{c.2^n}{2^n}$$

$$0 \leq 1 + \frac{n^3}{2^n} + \frac{25}{2^n} \leq c$$

$$\forall n > 9, \frac{n^3}{2^n} < 1 \text{ et } \forall n > 4, \frac{25}{2^n} < 1 \Longrightarrow c = 3 \text{ et } n_0 = 9$$

3. $T_3(n) = O(n)$
   Selon la définition de la notations de Landau O, on a :

$$0 \leq k + n \leq c.n \text{ où } k \leq n$$

$$0 \leq \frac{k+n}{n} \leq \frac{c.n}{n} \text{ où } k \leq n$$

$$0 \leq \frac{k}{n} + 1 \leq c \text{ où } k \leq n$$

$$\forall n > 0, \frac{k}{n} \leq 1 \Longrightarrow c = 2 \text{ et } n_0 = 0$$

4. $T_4(n) = O(n^2)$
   Selon la définition de la notations de Landau O, on a :

$$0 \leq n^2 + 3^{\log_4 n} \leq c.n^2$$

$$0 \leq \frac{n^2 + 3^{\log_4 n}}{n^2} \leq \frac{c.n^2}{n^2}$$

$$0 \leq 1 + \frac{3^{\log_4 n}}{n^2} \leq c$$

$$\forall n > 0, \frac{3^{\log_4 n}}{n^2} \leq 1 \Longrightarrow c = 2 \text{ et } n_0 = 0$$

Exercice 10

1. Soit $f(n)$ et $g(n)$ deux fonctions positives asymptotique. En s'aidant de la définition de base de la notation Θ, prouver que :
$$max(f(n), g(n)) = \Theta(f(n) + g(n))$$
2. Toujours à la base de la définition de base de la notation Θ, montrez que l'affirmation suivante est correcte : $3(n^2)^n + 6.2^n = \Theta((n^2)^n)$.

Solution

Étant donné deux fonctions f et g de $\mathbb{N}$ dans $\mathbb{R}^+$,

$$f = \Theta(g) \Leftrightarrow \exists c, d \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N} \text{ tel que } \forall n > n_0, d.g(n) \leq f(n) \leq c.g(n)$$

1. Clarifions ce qu'est la fonction $max(f(n), g(n))$. Définissons la fonction $h(n) = max(f(n), g(n))$.

$$h(n) = \begin{cases} f(n) & \text{si } f(n) \geq g(n) \\ g(n) & \text{sinon} \end{cases}$$

   Comme $f(n)$ et $g(n)$ sont asymptotiquement non négatif, il existe n_0 tel que $f(n) \geq 0$ et $g(n) \geq 0$ pour tout $n > n0$.
   Ainsi, pour $n > n0$, $f(n) + g(n) \geq f(n) \geq 0$ et $f(n) + g(n) \geq g(n) \geq 0$. Étant donné que pour tout n particulier,

$h(n)$ est soit $f(n)$ ou $g(n)$, nous avons $f(n)+g(n) \geq h(n) \geq 0$, ce qui montre que $h(n) = max(f(n), g(n)) \leq c(f(n)+g(n))$ pour tout $n > n_0$ (avec $c = 1$ dans la définition de Θ).
De même, puisque pour tout n particulier, $h(n)$ est la plus grande de $f(n)$ et $g(n)$, nous avons pour tout $n > n_0$, $0 \leq f(n) \leq h(n)$ et $0 \leq g(n) \leq h(n)$. L'ajout de ces deux inégalités donne $0 \leq f(n)+g(n) \leq 2h(n)$, ou de manière équivalente $0 \leq \frac{(f(n)+g(n))}{2} \leq h(n)$. Ce qui montre que $h(n) = max(f(n), g(n)) \geq d(f(n)+g(n))$ pour tout $n > n_0$ (avec $d = \frac{1}{2}$ dans la définition de Θ).

2. Selon la définition de la notations de Landau Θ, on a :

$$d.(n^2)^n \leq 3(n^2)^n + 6.2^n \leq c.(n^2)^n$$

$$\frac{d.(n^2)^n}{(n^2)^n} \leq \frac{3(n^2)^n + 6.2^n}{(n^2)^n} \leq \frac{c.(n^2)^n}{(n^2)^n}$$

$$d \leq 3 + \frac{6.2^n}{(n^2)^n} \leq c$$

$$\forall n > 2, 0 \leq \frac{6.2^n}{(n^2)^n} \leq 1 \Longrightarrow d = 3,\ c = 4 \text{ et } n_0 = 2$$

Exercice 11

Peut-on écrire : $2^{n+1} = O(2^n)$? $2^{2n} = O(2^n)$?
Justifiez votre réponse à la base de la notations de Landau O.

Solution

$2^{n+1} = O(2^n)$, mais $2^{2n} \neq O(2^n)$

Justification :

Étant donné deux fonctions f et g de $\mathbb{N}$ dans $\mathbb{R}^+$,

$$f = O(g) \Leftrightarrow \exists c \in \mathbb{R}^{+*}, \exists n_0 \in \mathbb{N} \text{ tel que } \forall n > n_0, f(n) \leq c.g(n)$$

1. Pour montrer que $2^{n+1} = O(2^n)$, nous devons trouver les constantes $c > 0$, $n_0 \geq 0$ telles que : $0 \leq 2^{n+1} \leq c.2^n$ pour tout $n > n_0$.
Sachant que $2^{n+1} = 2.2^n$, pour tout n, la définition est satisfaite avec $c = 2$ et $n_0 = 0$.
2. $2^{2n} \neq O(2^n)$
On suite un raisement pas l'absurde [4]. On suppose qu'il existe les constantes $c > 0$, $n_0 \geq 0$ telles que : $0 \leq 2^{2n} \leq c.2^n$ pour tout $n > n_0$.
$2^{2n} = 2^n.2^n \leq c.2^n \Rightarrow 2^n \leq c$. Mais il n'existe pas de constantes $c > 0$, $n_0 \geq 0$ telles que : $0 \leq 2^n \leq c$ pour tout $n > n_0$ (autrement dit, il n'existe pas de constant c supérieure à 2^n). L'hypothèse conduit donc à une contradiction.

1.7 Optimalité d'un algorithme

Exercice 12

Soit l'algorithme RechMax qui recherche du plus grand élément d'une liste.

Algorithme 12 : RechMax(L)

Data : L : une liste non vide à n éléments entiers
Result : Recherche le plus grand élément M de la liste non-vide L

```
1 M ← L[1] ;
2 for i ← 2 to n do
3 |   if L[j] > M then
4 |   |   M ← L[j] ;
5 |   end
6 end
```

Quelle que soit la mesure d'évaluation considérée, le nombre de comparaisons est égal à $n-1$:

$$Min_A(n) = Moy_A(n) = Max_A(n) = n - 1$$

— Démontrez que cet algorithme est optimal en nombre de comparaisons.

4. Le raisonnement par l'absurde (ou apagogie) est un raisonnement qui permet de démontrer qu'une affirmation est vraie en montrant que son contraire est faux. Il s'appuie sur la règle logique que : Si "non P" est faux, alors P est vraie.

Solution (Gaudel et al., 1987)
Considérons la classe C des algorithmes qui résolvent le problème de la recherche du maximum de n éléments en utilisant comme opération fondamentale, les comparaisons entre éléments. On montre, par une analyse directe du problème, que tout algorithme de C effectue au moins $n-1$ comparaisons, quelle que soit la donnée sur laquelle il travaille. Prouvons cela.

Proposition 1. *Étant donné un algorithme E de C et un tableau quelconque, tout élément autre que le maximum est comparé au moins une fois avec un élément qui lui est plus grand.*

Démonstration. Soit i_0 l'indice du tableau L où se trouve le maximum M, résultat de l'algorithme E. Raisonnons par l'absurde en supposant qu'il existe $j_0 \neq i_0$ tel que $L[j_0]$ n'a pas été comparé avec le maximum $L[i_0]$. Construisons alors le tableau L' identité à L, sauf pour l'indice j_0, où il vaut $M+1$.
L'algorithme E effectuera les mêmes comparaisons sur L' que sur L, et par suite ne comparera pas $L'[j_0]$ et $L'[i_0]$. Il donnera donc comme maximum $L'[i_0]$ et sera incorrect. D'où la contradiction. □

Il résulte de cette dernière proposition qu'il est impossible de déterminer l'élément maximum d'un tableau quelconque de n éléments en moins de $n-1$ comparaisons. L'algorithme est donc optimal en nombre de comparaisons.

1.8 Equation de récurrences et technique de résolution

Exercice 13
Soit l'équation récurrente suivante :

$$T(n) = \begin{cases} 2 & \text{si } n = 2 \\ 2T(\frac{n}{2}) + n & \text{si } n = 2^k, \text{ pour } k > 1 \end{cases}$$

Sachant que n une puissance exacte de 2 :

1. Trouvez la solution pour $T(n)$ avec la méthode "Plug-and-Chug" (en illustrant un algorithme qui a ce comportement).
2. Démontrez la véracité de la solution obtenue par induction.

Solution

1. Application de la mèthode "Plug-and-Chug" [5] (force brute) (Geurts, 2013/2014).
   — **"Plug" (appliquer l'équation récurrente) et "Chug" (simplifier)**

$$\begin{aligned} T(n) =& 2T(\frac{n}{2}) + n \\ =& 2(2T(\frac{n}{2^2}) + \frac{n}{2}) + n \\ =& 2^2T(\frac{n}{2^2}) + 2n \\ =& 2^2(2T(\frac{n}{2^3}) + \frac{n}{2^2}) + 2n \\ =& 2^3T(\frac{n}{2^3}) + 3n \\ =& 2^4T(\frac{n}{2^4}) + 4n \\ =& ... \end{aligned}$$

   — **Identifier et vérifier un "pattern"**
   - Identification :

$$T(n) = 2^iT(\frac{n}{2^i}) + in$$

   - Vérification en développant une étape supplémentaire :

$$\begin{aligned} T(n) =& 2^i(2(T(\frac{n}{2^{i+1}})) + \frac{n}{2^i}) + in \\ =& 2^{i+1}T(\frac{n}{2^{i+1}}) + (i+1)n \end{aligned}$$

5. Cette méthode est appelée téléscopage ou méthode des facteurs sommants.

— **Exprimer le n^{e} terme en fonction des termes précédents**
Notons que le cas de base, $T(2) = 2$, se produit lorsque $n = 2^{i+1}$. C'est-à-dire que $i = \log n - 1$, d'où :

$$\begin{aligned} T(n) =& 2^{\log n - 1} T(2) + (\log n - 1)n \\ =& \frac{n}{2} 2 + n \log n - n \\ =& n \log n \end{aligned}$$

L'algorithme qui a ce comportement est le tri par fusion (merge sort) dans le pire, meilleur et moyen des cas.

2. Démonstration par induction sur n que $P(n) =$ "$T(n) = n \log n$" est vrai.
Puisque n est une puissance de 2 (pour $n \geq 2$), donc $n = 2^k$ pour $k \geq 1$. $S(k) = T(2^k)$, démontrer par induction sur n que $P(n) =$ "$T(n) = n \log n$" est vrai revient à prouver par induction sur k que $P'(k) =$ "$S(k) = 2^k \times k$" est vrai.

$$S(k) = \begin{cases} 2 & \text{si } k = 1 \\ 2T(\frac{2^k}{2}) + 2^k = 2T(2^{k-1}) + 2^k & \text{, pour } k > 1 \end{cases}$$

$$S(k) = \begin{cases} 2 & \text{si } k = 1 \\ 2S(k-1) + 2^k & \text{, pour } k > 1 \end{cases}$$

Cas de base : $P'(1)$ est vrai car $S(1) = 2 = 2^1 \times 1$
Cas inductif : Montrons que $S(k) = 2^k \times k$ (pour $k > 1$) est vrai dès que $S(k-1) = 2^{k-1} \times (k-1)$ est vrai :

$$\begin{aligned} S(k) =& 2S(k-1) + 2^k \\ =& 2(2^{k-1} \times (k-1)) + 2^k \\ =& 2^k \times (k-1) + 2^k \\ =& 2^k \times k - 2^k + 2^k \\ =& 2^k \times k \end{aligned}$$

Exercice 14

Soit les équations récurrentes suivantes :

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \\ T(n-1) + n & \text{, pour} n > 0 \end{cases}$$

$$S(n)^{\,6} = \begin{cases} 0 & \text{si } n = 0 \\ 2S(n-1) + 1 & \text{, pour} n > 0 \end{cases}$$

— Trouvez la solution pour T(n) et S(n).
— Démontrez par induction que les solutions sont correctes.

Solution (inspirée de (Geurts, 2013/2014))

1. La première équation de récurrence ($T(n)$) est une récurrence linéaire d'ordre 1 de la forme :

$$T_n = T_{n-1} + f(n)$$

Le plug-and-chug appliqué à une récurrence de ce type donne directement :

$$T_n = T_0 + \sum_{k=1}^{n} f(k)$$

Donc,

$$\begin{aligned} T(n) =& 1 + \sum_{k=1}^{n} k \\ =& 1 + \frac{n(n+1)}{2} \\ =& \frac{n^2}{2} + \frac{n}{2} + 1 \end{aligned}$$

6. Le nombre de déplacements de disques d'une tour de n disques d'une tige vers une autre dans le jeu des tours de Hanoï.

Théorème 4. *$T(n) = \frac{n^2}{2} + \frac{n}{2} + 1$ satisfait la récurrence :*

$$T(n) = \begin{cases} 1 & si\ n = 0 \\ T(n-1) + n & ,\ pour\ n > 0 \end{cases}$$

Démonstration. Par induction sur n que $P(n) =$ "$T(n) = \frac{n^2}{2} + \frac{n}{2} + 1$" est vrai.
Cas de base : $P(0)$ est vrai car $T(0) = 1 = \frac{0^2}{2} + \frac{0}{2} + 1$.
Cas inductif : Montrons que $T(n) = \frac{n^2}{2} + \frac{n}{2} + 1$ (pour $n > 0$) est vrai dès que $T(n-1) = \frac{(n-1)^2}{2} + \frac{(n-1)}{2} + 1$ est vrai.

$$\begin{aligned} T(n) =& T(n-1) + n \\ =& \frac{(n-1)^2}{2} + \frac{(n-1)}{2} + 1 + n \\ =& \frac{n^2 + 1 - 2n}{2} + \frac{n-1}{2} + 1 + n \\ =& \frac{n^2}{2} + \frac{1}{2} - \frac{2n}{2} + \frac{n}{2} - \frac{1}{2} + 1 + n \\ =& \frac{n^2}{2} + \frac{n}{2} + 1 \end{aligned}$$

□

2. La deuxième équation de récurrence ($S(n)$) est aussi linéaire d'ordre 1, mais de la forme :

$$T_n = g(n)T_{n-1} + f(n) \quad \text{où le coefficient de } T_{n-1} \text{ n'est pas 1}$$

En divisant gauche et droite par $h(n) = g(n).g(n-1).g(n-2)...g(1)$ permet de se ramener à une récurrence de la forme :

$$\frac{T_n}{h(n)} = \frac{T_{n-1}}{h(n-1)} + \frac{f(n)}{h(n)} \Rightarrow T_n = h(n)(\frac{T_0}{h(0)} + \sum_{k=1}^{n} \frac{f(k)}{h(k)})$$

Appliquons la même démarche pour trouver la solution de $S(n)$.
$S(0) = 0$, $S(n) = 2S(n-1) + 1$ pour $n > 0$.
En divisant gauche et droite par $h(n) = 2.2...2 = 2^n$, on obtient la récurrence :

$$\begin{aligned} \frac{S(n)}{2^n} = \frac{S_{n-1}}{2^{n-1}} + \frac{1}{2^n} \Rightarrow S(n) =& 2^n(0 + \sum_{k=1}^{n} \frac{1}{2^k}) \\ =& 2^n \times \frac{1}{2} \times (\frac{1 - (\frac{1}{2})^n}{\frac{1}{2}}) \\ =& 2^n \times \frac{2^n - 1}{2^n} \\ =& 2^n - 1 \end{aligned}$$

Théorème 5. *$S(n) = 2^n - 1$ satisfait la récurrence :*

$$S(n) = \begin{cases} 0 & si\ n = 0 \\ 2S(n-1) + 1 & ,\ pour\ n > 0 \end{cases}$$

Démonstration. Par induction sur n que $P(n) =$ "$S(n) = 2^n - 1$" est vrai.
Cas de base : $P(0)$ est vrai car $S(0) = 0 = 2^0 - 1$.
Cas inductif : Montrons que $S(n) = 2^n - 1$ (pour $n > 0$) est vrai dès que $S(n-1) = 2^{n-1} - 1$ est vrai :

$$\begin{aligned} S(n) =& 2S(n-1) + 1 \\ =& 2(2^{n-1} - 1) + 1 \\ =& 2^n - 1 \end{aligned}$$

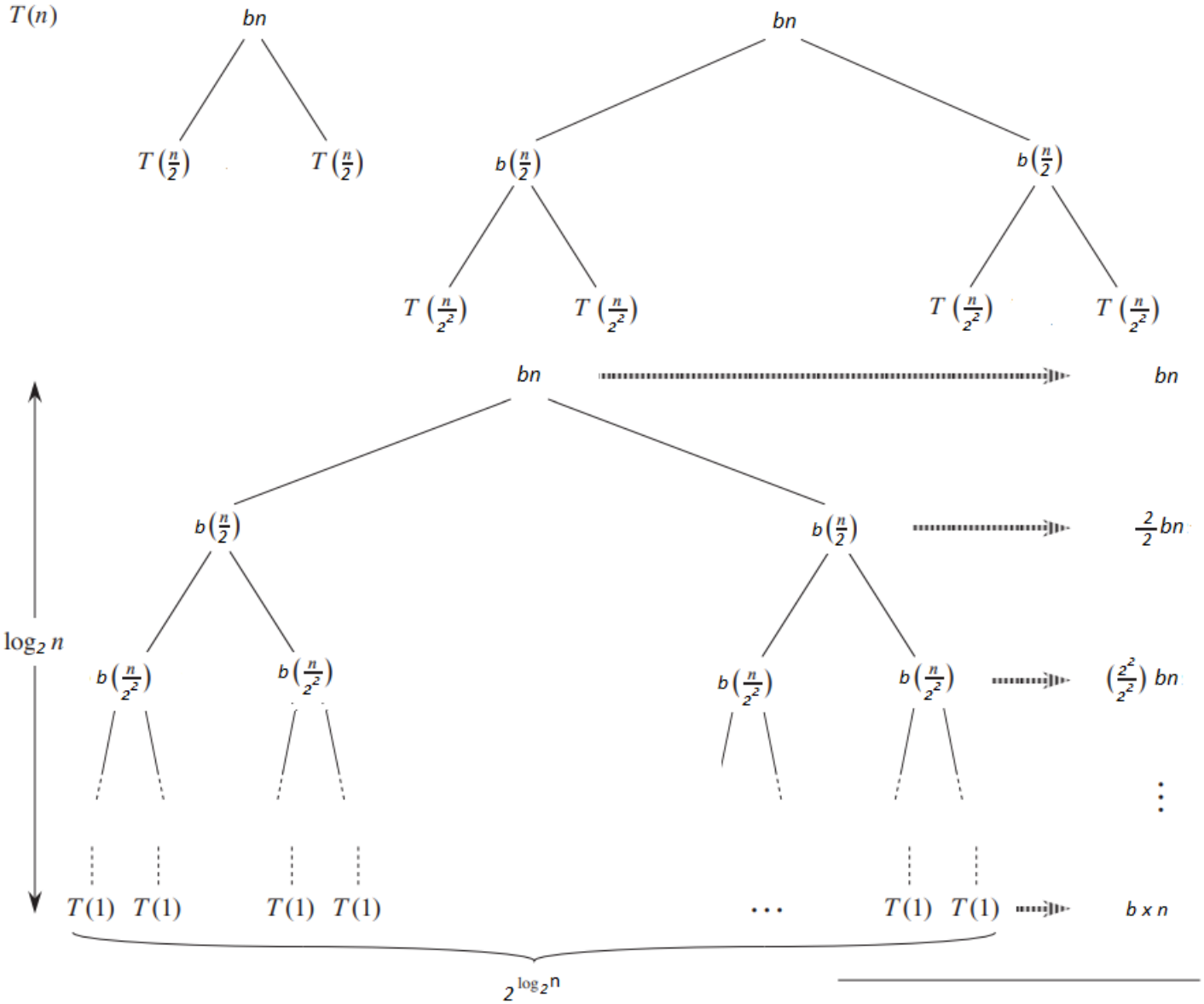
□

Exercice 15 (inspiré de (Goodrich, 2004))
Soit la relation de récurrence suivante :

$$T(n) = \begin{cases} b & \text{si } n < 2 \\ 2T(\frac{n}{2}) + bn & \text{, pour } n \geq 2 \end{cases}$$

— Résolvez cette équation en dessinant son arbre de récurrence.

Solution
L'arbre de récursion qui correspond à $T(n)$:



Le coût total est la somme du coût de chaque niveau de l'arbre :

$$\begin{aligned} T(n) =& bn + \overbrace{bn + ... + bn}^{\log n fois} \\ =& bn + bn \log n \end{aligned}$$

Exercice 16
Soit $F(n)$ le nombre d'additions effectuées par l'algorithme récursif calcul de la valeur du nombre de Fibonacci au rang n :

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \text{ ou } n = 1 \\ F(n-1) + F(n-2) + 1 & \text{, pour } n > 1 \end{cases}$$

— Calculez la complexité de cet algorithme.

Solution

Résolution des récurrences linéaires (Geurts, 2013/2014)
Soit une récurrence de la forme :

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k) + g(n)$$

avec k conditions initiales : $f(0) = v_0, f(1) = v_1, ..., f(k-1) = v_{k-1}$
Étape 1 : Trouver les racines de l'équation caractéristique de la RRHA (Relation de Récurrence Homogène Associée).

$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - ... - a_{k-1} r - a_k = 0$$

Remarque : Le terme $g(n)$ n'est pas pris en compte dans l'équation caractéristique.

- **Racine caractéristique :** solution r_0 de l'équation caractéristique
- **Multiplicité m d'une racine r_0 :**

$$r^k - a_1 r^{k-1} - a_2 r^{k-2} - ... - a_{k-1} r - a_k = (r - r_0)^m Q(r)$$

où r_0 n'est pas une racine de $Q(r)$.

Étape 2 : Trouver une solution homogène $h(n)$, sans tenir compte des conditions initiales.

- Si l'équation caractéristique admet t racines distinctes $r_1, r_2, ..., r_t$ de multiplicités $m_1, ..., m_t$ respectivement ($m_i \geq 1$ et $m_1 + m_2 + + m_t = k$), alors la solution $\{h(n)\}$ de la relation de récurrence est de la forme :

$$h(n) = p_1(n) r_1^n + p_2(n) r_2^n + ... + p_t(n) r_t^n$$

où $p_i(n)$ est un polynôme de degré $m_i - 1$ en la variable n.

Étape 3 : Trouver une solution particulière $p(n)$, sans tenir compte des conditions initiales.
Dans le cas où $g(n) = Q(n)s^n$, avec $Q(n)$ un polynôme de degré t et $s \in \mathbb{R}$, alors

- si s n'est pas une racine de l'équation caractéristique de la RRHA, alors

$$p(n) = (\beta_0 + \beta_1 n + ... + \beta_{t-1} n^{t-1} + \beta_t n^t) s^n$$

- sinon si s est une racine de l'équation caractéristique de multiplicité m de la RRHA, alors

$$p(n) = n^m (\beta_0 + \beta_1 n + ... + \beta_{t-1} n^{t-1} + \beta_t n^t) s^n$$

Étape 4 : Former une solution générale, sans tenir compte des conditions initiales.
Il suffit d'additionner la solution homogène et la solution particulière :
Toute solution $\{s(n)\}$ de la relation de récurrence :

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + ... + a_k f(n-k) + g(n)$$

est de la forme :

$$s(n) = p(n) + h(n)$$

où $\{p(n)\}$ est une solution particulière et $\{h(n)\}$ est une solution de la RRHA
Étape 5 : Déterminer les valeurs des constantes introduites à l'étape 2

- Pour chaque condition initiale, appliquer la solution générale à cette condition. On obtient une équation en fonction des constantes à déterminer
- Résoudre le système formé par ces équations

$F(n)$ est une récurrence linéaire non homogène (générale) de degré 2 à coefficients constants. Pour la résoudre, on procède en 5 étapes :

- **Étape 1 :** Trouver les racines de l'équation caractéristique de la RRHA [7]
  On identifie la RRHA : $F(n) = F(n-1) + F(n-2)$
  On détermine l'équation caractéristique de la RRHA : $r^2 - r - 1 = 0$
  On calcule les racines du polynôme caractéristique de la RRHA : $r_1 = \frac{1-\sqrt{5}}{2}$ et $r_2 = \frac{1+\sqrt{5}}{2}$
- **Étape 2 :** Trouver une solution homogène $h(n)$, sans tenir compte des conditions initiales
  On détermine la forme de la solution de la RRHA. Puisque le polynome possède deux racines distinctes, la

7. La relation de récurrence homogène associée

solution homogène $h(n)$ sera de la forme :

$$\begin{aligned} h(n) =& \alpha_1 (r_1)^n + \alpha_2 (r_2)^n \\ =& \alpha_1 \left(\frac{1-\sqrt{5}}{2}\right)^n + \alpha_2 \left(\frac{1+\sqrt{5}}{2}\right)^n \end{aligned}$$

— **Étape 3 :** Trouver une solution particulière $p(n)$, sans tenir compte des conditions initiales
Comme $g(n) = 1 = (1)^n$, on a $s = 1$ (qui n'est pas une racine) et $t = 0$ (le degré de 1). La solution $p(n)$ est donc de la forme :

$$p(n) = \beta_0$$

On détermine la valeur du coefficient β_0 : comme $p(n)$ est une solution particulière, elle satisfait la relation de récurrence, c-à-d $p(n) = p(n-1) + p(n-2) + 1$. Ainsi,

$$\beta_0 = 2\beta_0 + 1 \Longleftrightarrow \beta_0 = -1$$

La solution particulière est donc : $p(n) = -1$

— **Étape 4 :** Former une solution générale $s(n)$, sans tenir compte des conditions initiales

$$s(n) = \alpha_1 \left(\frac{1-\sqrt{5}}{2}\right)^n + \alpha_2 \left(\frac{1+\sqrt{5}}{2}\right)^n - 1$$

— **Étape 5 :** Déterminer les valeurs des constantes introduites à l'étape 2
On détermine les valeurs des coefficients. Comme il y a deux coefficients à trouver, il nous faut donc aussi 2 équations :

$$\begin{cases} s(0) = & \alpha_1 + \alpha_2 - 1 = 0 = F(0) \\ s(1) = & \alpha_1 \left(\frac{1-\sqrt{5}}{2}\right) + \alpha_2 \left(\frac{1+\sqrt{5}}{2}\right) - 1 = 0 = F(1) \end{cases} \Longrightarrow \alpha_1 = \frac{-1+\sqrt{5}}{2\sqrt{5}} \text{ et } \alpha_2 = \frac{1+\sqrt{5}}{2\sqrt{5}}$$

La solution est :

$$\boxed{s(n) = \frac{-1+\sqrt{5}}{2\sqrt{5}} \left(\frac{1-\sqrt{5}}{2}\right)^n + \frac{1+\sqrt{5}}{2\sqrt{5}} \left(\frac{1+\sqrt{5}}{2}\right)^n - 1 = O((\tfrac{1+\sqrt{5}}{2})^n)}$$

Exercice 17

Soit $F(n)$ et $T(n)$ des relations de récurrence linéaires non homogènes de degré k ($k \geq 1$) à coefficients constants :

$$F(n) = \begin{cases} 1 & \text{si } n = 0 \\ 8F(n-1) + 10^{n-1} & \text{, pour } n > 0 \end{cases}$$

$$T(n) = \begin{cases} 2 & \text{si } n = 1 \\ 3 & \text{si } n = 2 \\ 5 & \text{si } n = 3 \\ 5T(n-1) - 8T(n-2) + 4T(n-3) + n^2 2^n & \text{, pour } n > 3 \end{cases}$$

— Trouvez la solution de chaque récurrence en passant les cinq étapes de résolution.

Solution

1. La solution pour $F(n)$:
   — **Étape 1 :** Trouver les racines de l'équation caractéristique de la RRHA
   On identifie la RRHA : $F(n) = 8F(n-1)$ On détermine l'équation caractéristique de la RRHA : $r - 8 = 0$
   On calcule les racines du polynôme caractéristique de la RRHA : $r_1 = 8$
   — **Étape 2 :** Trouver une solution homogène $h(n)$, sans tenir compte des conditions initiales
   On détermine la forme de la solution de la RRHA. Puisque le polynome possède une racine de multiplicité 1, la solution homogène $h(n)$ sera de la forme :

$$h(n) = \alpha_1 (r_1)^n = \alpha_1 (8)^n$$

— **Étape 3 :** Trouver une solution particulière $p(n)$, sans tenir compte des conditions initiales
Comme $g(n) = 10^{n-1} = \frac{1}{10}(10)^n$, on a $s = 10$ (qui n'est pas une racine) et $t = 0$ (le degré de $\frac{1}{10}$). La solution $p(n)$ est donc de la forme :

$$p(n) = \beta_0 10^n$$

On détermine la valeur du coefficient β_0 : comme $p(n)$ est une solution particulière, elle satisfait la relation de récurrence, c-à-d $p(n) = 8p(n-1) + 10^{n-1}$. Ainsi,

$$\beta_0 10^n = 8\beta_0 10^{n-1} + 10^{n-1} \Longleftrightarrow \beta_0 10 = 8\beta_0 + 1 \Longleftrightarrow \beta_0 = \frac{1}{2}$$

La solution particulière est donc : $p(n) = \frac{1}{2}10^n = \frac{10^n}{2}$

— **Étape 4 :** Former une solution générale $s(n)$, sans tenir compte des conditions initiales

$$s(n) = \alpha_1(8)^n + \frac{10^n}{2}$$

— **Étape 5 :** Déterminer les valeurs des constantes introduites à l'étape 2
On détermine la valeur du coefficient α_1. Comme il y a un seul coefficient à trouver, il nous faut donc aussi une équation :

$$s(0) = \alpha_1 + \frac{1}{2} = 1 = F(0) \Longrightarrow \alpha_1 = \frac{1}{2}$$

La solution est :

$$\boxed{s(n) = \frac{(8)^n + 10^n}{2}}$$

2. La solution pour T(n) :

— **Étape 1 :** Trouver les racines de l'équation caractéristique de la RRHA
On identifie la RRHA : $T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3)$ On détermine l'équation caractéristique de la RRHA : $r^3 - 5r^2 + 8r + 4 = 0$
On calcule les racines du polynôme caractéristique de la RRHA : $r_1 = 2$ (de multiplicité 2) et $r_2 = 1$ (de multiplicité 1)

— **Étape 2 :** Trouver une solution homogène $h(n)$, sans tenir compte des conditions initiales
On détermine la forme de la solution de la RRHA. Puisque le polynome possède deux racine, r_1 et r_2 de multiplicité respectivement 2 et 1, la solution homogène $h(n)$ sera de la forme :

$$h(n) = (\alpha_1 + \alpha_2 n)(r_1)^n + \alpha_3(r_2)^n = (\alpha_1 + \alpha_2 n)(2)^n + \alpha_3$$

— **Étape 3 :** Trouver une solution particulière $p(n)$, sans tenir compte des conditions initiales
Comme $g(n) = n^2 2^n$, on a $s = 2$ (qui est une racine) et $t = 2$ (le degré de n^2). La solution $p(n)$ est donc de la forme :

$$p(n) = n^2(\beta_0 + \beta_1 n + \beta_2 n^2)2^n$$

On détermine les valeurs des coefficients β_0, β_1 et β_2 : comme $p(n)$ est une solution particulière, elle satisfait la relation de récurrence, c-à-d $p(n) = 5p(n-1) - 8p(n-2) + 4p(n-3) + n^2 2^n$. Ainsi,
$(n)^2(\beta_0 + \beta_1 n + \beta_2(n)^2)2^n = 5((n-1)^2(\beta_0 + \beta_1(n-1) + \beta_2(n-1)^2)2^{n-1}) - 8((n-2)^2(\beta_0 + \beta_1(n-1) + \beta_2(n-2)^2)2^{n-3}) + 4((n-3)^2(\beta_0 + \beta_1 n + \beta_2(n-3)^2)2^{n-3}) + n^2 2^n$

$$\Longrightarrow \beta_0 = \frac{11}{6},\ \beta_1 = 0 \text{ et } \beta_3 = \frac{1}{6}$$

La solution particulière est donc : $\frac{1}{6}(11 + n^2)n^2 2^n$

— **Étape 4 :** Former une solution générale $s(n)$, sans tenir compte des conditions initiales

$$s(n) = (\alpha_1 + \alpha_2 n)(2)^n + \alpha_3 + \frac{1}{6}(11 + n^2)n^2 2^n$$

— **Étape 5 :** Déterminer les valeurs des constantes introduites à l'étape 2
La solution est :

$$\boxed{s(n) = \left(\frac{157}{2} - 32n + \frac{11n^2}{6} + \frac{n^4}{6}\right)2^n - 95}$$

Exercice 18 (inspiré de (Bari, 2018) et de ("Complexité d'un algorithme", 2021/2022))
Pour chacune des récurrences suivantes, donnez une expression pour le temps d'exécution $T(n)$ si la récurrence peut être résolue avec le théorème général (master theorem). Sinon, indiquez pourquoi le théorème ne s'applique pas.

1. $T(n) = 9T\left(\frac{n}{3}\right) + n$
2. $T(n) = T\left(\frac{2n}{3}\right) + 1$
3. $T(n) = 3T\left(\frac{n}{4}\right) + n\log n$
4. $T(n) = 2^n T\left(\frac{n}{2}\right) + n$
5. $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$
6. $T(n) = 0.5T\left(\frac{n}{2}\right) + n$
7. $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$
8. $T(n) = 64T\left(\frac{n}{2}\right) + n^2\log n$

Solution

Théorème 6. *(Théorème général)(Bari, 2018) Si la récurrence est de la forme suivante :*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

où n : la taille du problème
a : le nombre de sous-problèmes dans la récursivité et $a \geq 1$
$\frac{n}{b}$: la taille de chaque sous-problème
$b > 1$ et $f(n) = \Theta(n^k \log^p n)$ une fonction avec $k \geq 0$ et p est un nombre réel
$T(n)$ peut alors être bornée asymptotiquement comme suit :

1. *Si $a > b^k$, alors $T(n) = \Theta(n^{\log_b a})$*
2. *Si $a = b^k$, alors :*
   (a) $p > -1 \longrightarrow T(n) = \Theta(n^{\log_b a}\log^{p+1} n)$
   (b) $p = -1 \longrightarrow T(n) = \Theta(n^{\log_b a}\log\log n)$
   (c) $p < -1 \longrightarrow T(n) = \Theta(n^{\log_b a})$
3. *Si $a < b^k$, alors :*
   (a) $p \geq 0 \longrightarrow T(n) = \Theta(n^k \log^p n)$
   (b) $p < 0 \longrightarrow T(n) = \Theta(n^k)$

Les solutions :

1. $T(n) = 9T(\frac{n}{3}) + n$
   Pour cette récurrence, on a $a = 9$, $b = 3$ et $f(n) = n = \Theta(n^1 \log^0 n)$ ($k = 1$ et $p = 0$)
   — $a > b^k (9 > 3^1) \longrightarrow T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$
2. $T(n) = T\left(\frac{2n}{3}\right) + 1$
   Pour cette récurrence, on a $a = 1$, $b = \frac{3}{2}$ et $f(n) = 1 = \Theta(n^0 \log^0 n)$ ($k = 0$ et $p = 0$)
   — $a = b^k (1 = \left(\frac{3}{2}\right)^0) \longrightarrow T(n) = \Theta(n^{\log_{\frac{3}{2}} 1}\log^1 n) = \Theta(\log n)$ ($p > -1$)
3. $T(n) = 3T\left(\frac{n}{4}\right) + n\log n$
   Pour cette récurrence, on a $a = 3$, $b = 4$ et $f(n) = n\log n = \Theta(n^1 \log^1 n)$ ($k = 1$ et $p = 1$)
   — $a < b^k (3 < 4^1) \longrightarrow T(n) = \Theta(n^1 \log^1 n) = \Theta(n\log n)$ ($p \geq 0$)
4. $T(n) = 2^n T\left(\frac{n}{2}\right) + n$
   Le théorème général ne s'applique pas : le paramètre a n'est pas une constante ; le nombre de sous-problèmes devrait ne pas varier.
5. $T(n) = 2T\left(\frac{n}{2}\right) + n\log n$
   Pour cette récurrence, on a $a = 2$, $b = 2$ et $f(n) = n\log n = \Theta(n^1 \log^1 n)$ ($k = 1$ et $p = 1$)
   — $a = b^k (2 = 2^1) \longrightarrow T(n) = \Theta(n^{\log_2 2}\log^2 n) = \Theta(n\log^2 n)$ ($p > -1$)
6. $T(n) = 0.5T\left(\frac{n}{2}\right) + n$
   Le théorème général ne s'applique pas : le paramètre a est plus petit que 1 ($a < 1$); cela n'a pas de sens d'avoir moins d'un sous-problème.
7. $T(n) = 4T\left(\frac{n}{2}\right) + \frac{n^2}{\log n}$
   Pour cette récurrence, on a $a = 4$, $b = 2$ et $f(n) = \frac{n^2}{\log n} = \Theta(n^2 \log^{-1} n)$ ($k = 2$ et $p = -1$)

— $a = b^k (4 = 2^2) \longrightarrow T(n) = \Theta(n^{\log_2 4} \log\log n) = \Theta(n^2 \log\log n)$ $(p = -1)$

8. $T(n) = 64T\left(\frac{n}{2}\right) - n^2 \log n$
   Le théorème général ne s'applique pas : $f(n)$ qui est le coût de la décomposition-recombinaison est négatif.

1.9 Conclusion

La sélection et l'application de méthodes rigoureuses sont essentielles au développement d'algorithmes efficaces. Les performances du programme doivent être aussi indépendantes que possible du matériel. Cela doit d'abord dépendre de la manière de concevoir et de spécifier les algorithmes.

Ce chapitre a proposé un ensemble d'exercices sur la complexité algorithmique et des notions sur la récurrence en algorithmique. L'objectif est de présenter les concepts de base de la théorie de la complexité algorithmique, qui fournit un ensemble d'outils pour analyser, comparer et optimiser des algorithmes et par la suite des programmes informatiques.

Chapitre 2

Complexité des problèmes

2.1 Introduction

Certains problèmes d'optimisation combinatoire disposent d'algorithmes polynomiaux, tandis que d'autres n'en ont toujours pas [1]. La question qui se pose : Existe-t-il réellement une classe de problèmes combinatoires pour lesquels on ne trouvera jamais d'algorithmes polynomiaux, ou est ce que les problèmes diffciles ont en fait de tels algorithmes, mais non encore découverts ? On conjecture depuis longtemps l'existence d'une classe de problèmes intrinsèquement difficiles, car plusieurs problèmes difficiles (comme le problème du voyageur de commerce) résistent depuis plus de 50 ans à l'assaut des travaux de recherche : malgré ces efforts, aucun algorithme polynomial n'a été trouvé.

Ce chapitre présente des exercices corrigés sur les classes P (Polynomial) et NP (Non-deterministic Polynomial), la réduction polynomiale entre problèmes et les problèmes NP-complets. Il s'achève par des exercices sur les machines de turing.

2.2 Classes P et NP

Exercice 1 (inspiré de (Cohen, 2018))
Le graphe G est eulérien si il existe un cycle en empruntant exactement une fois chaque arête du graphe G. On rappelle qu'un graphe connexe est eulérien si et seulement si chacun de ses sommets a un degré pair.

1. Écrivez le problème de décision qui lui est associé.
2. Trouvez un algorithme polynomial qui détermine si le graphe est eulérien.

Solution

1. La problème de décision :

   Données : Un graphe non orienté connexe $G = (V, E)$

   Question : Existe-t-il un cycle eulérien dans G
2. Un algorithme polynomial qui détermine si le graphe est eulérien :

   Théorème 7. *Un graphe G connexe a un cycle eulérien si et seulement si chaque sommet est de degré pair.*

   Démonstration. Soit C un cycle eulérien. Si on regarde un sommet x quelconque, alors, suivant le cycle C, pour chaque arête qui "arrive" dans x, il existe un arête qui "part" de x. Un cycle eulérien parcourt chaque arête, alors chaque sommet a un nombre pair d'arêtes, donc est de degré pair.
   Réciproquement, on suppose que tous les sommets de G soient de degré pair. On forme une chaîne simple c_1, aussi longue que possible, à partir d'un sommet arbitraire x_0. Cette chaîne c_1 est en fait un cycle, sinon, son extrémité finale serait de degré impair. Si ce cycle c_1 contient toutes les arêtes du graphe G, c_1 est le cycle eulérien cherché. Dans le cas contraire, on considère le sous-graphe H obtenu à partir de G en éliminant les arêtes de c_1 et ses sommets qui ne sont incidents à aucune des arêtes restantes. Comme G est connexe, H possède au moins un sommet commun avec le cycle c_1. Soit x_1 un tel sommet. Les sommets de H sont encore de degré pair. On construit alors, de la même manière que précédemment, un cycle c_2 dans H à partir de x_1. On rallonge le cycle c_1 en insérant à partir du sommet x_1 le cycle c_2 pour former un cycle c_1' de x_0 à x_0. Si ce cycle c_1' possède toutes les arêtes de G, c_1' est le cycle eulérien cherché. Sinon, on continue ce processus, qui se terminera car les sommets du graphe G sont en nombre fini. □

   L'algorithme polynomial qui détermine si le graphe est eulérien :

1. Complexité d'un problème représente la complexité du meilleur programme pour le résoudre.

Algorithme 13 : EULERIEN(G)

Data : Un graphe non orienté connexe $G = (V, E)$.
Result : Un booléen b qui indique si le graphe est eulérien ou non.

```
1 b ← True ;
2 for tout sommet v do                    // La boucle est exécutée O(|V|) fois
3 |   Calculer le degré de v ;             // O(|V|) opérations au pire
4 |   if le degré de v est impair then
5 |   |   b ← False ;
6 |   end
7 end
8 return b ;
```

La complexité de l'algorithme dans le pire des cas est $O(|V|^2)$.

Exercice 2

Mettre sous forme de problème de décision les problèmes suivants :

1. Problème de savoir s'il existe un chemin entre deux sommets disjoints dans un graphe.
2. Problème de connaitre le plus court chemin entre deux sommets disjoints dans un graphe non pondéré.
3. Problème de connaitre la plus longue chaîne élémentaire dans un graphe pondéré. Une chaîne est élémentaire si elle ne passe pas deux fois par le même sommet.
4. Problème qui consiste à trouver le plus petit diviseur non trivial [2] d'un entier naturel n

Solution

1. **Données :** Un graphe orienté $G = (V, E)$, deux sommets distincts u et v dans V.
   Question : Existe-t-il un chemin entre u et v dans G ?
2. **Données :** Un graphe orienté $G = (V, E)$, deux sommets distincts u et v dans V, un entier k.
   Question : Existe-t-il un chemin élémentaire entre u et v dans G tel que sa longueur soit inférieure à k ?
3. **Données :** Un graphe non orienté $G = (V, E, w)$ muni d'une fonction $w : E \longrightarrow \mathbb{N}$, deux sommets distincts u et v dans G et un entier k.
   Question : Existe-t-il une chaîne élémentaire entre u et v dans G tel que sa longueur soit supérieure à k ?
4. **Données :** Deux entiers naturels n et k
   Question : Existe-t-il un diviseur non trivial de n qui soit inférieur à k ?

Exercice 3

Les problèmes suivants sont-ils dans NP, dans P ? Justifier votre réponse.

1. Problème P_1 :
   Données : Un graphe $G = (V, E)$
   Question : Existe-t-il un cycle de longueur égale à 4
2. Problème P_2 :
   Données : Un graphe $G = (V, E)$, deux sommets u, v distincts de G et un entier k.
   Question : Existe-il un simple chemin entre u, v de longueur inférieure ou égale à k

Solution

1. Problème P_1 :
   Le problème P_1 est dans NP car étant donnée une suite de sommets S, on peut vérifier en temps polynomial que S est un cycle de longueur égale à 4 dans G (en $O(|V|)$ opérations).
   Le problème P_1 est dans P. Considérons l'algorithme suivant. Pour tout 4-uplets de sommets (x, y, z, t) vérifier s'il est un cycle de longueur égale à 4. Si c'est un cas, pour au moins un 4-uplet, alors répondre **oui** sinon répondre **non**. La complexité de cet algorithme est $O(|V|^5)$ (il y a $O(|V|^4)$ 4-uplets et tester si un 4-uplet est un cycle de longueur égale à 4 nécessite $O(|V|)$ opérations)

2. Un diviseur non trivial d'un entier naturel n est un entier naturel diviseur de n mais distinct de n et de 1 (qui sont ses diviseurs triviaux)

2. Problème P_2 :
   Le problème P_2 est dans NP car étant donnée une suite de sommets S, on peut vérifier en temps polynomial que S est un simple chemin entre u et v de longueur inférieure ou égale à k (en $O(|V|)$ opérations.
   Le problème P_2 est dans P. Consédérons l'algorithme suivant. Il suffit simplement de calculer le plus court chemin entre u et v. Si la longueur du chemin est plus grande que k, ou si il n'existe pas de chemin entre u et v alors répondre non sinon répondre **oui**.

Exercice 4

Montrez que les problème suivants sont dans la classe de complexité P :

1. Problème DNF-SAT :

   Données : Une formules propositionnelle φ sous forme normale disjonctive.

   Question : Déterminer s'il existe une assignation des variables propositionnelles de façon à rendre φ satisfaisable.

2. Problème CNF-Taut :

   Données : Une formules propositionnelle φ sous forme normale conjonctive.

   Question : Déterminer si φ tout le temps vraie (quelle que soit l'affectation).

Solution

1. Une disjonction de termes est satisfaisable ssi l'un des termes est satisfaisable, ce qui est possible ssi il ne contient pas une variable et sa négation. Il suffit donc de tester si cela se produit ou non, ce qui se fait en temps polynomiale. Par conséquent, DNF-SAT $\in P$.
2. Une formule propositionnelle est une tautologie ssi sa négation est tout le temps fausse, ce qui veut dire que sa négation n'est pas satisfaisable. Comme la négation d'une formule sous forme normale conjonctive est facilement équivalente à une formule sous forme normale disjonctive, ceci donne une réduction polynomiale de CNF-Taut à DNF-Sat. On peut décider si une une formule DNF n'est pas satisfaisable en temps polynomiale et la transformation CNF-Taut $\leq$ DNF-SAT est polynomiale (la négation d'une formule sous forme normale conjonctive) ; par conséquent, CNF-Taut $\in P$.

Exercice 5 (inspiré de (Cohen, 2018) et de ("Réductions, Problèmes P, NP-complétude", Mars 2017))

Soit le problème 2-SAT définit comme suit :

Données : Un ensemble de variables $U = \{x_1, ..., x_n\}$ et une formule $\varphi = c_1 \wedge c_2 \wedge ... \wedge c_p$ avec $c_i = y_{i,1} \vee y_{i,2}$ où pour tout i, j, $y_{i,j}$ est un littéral, c'est-à-dire $y_{i,j}$ est soit x_k, soit $\neg x_k$ pour l'un des x_k

Question : Existe-t-il une affectation de valeurs de vérités aux variables qui rend φ vrai ?

1. Montrez que $u \vee v = (\neg u \Rightarrow v) \wedge (\neg v \Rightarrow u)$.
   À partir d'une instance (U, φ) de 2-SAT, on construit un graphe orienté $G_\varphi = (V, E)$ tel que :
   - Un sommet par littéral de U.
   - Un arc par implication (en transformant chaque clause par deux implications).
2. Dessinez les graphes G_{φ_1} et G_{φ_2} tels que :

$$\varphi_1 = (y \vee \neg z) \wedge (\neg y \vee z) \wedge (y \vee \neg x)$$

$$\varphi_2 = (y \vee z) \wedge (\neg y \vee z) \wedge (\neg z \vee x) \wedge (\neg z \vee \neg x)$$

3. Montrez que si G_φ possède un chemin entre u et v, alors il possède aussi un chemin entre $\neg v$ et $\neg u$.
4. Montrez qu'il existe une variable u de U tel que G_φ contient un cycle entre u vers $\neg u$ si et seulement si φ n'est pas satisfaisable.
5. Montrez qu'un algorithme en temps polynomial peut résoudre le problème 2-SAT.

Solution

1. Montrez que $u \vee v = (\neg u \Rightarrow v) \wedge (\neg v \Rightarrow u)$.

u	v	$(u \Rightarrow v)$	$(\neg u \Rightarrow v)$	$(\neg v \Rightarrow u)$	$(\neg u \Rightarrow v) \wedge (\neg v \Rightarrow u)$	$(u \vee v)$
0	0	1	0	0	0	0
1	0	0	1	1	1	1
0	1	1	1	1	1	1
1	1	1	1	1	1	1

2. Dessinez les graphes G_{φ_1} et G_{φ_2}.

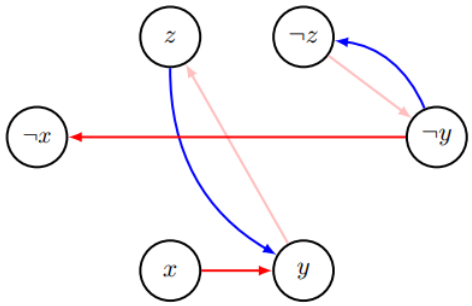


Figure 2.1 – Le graphe G_{φ_1}. $\varphi_1 = (y \vee \neg z) \wedge (\neg y \vee z) \wedge (y \vee \neg x)$

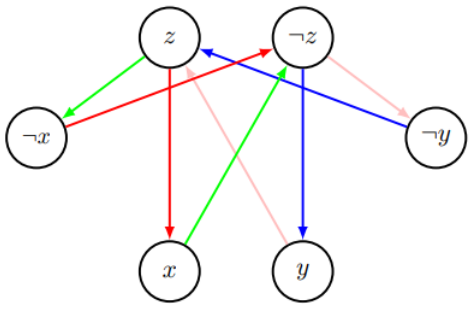


Figure 2.2 – Le graphe G_{φ_2}. $\varphi_2 = (y \vee z) \wedge (\neg y \vee z) \wedge (\neg z \vee x) \wedge (\neg z \vee \neg x)$

3. Montrez que si G_φ possède un chemin entre u et v, alors il possède aussi un chemin entre $\neg v$ et $\neg u$.

Démonstration. On suppose qu'il existe un chemin A dans G_φ entre u et v. On note $A = \{u_1 = u, u_2, ..., u_t = v\}$. On remarque que tout $1 \leq i < t$, G_φ a un arc $(u_i \Rightarrow u_{i+1})$ et G_φ possède aussi l'arc $(\neg u_{i+1} \Rightarrow \neg u_i)$ (Rappel : $u \vee v = (\neg u \Rightarrow v) \vee (\neg v \Rightarrow u)$). On note $A^c = \{\neg u_t = \neg v, \neg u_{t-1}, ..., \neg u_1 = \neg u\}$ existe dans G_φ.
Donc il possède aussi un chemin entre $\neg v$ et $\neg u$. □

4. Montrez qu'il existe une variable u de U tel que G_φ contient un cycle entre u vers $\neg u$ si et seulement si φ n'est pas satisfaisable.

Démonstration.
$\Rightarrow$ On suppose qu'il existe un cycle A dans G_φ tel qu'il contient un cycle entre u vers $\neg u$. On note $A = \{u_0 = u, u_1, ..., u_k = \neg u, .., u_t = u\}$.On raisonne par l'absurde [3]. On suppose qu'il existe une affectation t telle qu'elle satisfasse φ. On suppose que $t(u) = vrai$. On considère le chemin $\{u_0 = u, u_1, ..., u_k = \neg u\}$. Comme il y a un arc entre u_0 et u_1, cela implique qu'il existe la clause $(\neg u_0 \vee u_1)$ dans φ. Comme $t(\varphi) = vrai$, la clause $(\neg u_0 \vee u_1)$ est satisfaite par t et ainsi $t(u_1) = vrai$. Donc, on peut étendre ce raisonnement et prouver que pour $0 \leq \ell \leq k$ on a $t(u_\ell) = vrai$. Cela signifie que $t(u_k) = vrai$, et que $t(\neg u) = vrai$. Ceci contredit le fait que $t(u) = vrai$.
On peut appliquer le même raisonnement si $t(u) = faux$ en considérant le chemin $\{u_k = \neg u, .., u_t = u\}$.
Donc, s'il existe une variable u de U tel que G_φ contient un cycle entre u vers $\neg u$, alors φ n'est pas satisfaisable.
$\Leftarrow$ On suppose que φ est insatisfaisable. Donc, il existe une conjonction d'implications $\varphi_1 = (u_0 \Rightarrow u_1) \wedge (u_1 \Rightarrow u_2) \wedge ... \wedge (u_{k-1} \Rightarrow u_k)$, pour $k > 0$, dans φ telle que u_0 vaut $vrai$ et u_k vaut $faux$. Pour que cela soit vérifié, il faut que u_0 soit une variable u de U et u_k sa négation $(\neg u)$. Mais, on peut avoir $u_0 = faux$ et $u_k = vrai$. Pour cela il faut qu'il y ait dans φ une autre conjonction d'implication $\varphi_2 = (\neg u \Rightarrow u_{k+1}) \wedge (u_{k+1} \Rightarrow u_{k+2}) \wedge ... \wedge (u_{t-1} \Rightarrow u)$, pour $t - k > 0$. De cette manière, on aura si $u = vrai$ $(\neg u = faux)$, φ_1 sera toujours $faux$. Et, si $u = faux$ $(\neg u = vrai)$, $\varphi_2 = faux$. φ_1 et φ_2 se traduisent dans le graphe d'implication G_φ par un cycle entre u vers $\neg u$.
Par conséquent, si φ n'est pas satisfaisable, alors il existe une variable u de U tel que G_φ contient un cycle entre u vers $\neg u$. □

5. Montrez qu'un algorithme en temps polynomial peut résoudre le problème 2-SAT.

Algorithme 14 : Décider s'il existe une affectation de valeurs de vérités aux variables qui satisfait φ.

Data : Un ensemble U de variables, une formule logique φ de clauses de deux littéraux.
Result : Un booléen b qui indique si φ est satisfaisable.

```
b ← Vrai ;
Construire le graphe G_φ ;
foreach variable u dans U do
    if il existe un chemin de u vers ¬u then
        if il existe un chemin de ¬u vers u then
            b ← Faux ;
        end
    end
end
return b ;
```

L'existence d'un chemin d'un nœud à un autre peut être déterminée par des algorithmes de parcours de graphe triviaux tels que BREADTH FIRST SEARCH (en largeur) ou DEPTH FIRST SEARCH (en profondeur). BFS et DFS prennent tous deux un temps polynomial en $O(|V| + |E|)$. Ainsi prouvé que 2-SAT est dans P.

3. Pour montrer que $P \Rightarrow Q$, on suppose que cette implication est fausse et on cherche une contradiction.

2.3 Réduction polynomiale

Exercice 6 (inspiré de (Cohen, 2018))
Soit A et B deux problèmes de décision.
On note $X \subseteq D$ l'ensemble des instance positive de A, et $Y \subseteq D'$ l'ensemble des instances positives de B
Une réduction polynomiale de A vers B est un algorithme polynomial (fonction calculable en temps polynomial) $f : D \longrightarrow D'$, telle que :

$$x \in X \Longleftrightarrow f(x) \in Y$$

Nous notons $A \leq_p B$ lorsque A se réduit polynomialement à B. $A \leq_p B$ signifie que le problème A est donc plus facile que le problème B

1. Montrez que $\leq_p$ est réflexive ($A \leq_p A$)
2. Montrez que $\leq_p$ est transitive ($A \leq_p B$ et $B \leq_p C \Longrightarrow A \leq_p C$)
3. Montrez que si $B \in NP$, $A \leq_p B$ et $A \in NPC$ alors $B \in NPC$
4. Montrez que si $A \leq_p B$ et $B \in P$ alors $A \in P$
5. Montrez que $P = NP$ si et seulement si $SAT \in P$
6. Soit A un problème NP-complet. Montrez que $P = NP$ si et seulement si $A \in P$.

Solution

1. $\leq_p$ est réflexive ($A \leq_p A$)

   Démonstration. Intuitivement : un problème est aussi facile (et difficile) que lui-même.
   On considère la fonction identité pour f ($f(x) = x$) : $f : D \longrightarrow D$. On a $x \in X \Longleftrightarrow f(x) \in X$. f est calculable en temps polynomial (elle fait 0 calcule).
   Donc, A peut se réduire polynomialement à A ($A \leq_p A$). Par conséquent, $\leq_p$ est réflexive. □

2. $\leq_p$ est transitive ($A \leq_p B$ et $B \leq_p C \Longrightarrow A \leq_p C$)

   Démonstration. Intuitivement : la relation "être plus facile que" est transitive.
   On note $Z \subseteq D$" l'ensemble des instances positives de C.
   $A \leq_p B$ et $B \leq_p C$ signifie qu'il existe deux fonctions calculables en temps polynomial $f : D \longrightarrow D'$ et $f : D' \longrightarrow D$", telles que :

   $$\begin{cases} x \in X \Longleftrightarrow & f(x) \in Y \\ y \in Y \Longleftrightarrow & g(y) \in Z \end{cases} \Longrightarrow x \in X \Longleftrightarrow f(g(x)) \in Z$$

   La composée de deux fonctions calculable en temps polynomial est aussi calculable en temps polynomial.
   Par conséquent, $\leq_p$ est transitive. □

3. Si $B \in NP$, $A \leq_p B$ et $A \in NPC$ alors $B \in NPC$

   Démonstration. Puisque A est NP-complet, pour tout problème $X \in NP$, $X \leq_p A$. Si $A \leq_p B$ alors, par transitivité, pour tout problème $X \in NP$, $X \leq_p B$ (voir la démonstration 2). On sait que $B \in NP$. Par conséquent, $B \in NPC$ [4]. □

4. si $A \leq_p B$ et $B \in P$ alors $A \in P$

   Démonstration. $B \in P$ signifie qu'il existe un algorithme polynomial ALG pour B.
   $A \leq_p B$ veut dire q'on peut transformer polynomialement les instances de A en instances de B, puis exécuter l'algorithme pour B. Soit ALG' l'algorithme ainsi construit. Puisque ALG est polynomial pour B, alors ALG' est aussi polynomial.
   Par conséquent, A est polynomial ($A \in P$). □

5. $P = NP$ si et seulement si $SAT \in P$

   Démonstration. Puisque SAT est dans NP, si $P = NP$, alors SAT $\in P$.
   Réciproquement, puisque SAT est complet, pour tout problème $X \in NP$, $X \leq_p$ SAT et donc $X \in P$ si SAT $\in P$ (par la démonstration 3).
   Cette démonstration peut être généralisée pour démontrer que $P = NP$ si et seulement si $Y \in P$ (Y est un problème NP-complet). □

4. Un problème NP-complet est un problème de NP en lequel se transforme polynomialement tout autre problème NP.

Exercice 7

Soit le problème 2-COLORABILITE définit comme suit :

Données : Un graphe $G = (V, E)$ non-orienté

Question : Existe-t-il un coloriage du graphe utilisant au plus 2 couleurs de sorte que deux sommets adjacents ne reçoivent pas la même couleur ?

— Démontrez que le problème 2-COLORABILITE se réduit polynomialement à 2-SAT. En déduire qu'il est dans P.

Solution

Démonstration. ("Réductions, Problèmes P, NP-complétude", Mars 2017) À toute instance G de 2-COLORABILITE, on associe une instance φ de 2-SAT tq.

$$\text{Le graphe } G \text{ est 2-coloriable} \iff \text{La formule } \varphi \text{ est satisfaisable}$$

Soit $G = (V, E)$. On construit la formule φ comme suit :

— À chaque sommet u, on associe une varaible x_u.

— Pour chaque arête $(u, v) \in E$, on construit deux clauses $(x_u \vee x_v)$ et $(\neg x_v \vee \neg x_u)$.

— L'instance de 2-SAT φ est l'union (et) de toutes ces clauses.

L'algorithme de construction de l'instance est clairement polynomial en la taille du graphe.
Il reste à montrer que G est 2-colorable (exemple : par le bleu et le rouge) ssi la formule φ construite est satisfaisable :

— Si G est 2-colorable, alors en prenant la valuation induite par les couleurs : $\begin{cases} bleu \to vrai \\ rouge \to faux \end{cases}$, on obtient la formule φ sera vraie.

— Si la formule φ est satisfaisable, alors en prenant la coloration induite par la valuation : $\begin{cases} x_u = vrai \to couleur(u) = bleu \\ x_u = faux \to couleur(u) = rouge \end{cases}$, on obtient bien G 2-colorable.

□

2-COLORABILITE $\leq_p$ 2-SAT et 2-SAT $\in P$ alors 2-COLORABILITE $\in P$ (démonstration 4 de l'exercice précédent).

2.4 Problèmes NP-complets

Exercice 8

Soit le problème NAESAT dont la définition est :

Données : Un ensemble de variables $\{x_1, ..., x_n\}$ et un ensemble de clauses $c_1, c_2, ..., c_l$ avec $c_i = y_{i,1} \vee ... \vee y_{i,k_i}$ où pour tout i, j, $y_{i,j}$ est un littéral, c'est-à-dire $y_{i,j}$ est soit x_k, soit $\neg x_k$ pour l'un des x_k

Question : Existe-t-il une affectation de valeurs de vérités aux variables $x_i \in \{0, 1\}$ de telle sorte que chaque clause contienne au moins un littéral vrai et au moins un littéral faux (c'est-à-dire, pour tout i, il y a un j et un k avec $y_{i,j} = 1$ et $y_{i,k} = 0$)

— Prouvez que ce problème est NP-complet à partir d'une réduction du problème SAT.

Solution

Théorème 8. *Le problème NAESAT est NP-complet.*

Démonstration. (Cori & Steyaert, 2010) Pour démontrer ce théorème, on va prouver que :

1. NAESAT $\in NP$
2. SAT $\leq_p$ NAESAT (tq. SAT est un problème NP-complet)

1. NAESAT $\in NP$?
Un problème qui appartient à la classe NP devrait avoir un vérificateur polynomial, c'est-à-dire étant donnée un certificat, on devrait être en mesure de vérifier en temps polynomial s'il s'agit d'une solution au problème.
Un vérificateur pour NAESAT prend en entrée une formule CNF et une valuation de ses variables (certificat), et vérifie que chaque clause contienne au moins un littéral vrai et au moins un littéral faux. Cet algorithme ce fait en $O(n \times p)$ (n et p sont respectivement le nombre de variables et de clauses) dans le pire des cas. Donc, le vérificateur est polynomial.
$\Longrightarrow$ Par conséquent, NAESAT $\in NP$

2. SAT $\leq_p$ NAESAT ?
   À tout instance φ de SAT, on va associer une instance $\widetilde{\varphi}$ de NAESAT tq.

$$\varphi \text{ est satisfaisable } \iff \widetilde{\varphi} \text{ est NAESAT satisfaisable}$$

Remarque : On va construire $\widetilde{\varphi}$ à partir de φ en temps polynomial (reduction polynomial) par rapport à la taille $|\varphi|$ de φ.
Si $\varphi = c_1 \wedge ... \wedge c_k$ où chaque c_i est une clause, on construit l'ensemble $\widetilde{\varphi} = c'_1 \wedge ... \wedge c'_k$, où :

- On ajoute une unique variable distincte z et on forme les clauses pour NAESAT en remplaçant chaque clause $c_i = y_{i,1} \vee ... \vee y_{i,k_i}$ de φ en $c'_i = y_{i,1} \vee ... \vee y_{i,k_i} \vee z$

Cette transformation se réalise bien en temps polynomial. On vérifie qu'avec la construction précédente :

- Si une affectation des variables rend chaque c_i vraie, la même affectation de variables tout en fixant pour z à 0 fournit une affectation NAESAT valide pour chaque c'_i
- Réciproquement, supposant que $\widetilde{\varphi}$ soit NAESAT satifaisable. Si la valeur de vérité de z dans l'affectation correspondante est 0, alors les valeurs des variables x_i dans l'affectation donnent une affectation valide pour la formule φ (pour l'instance SAT). Si au contraire z vaut 1, on change toutes les valeurs de toutes les variables. L'affectation reste valide pour NAESAT car au moins un littéral par clause dans l'affectation initaiale valait 0, et vaut donc maintenant 1, tandis que z vaut 0. On a donc construit une affectation dans laquelle z vaut 0, et en vertu du cas précédent l'instance de SAT φ est satisfaisable.
- On a donc prouver l'équavalence entre satisfaisabilité de φ et l'instance correspondante pour NAESAT $\widetilde{\varphi}$:

$$\varphi \text{ est satisfaisable } \iff \widetilde{\varphi} \text{ est NAESAT satisfaisable}$$

Le temps de calcul se réduit à ajouter la variable à chaque clause de c_i de φ qui se fait en $O(k)$ (où k représente le nombre de clauses). L'ensemble du processus de réduction se réalise donc bien en temps polynomial.
Par conséquent :

$$\text{SAT} \leq_p \text{NAESAT}$$

Conclusion :

$$\begin{cases} \text{NAESAT} \in NP \\ \text{SAT} \leq_p \text{NAESAT} \end{cases} \implies \textbf{NAESAT est NP-complet}$$

□

Exercice 9

Une clique dans un graphe est un ensemble de sommets où chaque sommet partage une arête avec tous les autres sommets. Ainsi, une clique dans un graphe est un sous-graphe qui est un graphe complet.
Le problème CLIQUE est défini comme suit :

Données : Un graphe $G(V, E)$ non orienté et un entier k.

Question : G contient-t-il une clique de taille k ?

- Prouvez la NP-complétude ce problème à partir d'une réduction du problème 3-SAT, sachant que 3-SAT est NP-complet (dans le cours, on a utilisé le problème STABLE pour cette démonstration).

Solution

Théorème 9. *Le problème CLIQUE est NP-complet.*

Démonstration. (Lebbah, 2021) Pour démontrer ce théorème, on va prouver que :

1. NAESAT $\in NP$
2. 3-SAT $\leq_p$ CLIQUE (tq. 3-SAT est un problème NP-complet)

1. CLIQUE $\in NP$?
   (a) **Certificat :** Soit le certificat un ensemble S composé de noeuds dans la clique et S est un sous-graphe de G.
   (b) **Vérification :** Vérifier si le nombre de noeuds dans S est égal à k prend un temps $O(1)$. Vérifier si chaque sommet a un degré extérieur de $(k-1)$ prend un temps $O(k^2)$. (Puisque dans un graphe complet, chaque sommet est connecté à tous les autres sommets par une arête. Donc, le nombre total d'arêtes dans un graphe complet est égal à $\frac{k\times(k-1)}{2}$). Par conséquent, pour vérifier si le graphe formé par les k nœuds dans S est complet ou non, il faut $O(k^2) = O(|V|^2)$ temps (puisque $k \leq |V|$).
   Donc, le problème de décision de CLIQUE a un vérificateur polynomial.
   $\implies$ CLIQUE $\in NP$

2. 3-SAT $\leq_p$ CLIQUE ?
   À toute instance φ de 3-SAT, on associe une instance G_φ, k_φ de CLIQUE tq.

$$\varphi \text{ est satisfaisable } \iff G_\varphi \text{ a une clique de taille } k_\varphi$$

Remarque : On va construire G_φ, k_φ à partir de φ en temps polynomial par rapport à la taille de φ.
Soit $\varphi = \bigwedge_{1\leq j\leq k} (x_{1j} \vee x_{2j} \vee x_{3j})$. On construit un graphe $G_\varphi = (V, E)$ avec $3k$ sommets dans V, un pour chaque occurrence d'un littéral dans une clause.
Pour chaque paire de sommets distincts $u, v \in V$, on crée une arête (u, v) dans E si :

- Les littéraux correspondant à u, v ne sont pas dans la même clause.
- Les littéraux correspondant à u, v ne sont pas des négations l'un de l'autre

On choisit l'entier k_φ égal à k (le nombre de clauses de φ).
On démontre maintenant que φ est satisfaisable ssi G_φ possède une clique de taille k_φ :

- Si φ est satisfaisable, on consière une assignation des varaibles satisfaisant φ. Pour chaque clause c de φ , on choisit y_c un littéral de c rendu vrai par l'assignation. De toute évidence, deux littéraux choisis ne peuvent pas être des négations l'un de l'autre (car x et $\neg x$ ne peuvent pas être tous les deux vrais). Cela définit k sommets formant une clique de G_φ.
- Réciproquement, soient $v_1, v_2, ..., v_k$ les sommets de la k-clique dans G_φ. Leurs littéraux correspondants doivent provenir de clauses différentes (car il n'existe pas d'arête entre les sommets de deux littéraux de la même clause). De plus, les littéraux correspondant à $v_1, v_2, ..., v_k$ ne peuvent pas être des négations l'un de l'autre (car aucune arête n'existe entre les sommets de deux littéraux qui sont des négations l'un de l'autre). Nous pouvons donc construire une affectation de vérité en fixant ces k littéraux à vrai.
- Donc,

$$\varphi \text{ est satisfaisable } \iff G_\varphi \text{ a une clique de taille } k_\varphi$$

Le temps de calcul se réduit à construire les arêtes dans G_φ qui se fait en $O(k^2)$ (on a $3^2(\frac{k(k-1)}{2})$ arêtes à créer dans le pire des cas). Il polynomial par rapport à la taille de φ.
Par conséquent,

$$\text{3-SAT} \leq_p \text{CLIQUE}$$

Conclusion :

$$\begin{cases} \text{CLIQUE} \in NP \\ \text{3-SAT} \leq_p \text{CLIQUE} \end{cases} \implies \textbf{CLIQUE est NP-complet}$$

□

Exercice 10

Soit le problème de RECOUVREMENT DE SOMMETS défini comme suit :

Données : Un graphe $G = (V, E)$ non orienté et un entier k

Question : Existe-t-il un sous-ensemble $V' \subset V$, avec $|V'| = k$, tel que toute arête de G ait au moins une extrémité dans V'.

— Démontrez que ce problème est NP-complet (utilisez le problème CLIQUE).

Solution

Théorème 10. *Le problème RECOUVREMENT DE SOMMETS est NP-complet*

Démonstration. (Cori & Steyaert, 2010) Pour démontrer ce théorème, on va prouver que :

1. RECOUVREMENT DE SOMMETS $\in NP$
2. CLIQUE $\leq_p$ RECOUVREMENT DE SOMMETS

1. RECOUVREMENT DE SOMMETS $\in NP$?
   (a) **Certificat :** Un sous-ensemble S de V, qui contient les sommets de la couverture de sommets.
   (b) **Vérification :** On peut vérifier si l'ensemble S est une couverture de sommets de taille égale à k en utilisant la stratégie suivante (pour un graphe $G(V, E)$) :

Algorithme 15 : VERIFVERTEXCOVER(G,k,S)

Data : Un graphe $G = (V, E)$, un entier k, un ensemble $S \subset V$
Result : Un booléen b indiquant si l'ensemble S est une couverture de sommets de taille égale à k.

```
1  b ← FAUX ;
2  cpt ← 0 ;
3  for tout sommet v ∈ S do
4  |  Supprimer toutes les arêtes adjacentes à v de l'ensemble E ;
5  |  cpt ← cpt + 1 ;
6  |  if E est vide then
7  |  |  if cpt = k then
8  |  |  |  cpt ← VRAI ;
9  |  |  |  return ;
10 |  |  else
11 |  |  |  cpt ← FAUX ;
12 |  |  |  return ;
13 |  |  end
14 |  end
15 end
```

Il est clair que cela peut être fait en temps polynomial. Donc, le problème de décision de RECOUVREMENT DE SOMMETS possède un vérificateur polynomial.
$\Longrightarrow$ RECOUVREMENT DE SOMMETS $\in NP$

2 CLIQUE $\leq_p$ RECOUVREMENT DE SOMMETS ?
À toute instance G, k de CLIQUE, on associe une instance G', k' de RECOUVREMENT DE SOMMETS tq.

$$G \text{ a une clique de taille } k \Longleftrightarrow G' \text{ a un recouvrement de sommets de taille } k'$$

Remarque : On va construire G', k' à partir de G en temps polynomial par rapport à la taille de G
Soit le graphe $G' = (V', E')$ qui se compose de toutes les arêtes non pas dans $G = (V, E)$, mais dans le graphe complet utilisant tous les sommets de G ($V' = V$). On appelle cela le complément de G.
Maitenant, on démontre que G a une clique de taille k ssi G' a une couverture de sommets de taille $|V| - k$ (k') :

- On suppose qu'il existe une clique de taille k dans G. Soit V_1 l'ensemble des sommets de la clique. Cela signifie $|V_1| = k$. Dans le graphe complémentaire G', on choisit une arête quelconque (u, v). Alors au moins l'un de u ou v doit être dans l'ensemble $V - V_1$. En effet, si u et v appartenaient tous deux à l'ensemble V_1, alors l'arête (u, v) appartiendrait à la clique V_1, ce qui signifierait à son tour que l'arête (u, v) est dans G. Ce n'est pas possible puisque (u, v) n'est pas dans G. Ainsi, toutes les arêtes de G' sont couvertes par les sommets dans l'ensemble $V - V_1$.
- On suppose maintenant qu'il existe une couverture de sommet V_2 de taille $|V| - k$ in G'. Cela signifie que toutes les arêtes de G' sont connectées à un sommet de V_2. Donc, si nous choisissons une arête (u, v) de G', les deux ne peuvent pas être en dehors de l'ensemble V_2. Cela signifie que toutes les arêtes (u, v) telles que u et v sont à l'extérieur de l'ensemble V_2 sont dans G, c'est-à-dire que ces arêtes constituent une clique de taille k.
- Donc,
$$G \text{ a une clique de taille } k \Longleftrightarrow G' \text{ a un recouvrement de sommets de taille } |V| - k$$

Le temps de calcul se réduit à calculer le graphe complémentaire G' qui se fait dans le pire des cas en $O(|V|^2)$. Ainsi, le processus de réduction se réalise bien en temps polynomial. Par conséquent,

$$\text{CLIQUE} \leq_p \text{RECOUVREMENT DE SOMMETS}$$

Conclusion :

$$\begin{cases} \text{RECOUVREMENT DE SOMMETS} \in NP \\ \text{CLIQUE} \leq_p \text{RECOUVREMENT DE SOMMETS} \end{cases} \Longrightarrow \textbf{RECOUVREMENT DE SOMMETS est NP-complet}$$

Exemple :
Pour comprendre la preuve, on considère l'exemple de graphe suivant et son complément :

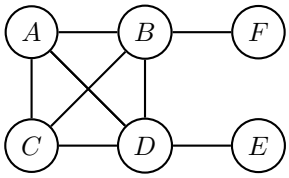


Figure 2.3 – Un graphe G avec une clique V_1 de taille 4 : $V_1 = \{A, B, C, D\}$.

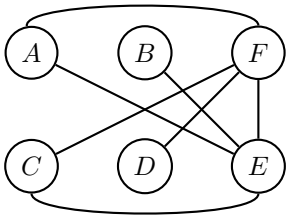


Figure 2.4 – Le graphe G' (le complément de G) qui contient une couverture de sommets V_2 de taille $|V| - 4 = 2$: $V_2 = \{F, E\}$.

□

2.5 Machines de Turing

Exercice 11

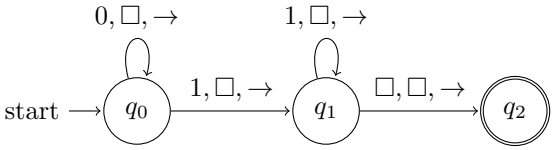
Soit la machine de Turing à une bande $M = (Q, q_0, F, \Sigma, \Gamma, \delta)$:

- L'ensemble des états Q est $\{q_0, q_1, q_2\}$.
- L'état initial est q_0.
- Le seul état final est l'état q_2.
- L'alphabet d'entrée $\Sigma = \{0, 1\}$
- L'alphabet de la bande est $\Gamma = \{0, 1, \square\}$
- L'ensemble de transitions $\delta = \{(q_0, 0, \square, \rightarrow, q_0), (q_0, 1, \square, \rightarrow, q_1), (q_1, 1, \square, \rightarrow, q_1), (q_1, \square, \square, \rightarrow, q_2)\}$

1. Représentez graphiquement la machine de Turing M.
2. Quel langage reconnaît-elle ?
3. Écrivez le calcul de la MT M au cours de son exécution sur l'entrée 001110.

Solution

1. La représentation graphique de la MT M :



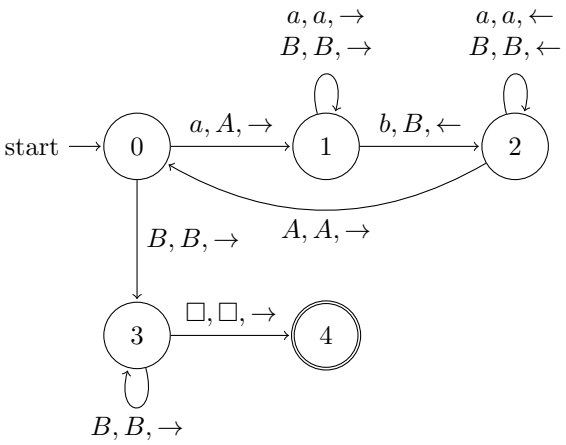
2. Cette machine reconnaît le langage 0^*1^+.
3. Le calcul de la machine de Turing sur l'entrée 001110 :
   $\mathbf{q_0}001110 \vdash \square\mathbf{q_0}01110 \vdash \square\square1110 \vdash \square\square\square\mathbf{q_1}110 \vdash \square\square\square\square\mathbf{q_1}10 \vdash \square\square\square\square\square\mathbf{q_1}0$
   Le calcul se termine sur un état non final (q_1), le mot 001110 n'est donc pas accepté par la MT M.

Exercice 12 (inspiré de (Muscholl, 2021))

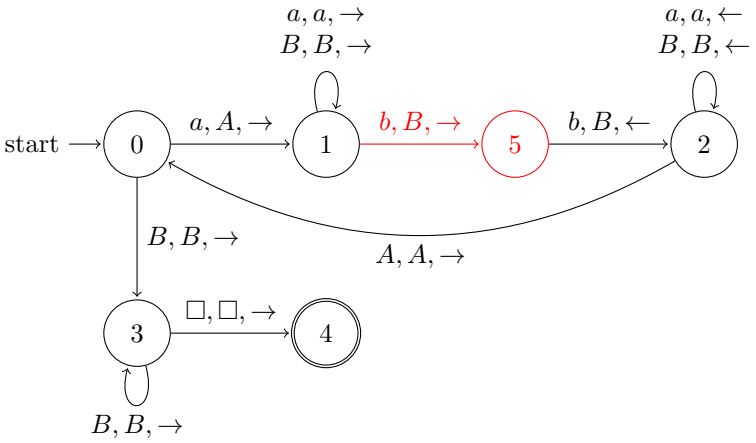
1. Construisez une machine de Turing M acceptant le langage $\{a^n b^n | n \geq 0\}$.
2. Donnez le calcul de la machine avec l'entrée $aaabbb$.
3. Modifiez cette machine pour reconnaître le langage $\{a^n b^{2n} | n \geq 0\}$.

Solution

1. MT M acceptant le langage $\{a^n b^n | n > 0\}$:
   La machine de Turing à une bande $M = (Q, q_0, F, \Sigma, \Gamma, \delta)$ est la suivante :
   - L'ensemble des états Q est $\{0, 1, 2, 3\}$.
   - L'état initial est 0.
   - Le seul état final est l'état 4.
   - L'alphabet d'entrée $\Sigma = \{a, b\}$
   - L'alphabet de la bande est $\Gamma = \{a, b, A, B, \square\}$
   - L'ensemble de transitions est décrit par la représentation graphique ci-dessous. Cette machine est déterministe.



2. Le calcul de la MT M avec l'entrée $aaabbb$:
   $\mathbf{0}aaabbb \vdash A\mathbf{1}aabbb \vdash Aa\mathbf{1}abbb \vdash Aaa\mathbf{1}bbb \vdash Aa\mathbf{2}aBbb \vdash A\mathbf{2}aaBbb \vdash \mathbf{2}AaaBbb \vdash A\mathbf{0}aaBbb \vdash AA\mathbf{1}aBbb \vdash AAa\mathbf{1}Bbb \vdash AAaB\mathbf{1}bb \vdash AAa\mathbf{2}BBb \vdash AA\mathbf{2}aBBb \vdash A\mathbf{2}AaBBb \vdash AA\mathbf{0}aBBb \vdash AAA\mathbf{1}BBb \vdash AAAB\mathbf{1}Bb \vdash AAABB\mathbf{1}b \vdash AAAB\mathbf{2}BB \vdash AAA\mathbf{2}BBB \vdash AA\mathbf{2}ABBB \vdash AAA\mathbf{0}BBB \vdash AAAB\mathbf{3}BB \vdash AAABB\mathbf{3}B \vdash AAABBB\mathbf{3}□ \vdash AAABBB□\mathbf{4}$
   Le calcul se termine sur un état final ($\mathbf{4}$), le mot $aaabbb$ est donc accepté par la MT M.
3. MT M' (modifiant M) acceptant le langage $\{a^n b^{2n} | n > 0\}$:



Exercice 13 (inspiré de (Muscholl, 2021))
Construisez une machine de Turing qui :

1. interprête son entrée comme un entier en binaire n, le remplace par $\lfloor \frac{n}{2} \rfloor$ et s'arrête.
2. effectue l'incrément en binaire.
3. effectue l'addition de deux entiers en unaire.

Solution

On peut utiliser les MT pour **accepter des langages** ou, plus généralement, pour **calculer des fonctions**.

— Une MT déterministe acceptant un langage L calcule la fonction caractéristique de L définie par :

$$f : \begin{array}{l} \Sigma^* \to \{0,1\} \\ w \to \begin{cases} 0 & \text{si } w \notin L, \\ 1 & \text{si } w \in L. \end{cases} \end{array}$$

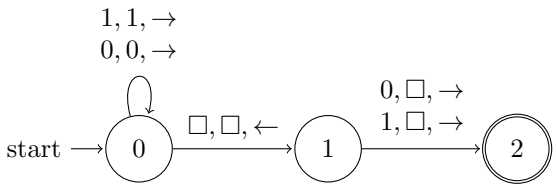
— Plus généralement, on peut associer à une MT déterministe M une fonction $f_M : \Sigma^* \to \Gamma$:

- On écrit la donnée $w \in \Sigma^*$ sur la bande,
- Si la MT s'arrête avec sur la bande le mot $z \in \Gamma^*$, la fonction est définie par :

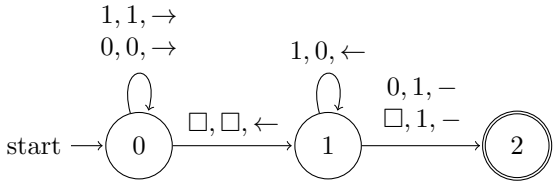
$$f_M(w) = z$$

.

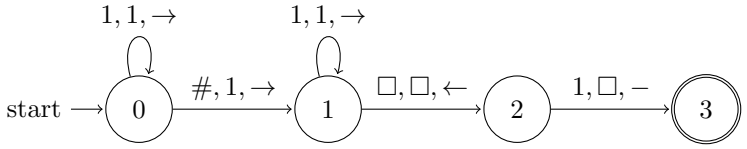
1. Calcul de $\lfloor \frac{n}{2} \rfloor$:



2. Incrément en binaire :



3. Addition en unaire :
Le mot d'entrée est de la forme $1^n\#1^m$ interprété comme la donnée des entiers n et m.



2.6 Conclusion

Le question d'existence des algorithmes de résolution de complexité polynomiale nous amène à définir des classes de complexité : intuitivement nous voulons une classe de programmes qui peuvent être résolus en temps polynomial, une classe de problèmes plus complexes, et un moyen de déterminer à quelle classe appartient un problème. Les classes sont P (des problèmes résolubles en temps plynomiale) et NP (des problèmes vérifiables en temps polynomiale). Une des raisons qui laissent à penser que P $\neq$ NP est l'existence de la classe des problèmes NP-complets : si un seul problème NP-complet peut être résolu en temps polynomial, alors tous les problèmes de NP peuvent être résolus en temps polynomial et P = NP. Mais aucun algorithme polynomial n'a jamais été découvert pour aucun problème NP-complet. Les problèmes NP-complets sont, dans un certain sens, les problèmes les plus "difficiles" de NP.

Chapitre 3

Résolution de Problèmes et Intelligence Artificielle

3.1 Introduction

Résoudre des problèmes signifie choisir et organiser des actions en séquences pour atteindre des objectifs donnés en respectant les limites d'un environnement déterminé.

- **Composition du processus :** Un état initial (le problème à résoudre), un état final (la solution au problème), un ensemble d'actions permettant de passer d'un état à un autre.
- **Objectif du processus :** Trouver une séquence d'actions à exécuter pour passer de l'état initial à l'état final.
- **Complexité du processus :** À partir d'un état donné, on peut effectuer plusieurs actions différentes qui débouchent sur des états différents.

La méthode qui consiste à tenter successivement toutes les options jusqu'à trouver une solution est une méthode de recherche fortement **combinatoire**. On est souvent amené à choisir entre plusieurs choix ou à choisir la meilleur solution, ce qui est un **problème d'optimisation combinatoire**. L'IA apporte une aide via **les heuristiques**, qui sont des méthodes pour choisir les actions les plus prometteuses en premier (Bourahla, 2020).

On commence ce chapitre par des exercices avec solution sur la modélisation d'un problème puis sur les méthodes de résolution de solutions dans l'espace d'états avec les algorithmes non-informés ou "aveugles" (recherche en largeur, recherche en profondeur, recherche en profondeur limitée, recherche par approfondissement itératif) et les algorithmes informés ou heuristiques (algorithme glouton (greedy), algorithme A^*). On termine le chapitre par des exercices sur une autre famille de méthodes de résolution qui est la résolution par décomposition de problèmes.

3.2 Modélisation d'un problème

Exercice 1 (inspiré du livre (Russell, Russell, Norvig, & Davis, 2010) et de (Bienvenu, 2006))
Soit les problèmes suivants :

1. **Le problème de l'aspirateur** est un problème (voir la figure 1) de recherche bien connu pour un agent [1] qui travaille sur l'intelligence artificielle. Dans ce problème, notre aspirateur est notre agent. C'est un agent basé sur un objectif, et le but de cet agent, qui est l'aspirateur, est de nettoyer toute la zone. Ainsi, dans le problème classique de l'aspirateur, nous avons deux pièces et un aspirateur. Il y a de la saleté dans les deux chambres et il faut la nettoyer. L'aspirateur est présent dans chacune de ces pièces. On doit donc atteindre un état dans lequel les deux pièces sont propres et exemptes de poussière.
2. **Le taquin à 8 pièces**, dont un exemple est illustré à la figure 2, se compose d'un plateau de 3×3 cases dont huit sont occupées par des pièces numérotées et dont une est vide. Une pièce adjacente à l'espace vide peut être déplacée dans cet espace. L'objectif est d'atteindre un état but spécifique, tel que celui illustré à droite de la figure.
3. L'objectif du **problème des 8 reines** est de placer huit reines sur un échuiquier de façon à ce qu'aucune d'elles n'en menace une autre (une reine menace toute pièce située dans la même ligne, colonne ou diagonale). La figure 3 illustre une solution inadéquate : la reine située dans l'angle inférieur droit est menacée par celle du coin supérieur gauche.
4. Le problème du voyage en avion que doit résoudre un moteur de recherche de vols sur le Web.

1. En intelligence artificielle, un agent intelligent est une entité autonome capable de percevoir son environnement grâce à des capteurs et aussi d'agir sur celui-ci via des effecteurs afin de réaliser des buts.

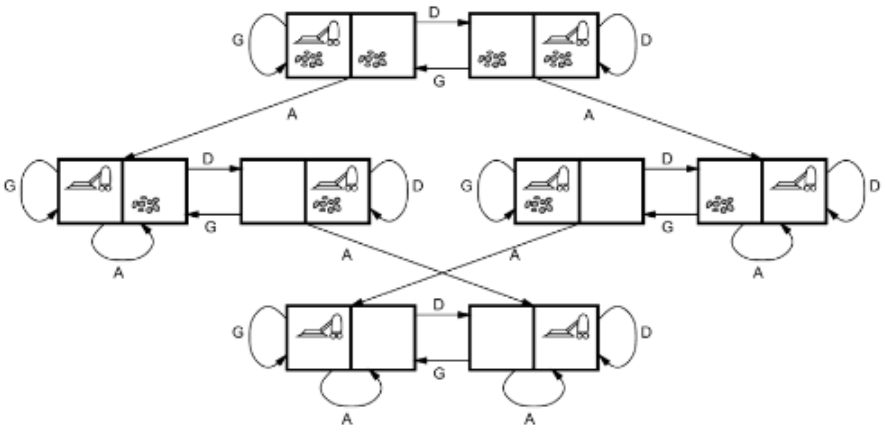


Figure 3.1 – L'espace des états pour le problème d'aspirateur (Russell et al., 2010). Les liens d'actions : $G = Gauche$, $D = Droite$, $A = Aspirer$.

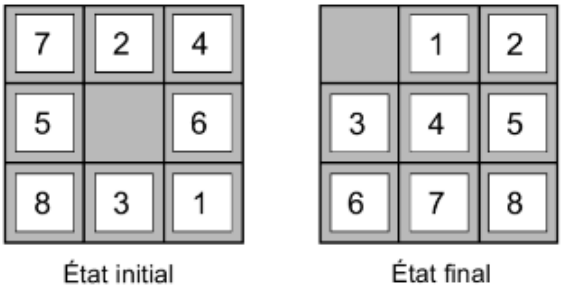


Figure 3.2 – Un exemple typique de taquin à 8 pièces (Russell et al., 2010).

— Donnez la formulation standard de ces problèmes.

— Classez les en tant que problèmes de planification ou de satisfaction de contraintes.

Solution

1. Le problème d'aspirateur :

   États L'état est déterminé par l'emplacement de l'agent et les emplacement de la poussière. L'agent est dans un des deux emplacements, qui peuvent contenir de la salté ou non. Ainsi, il y a 2×2^2 états possibles du monde. Un plus grand environnement, doté de n emplacements, aurait $n \times 2^n$ états.

   État initial N'importe quel état peut être pris comme état initial.

   Actions Dans cet envirenement élémentaire, chaque état a juste trois actions possibles : *Gauche*, *Droite*, et *Aspirer*. Un plus grand environnement pourrait aussi disposer des actions $Haut$ et Bas.

   Fonction de successeur Les actions ont leurs effets prévus, sauf que les actions *Gauche* dans le carré le plus à gauche, *Droit* dans le carré le plus àl'extrême droite, et *Aspirer* dans un endroit propre n'ont aucun effet. La figure 1 illustre l'espace des états complet.

   Test de but Il vérifie si tous les carrés sont propres.

   Coût des actions Chaque étape coûte 1, et le coût de chemin est donc le nombre d'étapes dans le chemin.

2. Le taquin à 8 pièces [2] :

   États La description d'un état spécifie l'emplacement des huit pièces et celui de la case vide.

   État initial N'importe quel état peut être pris comme état initial.

   Actions Dans la formulation la plus simple, les actions sont définies comme des mouvements de la case vide : *Gauche*, *Droite*, *Haut*, ou *Bas*. Différents sous-ensembles d'actions sont possibles selon l'endroit où se trouve la case vide.

2. Le taquin est souvent utilisé pour tester les algorithmes de recherche. En augmentant la taille de la grille, les problèmes deviennent de plus en plus complexes.

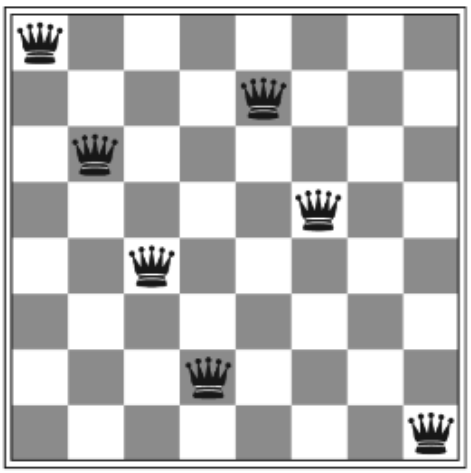


Figure 3.3 – Une quasi-solution eu problème des huit reines (Bienvenu, 2006).

Fonction de successeur Étant donné un état et une action, cela renvoie l'état résultant ; par exemple, si on applique l'action *Gauche* à l'état initial à la figure 2, on aboutit à un état résultanr où le 5 et le vide ont échangé leurs positions.

Test de but Il vérifie si l'état correspond à la configuration finale de la figure 2 (d'autres configurations finales sont envisageables).

Coût des actions Chaque étape coûte 1 et le coût de chemin est donc le nombre d'étapes dans le chemin.

3. Le problème des huit reines : On distingue deux types de formulations. **Une formulation incrémentale**[3] qui à chaque action ajoutera une reine à l'état courant. **Une formulation complète** commence avec les huits reines sur l'échiquier et deplace celles-ci.
Une formulation incrémentale :

États Toute disposition de 0 à 8 reines sur l'échiquier constitue un état.

État initial Aucune reine n'est présente sur l'échiquier.

Actions Poser une reine sur l'une des cases vides.

Fonction de successeur Retourner l'échiquier avec une reine supplémentaire sur une case donnée.

Test de but Huit reines sont présentes sur l'échiquier, et aucune n'est menacée par une autre.

Coût des actions Ce pourrait être 0, ou un coût constant pour chaque action. On s'intéresse pas au chemin, seulement à l'état but obtenu

Dans cette formulation, on a $64.63...57 \approx 1,8 \times 10^{14}$ séquences possibles à examiner.
Une meilleure formulation interdirait de placer une reine dans une case qui est déjà menacée :

États Tous les arrangements possibles de n reines ($0 \leq n \leq 8$), une par colonne dans les n colonnes les plus à gauche, aucune reine n'en menace une autre.

Actions Ajouter une reine à n'importe quelle place dans la plus à gauche de façon à ce qu'elle ne soit attaquée par aucune autre reine.

Cette formulation réduit l'espace des états du problème des 8 reines de $1,8 \times 10^{14}$ à 2057.
Une autre possibilité (formulation complète) serait de commencer avec les huits reines sur la grille (une par colonne) et puis de changer la position d'une reine à chaque tour :

État initials Une configuration de 8 reines telle qu'il n'y a qu'une seule reine par colonne

État initial Un état choisi aléatoirement

Actions Changer la position d'une reine dans sa colonne

4. Le problème du voyage en avion :

États Chaque état comprend evidement un lieu (exemple un aéroport) et l'heure.

État initial Il est spécifié par la requête de l'utilisateur.

Actions Prendre n'importe quel vol au départ du lieu courant, partant après l'heure courante, en laissant assez de temps pour la correspondance s'il y a lieu.

3. La formulation incrémentale fait appel à des opérateurs qui augmentent la description de l'état courant en commençant par l'état initial.

Fonction de successeur L'état résultant du vol aura la destination du vol comme lieu courant et l'heure d'arrivée du vol comme heure courante.

Test de but Est-on arrivé à la destination finale spécifiée par l'utilisateur ?

Coût des actions Cela va dépendre des préférences du client : 1 (pour chaque opération pour minimiser le nombre de vols), délai d'atendre, la durée de vol, le prix des trajets, etc.

— **Problèmes de planification** [4] **:** Le problème de l'aspirateur, le taquin à 8 pièces le problème du voyage en avion.
Problèmes de satisfaction de contraintes [5] **:** Le problème des 8 reines.

Exercice 2

Un fermier a une chèvre, un loup et une laitue sur la rive ouest d'une rivière. Il veut amener ses animaux et sa laitue de l'autre côté de la rivière, sur la rive est. Le fermier a un petit bateau à rame et il n'y a de la place que pour lui et une autre chose. De plus, le loup peut manger la chèvre sans présence du fermier et la chèvre peut manger la laitue sans présence du fermier. Comment le fermier peut faire pour transporter tous les éléments sur la rive est ?

1. Formulez ce problème comme un problème de recherche. C'est-à-dire, donner la représentation des états, l'état de départ, les actions pour passer d'un état à un autre, la fonction successeur, l'état but et le coût des actions.
2. Résolvez ce problème de recherche (en utilisant la méthode de votre choix). Dessinez l'arbre de recherche et donner la solution finale

Solution

1. Il y a plusieurs manières possibles de formuler un problème de recherche. Mais voici une des réponses possibles :

   État Les états peuvent être représentés par deux ensembles W et E contenant les éléments sur la rive ouest et est respectivement. Pour les éléments, on peut utiliser les symboles $fermier$, $chèvre$, $loup$ et $laitue$.

   État initial $W = \{fermier, chèvre, loup, laitue\}$, $E = \{\}$

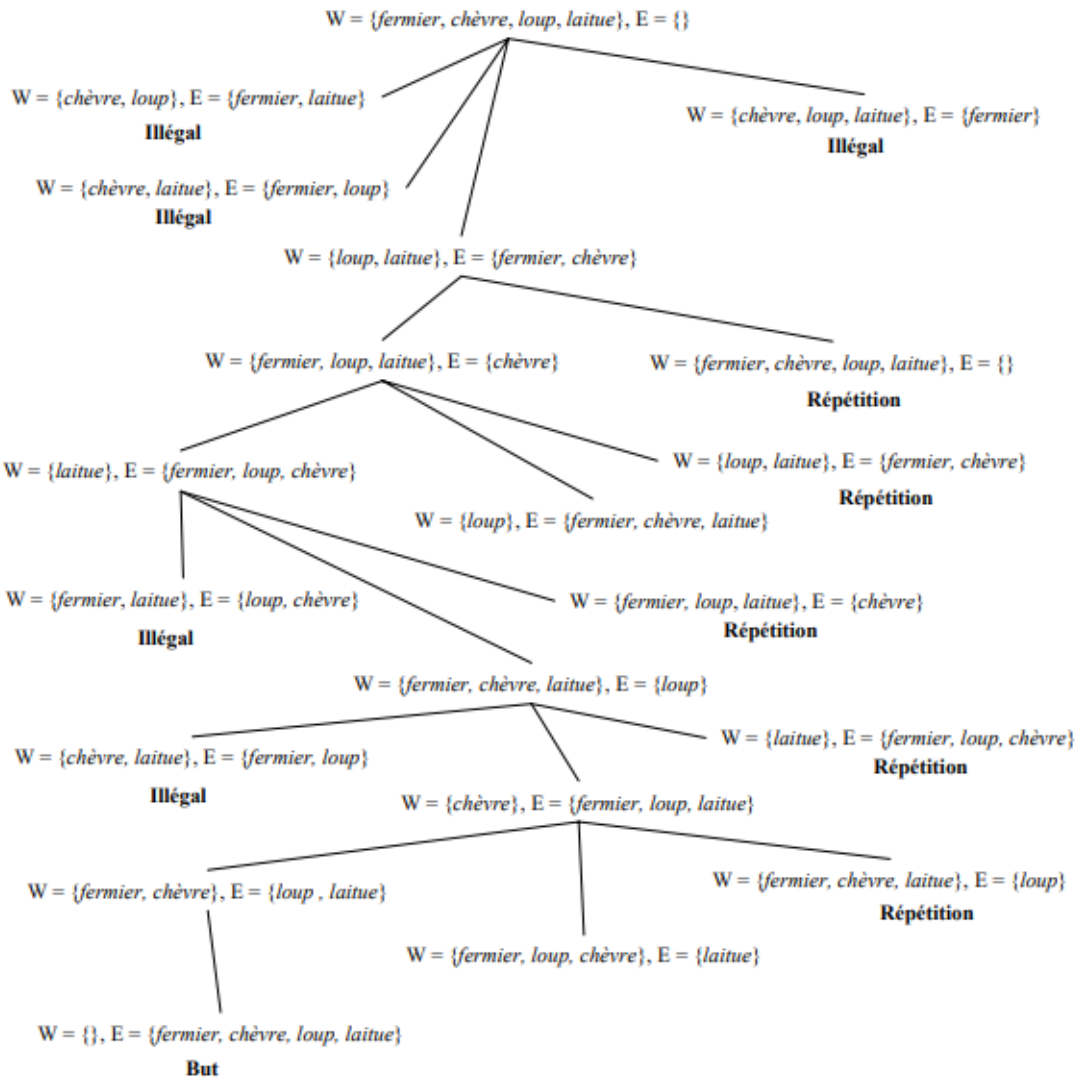
   Actions Déplacer fermier de l'ensemble E à l'ensemble W et vice versa. Déplacer $fermier$ et un des éléments $chèvre$, $loup$ ou $laitue$ de l'ensemble E à l'ensemble W et vice versa.

   Fonction de successeur Les ensembles W et E résultant de l'éxécution d'une action dans l'état courant.

   Test de but $W = \{\}$, $E = \{fermier, chèvre, loup, laitue\}$

   Coût des actions Chaque déplacement a coût de 1 (pour trouver une solution avec le moins de déplacements)
2. Voici un arbre de recherche possible :

4. Dans les problèmes de planification, on cherche les étapes à effectuer pour arriver sur un état but connu.
5. Dans les problèmes de satisfaction de contraintes, on cherche un état but respectant les contraintes, peu importe le chemin parcouru



3.3 Résolution de solutions dans l'espace d'états

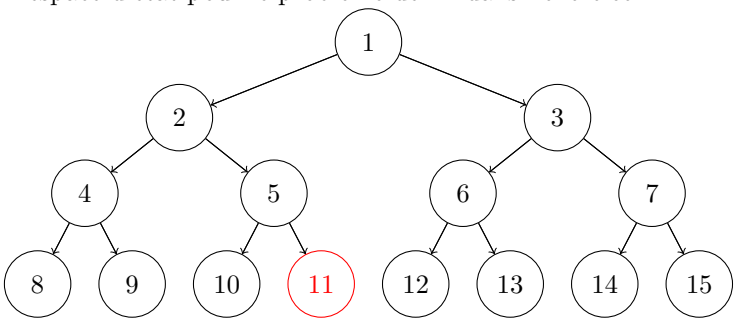
Exercice 3

Considérez un espace de recherche dans lequel l'état initial est 1 et chaque état k possède deux successeurs: nombres $2k$ et $2k + 1$

1. Dessinez la partie de l'espace de recherche contenant les noeuds de 1 a 15.
2. Supposez que le but soit 11. Donnez l'ordre de parcours des noeuds pour les algorithmes :
   - largeur d'abord
   - profondeur d'abord
   - profondeur d'abord limitée à 2
   - profondeur itérative

Solution (inspirée de (Russell & Norvig, 2003))

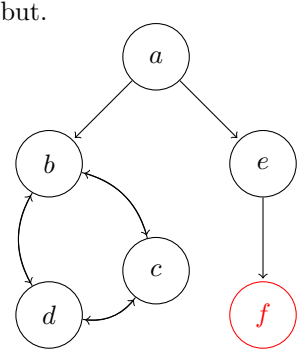
1. L'espace d'état pour le problème défini dans l'exercice :



2. Recherche en largeur : $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$
   Recherche en profondeur : $\{1, 2, 4, 8, 9, 5, 10, 11\}$
   Recherche en profondeur limitée à 2 : $\{1, 2, 4, 5, 3, 6, 7\}$
   Recherche par approfondissement itératif : $\{1; 1, 2, 3; 1, 2, 3; 1, 2, 4, 5, 3, 6, 7; 1, 2, 4, 8, 9, 5, 10, 11\}$

Exercice 4

Considérons le problème de recherche représenté dans la figure ci-dessous, où a représente l'état initial et f est le but.



1. Préférez-vous DFS (recherche en profondeur) ou BFS (recherche en largeur) pour ce problème ? Pourquoi?
2. Quelles séquences de chemins sont explorées par BFS et DFS dans ce problème ?

Solution

1. Si on exécute simplement l'algorithme DFS pur (pas d'élagage ni de vérification de boucle), on préférere BFS, car DFS pourrait rester bloqué dans une boucle infinie. On note que DFS est sensible à l'ordre d'exploration des noeuds. S'il explore d'abord à gauche, il restera coincé dans la boucle, alors que s'il explore d'abord à droite, il trouvera le but très rapidement.
2. DFS explore $a \to b \to d \to b \to d$ et continue d'osciller entre les deux noeuds b et d (bien sûr, s'il explore d'abord à gauche).
   BFS commence par les chemins $a \to b$, $a \to e$ ensuite ils deviennent $a \to b \to d$, $a \to b \to c$ et $a \to e \to f$ (attentient le noeud correspndant à l'état f n'est pas encore sélectionné, donc on ne s'arrête pas). Après, on obtient les chemins $a \to b \to d \to b$, $a \to b \to d \to c$, $a \to b \to c \to b$, $a \to b \to c \to d$. Finalement, le noeuds correspondant à f est sélectionné de la liste des noeuds à traiter. L'état but est atteint, donc l'algorithme s'arrête et renvoie le chemin $a \to e \to f$.

Exercice 5

Le graphe de la figure ci-dessous montre l'espace d'états d'un problème de recherche hypothétique. Les états sont désignés par des lettres et le coût de chaque action est indiqué sur l'arête correspondante. Notez que les actions ne sont pas réversibles, puisque le graphe est orienté. Le tableau à côté de l'espace d'états montre la valeur d'une fonction heuristique admissible, en considérant G comme l'état de but (il est facile de vérifier qu'une telle heuristique ne surestime jamais le coût réel du chemin minimum d'un état donné à l'état de but G).

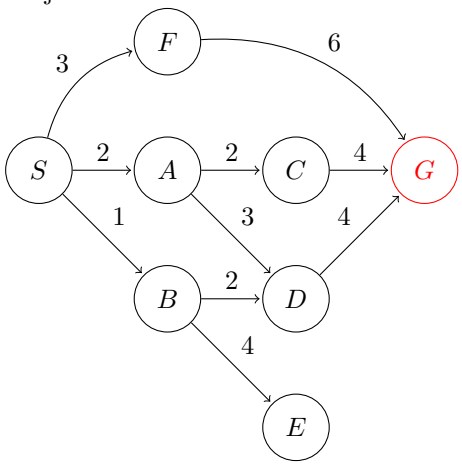


Figure 3.4 – L'espace des états.

S	A	B	C	D	E	F	G
6	4	5	2	2	8	4	0

Figure 3.5 – La fonction heuristique (état du but : G)

En considérant S comme état initial, résolvez le problème de recherche ci-dessus en utilisant une :

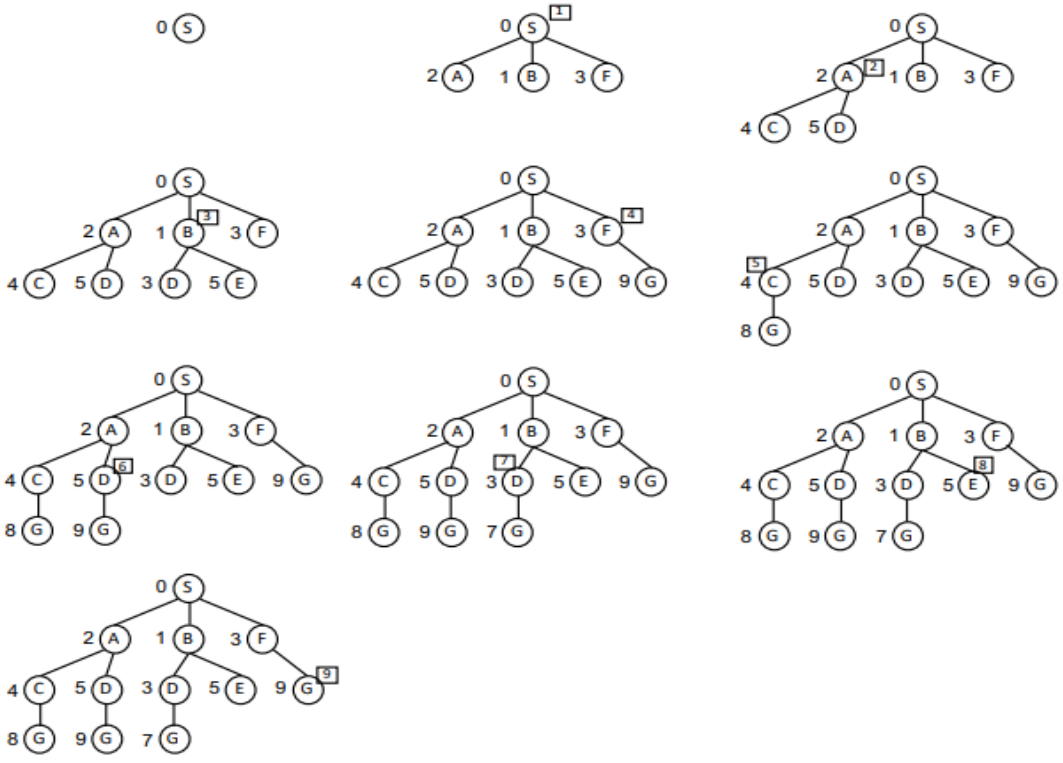
1. Recherche en largeur d'abord ou BFS (Breadth First Search)

2. Recherche en profondeur d'abord ou DFS (Depth First Search)
3. Recherche meilleur d'abord gloutonne (Greedy best-first search)
4. Recherche A*

Lorsque vous dessinez l'arbre de recherche, vous devez clairement indiquer : l'ordre d'expansion de chaque noeud (par exemple, en numérotant les nœuds élargis selon l'ordre de leur expansion) ; l'action correspondant à chaque arête de l'arbre ; l'état, le coût du chemin et la valeur de l'heuristique de chaque noeud.

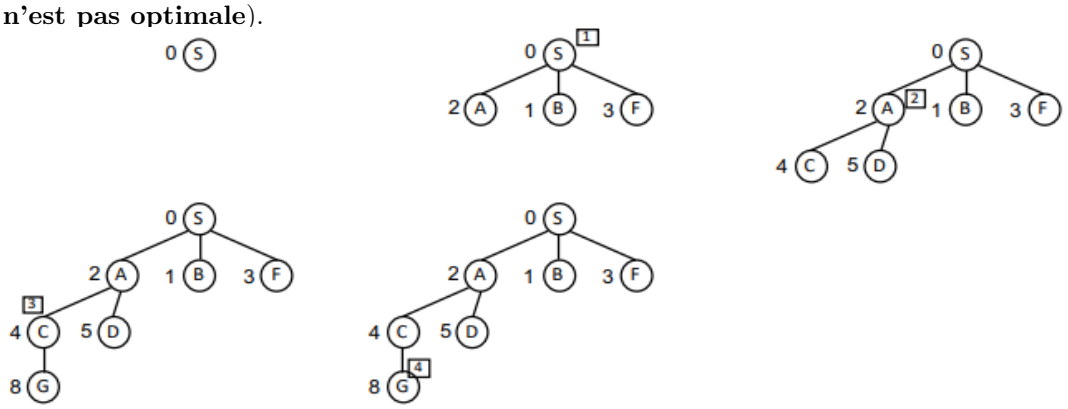
Solution (inspirée de (Fumera, 2019/2020b))
Les arbres de recherche construits par les trois stratégies sont présentés ci-dessous. Le numéro à gauche de chaque noeud indique son coût de chemin pour la recherche en largeur et en profondeur d'abord (notez cependant que le coût de chemin n'est pas utilisé par ces stratégies pour choisir le noeud feuille suivant à développer), la valeur de l'heuristique pour la recherche meilleur d'abord gloutonne et la somme de son coût de chemin et de son heuristique pour la recherche A*. L'ordre d'expansion est indiqué par les petits nombres à l'intérieur des carrés. Par souci de clarté, l'arbre de recherche obtenu après chaque expansion est présenté.
N'oubliez pas que lorsque plusieurs nœuds feuilles peuvent être choisis pour l'expansion (c'est-à-dire lorsqu'il y a plus d'un noeud feuille à la profondeur la plus basse pour la recherche en largeur d'abord, ou à la profondeur la plus élevée pour la recherche en profondeur d'abord, ou avec la valeur la plus basse de la fonction d'évaluation pour la recherhe meilleur d'abord gloutonne et A*[6]) un choix aléatoire doit être fait parmi eux ; dans cet exercice, on suppose que le choix est fait en considérant l'ordre alphabétique parmi les états correspondants (par exemple, si le choix se fait parmi les noeuds A, B et F, le noeud A est choisi).
Enfin, rappelez-vous que le test de but est appliqué lorsqu'un noeud est sélectionné pour l'expansion : par conséquent, une solution n'est pas trouvée lorsqu'un noeud contenant un état but est généré (en développant son noeud parent), mais lorsqu'il est sélectionné pour l'expansion.

1. Recherche en largeur d'abord : L'arbre de recherche construit par la recherche en largeur d'abord, étape par étape, est illustré dans la figure ci-dessous (de haut en bas et de gauche à droite). Notez que différents noeuds peuvent correspondre à un même état (c'est le cas des états D et G) : cela signifie qu'un tel état peut être atteint à partir de l'état initial par différents chemins dans l'espace d'états (c'est-à-dire différentes séquences d'actions), éventuellement avec des coûts de chemin différents : par conséquent, ces noeuds doivent rester distincts dans l'arbre de recherche. Notez également que le noeud E n'a pas de successeurs (aucun autre état ne peut être atteint à partir de l'état E, selon l'espace d'états) : par conséquent, lorsqu'il est sélectionné pour l'expansion (à l'étape 8), aucun noeud enfant n'est ajouté à l'arbre. Enfin, notez que la solution trouvée par la recherche en largeur d'abord ($S \to F \to G$) n'est pas celle avec un coût de chemin minimum, car cela peut arriver du fait que **cette stratégie de recherche n'est pas optimale**.

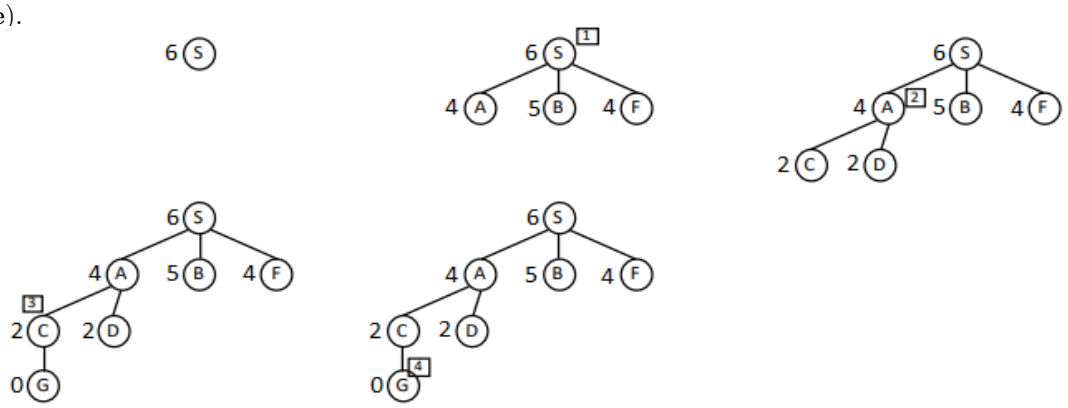


6. La fonction d'évaluation pour la recherche A* est la somme du coût du chemin et de la fonction heuristique.

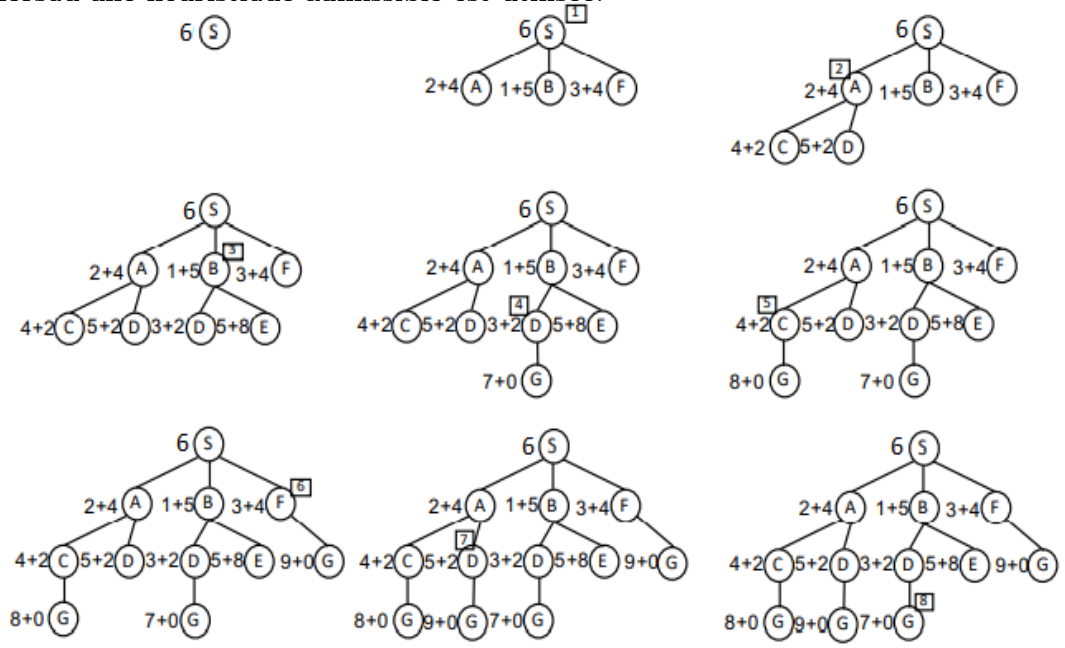
2. Recherche en profondeur d'abord : L'arborescence de recherche construite par la recherche en profondeur d'abord est illustrée ci-dessous. Dans ce cas, une solution est trouvée le long du chemin qui est exploré en premier (lorsque le noeud correspondant à l'état G est sélectionné pour l'expansion à l'étape 4). Notez que dans ce cas également, la solution trouvée ($S \rightarrow A \rightarrow C \rightarrow G$) n'est pas la solution à coût minimum (**la recherche en profondeur d'abord n'est pas optimale**).



3. Recherche meilleur d'abord gloutonne : L'arbre de recherche construit par la recherche meilleur d'abord gloutonne est illustrée cidessous. Cet algorithme évalue les noeuds en utilisant uniquement la fonction heuristique : $f(n) = h(n)$, et examine ceux qui semblent les plus proches d'un état but, dans l'espoir d'aboutir plus vite à une solution. Pour l'expansion, on choisit le noeud feuille avec la valeur minimale de $f(n)$. La solution trouvée ($S \rightarrow A \rightarrow C \rightarrow G$) n'a pas le coût minimum (**la recherche meilleur d'abord gloutonne n'est pas optimale**).

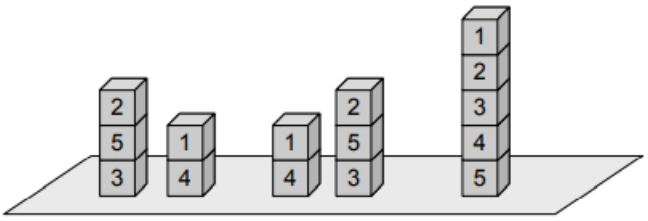


4. Recherche A* : L'arborescence de recherche construite par la recherche A* est illustrée ci-dessous. À côté de chaque noeud, le coût de chemin correspondant $g(n)$ et l'heuristique $h(n)$ sont représentés par $f(n) = g(n)+h(n)$. Rappelez-vous qu'à chaque étape, le noeud feuille avec la valeur minimale de $f(n)$ est choisi pour l'expansion. La solution trouvée à l'étape 8 ($S \rightarrow B \rightarrow D \rightarrow G$) est garantie à coût minimum, selon le fait que **A* est optimal lorsqu'une heuristique admissible est utilisée**.



Exercice 6

Le monde des blocs est un domaine de jouets bien connu utilisé en IA pour les problèmes de planification liés aux robots. On onsidère une version simple de ce problème, dans laquelle il y a cinq blocs de forme et de taille identiques, numérotés de 1 à 5. Les blocs peuvent être empilés en une ou plusieurs piles. La figure ci-dessous montre deux configurations possibles de tels blocs ; notez que la position relative des piles n'a pas d'importance, ainsi la configuration de gauche est équivalente à celle du milieu. Chaque bloc peut être soit sur la table (il n'y a pas de contraintes sur le nombre de piles) ou au sommet de n'importe quel autre bloc. Le but est d'empiler les blocs en une seule pile, dans l'ordre indiqué dans la configuration la plus à droite de la figure, à partir d'un ensemble donné de piles, en déplaçant le plus petit nombre possible de blocs. Seuls les blocs au sommet des piles actuelles peuvent être déplacés et ne peuvent être placés que sur la table ou au sommet d'une autre pile.



1. Formulez la version ci-dessus du monde des blocs comme un problème de recherche, en définissant précisément l'espace d'états, les états, l'état initial, l'ensemble des actions possibles, la fonction de successeur, le test de but et le coût du chemin.
2. On suppose que la configuration initiale est celle illustrée à gauche dans la figure ci-dessus, et considère une fonction heuristique définie comme le nombre de blocs qui ne sont pas dans la pile la plus élevée (ou dans l'une des piles les plus élevées). Sous ce paramètre, montrez comment la stratégie de recherche A* développe les quatre premiers noeuds de l'arbre de recherche, en évitant les états répétés. Comme dans l'exercice précédent, lorsque vous dessinez l'arbre de recherche, vous devez clairement indiquer : l'ordre d'expansion de chaque noeud ; l'action correspondant à chaque arête de l'arbre ; l'état, le coût du chemin et la valeur de l'heuristique de chaque noeud.
3. Définissez une autre heuristique admissible possible, et prouvez son admissibilité.

Solution (inspirée de (Fumera, 2019/2020b))

1. La formulation standard de la version simplifiée décrite ci-dessus du monde des blocs :

   États L'espace d'état est constitué de l'ensemble des arrangements distincts des cinq blocs en une ou plusieurs (jusqu'à cinq) piles. Un état peut être représenté par un ensemble [7] de piles, et chaque pile peut être représentée comme une séquence de numéros de blocs, où les numéros de gauche à droite correspondent, par exemple, aux blocs du haut vers le bas de la pile. Par exemple, les deux configurations équivalentes de piles (états) représentées à gauche et au milieu de la figure ci-dessus sont représentées par l'ensemble $\{(2,5,3),(1,4)\}$.

   État initial N'importe quel ensemble de piles donné.

   Actions Les actions peuvent être formellement décrites comme suit :
   Étant donné un état $\{(b_{1,1},...,b_{1,n_1}),...,(b_{p,1},...,b_{p,n_p})\}$, où p désigne le nombre de piles ($1 \leq p \leq 5$) et n_k le nombre de blocs dans la k^{e} pile ($n_k \geq 1$ pour chaque k, et $\sum_{k=1}^{p} n_k = 5$), les actions consistent à déplacer un des p blocs $b_{k,1}$ (sur le dessus d'une des piles) soit à la table, générant ainsi une nouvelle pile (uniquement si la pile d'origine contient plus d'un bloc, c'est-à-dire, $n_k > 1$), ou au sommet de l'un des autres $p-1$ piles (le cas échéant). Chaque action peut être décrite en indiquant le numéro du bloc qui a été déplacé, $b_{k,1}$, et sa nouvelle position, c'est-à-dire le numéro du bloc au sommet duquel il a été placé, ou la table (qui peut être désignée par le chiffre 0). Par exemple, de l'état $\{(2,5,3),(1,4)\}$ l'action $(2,0)$ consiste à déplacer le bloc 2 vers la table, conduisant à l'état $\{(5,3),(2),(1,4)\}$, et $(1,2)$ consiste à déplacer le bloc 1 au-dessus du bloc 2, conduisant à l'état $\{(1,2,5,3),(4)\}$

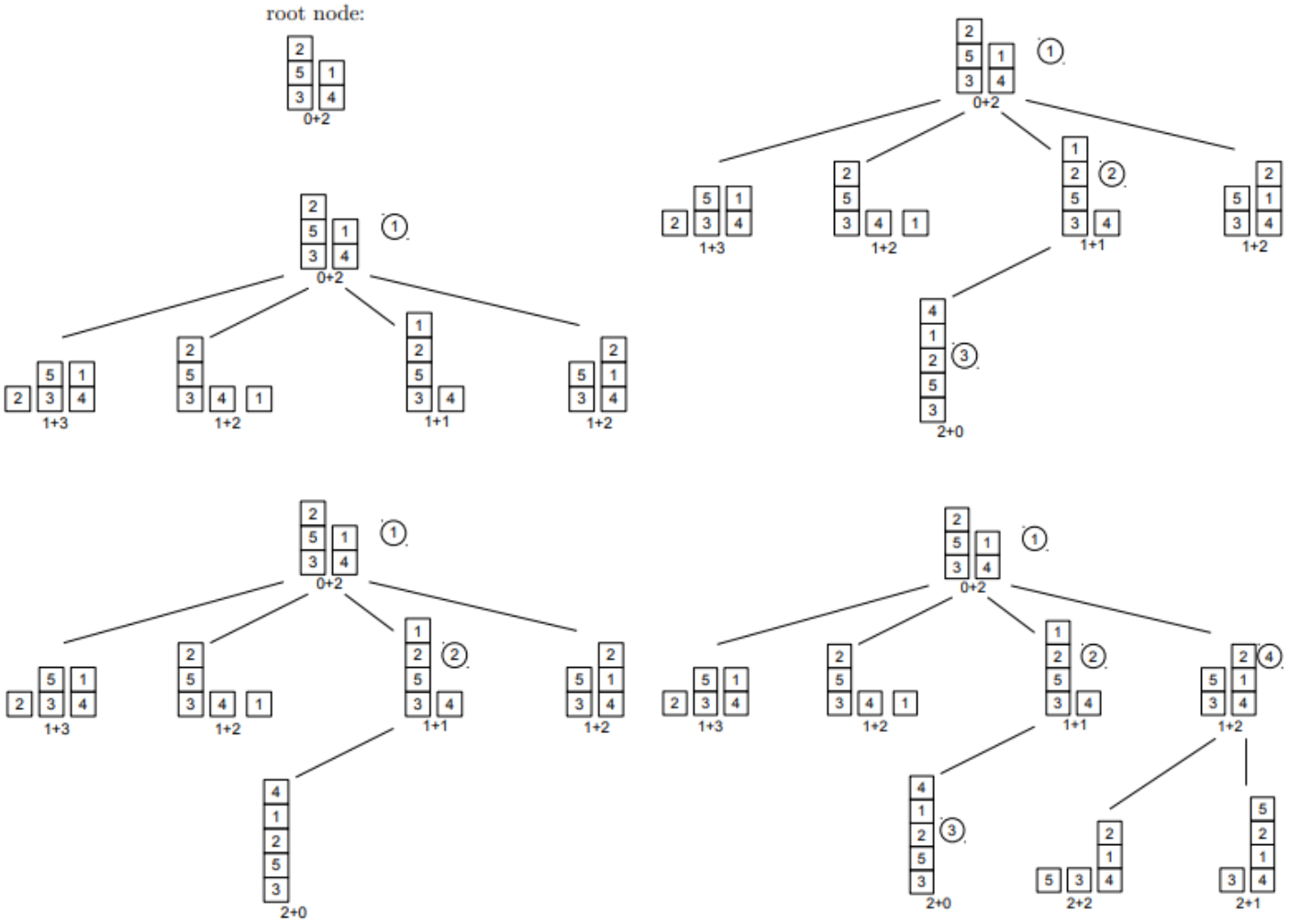
   Fonction de successeur Les actions peuvent être implémentées en tant que fonction successeur : il reçoit en argument la description s d'un état, et renvoie tous les couples (s', a) où s' est l'un des états obtenus comme décrit ci-dessus, et a la description de l'action correspondante.

   Test de but L'état du but est représenté par $\{(1,2,3,4,5)\}$.

   Coût des actions Puisque le but est d'atteindre la configuration cible en déplaçant le moins de blocs possible, il s'ensuit que le coût du chemin est donné par le nombre d'actions qui ont été effectuées pour atteindre un état donné depuis l'état initial, et donc chaque action a un coût égal à 1.

7. Selon la définition du problème, l'ordre entre les piles n'est pas important.

2. L'arbre de recherche obtenu par la stratégie de recherche A* après l'expansion des quatre premiers noeuds, en utilisant l'heuristique donnée dans le texte et en évitant les états répétés, est présenté dans la figure ci-dessous. Par souci de clarté, l'arbre obtenu après l'expansion de chaque noeud est représenté.
La fonction d'évaluation A* (coût du chemin + heuristique) est indiquée sous chaque état. Les nombres à l'intérieur des cercles indiquent l'ordre dans lequel le nœud correspondant a été développé. Notez que le troisième nœud n'a pas de successeurs, puisque la seule action pouvant être effectuée sur son état conduit à un état répété (celui de son nœud parent). Notez également qu'il y a deux nœuds feuilles candidats à développer à la quatrième itération de A* (les deuxième et quatrième noeuds à partir de la gauche sur la figure, à la profondeur 1), puisqu'ils présentent la même valeur de la fonction d'évaluation, qui est inférieure à celle des noeuds feuilles restants ; dans ce cas, un choix aléatoire doit être fait entre les noeuds candidats : dans la figure ci-dessous, le noeud de droite est choisi pour l'expansion.



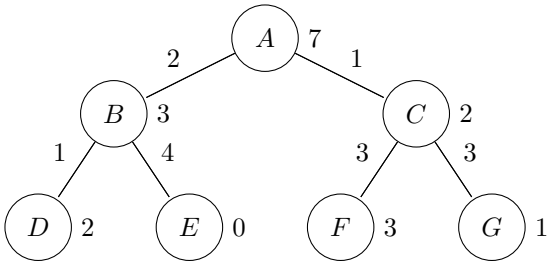
3. Une autre heuristique admissible possible est la suivante : soit n le nombre de blocs au-dessus duquel se trouve un bloc différent de celui dans l'état but ; alors l'heuristique est définie comme $\lfloor \frac{n}{2} \rfloor$. Par exemple, si l'état du but est $\{(1,2,3,4,5)\}$, dans l'état $\{(2,5,3),(1,4)\}$:
   - il n'y a pas de bloc au-dessus du bloc 1, qui compte pour 0 ;
   - il n'y a pas de bloc au-dessus du bloc 2, contrairement au but : cela compte pour 1 ;
   - le bloc 5, au lieu du bloc 2, est au-dessus du bloc 3 : cela compte comme 1 ;
   - le bloc 1, au lieu du bloc 3, est au-dessus du bloc 4 : cela compte pour 1 ;
   - le bloc 2, au lieu du bloc 4, est au-dessus du bloc 5 : cela compte pour 1 ;

   La valeur de la fonction heuristique pour l'état $\{(2,5,3),(1,4)\}$ est donc $\lfloor \frac{4}{2} \rfloor = 2$; en d'autres termes, au moins deux blocs doivent être déplacés pour atteindre l'état cible.
   Il n'est pas difficile de voir que cette heuristique est admissible : en déplaçant un bloc de n'importe quel état s', au plus deux autres blocs auront un bloc différent au sommet dans l'état résultant s", et donc dans s" au plus deux blocs supplémentaires que dans s' peuvent avoir le bloc correct au sommet comme dans l'état de but.

Exercice 7

On suppose que l'arbre de recherche ci-dessous a été obtenu pour un problème de recherche après les trois premières étapes d'expansion par la stratégie A*. Les nombres sur les arêtes indiquent le coût des actions correspondantes, les lettres à l'intérieur de chaque nœud indiquent l'état correspondant et les nombres à côté de chaque nœud indiquent la valeur de la fonction heuristique (qui est supposée être admissible) pour le même état.

— Quel noeud A* sélectionnera-t-il pour l'expansion à l'étape suivante ?

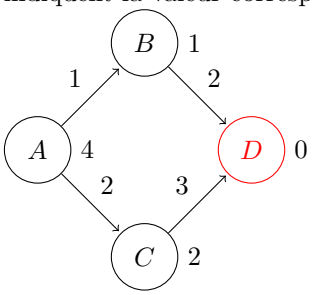


Solution

A* sélectionne pour l'expansion le noeud feuille n avec la valeur la plus basse de la fonction d'évaluation $f(n) = g(n) + h(n)$, où g et h désignent le coût de chemin de n (c'est-à-dire la somme des les coûts d'étapes - coûts des actions - dans le chemin du noeud racine à n) et la valeur de la fonction heuristique pour l'état correspondant, respectivement. Il s'ensuit immédiatement que A* sélectionnera (au hasard) le noeud feuille le plus à gauche ou le plus à droite, dont la fonction d'évaluation est égale à 5.

Exercice 8

1. Considérez l'espace d'états d'un problème de recherche donné dans la figure ci-dessous, où D est l'état du but, les nombres sur chaque arête indiquent le coût de l'action correspondante et les nombres à côté de chaque état indiquent la valeur correspondante d'une fonction heuristique.



   — La fonction heuristique considérée est-elle admissible ?

2. Proposez deux heuristiques admissibles pour le problème du taquin à 8 pièces.

Solution

Une fonction heuristique est admissible si elle ne surestime jamais le coût minimum d'un état à l'état cible.

1. Il est facile de voir que la condition de recevabilité n'est pas remplie par l'heuristique considérée :
   — Le coût minimum des états B et C à l'état cible D est respectivement de 2 et 3, ce qui est supérieur à l'heuristique correspondante respectivement, 1 et 2) ;
   — Le coût minimum de l'état A à D est 3, ce qui correspond au chemin $A \rightarrow B \rightarrow D$: il est cependant inférieur à l'heuristique correspondante (4).
2. Deux exemples différents d'heuristiques admissibles qui s'appliquent au problème :
   — $h_1(n)$: La distance de Hamming qui est le nombre total de pièces mal placées. Il est clair que cette heuristique est admissible puisque le nombre total de déplacements pour ordonner correctement les pièces est au moins égal au nombre de pièces mal placées (chaque pièce non en place doit être déplacée au moins une fois). Le coût (nombre de déplacements) jusqu'au but est au moins égal à la distance de Hamming.
   — $h_2(n)$: La distance de Manhattan qui est la somme des distances de chaque pièce à partir de sa position dans l'état but. Elle est une heuristique admissible car chaque pièce devra être déplacée d'au moins le nombre de points entre elle-même et sa position correcte.

Exemple : On calcule la distance de Hamming et de Manhattan entre l'état initial et l'état final dans l'exemple de taquin à 8 pièces de la figure 2.
$h_1(n) = 8$
$h_2(n) = 3 + 1 + 2 + 2 + 3 + 2 + 2 + 3 = 18$

Exercice 9

Considérez le problème du taquin à 4 pièces, qui est une version plus simple du taquin à 8 pièces où le plateau est de 2×2 et il y a trois pièces, numérotées 1, 2 et 3, et une case vierge. Il y a quatre opérateurs qui déplacent le blanc vers le haut, le bas, la gauche et la droite. Les états de départ et de but sont donnés ci-dessous. Montrez comment le chemin vers le but peut être trouvé en utilisant une :

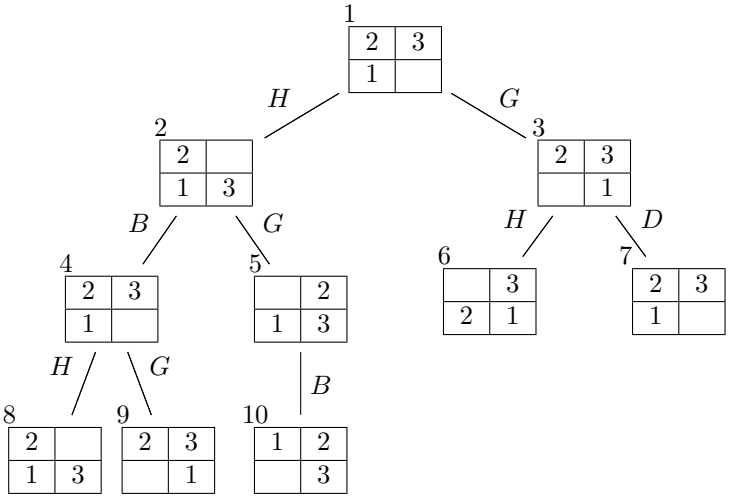
1. Recherche en largeur d'abord ou BFS (Breadth First Search)
2. Recherche en profondeur d'abord ou DFS (Depth First Search)
3. Recherche A*

L'heuristique étant la somme du nombre de coups et du nombre de pièces mal placées.
On suppose qu'il n'y ait aucune possibilité de se souvenir des états qui ont été visités plus tôt. Utilisez également les opérateurs donnés dans l'ordre indiqué, sauf indication contraire de la méthode de recherche. Étiquetez chaque nœud visité avec un numéro indiquant l'ordre dans lequel ils sont visités. Si une méthode de recherche ne trouve pas de solution, expliquez pourquoi cela s'est produit.

Solution

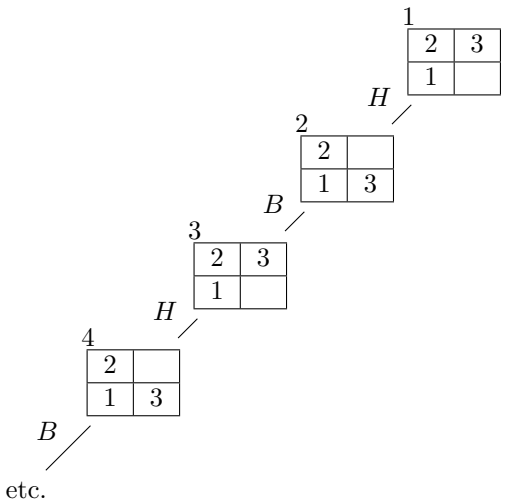
1. Recherche en largeur d'abord ou BFS (Breadth First Search) :

La recherche en largeur d'abord commencera à partir du noeud racine, puis étendra tous les successeurs de ce noeud, puis tous leurs successeurs et ainsi de suite. La recherche en largeur d'abord s'arrête lorsque la première solution est trouvée.

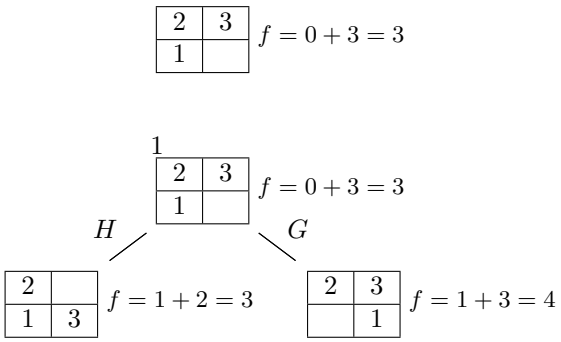


2. Recherche en profondeur d'abord ou DFS (Depth First Search) :

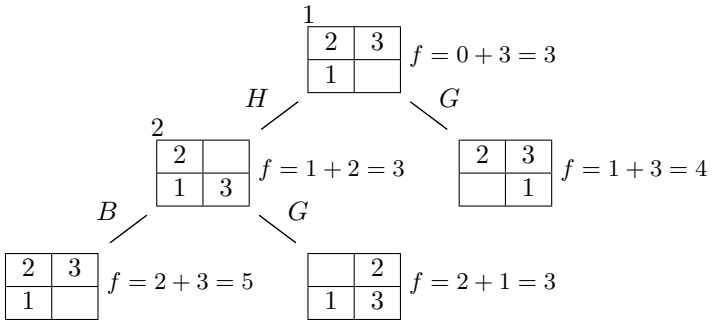
La recherche en profondeur d'abord développe le nœud le plus profond dans l'arbre de recherche. Notez qu'il n'y avait aucun mécanisme pour se souvenir des états qui ont été visités plus tôt. La profondeur d'abord ne trouvera pas de solution car elle commencera à osciller entre les mouvements H et B.



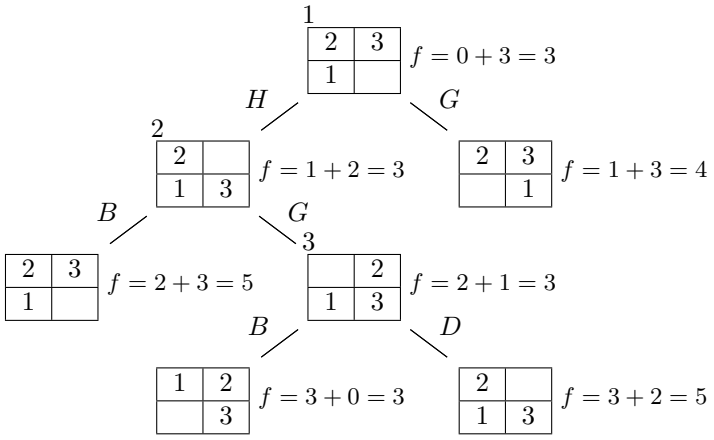
3. Recherche A* :
   A* est une méthode de recherche qui ouvre le noeud avec la plus petite valeur de fonction d'évaluation $f(n) = g(n) + h(n)$ ($g(n)$ le coût du chemin entre l'état initial et le noeud et $h(n)$ la valeur de la fonction heuristique). La recherche commence à partir de l'état de départ.
   (a) Lorsque l'état initial est ouvert, on voit deux états.



   (b) L'état le plus à gauche a le coût le plus bas, donc celui-là est choisi et ouvert. On voit maintenant des états avec des coûts 5, 3 et 4.

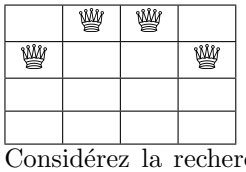


   (c) L'état avec le coût le plus bas est ouvert et on voit deux autres noeuds, y compris le noeud de but. On remarque que le noeud d'objectif a le coût le plus bas, on le choisit donc et peut terminer la recherche. Si un autre noeud avec une valeur de fonction de coût inférieure était encore visible, la recherche A* le choisirait au lieu du nœud de but qui a un coût plus élevé. C'est parce que A* essaie de trouver le chemin avec le coût le plus bas.



Exercice 10

Soit l'état initial du problème des 4 reines illustré ci-dessous :

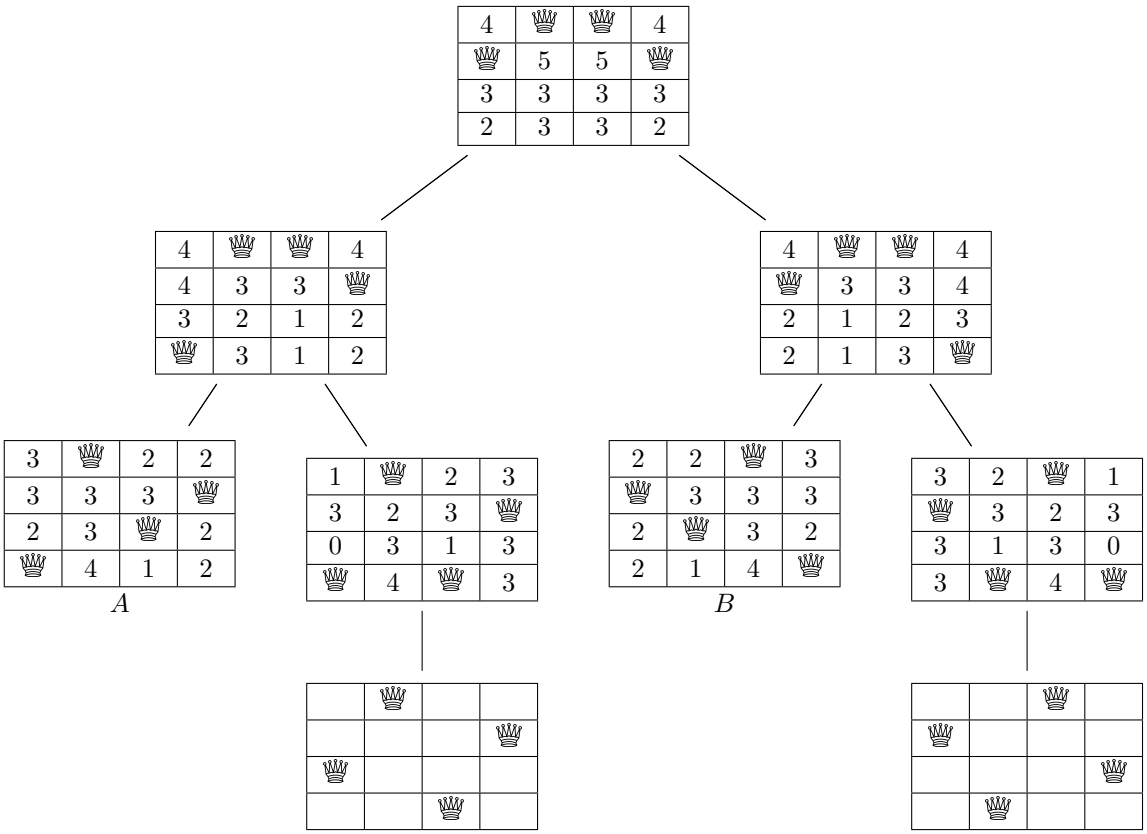


Considérez la recherche locale gloutonne (hill-climbing) pour trouver un état valide du problème des 4 reines. La valeur heuristique d'un état est le nombre de paires de reines distinctes qui peuvent s'attaquer les unes aux autres. Les successeurs d'un état sont tous les états possibles générés en déplaçant une seule reine vers une autre case de la même colonne. L'algorithme prendrait à chaque pas le successeur avec le plus petit nombre de conflits

1. Dessinez l'arbre des états que Hill-climbing explore à partir de l'état initial en utilisant l'heuristique de conflit minimum. Les successeurs de chaque état sont les états qui ont le moins de conflits. Dessinez l'arbre correspondant à tous les chemins d'exploration possibles de Hill-climbing
2. Toutes les feuilles de l'arbre construit dans 1) sont-elles des solutions au problème des 4 reines ? Sinon, comment faire pour améliorer les états des feuilles qui ne sont pas des solutions ?

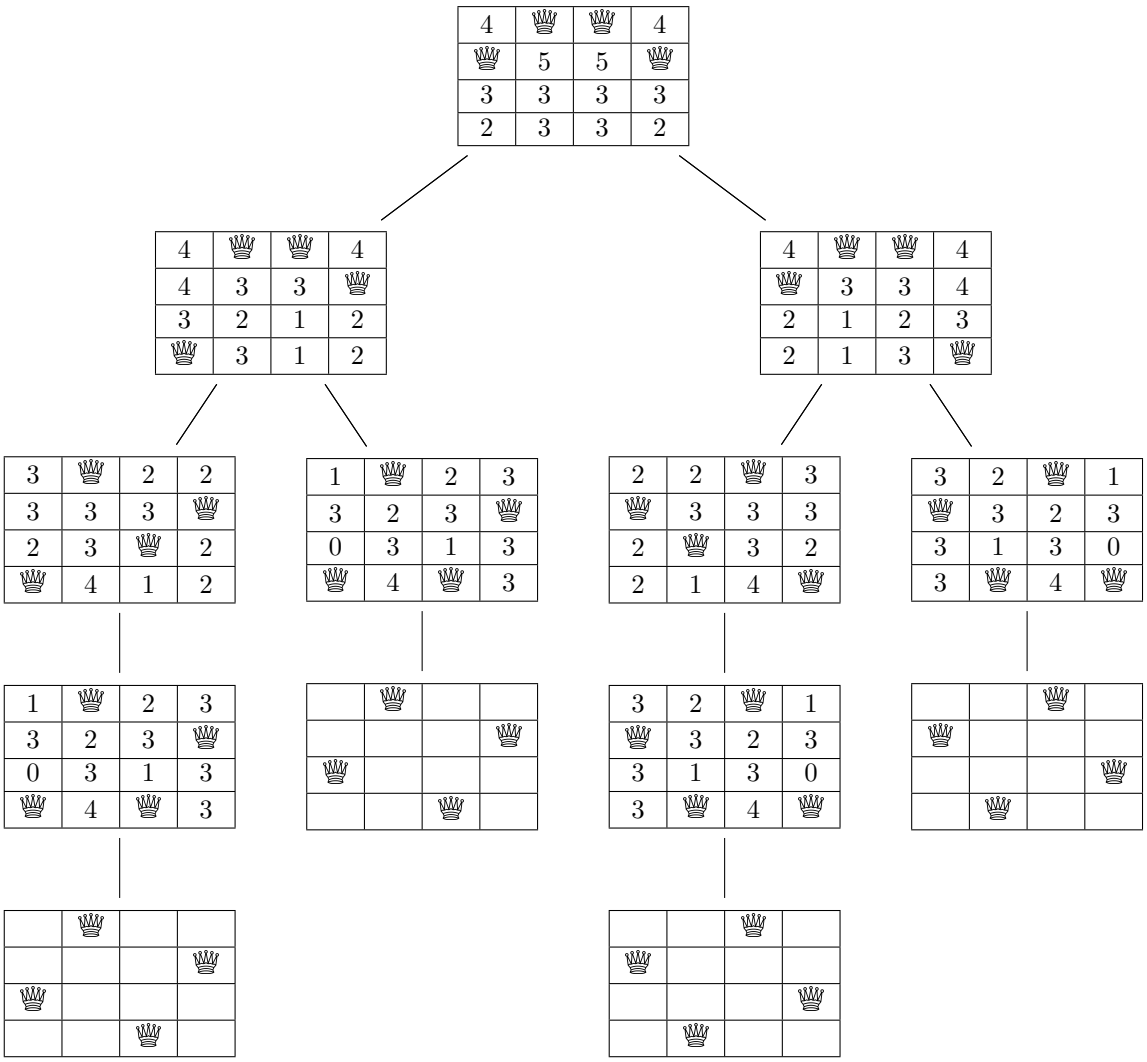
Solution

1. L'arbre pour la recherche locale gloutonne (hill-climbing) :



2. Toutes les feuilles de cet arbre ne sont pas des solutions. Par exemple, les feuilles A et B sont des optima locaux. La stratégie ici consiste à permettre des déplacements latéraux dans la recherche. Dans la recherche locale gloutonne standard, on arrête la recherche quand $neighbor$.Value $\geq$ $current$.Value (pour un problème de minimisation). Cela signifie qu'on arrête la recherche lorsque la valeur objective de l'état successeur est la même que celle de l'état courant. C'est exactement ce qui s'est passé dans 1) pour les états A et B. En A et B, tous les états successeurs ont $h = 1$, ce qui est identique à la valeur h de A et B.
   Ainsi, à la place du critère d'arrêt, on peut utiliser $neighbor$.Value $>$ $current$.Value[8]. Cela signifie qu'on ne s'arrête que lorsque la valeur h de l'étape suivante est strictement supérieure à la valeur h de l'étape précédente. Cela permet à notre algorithme de se déplacer dans des zones plates et d'explorer un peu plus. Lorsque cela est fait, on obtient ce qui suit :

8. Permettre à l'algorithme de visiter les successeurs ayant la même valeur que l'état courant.



Exercice 11 (inspiré de (Hermawanto, 2013))
On voudrait trouver en utilisant l'algorithme génétique la valeur de a, b, c et d qui satisfait l'équation suivante $a + 2b + 3c + 4d = 30$. Pour ce problème, l'objectif est de minimiser la valeur de la fonction $f(x)$ où $f(x) = |((a + 2b + 3c + 4d) - 30)|$. Puisqu'il y a quatre variables dans l'équation, à savoir a, b, c et d, on peut composer le chromosome (l'état) comme suit :

a	b	c	d

Pour accélérer le calcul, on limite les valeurs des variables a, b, c et d à entiers compris entre 0 et 30.

— Déroulez une itération de l'algorithme génétique.

Solution

1. **Initialisation**
   Par exemple, on définit le nombre de chromosomes (états) dans la population à 6, puis on génère une valeur aléatoire du gène a, b, c, d pour 6 chromosomes.
   — Etat[1] = [12,05,23,08]
   — Etat[2] = [02,21,18,03]
   — Etat[3] = [10,04,13,14]
   — Etat[4] = [20,01,10,06]
   — Etat[5] = [01,04,13,19]
   — Etat[6] = [20,05,17,01]
2. **Sélection**
   On calcule la valeur de la fonction objectif pour chaque chromosome produit lors de l'étape d'initialisation :

F_obj[1] = $|((12 + 2 * 05 + 3 * 23 + 4 * 08) - 30)| = 93$
F_obj[2] = $|((02 + 2 * 21 + 3 * 18 + 4 * 03) - 30)| = 80$
F_obj[3] = $|((10 + 2 * 04 + 3 * 13 + 4 * 14) - 30)| = 83$
F_obj[4] = $|((20 + 2 * 01 + 3 * 10 + 4 * 06) - 30)| = 46$
F_obj[5] = $|(01 + 2 * 04 + 3 * 13 + 4 * 19) - 30)| = 94$
F_obj[6] = $|((20 + 2 * 05 + 3 * 17 + 4 * 01) - 30)| = 55$

Les chromosomes les plus aptes ont une probabilité plus élevée d'être sélectionnés pour la reproduction. Pour calculer la probabilité de fitness, on doit calculer la fitness de chaque chromosome. Pour éviter le problème de division par zéro, la valeur de F_obj est additionnée de 1.

$$\text{Fitness}[1] = \frac{1}{(1 + \text{F\_obj}[1])} = \frac{1}{94} = 0.0106$$
$$\text{Fitness}[2] = \frac{1}{(1 + \text{F\_obj}[2])} = \frac{1}{81} = 0.0123$$
$$\text{Fitness}[3] = \frac{1}{(1 + \text{F\_obj}[3])} = \frac{1}{84} = 0.0119$$
$$\text{Fitness}[4] = \frac{1}{(1 + \text{F\_obj}[4])} = \frac{1}{47} = 0.0213$$
$$\text{Fitness}[5] = \frac{1}{(1 + \text{F\_obj}[5])} = \frac{1}{95} = 0.0105$$
$$\text{Fitness}[6] = \frac{1}{(1 + \text{F\_obj}[6])} = \frac{1}{56} = 0.0179$$

Total = $0.0106 + 0.0123 + 0.0119 + 0.0213 + 0.0105 + 0.0179 = 0.0845$

La probabilité pour chaque chromosome est formulée par :

$$\text{P[i]} = \frac{\text{Fitness[i]}}{\text{Total}}$$

P[1] = $0.0106/0.0845 = 0.1254$
P[2] = $0.0123/0.0845 = 0.1456$
P[3] = $0.0119/0.0845 = 0.1408$
P[4] = $0.0213/0.0845 = 0.2521$
P[5] = $0.0105/0.0845 = 0.1243$
P[6] = $0.0179/0.0845 = 0.2118$

D'après les probabilités ci-dessus, on peut voir que le chromosome 4 qui a la meilleure fitness, ce chromosome a la probabilité la plus élevée d'être sélectionné pour la recombinaison.

Pour le processus de sélection, on utilise le système de la roulette (Roulette Wheel selection [9]), pour cela on doit calculer les valeurs de probabilité cumulées :

C[1] = 0.1254
C[2] = $0.1254 + 0.1456 = 0.2710$
C[3] = $0.1254 + 0.1456 + 0.1408 = 0.4118$
C[4] = $0.1254 + 0.1456 + 0.1408 + 0.2521 = 0.6639$
C[5] = $0.1254 + 0.1456 + 0.1408 + 0.2521 + 0.1243 = 0.7882$
C[6] = $0.1254 + 0.1456 + 0.1408 + 0.2521 + 0.1243 + 0.2118 = 1.0$

Après avoir calculé la probabilité cumulée de sélection, le processus utilisant la roulette peut être effectué. Le processus consiste à générer un nombre aléatoire R dans la plage 0-1 comme suit.

Si le nombre aléatoire R est inférieur à C[1], sélectionnez Etat[1] pour la reproduction. Sinon si le nombre aléatoire R est supérieur à C[i] et inférieur à C[i+1], sélectionnez Etat[i+1] comme chromosome pour la recombinaison.

On a besoin d'obtenir six paires de chromosomes pour l'étape de suivante (croisement). Donc, on utilise la roulette pour sélectionner les état comme expliqué ci-dessus.

R[1] = 0.954 R[2] = 0.428	R[1] = 0.133 R[2] = 0.906	R[1] = 0.603 R[2] = 0.765	R[1] = 0.597 R[2] = 0.892	R[1] = 0.311 R[2] = 0.870	R[1] = 0.543 R[2] = 0.284
x_1 = Etat[6] y_1 = Etat[4]	x_2 = Etat[2] y_2 = Etat[6]	x_3 = Etat[4] y_3 = Etat[5]	x_4 = Etat[4] y_4 = Etat[6]	x_5 = Etat[3] y_5 = Etat[6]	x_6 = Etat[4] y_6 = Etat[3]

3. **Croisement**

Après la sélection des chromosomes, le processus suivant consiste à déterminer, pour chaque paire à accoupler, la position du point de croisement. Cela se fait en générant des nombres aléatoires entre 1 et 4 (longueur du

9. La sélection des individus par le système de la roulette s'inspire des roues de loterie. À chacun des individus de la population est associé un secteur d'une roue. L'angle du secteur étant proportionnel à la probabilité de l'individu(chromosome). Vous tournez la roue et vous obtenez un individu.

chromosome).
P_C[1] = 1
P_C[2] = 2
P_C[3] = 4
P_C[4] = 3
P_C[5] = 1
P_C[6] = 2
Après avoir obtenu le point de croisement, les parents chromosomes seront coupés au point de croisement et ses gens seront échangés. Si c est le point de croisement alors on procède comme suit :

$$\text{APPEND}(\text{SUBSTRING}(x, 1, c), \text{SUBSTRING}(y, c+1, n))$$

[20, 05, 17, 01]	[02, 21, 18, 03]	[20, 01, 10, 06]	[20, 01, 10, 06]	[10, 04, 13, 14]	[20, 01, 10, 06]
[20, 01, 10, 06]	[20, 05, 17, 01]	[01, 04, 13, 19]	[20, 05, 17, 01]	[20, 05, 17, 01]	[10, 04, 13, 14]
[20, 01, 10, 06]	[02, 21, 17, 01]	[20, 01, 10, 06]	[20, 01, 10, 01]	[10, 05, 17, 01]	[20, 01, 13, 14]

4. **Mutation**
Chaque état obtenu est soumis à une mutation aléatoire avec une petite probabilité indépendante. Supposant que cette probabilité est de 0.4.
Pour chaque chromosome, on génère un nombre aléatoire R dans la plage 0-1. Si R est inférieur à 0.4 le chromosome va muter. Sinon la mutation ne sera pas applique.
Le processus de mutation se fait en générant un entier aléatoire entre 1 et 4 (longueur du chromosome) pour indiquer le gène (le point) de mutation (P_M). La valeur du gène muté et du point de mutation est remplacée par un nombre aléatoire compris entre 0 et 30.

Av_mu [10]	[20, 01, 10, 06]	[02, 21, 17, 01]	[20, 01, 10, 06]	[20, 01, 10, 01]	[10, 05, 17, 01]	[20, 01, 13, 14]
R	0.5 > 0.4	0.1 <= 0.4	0.8 > 0.4	0.7 > 0.4	0.05 <= 0.4	0.7 > 0.4
P_M	/	3	/	/	2	/
Ap_mu [11]	[20, 01, 10, 06]	[02, 21, 29, 01]	[20, 01, 10, 06]	[20, 01, 10, 01]	[20, 06, 13, 14]	[20, 01, 13, 14]

En terminant le processus de mutation, on a alors une itération ou une génération de l'algorithme génétique. Les états obtenus après mutation vont contstituer la nouvelle population (ou génération) et remplacer la population précédente (dans notre cas la population initiale).
Ces nouveaux chromosomes subiront le même processus que la génération précédente de chromosomes tels que **la sélection**, **le croisement** et **la mutation** et à la fin, ils produiront une nouvelle génération de chromosomes pour la prochaine itération. Ce processus sera répété jusqu'à ce qu'un individu soit convient suffisamment ou que suffisamment de temps se soit écoulé. Si au moins un de ces deux conditions soit vérifiée, l'algorithme génétique retourne le meilleur individu de la population, selon la fonction fitness.
Pour cet exemple, après avoir exécuté 50 générations, le meilleur chromosome est obtenu :
Chromosome = [07,05,03,01]
Cela signifie que : $a = 7$, $b = 5$, $c = 3$, $d = 1$
Si nous utilisons le nombre dans l'équation du problème :

$$a + 2b + 3c + 4d = 30 \Longrightarrow 7 + (2 * 5) + (3 * 3) + (4 * 1) = 30$$

On peut voir que la valeur de la variable a, b, c et d générée par l'algorithme génétique peut satisfaire cette égalité.

3.4 Résolution par décomposition de problèmes

Exercice 12

Soit le problème des Tours de Hanoi dans le cas particulier avec 3 disques situés sur la tour A et où le but est de placer les disques sur la tour C (voir la figure 3.6).

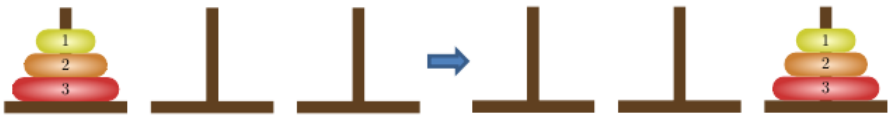


Figure 3.6 – Problème des Tours de Hanoi à 3 disques (Espinasse, 2004).

— Proposez une résolution par décomposition de ce problème en sous-problèmes jusqu'à arriver à des sous-problèmes "triviaux".

10. Avant mutation
11. Après mutation

Solution (inspirée de (Hadi, s. d.))
L'arbre ET qui résout par décomposition le problème des Tours de Hanoi à 3 disques est illustré dans la figure 3.7.

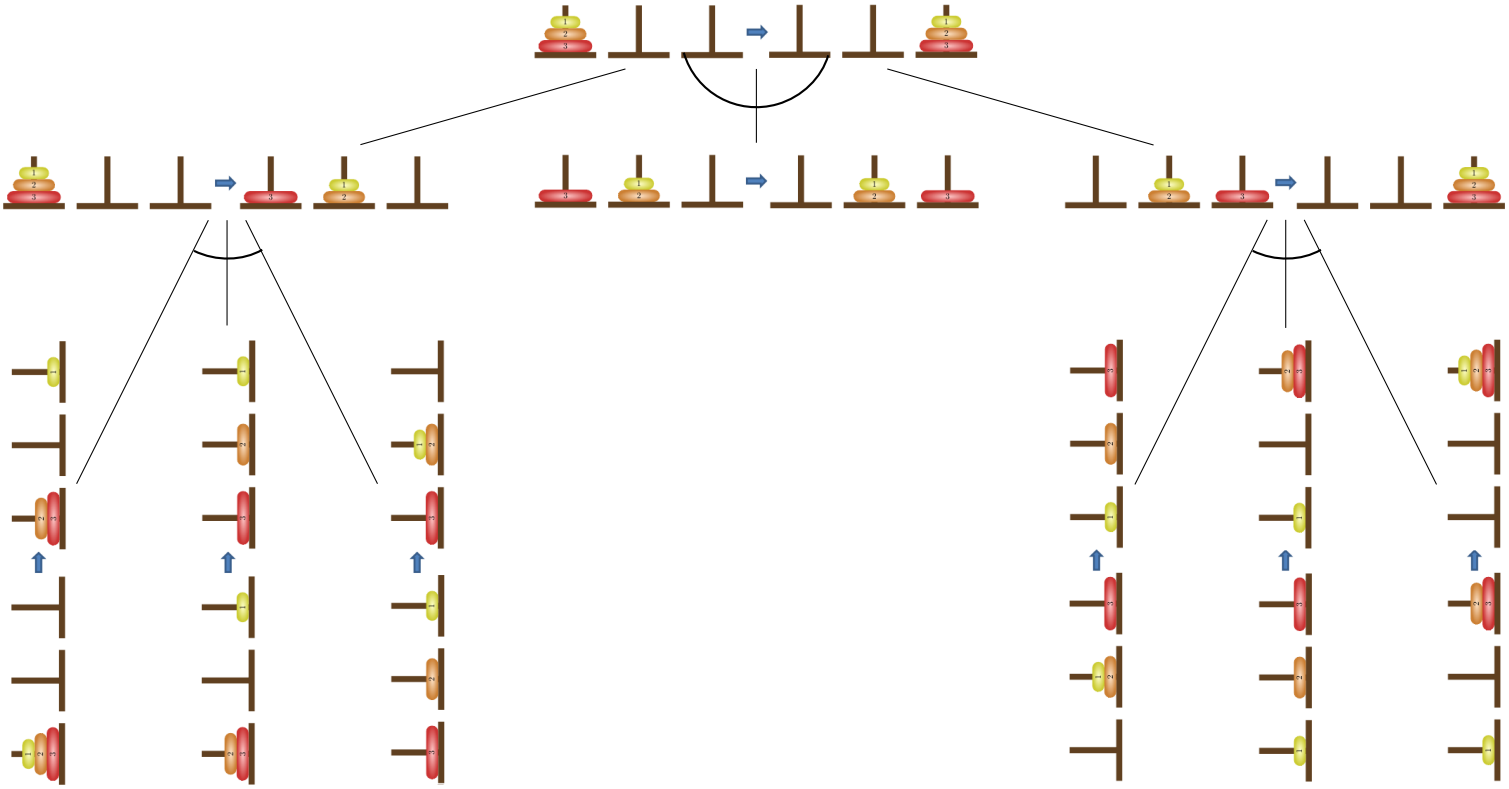
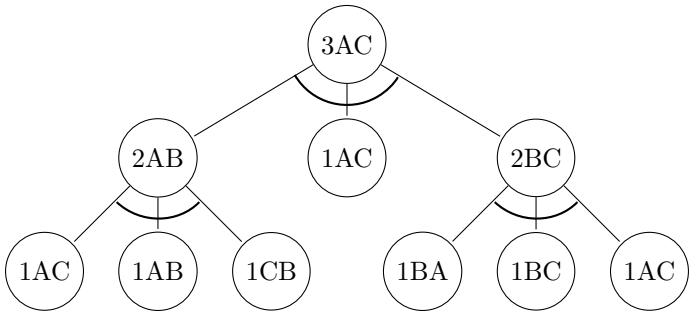


Figure 3.7 – Arbre ET montrant la solution de réduction de problème au problème des Tours de Hanoi à 3 disques.

Cet arbre peut être simpifié comme montré dans la figure suivante :



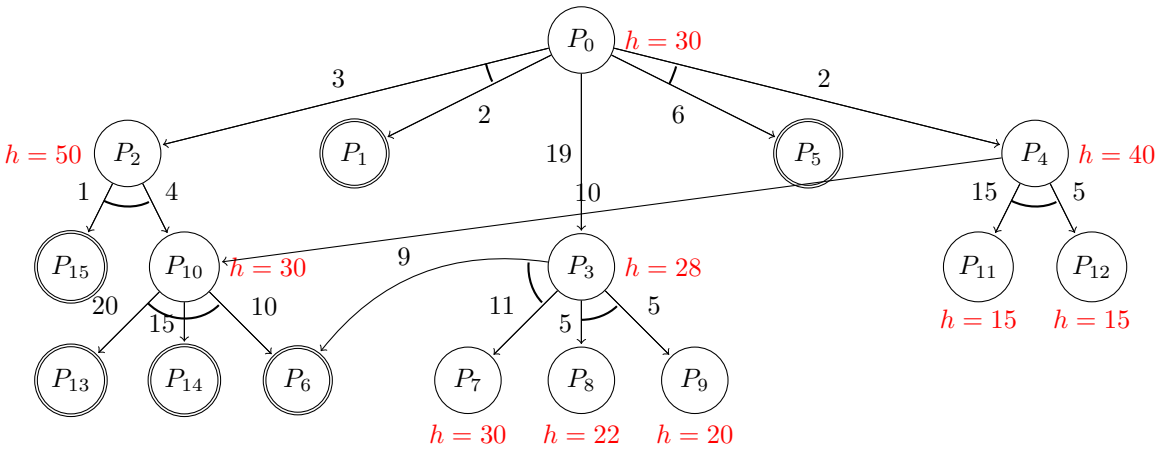
Le noeud racine, étiqueté "3AC" représente le problème d'origine de "transférer les 3 disques de la tour A à la tour C". Le problème peut être décomposé en trois sous-problème : 2AB, 1AC, 2BC. Pour acomplir le problème, les 3 sous-problèmes doivent être accomplis.

- Le sous-problème 2 est trivial, on sait le résoudre.
- Les sous-problèmes 1 et 3 ne le sont pas. Mais ils sont du même type que le problème de base, c-à-d, déplacer n disques d'une tour de départ à une tour d'arrivée. Alors on recommence, on décompose.
- En découpant le problème "2AB", on obtient les sous-problèmes "1AC", "1AB" et "1CB" qui sont tous triviaux.
- En découpant le problème "2BC", on obtient les sous-problèmes "1BA", "1BC" et "1AC" qui sont aussi tous triviaux.

Algorithme de recherche AO*

Exercice 13

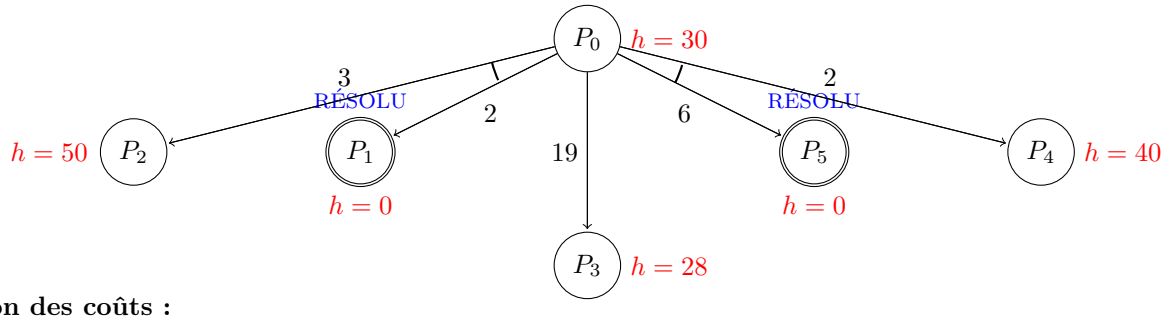
Soit le graphe ET-OU ci-dessous qui représente les différentes façons dont le problème initail (P_0) peut être décomposé en un ensemble de problèmes plus petits. Le coût estimé de résolution de chaque problème (h) et le coût de la connection entre problèmes sont indiqués sur le graphe. Les problèmes triviaux (qui correspondent aux noeuds terminaux dans le graphe) sont $\{P_1, P_5, P_6, P_{13}, P_{14}, P_{15}\}$, la fonction heuristique h vaut 0 pour ces derniers.



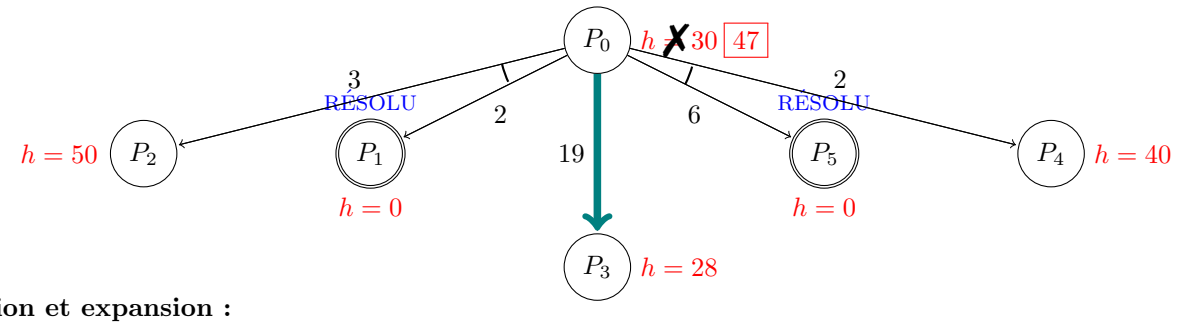
— Déterminez le chemin de moindre coût pour atteindre les noeuds terminaux (sous-graphe construit) et son coût en utilisant AO*.

Solution (inspirée de la vidéo (*AO* algorithm*, 2020))
Dans ce qui suit, on déroule l'algorithme AO* sur le graphe ET-OU ci-dessus pour trouver le chemin de solution (sous-graphe) à coût minimum à partir d'un problème initial P_0

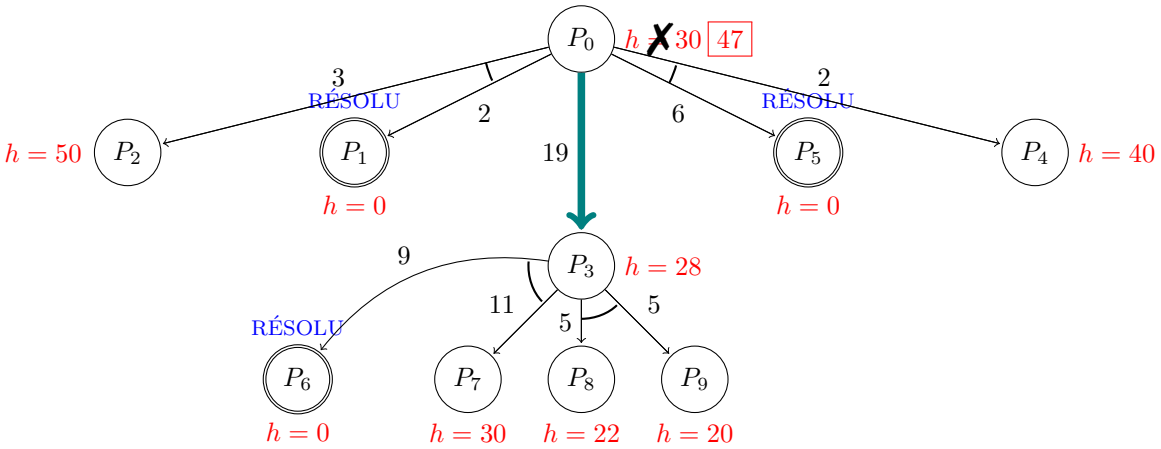
Sélection et expansion :



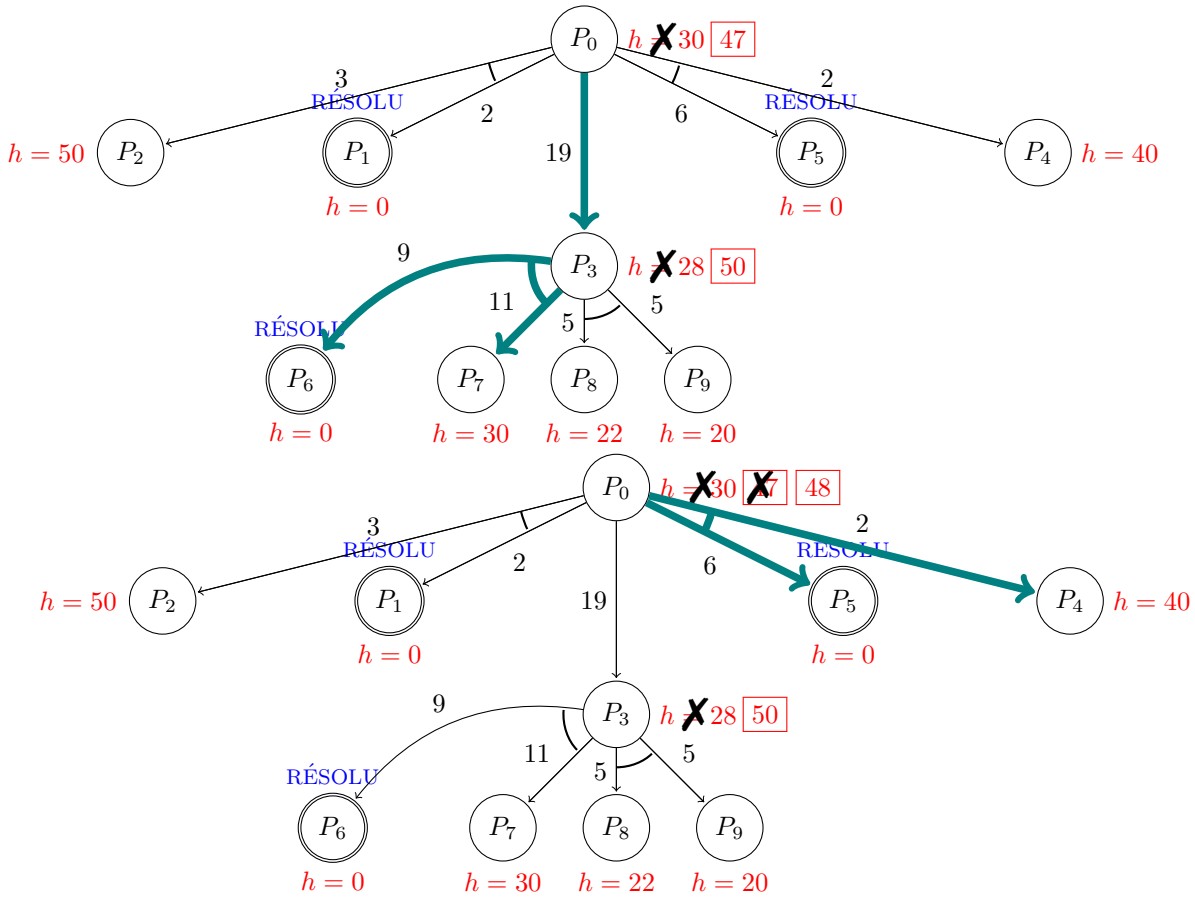
Révision des coûts :



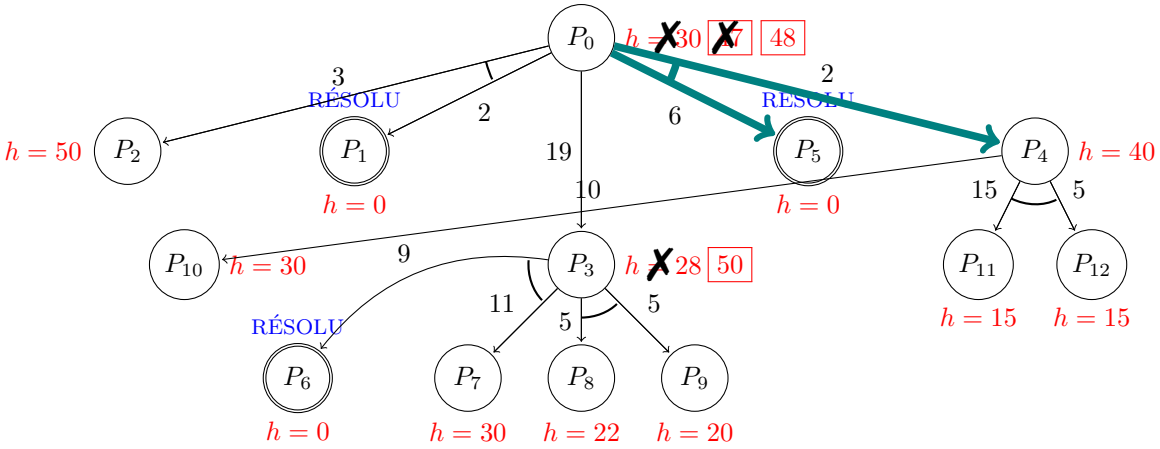
Sélection et expansion :



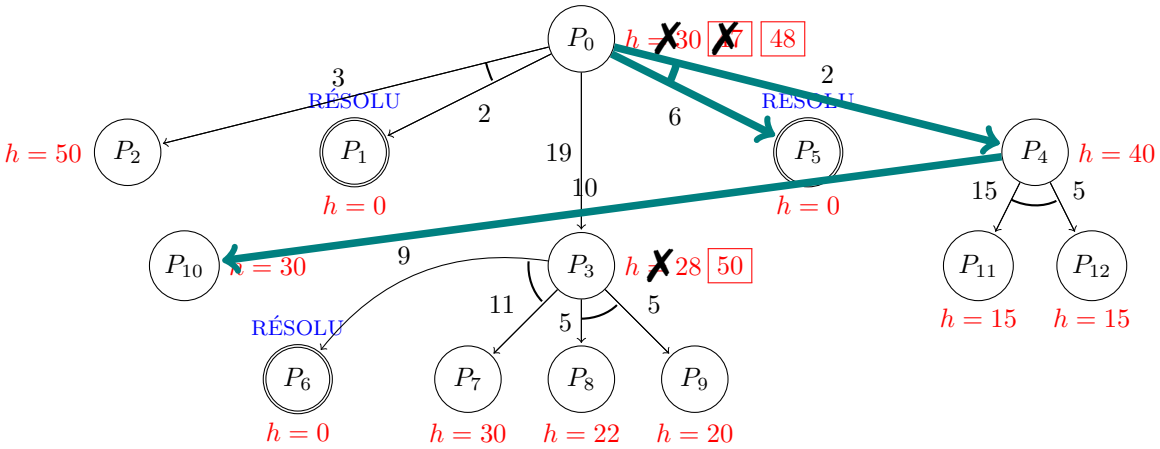
Révision des coûts :



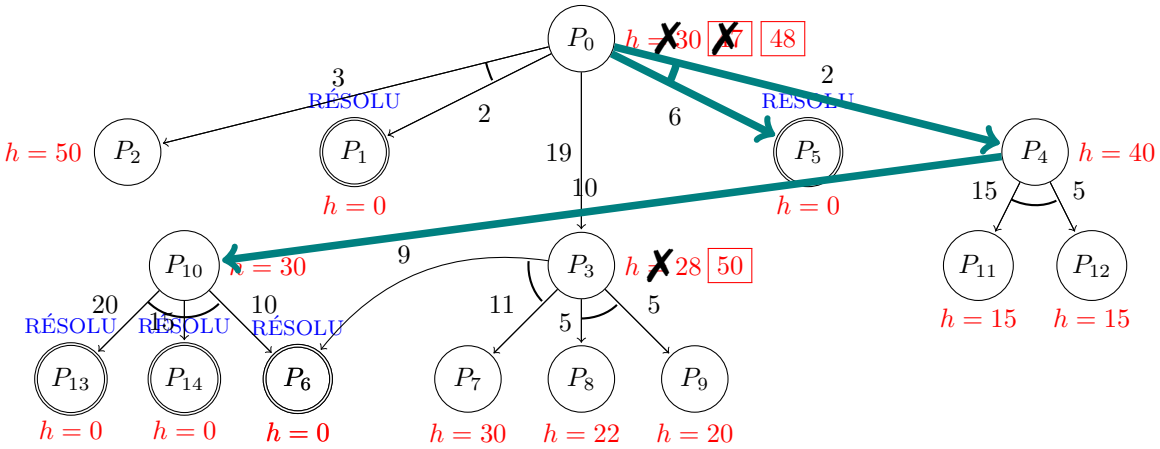
Sélection et expansion :



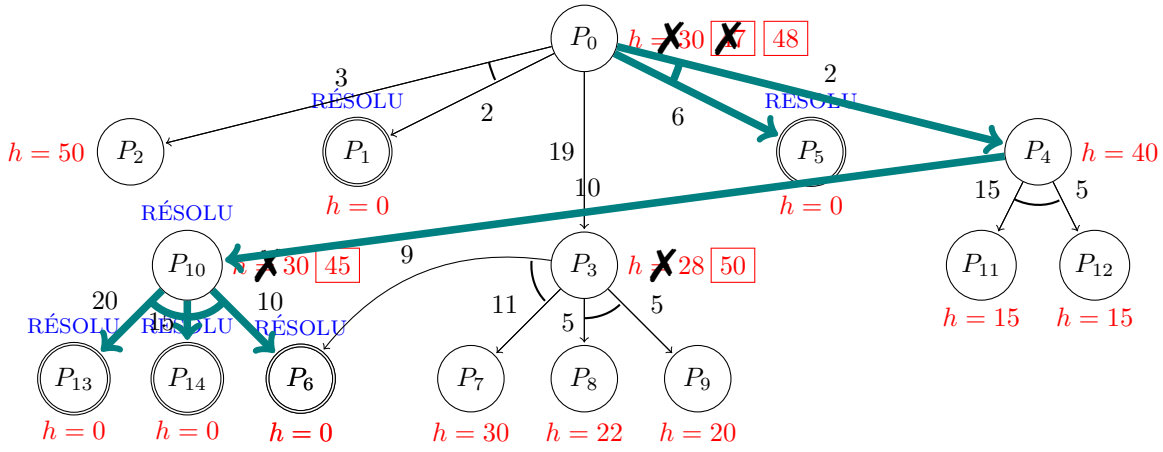
Révision des coûts :

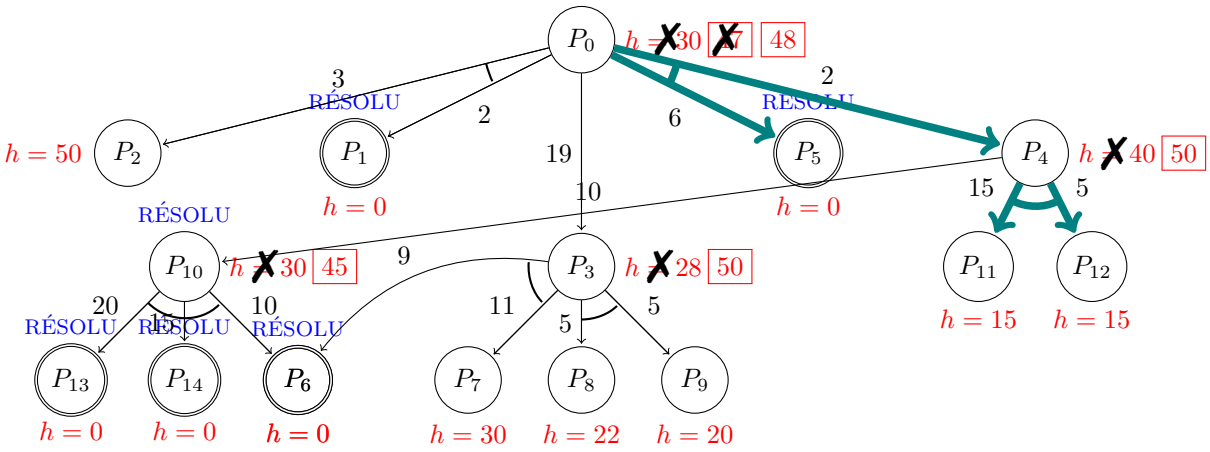


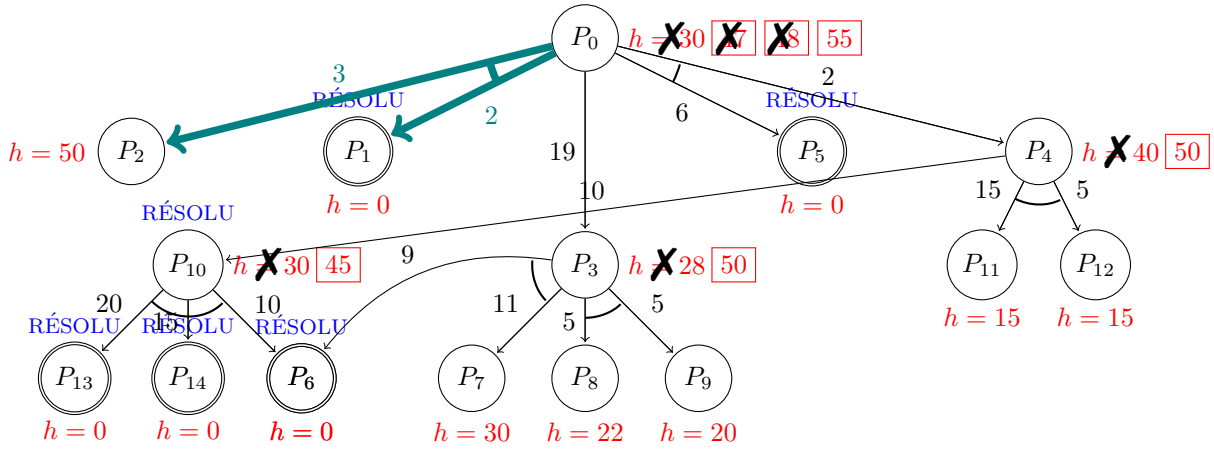
Sélection et expansion :



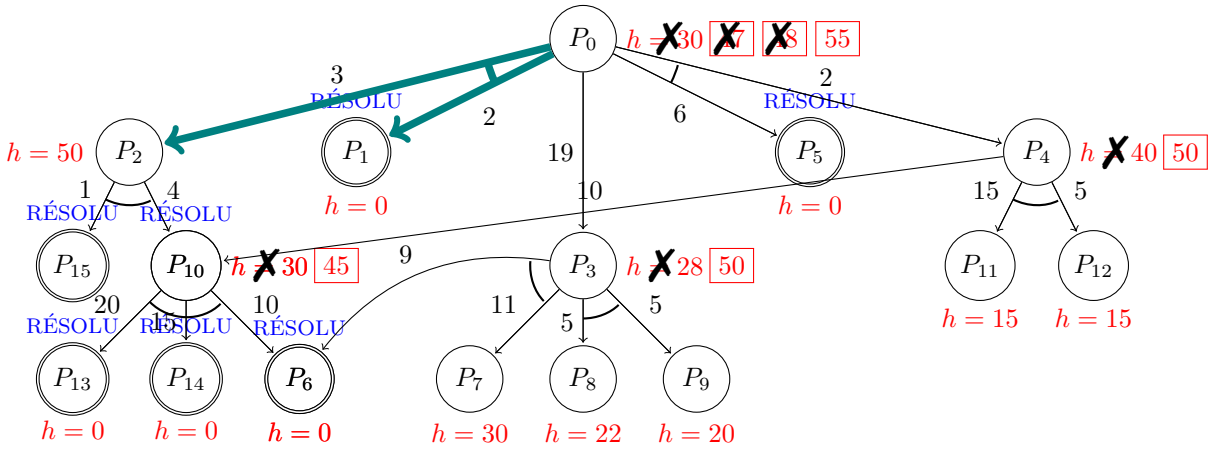
Révision des coûts :



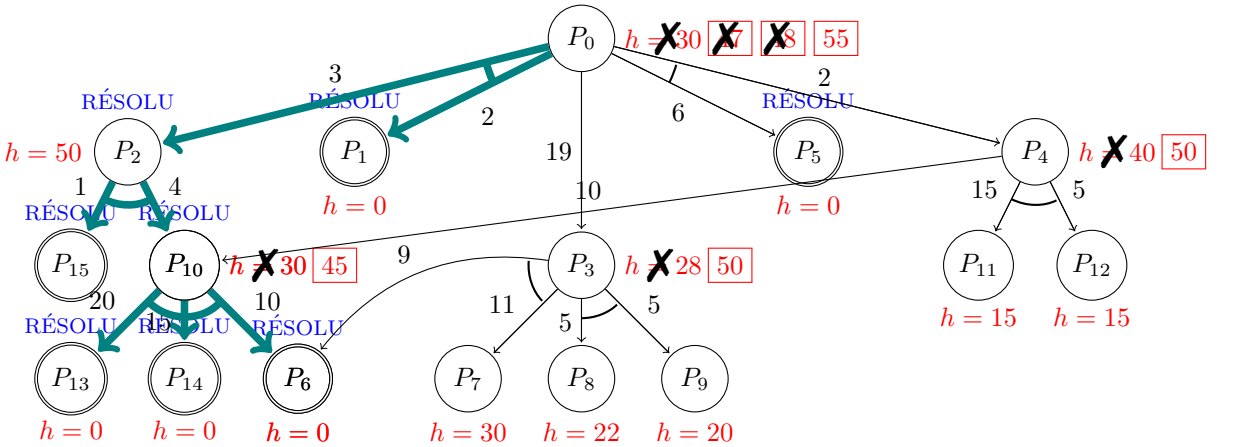


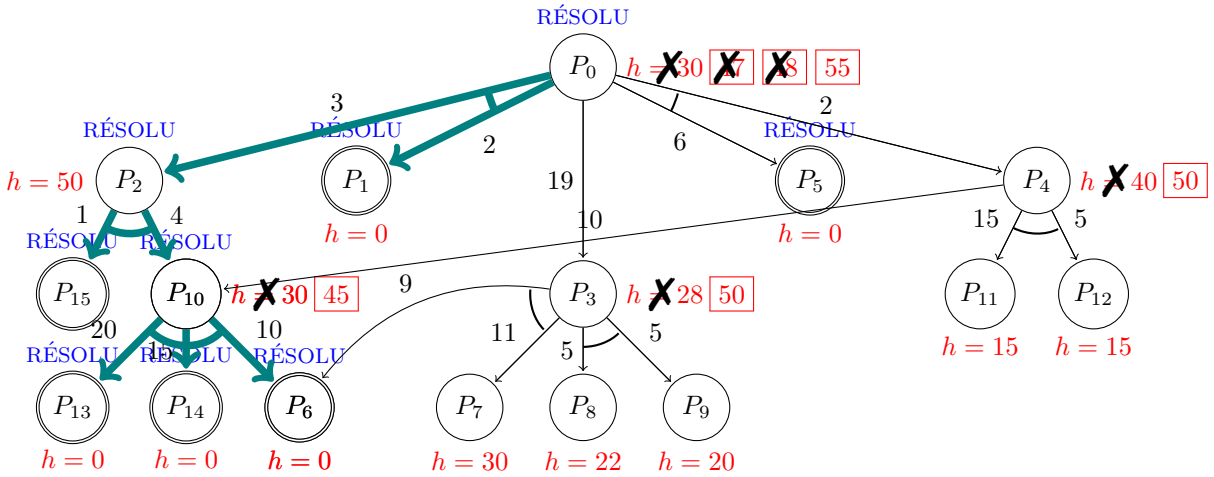


Sélection et expansion :



Révision des coûts :





Puisque le noeud initial est étiqueté **RÉSOLU**, alors l'agorithme se termine, et la meilleur façon de résoudre P_0 est en suivant le sous-arbre marqué. Le coût de cette résolution est de **55**.

3.5 Conclusion

Ce chapitre a présenté un aperçu sur l'apport de l'IA pour la résolution des problèmes.

Conclusion générale

Au début de l'informatique, les problèmes abordés étaient de taille relativement réduite, mais rapidement, on s'est rendu compte que les ordinateurs avaient des limites quant à leur capacité de calcul en fonction du temps, qui ont été révélées lorsque la taille des données a augmenté.

La théorie de la complexité étudie deux aspects:

- la complexité des algorithmes : comment évaluer en fonction de la taille des données le nombre des opérations des algorithmes.
- la complexité des problèmes et leur classification.

Il y a deux grandes familles de résolution de problèmes en intelligence artificielle selon la manière de construire la solution à partir d'un état de départ :

1. Recherche de solutions dans l'espace d'états :
   On commence toujours dans l'état initial et puis on exécute les étapes suivantes en boucle jusqu'à terminaison :
   - s'il n'y a plus d'états à traiter, renvoyer echec.
   - sinon, choisir un des états à traiter.
     - si l'état est un état but, renvoyer la solution correspondante.
     - sinon, supprimer cet état de l'ensemble des états à traiter, et le remplacer par ses états successeurs
2. Résolution par décomposition de problèmes :
   - On décompose le problème en sous-problèmes jusqu'à arriver à des sous-problèmes "triviaux".
   - On résout ces problèmes par la méthode précédente.
   - On remonte l'arbre de décomposition jusqu'à obtenir la solution au problème complet.

Pour aller plus loin, certains points restent à explorer :

- Les problèmes de satisfaction de contraintes ou CSP (Constraint Satisfaction Problems) qui sont des problèmes mathématiques où l'on cherche des états ou des objets satisfaisant un certain nombre de contraintes ou de critères. Pour avoir du détail sur cette partie de problèmes d'intelligence artificielle, le lecteur peut se référer au livre suivant (Alliot, Schiex, Brisset, & Garcia, 1994). Le lien suivant `https://aimacode.github.io/aima-exercises/csp-exercises/` donne des exercices sur des problèmes de satisfaction de contraintes.
- Une autre branche importante de l'intelligence artificielle est les jeux. L'ouvrage (Bienvenu, 2006) donne des explications sur comment la recherche peut être utilisée dans une classe de jeux en introduisant des algorithmes qui permettent d'obtenir les stratégies optimales pour cette dernière. Des exercices pratiques sur la recherche dans les jeux sont disponibles dans le document suivant (Ortiz, 2005/2006a), vous pouvez retrouver les solutions dans (Ortiz, 2005/2006b).

Références

Adam, E. (2008). Intelligence Artificielle - Résolution de Problèmes. Consulté sur http://emmanuel.adam.free.fr/site/IMG/pdf/resolution.pdf

Alliot, J.-M., Schiex, T., Brisset, P., & Garcia, F. (1994). *Intelligence artificielle et informatique théorique*. Cépaduès-éd.

Ao algorithm.* (2020). Vidéo en ligne. Consulté sur https://www.youtube.com/watch?v=bA2LJ2MyxTY

Arora, S., & Barak, B. (2009). *Computational complexity: a modern approach*. Cambridge University Press.

Atallah, M. J., & Blanton, M. (2009). *Algorithms and theory of computation handbook, volume 2: special topics and techniques*. CRC press.

Baase, S. (2009). *Computer algorithms: introduction to design and analysis*. Pearson Education India.

Bari, A. (2018). *Masters theorem in algorithms for dividing function.* Vidéo en ligne. Consulté sur https://www.youtube.com/watch?v=OynWkEj0S-s

Belaïd, A. (s. d.-a). Algorithmes de recherche aveugle. *Université de Nancy 2*. Consulté sur http://jul.andre.free.fr/Cavaliers/cours2_RechAveugle.pdf

Belaïd, A. (s. d.-b). Algorithmes de recherche heuristique. *Université de Nancy 2*. Consulté sur http://jul.andre.free.fr/Cavaliers/cours3_RechHeuristique.pdf

Bienvenu, M. (2006). Introduction to artificial intelligence. *Université de Provence, Marseille, France*. Consulté sur https://www.labri.fr/perso/meghyn/teaching.html

Bourahla, M. (2020). Chapitre 2 : L'IA et la résolution des problèmes. *Université de M'Sila*. Consulté sur https://elearning.univ-msila.dz/moodle/pluginfile.php/371253/mod_resource/content/1/_IAPA_papier_chapitre_2.pdf

Boyer, M. (s. d.). RELATIONS DE RÉCURRENCE. Consulté sur http://www.iro.umontreal.ca/~dift1063/cours16-17.pdf

Castaneda, H., Chaput, R., Guin, N., & Lefevre, M. (2021/2022). TD1 – Résolution de problèmes à l'aide de graphes d'états.

Cohen, J. (s. d.). GRAPHES ET COMPLEXITE. *LRI-CNRS*. Consulté sur https://www.lri.fr/~jcohen/documents/enseignement/Cours-glouton.pdf

Cohen, J. (2018). Exercices corrigés sur problèmes np-complets. Consulté sur https://www.lri.fr/~jcohen/documents/enseignement/exercices-NP.pdf

Complexité d'un algorithme. (2021/2022). *Lycée Michelet - classes de PCSI*. Consulté sur https://cahier-de-prepa.fr/psi-michelet/download?id=239

Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the third annual acm symposium on theory of computing* (pp. 151–158).

Cori, H. G. K. C., R., & Steyaert, J.-M. (2010). Conception et analyse d'algorithmes. *Cours de l'Ecole Polytechnique*.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. MIT press.

Doty, D. (s. d.). Who is the hardest. one of all? np-completeness. Consulté sur https://canvas.ucdavis.edu/courses/299448/files/5227263/download?wrap=1

Espinasse, B. (2004). Résolution de problemes et Intelligence Artificielle. *Université d'Aix-Marseille*.

Fumera, G. (2019/2020a). Artificial Intelligence - MSc Programs in: Computer Engineering, Cybersecurity and Artificial Intelligence Electronic Engineering Academic. *Department of Electrical and Electronic Engineering, University of Cagliari*.

Fumera, G. (2019/2020b). Exercises on search algorithms. *University of Cagliari*. Consulté sur https://unica.it/unica/protected/398005/0/def/ref/MAT238987/

Garey, M. R., & Johnson, D. S. (1979). Computers and intractability. *W. H. Freeman and Co*.

Gaudel, M.-C., Soria, M., & Froidevaux, C. (1987). *Types de données et algorithmes: Recherche, tri, algorithmes sur les graphes*. INRIA.

Geurts, P. (2013/2014). Introduction à la théorie de l'informatique. *Université de Liège, Institut Montefiore*. Consulté sur https://people.montefiore.uliege.be/geurts/Cours/iti/2013/cours-complet-2013-2014.pdf

Goldreich, O. (2008). Computational complexity: a conceptual perspective. *ACM Sigact News*, *39*(3), 35–39.

Goldreich, O. (2010). *P, np, and np-completeness: The basics of computational complexity*. Cambridge University Press.

Goodrich, M. (2004). Solving Recurrences.

Greef, A., & Reinecke, R. (1988). Problem solving using artificial intelligence techniques. *ORiON*, *4*(1).

Habermehl, P. (2005/2006). Algorithmes de recherche informés. Consulté sur https://www.irif.fr/~haberm/cours/ia05/cours3.pdf

Hadi, A. S. (s. d.). Problem Reduction. Consulté sur https://www.uobabylon.edu.iq/eprints/publication_12_357_213.pdf

Hartmanis, J. (1982). Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, *24*(1), 90.

Hazarika, S. (s. d.). Searching and-or graphs. *Department of Mechanical Engineering, Indian Institute of Technology - Guwahati*.

Hermawanto, D. (2013). Genetic algorithm for solving simple mathematical equality problem. *arXiv preprint arXiv:1308.4675*. Consulté sur https://arxiv.org/ftp/arxiv/papers/1308/1308.4675.pdf

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations* (pp. 85–103). Springer.

Knuth, D. E. (1997). *The art of computer programming* (Vol. 3). Pearson Education.

Lacomme, P. P., Prins, C., & Sevaux, M. (2003). *Algorithmes de Graphes*. Eyrolles. Consulté sur https://hal.archives-ouvertes.fr/hal-00009111 (ISBN 2-212-11385-4)

Lebbah, Y. (2021). Polycopié : Algorithmique avancée et complexité. *Département Informatique, Faculté des Sciences Exactes et Appliquées, Université d'Oran 1*. Consulté sur https://sites.google.com/site/ylebbah/teach#h.jsfrxtszqi7d

Lefevre, M. (2021). CM2 : Résolution de problèmes. *Université Claude Bernard Lyon 1*. Consulté sur https://perso.liris.cnrs.fr/marie.lefevre/ens/BIA/BIA2022-CM2-RP.pdf

Luger, G. F. (2005). *Artificial intelligence: structures and strategies for complex problem solving*. Pearson education.

Muscholl, A. (2021). Complexité et calculabilité. *Département ST, Université Bordeaux*. Consulté sur https://www.labri.fr/perso/anca/MC/poly.pdf

Ortiz, L. E. (2005/2006a). Practice problems on search in games. *MIT EECS*. Consulté sur http://www-personal.umd.umich.edu/~leortiz/teaching/6.034f/Fall06/games/games_probs.pdf

Ortiz, L. E. (2005/2006b). Solutions to practice problems on search in games. *MIT EECS*. Consulté sur http://www-personal.umd.umich.edu/~leortiz/teaching/6.034f/Fall06/games/games_probs_sol.pdf

Oussous, N.-E., & Wegrzynowski, E. (2008). Complexité des algorithmes (1).

Papadimitriou, C. H. (1994). *Computational complexity*. Addison-Wesley.

Perifel, S. (s. d.). COMPLEXITÉ ALGORITHMIQUE. Consulté sur https://www.irif.fr/~sperifel/complexite.pdf

Russell, S., & Norvig, P. (2003). *Instructor's manual: Exercise solutions for artificial intelligence a modern approach second edition.* Alan R. Apt.

Russell, S., Russell, S., Norvig, P., & Davis, E. (2010). *Artificial intelligence: A modern approach*. Prentice Hall. Consulté sur https://books.google.dz/books?id=8jZBksh-bUMC

Réductions, problèmes p, np-complétude. (Mars 2017). *University of Lyon 1*.

Schwarzentruber, F. (s. d.). Algorithmes sur les nombres.

Vivien, F. (2002). *Algorithmique avancée*. IUP 2. Consulté sur http://perso.ens-lyon.fr/frederic.vivien/Enseignement/Algo-2001-2002/Cours.pdf

Wegener, I. (2005). *Complexity theory: exploring the limits of efficient algorithms*. Springer Science & Business Media.

Zampieri, K., Riviere, S., & Amerein-Soltner, B. (2018). Complexité des algorithmes - Algorithmique. Consulté sur https://ressources.unisciel.fr/algoprog/s34plexite/emodules/cx00macours1/co/cx00macours1.html