

Introduction à l'informatique

pour les mathématiques, la physique et les sciences
computationnelles

Yann Thorimbert



**UNIVERSITÉ
DE GENÈVE**

CENTRE UNIVERSITAIRE
D'INFORMATIQUE

Partie 2 | Chapitre 5

Algorithmes sur graphes

Yann Thorimbert



UNIVERSITÉ
DE GENÈVE

CENTRE UNIVERSITAIRE
D'INFORMATIQUE



Chapitres du cours (seconde partie du cours)

- 0. Introduction à la complexité algorithmique
- 1. Algorithmes de tri naïfs
- 2. Structures de données : tableaux, listes et dictionnaires
- 3. Tri fusion et récursivité
- 4. Algorithmes de recherche au sein d'une séquence
- 5. Algorithmes sur graphes ←**

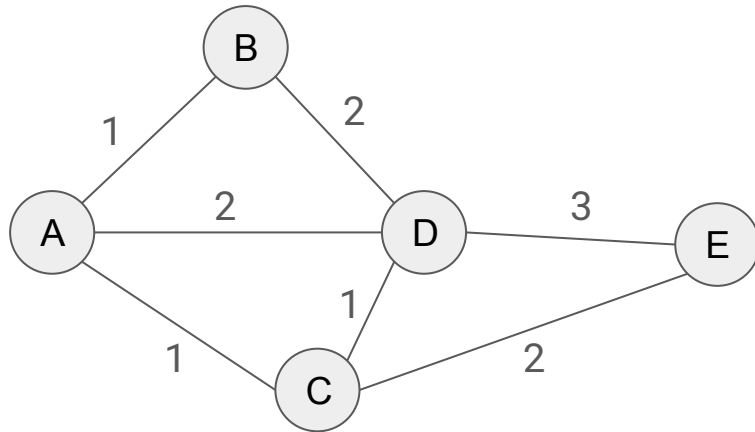
Contenu de ce chapitre

- Arbres :
 - Utilisation de structures en arbre pour représenter des séquences triées et y chercher des éléments.
 - Utilisation d'un arbre pour l'implémentation d'un dictionnaire.
 - Utilisation d'un arbre pour la compression de données.
 - Utilisation d'un arbre pour la recherche d'une solution à un problème combinatoire.
- Graphes :
 - Algorithmes de plus court chemin.

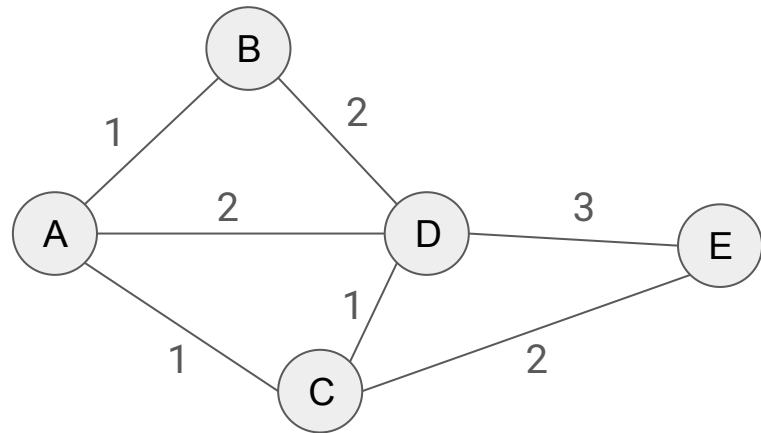
Graphes | Exemple de problème

Des villes sont reliées par des routes. Le temps mis pour joindre deux villes *via* une route dépend de la route.

Partant de A, quel est le plus court chemin pour aller à E ?



Graphes | Terminologie

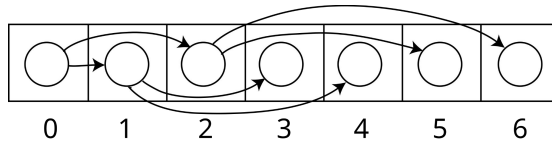




Arbres | Terminologie

Arbres binaires | Implémentation

- On peut définir un type abstrait correspondant à un arbre binaire. Ce type abstrait retranscrit l'objet mathématique discuté plus haut. Il peut par exemple être implémenté *via* une variante de liste chaînée, ou bien *via* un tableau de données.
- La pertinence d'une implémentation plutôt que l'autre dépend des garanties que l'on possède (par exemple : l'arbre est-il de taille fixe ? est-il proche d'être parfait ?)
- Exemple d'implémentation via un tableau, pertinente pour les arbres assez complets :

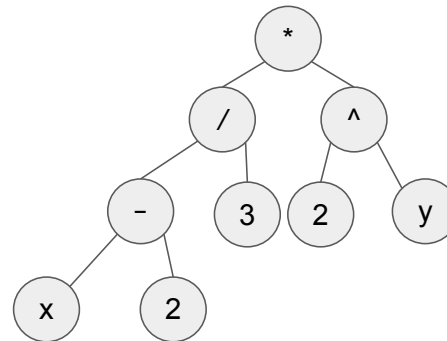


- Ici, nous travaillerons uniquement avec le type abstrait, sans nous soucier de l'implémentation choisie.

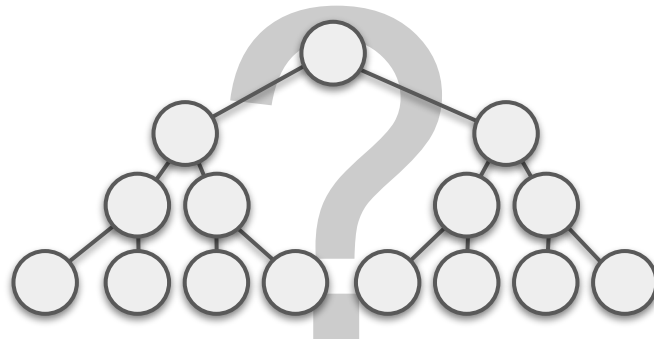
Commentaire

- De très nombreux systèmes intéressants peuvent être représentés par des arbres.
- Dans ce cours, nous sommes loin de pouvoir tous les aborder.
- Exemple : une expression algébrique s'exprime comme un arbre :

$$(x-2)/3 * 2^y$$



Arbre binaire de recherche

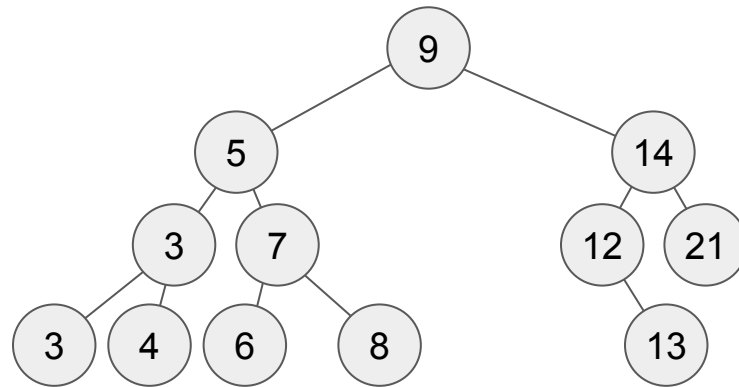


Arbre binaire de recherche

- Il est possible d'aborder le problème de la recherche au sein d'une séquence avec un tout autre point de vue.
- Rappel : nous avons traité de l'algorithme de recherche dichotomique, qui permet de trouver l'emplacement d'une valeur donnée au sein d'une séquence déjà triée.

Arbre binaire de recherche | Exemples d'insertions

$S = [9, 14, 5, 3, 7, 12, 3, 6, 13, 21, 8, 4]$



Cf. illustrations pas à pas

Arbre binaire de recherche | Commentaires finaux

- Une insertion est également $O(n)$ dans le pire des cas et $O(\log(n))$ en moyenne.
- La construction de l'arbre tout entier est $O(n^2)$ dans le pire des cas ; on peut la considérer comme un algorithme de tri.
- Certains algorithmes de tri $O(n \log(n))$ comme *heapsort* (non traité dans ce cours) sont spécialement conçus pour tirer profit de la structure d'arbre, avec des garanties en plus sur cette dernière.
- D'autres structures courantes, comme les files de priorité (*priority queue*), sont souvent implémentées grâce à un arbre également.



Arbre binaire de recherche | Implémentation

Comme toujours, toutes sortes d'implémentations sont envisageables.

Dans des langages le permettant, il est commode de définir une structure de donnée Noeud possédant les attributs suivants : `valeur`, `enfant_gauche` et `enfant_droite`. Ces deux derniers attributs pointent sur d'autres noeuds, à la manière d'une **liste chaînée**.

Voici un exemple en Python :

```
class Noeud:

    def __init__(self, valeur):
        self.valeur = valeur
        self.enfant_gauche = None #pas d'enfant initialement
        self.enfant_droite = None #pas d'enfant initialement
```

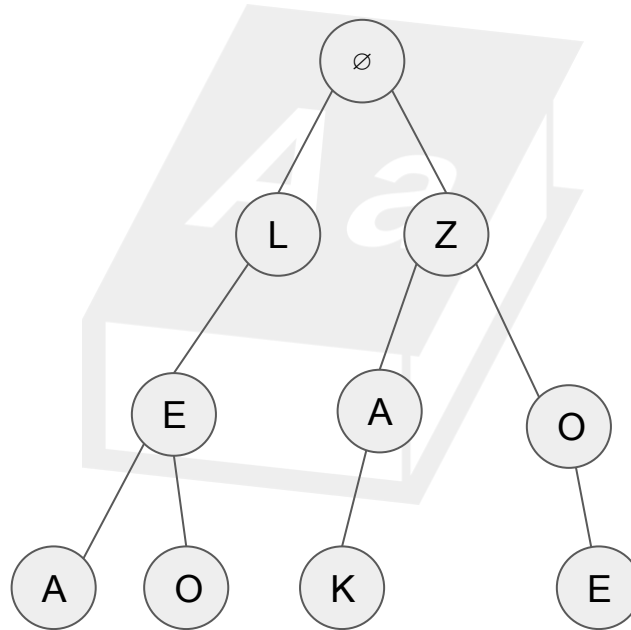


Arbre binaire de recherche | **Code pour la recherche**

Exemple récursif en Python à partir d'un noeud racine :

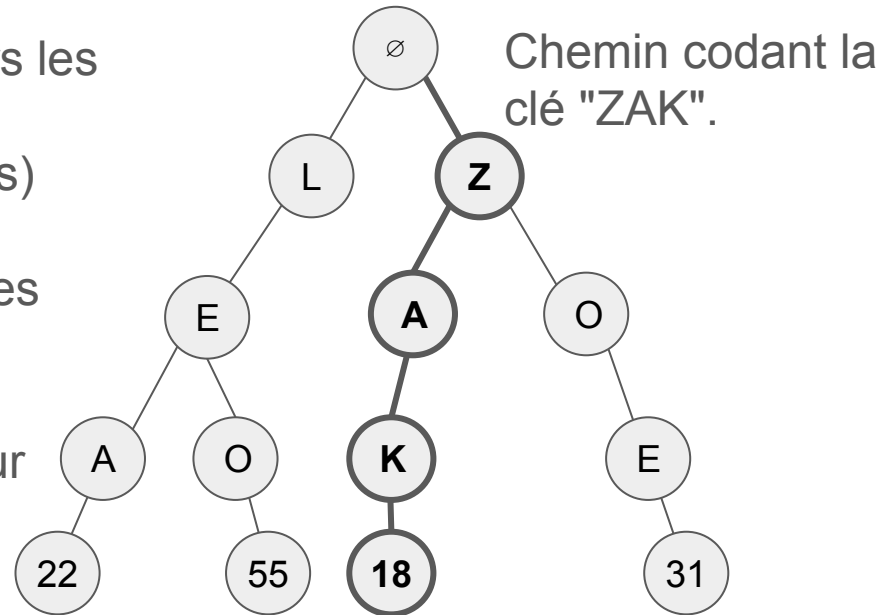
```
def cherche_noeud(racine, valeur):  
    if racine.valeur is None or racine.valeur == valeur:  
        return racine # contiendra None si valeur pas trouvée  
    elif valeur < racine.valeur:  
        return cherche_noeud(racine.enfant_gauche, valeur)  
    else:  
        return cherche_noeud(racine.enfant_droite, valeur)
```

Arbres pour représenter un dictionnaire



Arbres représentant un dictionnaire | **Idée générale**

- Observation : le chemin suivi à travers les branches d'un arbre décrit une combinaison unique (pas de collisions)
- Le **chemin** suivi à travers les branches peut donc décrire une **clé**.
- La **valeur** d'une **feuille** décrit la valeur associé à son chemin.





Arbres représentant un dictionnaire | **Idée générale**

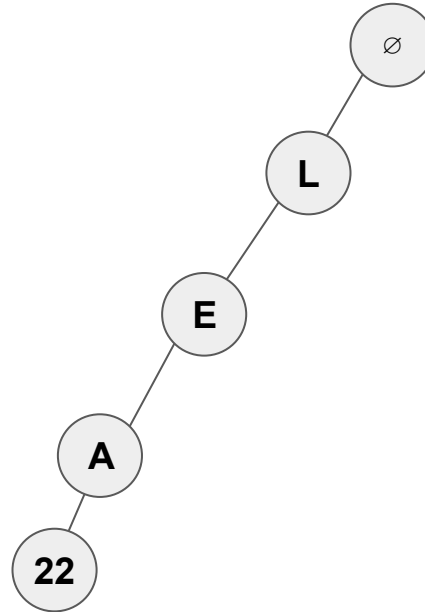
```
age = { } # dictionnaire vide
```



Arbres représentant un dictionnaire | **Idée générale**

```
age = { } # dictionnaire vide
```

```
age["LEA"] = 22
```

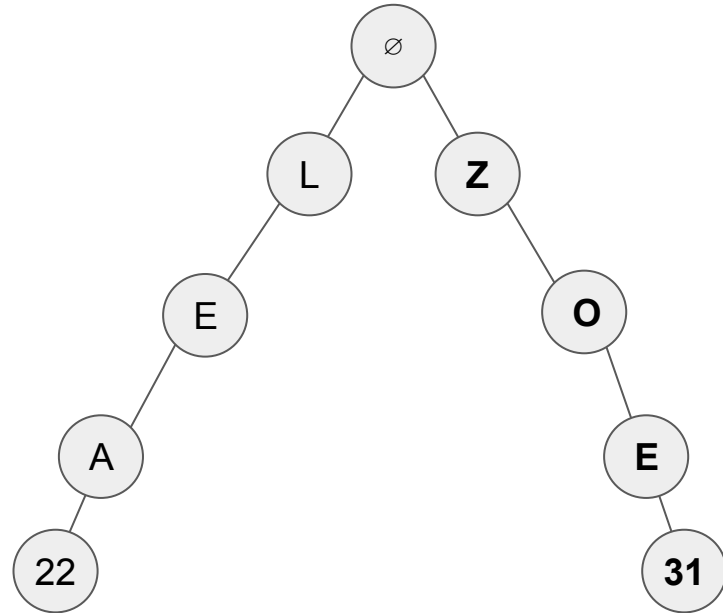


Arbres représentant un dictionnaire | **Idée générale**

```
age = { } # dictionnaire vide
```

```
age["LEA"] = 22
```

```
age["ZOE"] = 31
```



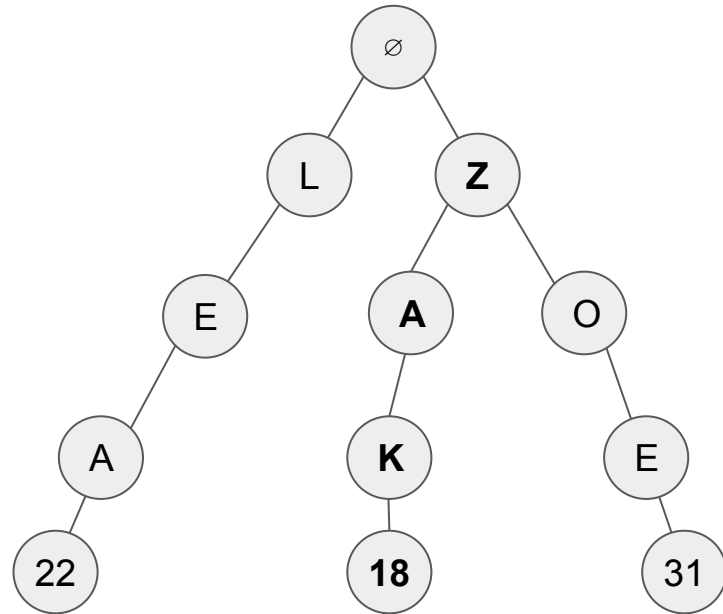
Arbres représentant un dictionnaire | **Idée générale**

`age = { } # dictionnaire vide`

`age["LEA"] = 22`

`age["ZOE"] = 31`

`age["ZAK"] = 18`



Arbres représentant un dictionnaire | Idée générale

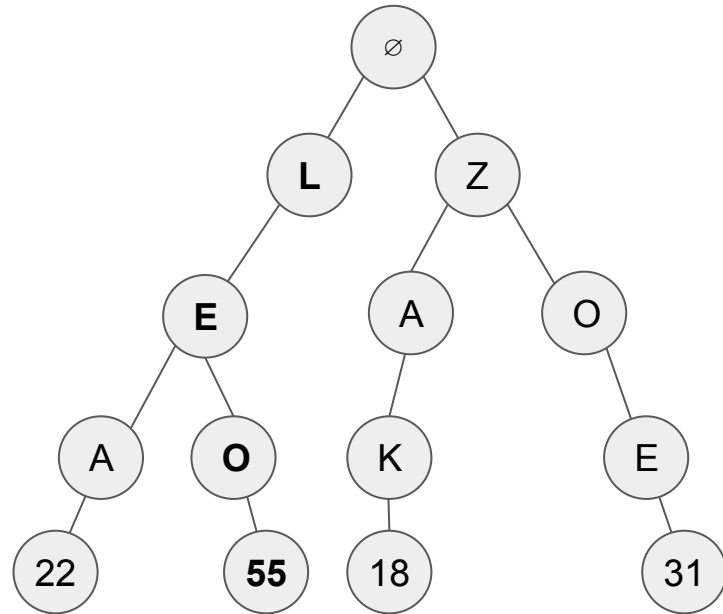
```
age = { } # dictionnaire vide
```

```
age["LEA"] = 22
```

```
age["ZOE"] = 31
```

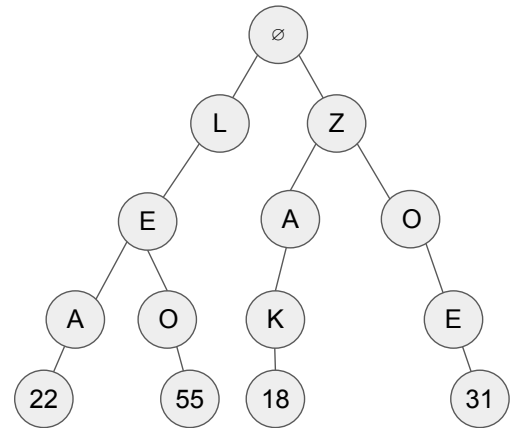
```
age["ZAK"] = 18
```

```
age["LEO"] = 55
```

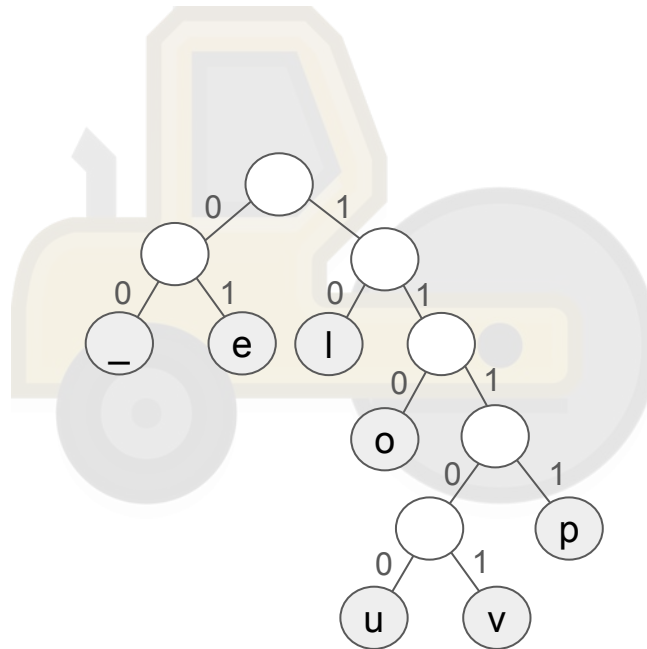


Arbres VS tables de hachage pour la recherche d'élément

- Pire des cas $O(\log(n))$ en utilisant des techniques d'équilibrage : mieux que le pire des cas d'une table de hachage, à savoir $O(n)$.
- Cas moyen en $O(\log(n))$ moins bon que pour une table de hachage, à savoir $O(1)$.
- Permet facilement d'avoir un dictionnaire ordonné, contrairement aux tables de hachage standard.



Arbres et compression d'information



Un autre exemple d'utilisation des arbres

- Dans le chapitre sur le codage des médias, nous avons mentionné les potentielles compressions d'information mises en oeuvre pour économiser de l'espace en mémoire.
- Une méthode de compression intéressante est le codage de Huffman, dont nous présentons ici l'idée appliquée à la compression d'un texte. On peut imaginer étendre l'idée à d'autres types de médias.
- Le résultat du codage de Huffman est un dictionnaire dont les clés sont des nombres binaires. Chaque clé nous dit comment encoder un symbole donné.
- Le codage de Huffman permet une compression **sans perte**. Il est mis en oeuvre (conjointement à d'autres méthodes) au sein de données utilisées au quotidien telles que les images au format png.

Idée de base de la compression de Huffman

- Approche naïve : s'il y a n symboles à coder, alors on peut les faire correspondre à des nombres en binaire codés sur $\log_2(n)$ bits.

Symbole	Codage
A	00
B	01
C	10
D	11

Idée de base de la compression de Huffman

- Approche naïve : s'il y a n symboles à coder, alors on peut les faire correspondre à des nombres en binaire codés sur $\log_2(n)$ bits.
- Cependant, on aimerait coder les caractères rares sur davantage de bits, et les caractères courant sur moins de bits...

Symbole	Codage
A	0
B	1
C	10
D	11

Idée de base de la compression de Huffman

- Approche naïve : s'il y a n symboles à coder, alors on peut les faire correspondre à des nombres en binaires codés sur $\log_2(n)$ bits.
- Cependant, on pourrait coder les caractères rares sur davantage de bits, et les caractères courant sur moins de bits...
- ... à condition de savoir comment délimiter les bits de chaque caractère !
- Comment s'en sortir ? Ici, le codage produit sera un code préfixe, c'est-à-dire qu'**aucun codage ne peut être le début d'un autre**.



Codage de Huffman | **Idée générale**

Codage de Huffman | **Exemple sur un texte**

Texte à compresser : "le_loup_vole_le_poele".

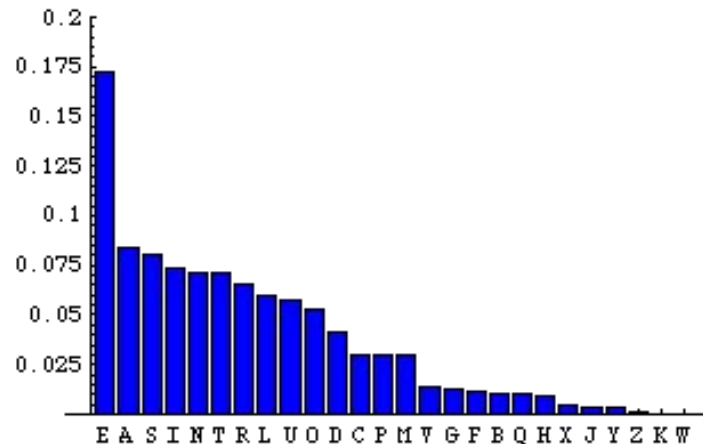
Cf. slides d'illustration
pas à pas.

Caractère	Fréquence
l	5
e	5
-	4
o	3
p	2
u	1
v	1

Attention : cet exemple est volontairement peu efficace en compression afin de voir plusieurs cas de figure en une seule application de l'algorithme.

Commentaire sur la fréquence des caractères

- La plupart du temps, la distribution de l'information n'est pas homogène.
- Fréquence des caractères des langues naturelles (exemple du français) :



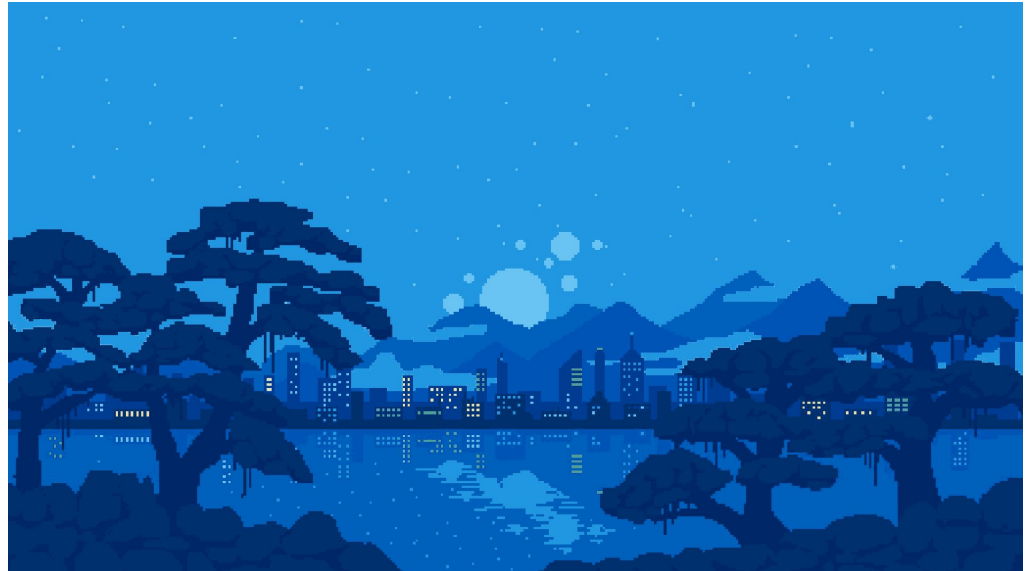
"Dans la dernière leçon, nous avons appris à tracer des rayons lumineux provenant d'une source de lumière comme cette ampoule. Nous avons aussi parlé du modèle corpusculaire de la lumière, dans lequel des grains de lumière appelés photons se déplacent en ligne droite et à vitesse constante. Quel est le lien entre les rayons que nous avons appris à tracer et ces photons ?

Nous allons ici répondre à cette question. Dans le modèle que nous utilisons, les photons sont émis par le filament de l'ampoule dans une direction aléatoire en réalité on ne peut pas voir les photons individuellement par ailleurs il se déplace bien plus vite et sont bien plus nombreux que sur le schéma que nous voyons même si nous l'étions plus rapide et plus nombreux comme ici pour bien les voir nous faisons exprès de les représenter plus grand plus lent et moins nombreux qui ne sont en réalité nous allons également cessé d'être représentés non pour la l'écran maintenant que nous avons dû les photons sont émis par une source de lumière le cas que nous voyons ici correspond à une source ponctuelle de lumière le cas que nous voyons ici correspond lui à une source de lumière non ponctuelle les photons peuvent être émis depuis n'importe quel point de la source"

Codage de Huffman | Exemple sur une image

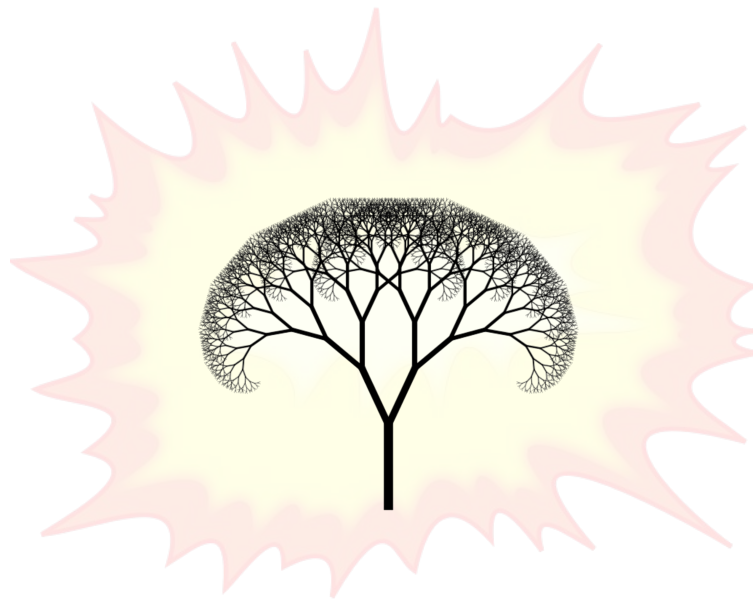
Image à compresser

Couleur	Fréquence [%]
A	50.8
B	19.6
C	11.6
D	9.1
E	5.0
F	2.6
G	1.1
H	0.1
I	0.1



Auteur : kejsirajbek - <https://whvn.cc/xl6v7v>

Arbres et problèmes combinatoires



Problème de la somme de sous-ensembles

- Soit E un ensemble d'entiers.
- Question : existe-t-il un sous-ensemble de E dont la somme des éléments vaut S ?
- Il s'agit d'un **problème de décision** (la réponse est "oui" ou "non").
- Exemples :
 - $E = \{3, 5, 7\}$ et $S = 8$ (la réponse est "oui").
 - $E = \{3, 5, 7\}$ et $S = 13$ (la réponse est "non").

Problème de la somme de sous-ensembles

- Soit E un ensemble d'entiers.
- Question : existe-t-il un sous-ensemble de E dont la somme des éléments vaut S ?
- Il s'agit d'un problème de décision (la réponse est "oui" ou "non").
- Exemples :
 - $E = \{3, 5, 7\}$ et $S = 8$ (la réponse est "oui").
 - $E = \{3, 5, 7\}$ et $S = 13$ (la réponse est "non").
- Nous dirons que ce problème est "**difficile**" car le nombre de sous-ensembles à tester **croît exponentiellement** avec le nombre n d'éléments de E .

Représentation du problème

- Il y a deux façons de se représenter le problème avec un arbre.
- Première méthode : arbre à facteur de branchement variable.
Chaque noeud possède comme enfants les nombres qui n'ont pas été utilisés le long du chemin courant.
- Dessin pour $E = \{2,3,4,5\}$:

Représentation du problème | **Arbre binaire**

- Seconde méthode : on peut numéroter E_i les éléments de E , que l'on considère maintenant comme une séquence. À chaque E_i on peut associer une valeur binaire u_i qui nous dit si le nombre fait partie ou pas du sous-ensemble considéré.
- Par exemple : $E = [2,3,4,5]$ et $u = [0,1,0,1]$ génère la somme $3 + 5 = 8$.
Dessin :

Représentation du problème | **Arbre binaire**

- Dans la méthode binaire, chaque chemin possible représente un sous-ensemble unique de E .
- Cela nous donne le nombre total de sous-ensembles : l'arbre étant de hauteur n , il existe donc $2^{n+1} - 1$ noeuds et donc chemins possibles. Le problème est bien exponentiellement grand en fonction de n .



Recherche en profondeur

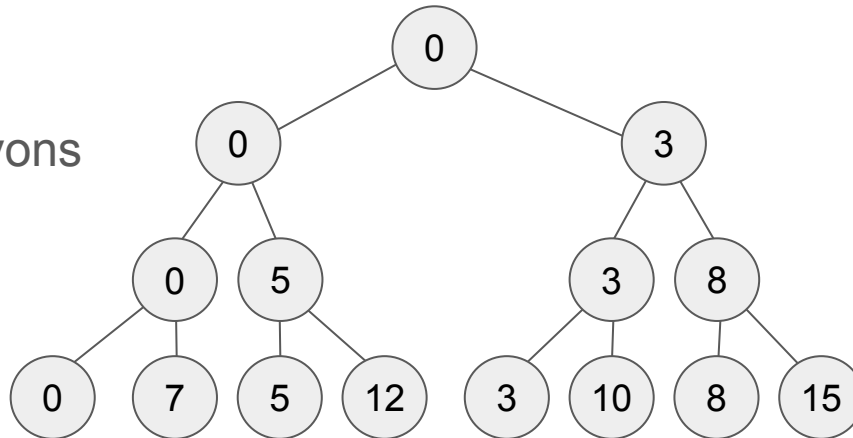
- On peut formuler un algorithme qui cherche en priorité dans la **profondeur** de l'arbre.
- L'algorithme effectue une recherche systématique. On a la garantie qu'il trouvera la bonne solution.
- Nous donnons ici l'idée générale.
- Les diapos détaillées sur la recherche en profondeur donnent une illustration pas à pas.

Exploration en profondeur de l'espace des solutions

Voici l'arbre complet, si on devait le construire.

Dans cet exemple, $E = [3, 5, 7]$

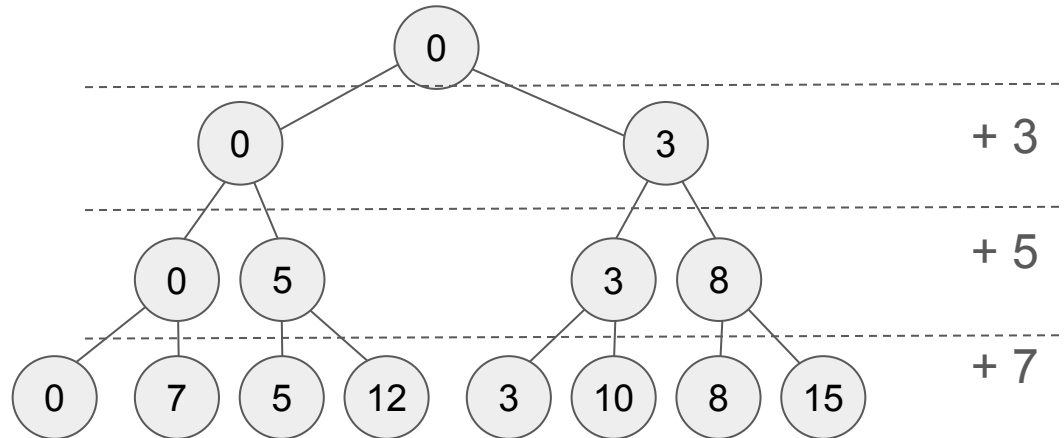
Gardons en tête que, précisément, nous devons construire cet arbre !



Exploration en profondeur de l'espace des solutions

Voici l'arbre complet, si on devait le construire.

Dans cet exemple, $E = [3, 5, 7]$

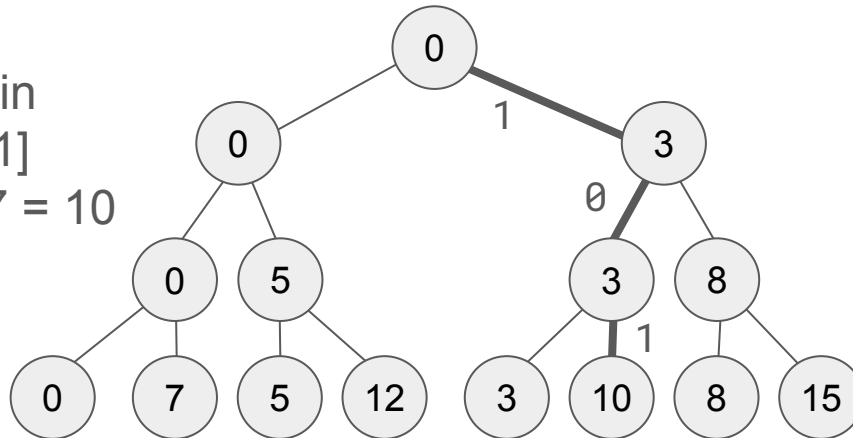


Exploration en profondeur de l'espace des solutions

Voici l'arbre complet, si on devait le construire.

Dans cet exemple, $E = [3, 5, 7]$

Par exemple, ce chemin correspond à $u = [1, 0, 1]$ et donc à $S = 3 + 0 + 7 = 10$



Recherche en profondeur | Commentaires

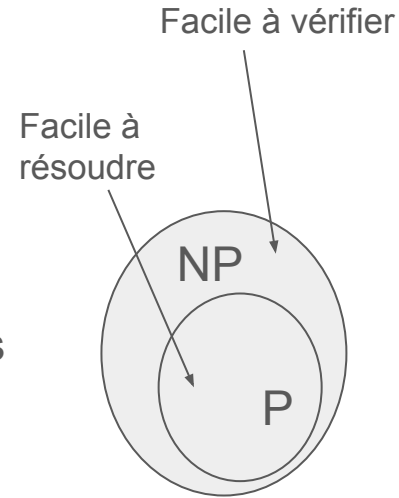
- De nombreux raffinements sont possibles selon les détails du problème. Par exemple, selon les contraintes sur E, on peut prédire à l'avance qu'une branche doit être abandonnée.
- De la même façon, on peut explorer l'ensemble des mots de passe de longueur donnée possibles, des parties d'échecs possibles, etc. Mais, en général, ces ensembles sont bien trop grands pour les ordinateurs même les plus puissants !
- Une recherche **exhaustive** est souvent nommée "approche **force brute**".

Recherche en profondeur | Commentaires

- La méthode qui vient d'être vue constitue une **recherche en profondeur**, ou DFS (*Depth-First Search*), car la priorité est systématiquement donnée aux nouveaux enfants avant l'exploration des noeuds frères.
- Comme la recherche est exhaustive, on a la garantie de trouver une solution s'il en existe une.
- Il existe d'autres méthodes, comme la recherche en largeur ou BFS (*Breadth-First Search*), qui est également exhaustive mais garantit de trouver la solution la plus courte (ici, le sous-ensemble le plus restreint).

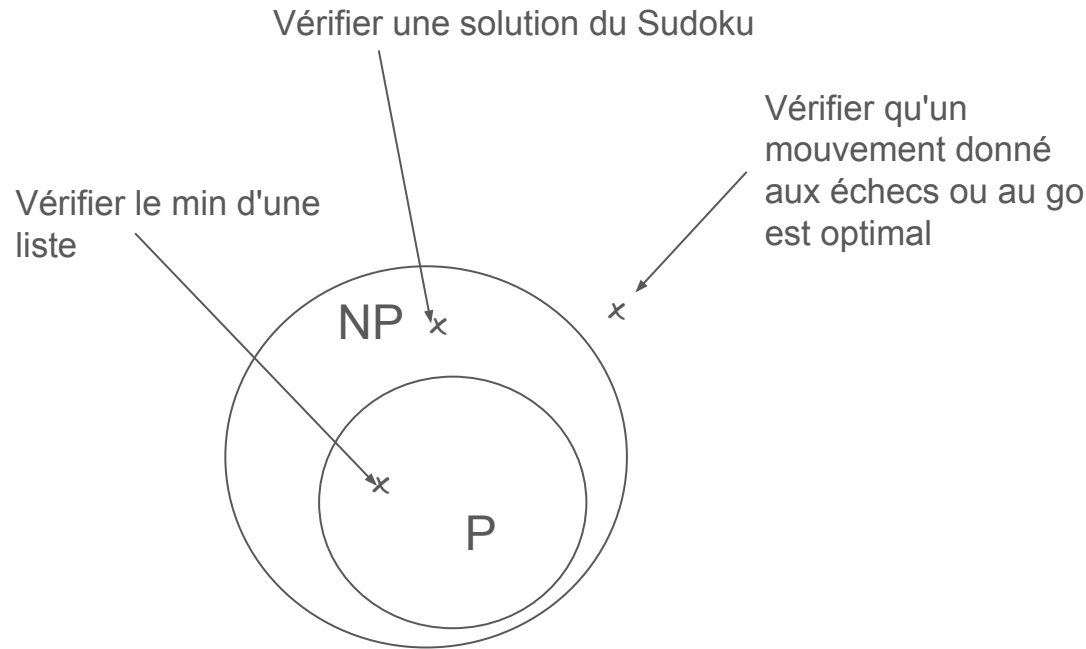
Commentaires sur les problèmes "difficiles" à résoudre

- Solution **difficile à trouver** : $O(2^n)$ solutions à tester.
- Dans ce cas, solution **facile à vérifier** : $O(n)$.
- Les problèmes de décision faciles à vérifier (complexité polynomiale) sont classés comme problèmes "NP".
- Les problèmes de décision à la fois difficiles à résoudre mais faciles à vérifier sont dits "NP-complets".
- Les problèmes faciles à résoudre (P) sont forcément faciles à vérifier (NP), donc P est inclus dans NP.
Mais la réciproque, qui implique $P = NP$, est-elle vraie ?



(Diapositive hors champ)

Commentaires sur les problèmes "difficiles" à résoudre



(Diapositive hors champ)

Commentaires sur les problèmes "difficiles" à résoudre

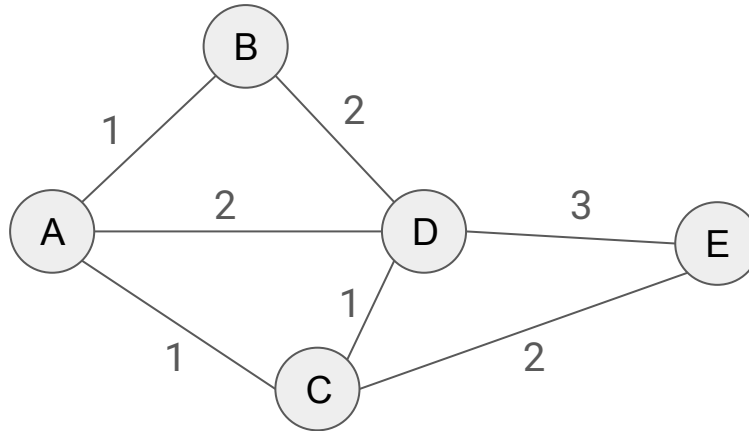
- Classer un problème dans P ou dans NP dépend du meilleur algorithme connu. C'est une chose qui peut changer avec les progrès. Par exemple, dans ce cours nous avons d'abord cru que le tri était $O(n^2)$ avant de voir qu'il était $O(n \log(n))$.
- On n'a jamais réussi à trouver un problème facile à **vérifier** dont on puisse prouver qu'il ne puisse pas être **résoluble** mieux qu'exponentiellement ! Il suffirait d'en trouver un pour montrer que $P \neq NP$.
- On a pu démontrer que tous les problèmes NP-complets sont en fait des formulations différentes d'un même problème (SAT) \Rightarrow on peut aussi essayer de trouver un algo de résolution polynomiale d'un problème NP-complet, car cela prouverait que $P = NP$.

(Diapositive hors champ)

Recherche de plus court chemin

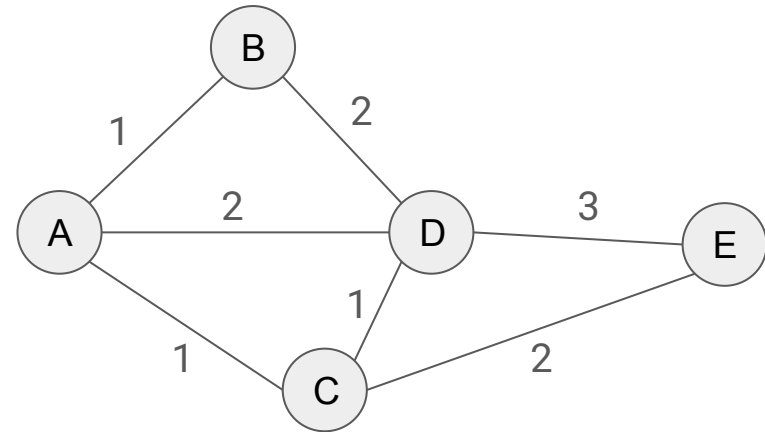
Des villes sont reliées par des routes. Le temps mis pour joindre deux villes *via* une route dépend de la route.

Partant de A, quel est le plus court chemin pour aller à E ?



Recherche de plus court chemin

- La structure mathématique que nous utilisons ici est un **graphe** dont les noeuds sont des villes, et les arêtes des routes.
- Le graphe est **non orienté** : les routes peuvent être prises dans les deux sens.
- Le graphe est **pondéré** : chaque route est associée à un certain poids.
- La solution au problème est une séquence de noeuds.



Comment implémenter une structure de graphe ?

- Tout comme pour les arbres, il existe plusieurs implémentations dont la pertinence dépend de la situation.
- Si le graphe est dense, alors il est bon de le décrire *via* sa matrice d'adjacence, qui est telle que $M_{ij} = 1$ si les noeuds i et j sont connectés, 0 sinon.
- Si le graphe est peu dense, il peut être intéressant de l'implémenter *via* une variante de liste chaînées.
- Ici, on ne s'intéresse pas à l'implémentation exacte, mais juste aux algorithmes exploitant la structure mathématique de graphe.

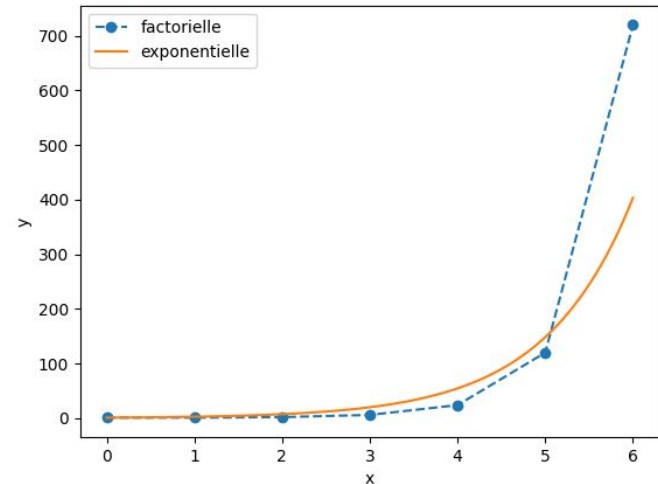
Matrice d'adjacence

- On peut même définir que le poids de l'arête connectant i et j vaut M_{ij} .
- Exemple pour $n = 5$ noeuds :
- En général, on définit $M_{ii} = 0$.
- Si le graphe est non orienté, on aura $M_{ij} = M_{ji}$, c'est-à-dire que la matrice est symétrique par rapport à la diagonale.
- Exemple sur le graphe des diapos précédentes :

	0	1	2	3	4
0	M_{00}	M_{01}	M_{02}	M_{03}	M_{04}
1	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}
2	M_{20}	M_{21}	M_{22}	M_{23}	M_{24}
3	M_{30}	M_{31}	M_{32}	M_{33}	M_{34}
4	M_{40}	M_{41}	M_{42}	M_{43}	M_{44}

Algorithme de plus court chemin naïf

- Tester tous les chemins possibles de A à E, et garder le plus court.
- Combien y a-t-il de chemins possibles ?
Pire des cas : chaque noeud est relié à tous les autres.
Dans ce cas, il existe $n!$ chemins possibles pour aller de A à E.
- La complexité d'un algorithme naïf est donc **particulièrement défavorable**.



Algorithme de Dijkstra | Idée générale

- L'algorithme construit, **pour un noeud de départ A donné**, le meilleur chemin vers chacun des autres noeuds.
- On associe à chaque noeud une **distance** depuis le noeud A. On associe également à chaque noeud un noeud **prédécesseur**.
- On maintient une liste des noeuds déjà **visités**. À chaque fois qu'on visite un noeud N, on regarde si depuis là on peut améliorer le score des voisins pas encore visités. Si on peut améliorer le score d'un voisins, alors son nouveau prédécesseur est le noeud N.
- Le prochain noeud à visiter est celui qui possède le plus petit score, parmi les noeuds pas encore visités.
- Pour reconstruire le chemin depuis un noeud, on "remonte" de prédécesseur en prédécesseur.

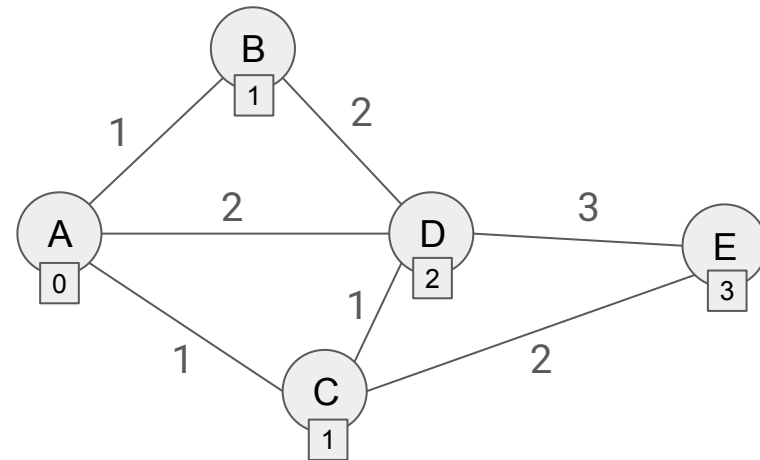
Algorithme de Dijkstra

1. `visites = []`
2. `prédécesseur[i] = ∅` pour tout noeud `i`
3. `dist[i] = ∞` pour tout noeud `i`
4. `dist[0] = 0`
5. Tant que `length(visites) ≠ n`:
 - a. `N = nœud hors de visites avec le plus petit score`
 - b. On insère `N` dans `visites`
 - c. Pour chaque voisin `V` de `N`, si `V ∉ visites`:
 - c.1 Si `dist[V] > dist[N] + MNV`:
 - c.1.1 `dist[V] = dist[N] + MNV`
 - c.1.2 `prédécesseur[V] = N`

Algorithme de Dijkstra | Exemple pas à pas

Cf. slides d'illustration pas à pas

Noeud	A	B	C	D	E
Score	0	1	1	2	3
Prédécesseur	∅	A	A	A	C



Algorithme de Dijkstra | Complexité

Tant que $\text{length}(\text{visites}) \neq n$: $O(n)$

N = nœud hors de visites avec le plus petit score $O(n)$

On insère N dans visites $O(n)$

Pour chaque voisin V de N , si $V \notin \text{visites}$: $O(n)$

Si $\text{dist}[V] > \text{dist}[N] + M_{NV}$: $O(1)$

$\text{dist}[V] = \text{dist}[N] + M_{NV}$ $O(1)$

$\text{prédécesseur}[V] = N$ $O(1)$

$O(1)$ au prix de certains raffinements non discutés ici.



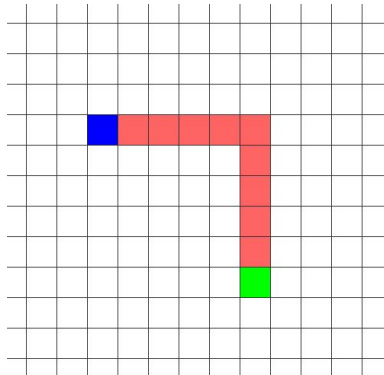
Algorithme de Dijkstra | Commentaires finaux

- Des variantes de l'algorithme, où certaines structures (*priority queues*) sont utilisées à bon escient, sont $O(E \log(V))$, avec E le nombre d'arêtes et V le nombre de sommets.
- La complexité de la version discutée ici, où `visites` et `dist` sont des tableaux de valeurs, est $O(n^2)$.
- Dijkstra est similaire à une recherche en largeur.

(Diapositive hors champ)

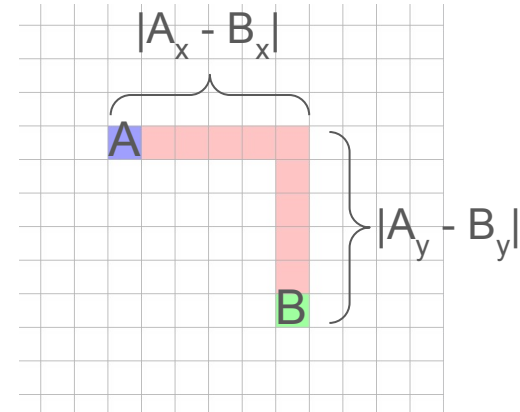
Algorithme de plus court chemin sur une grille (A*)

- Lorsque le chemin est à choisir parmi un **ensemble de cases adjacentes** (par exemple dans un labyrinthe ou sur un "échiquier"), on peut améliorer l'algorithme de Dijkstra en raison de la structure régulière du graphe sous-jacent.
- On introduit une **heuristique** qui **guide** la recherche : les noeuds proches de la destination en terme de **distance** sont alors favorisés. Cette distance est facilement calculable.



A* | Idée générale

- On introduit une **heuristique** qui **guide** la recherche : les noeuds dont la distance "physique" est proche de la destination sont alors favorisés. Cette distance est facilement calculable.
- Exemple avec la "distance de Manhattan" :
$$d(A,B) = |A_x - B_x| + |A_y - B_y|$$
- Les noeuds à visiter en **priorité** sont ceux qui rapprochent "physiquement" du but.
- Non prouvé dans ce cours : le chemin est optimal si l'heuristique est "optimiste", (*i.e.* elle sous-estime ou égale toujours la véritable distance).



A* | Différence avec Dijkstra

1. ~~visites = []~~ (il est possible de visiter plusieurs fois le même noeud)
2. $\text{prédécesseur}[i] = \emptyset$ pour tout noeud i
3. $\text{dist}[i] = \infty$ pour tout noeud i
4. $\text{dist}[0] = 0$
5. ~~Tant que $\text{length}(\text{visites}) \neq n$:~~ (tant qu'on n'est pas arrivé à destination)
 - a. N = nœud hors de visites avec le plus petit score
 - b. ~~On insère N dans visites~~
 - c. Pour chaque voisin V de N , ~~si $V \notin \text{visites}$:~~
 - i. Si $\text{dist}[V] > \text{dist}[N] + M_{NV}$:
 1. $\text{dist}[V] = \text{dist}[N] + M_{NV}$
 2. $\text{prédécesseur}[V] = N$



A* | Pseudocode

```
N = depart
candidats = []
Tant que dist(N, arrivee) ≠ 0:
    enfants = construire_enfants(case_visitee)
    ajouter enfants aux candidats
    Si candidats est vide:
        afficher("Pas de solution")
        retourner ∅
    N = candidat avec le plus petit score
retourner N
```

A* | Pseudocode

```
N = depart
candidats = []
meilleur_distance = {}
score = {}
Tant que dist(N, arrivee) ≠ 0:                                construire_enfants( )
    Pour chaque voisin V de N:
        nouvelle_dist_a_depart = dist_a_depart[N] + 1
        Si nouvelle_dist_a_depart < meilleur_distance[V]
            meilleur_distance[V] = nouvelle_dist_a_depart
            score[V] = nouvelle_dist_a_depart + manhattan(V, arrivee)
            ajouter V aux candidats
    Si candidats est vide:
        afficher("Pas de solution")
        retourner ∅
    N = candidat avec le plus petit score
retourner N
```

A* | Commentaires finaux

- Pire des cas $O(E \log(V))$, avec E le nombre d'arêtes et V le nombre de sommets.
- En pratique, très performant dans de nombreux problèmes.
- Dijkstra trouve le chemin optimal d'un noeud vers tous les autres ; A* ne trouve le chemin optimal que d'un noeud de départ vers une destination donnée.

