Graphes de visibilité

Trouver le plus court chemin

Christian Nguyen

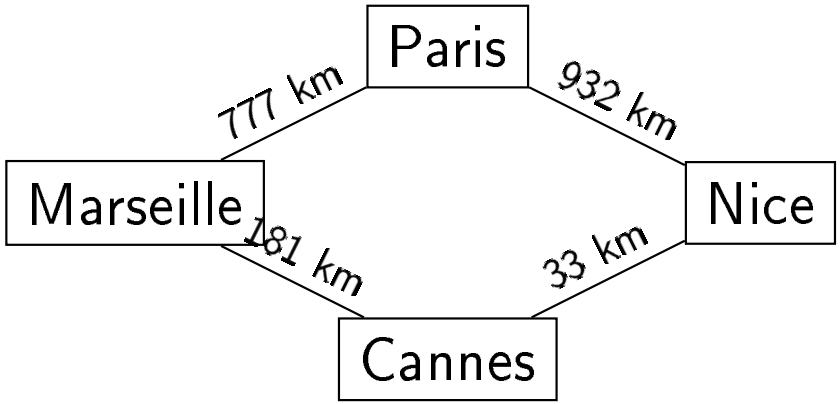
Département d'informatique
Université de Toulon

Plan

Motivation

Les graphes permettent de résoudre de nombreux problèmes concrets :

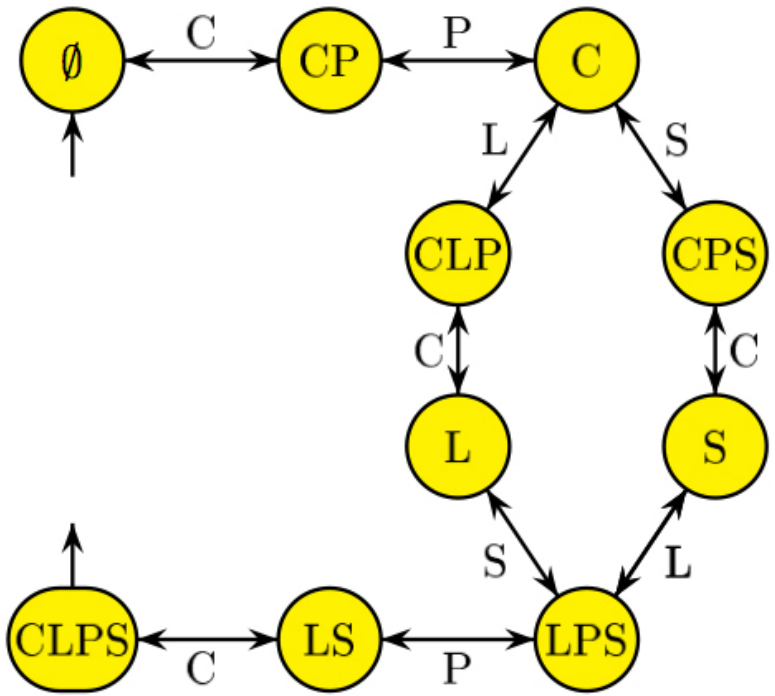
- réseaux routiers : les sommets sont les villes et les arêtes[1] sont les routes reliant des couples de villes ; ces arêtes peuvent être valuées par la longueur des routes correspondantes,



1. dénommées *arcs* dans un graphe orienté.

Motivation

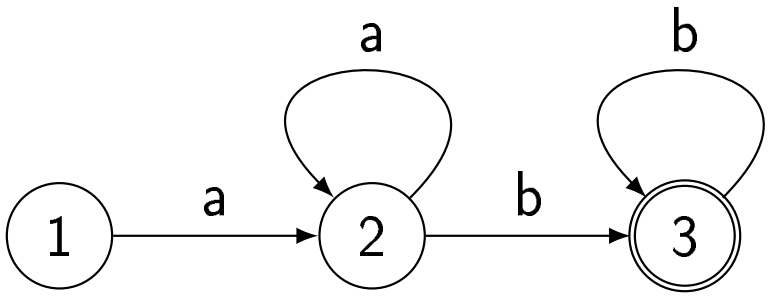
- processus à étapes : une famille de problèmes peut être définie par un état initial, un état final, des états intermédiaires et des règles de transition (par exemple, le problème « du chou, de la chèvre et du loup »),



Motivation

- automates finis : ils permettent de reconnaître un langage régulier et peuvent être représentés par un graphe orienté et étiqueté.

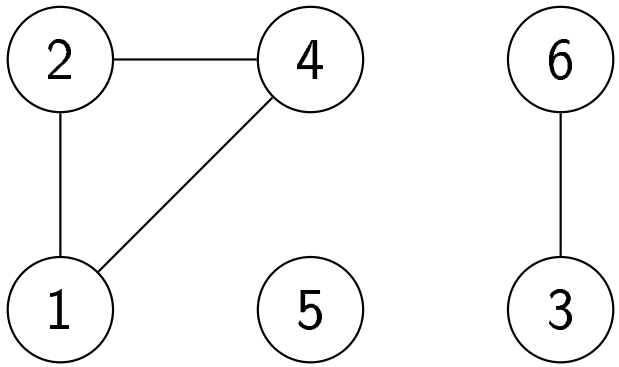
  Exemple, l'automate fini qui reconnait le langage des mots de la forme $a^n b^m$:



Définition

Un graphe non orienté est défini par un couple $G(N, E)$, formé d'un ensemble de nœuds N et d'un ensemble d'arêtes E.

Un couple de sommets $(n_i, n_j) \in E$ est équivalent au couple (n_j, n_i), la paire (n_i, n_j) est représentée graphiquement par $n_i - n_j$.

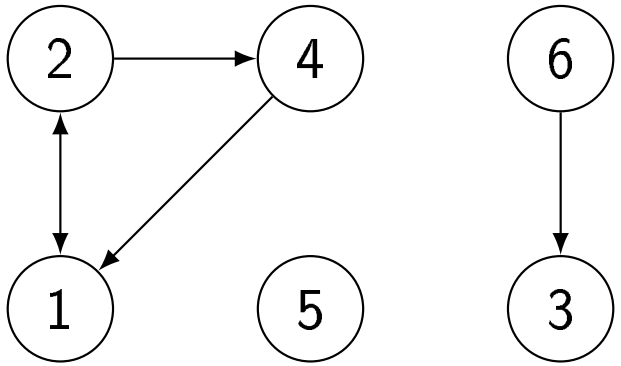


Le graphe $G(N, E)$ avec $N = \{1, 2, 3, 4, 5, 6\}$ et $E = \{(1, 2), (1, 4), (2, 4), (3, 6)\}$

Définition

Un graphe orienté est défini par un couple $G(V, A)$, formé d'un ensemble de sommets V et d'un ensemble d'arcs (arêtes orientées) A.

Un couple de sommets $(s_i, s_j) \in A$ est un couple ordonné, où s_i est le sommet initial et s_j est le sommet final, le couple (s_i, s_j) est représentée graphiquement par $s_i \rightarrow s_j$.



Le graphe $G(V, A)$ avec $V = \{1, 2, 3, 4, 5, 6\}$ et $A = \{(1, 2), (2, 1), (2, 4), (4, 1), (6, 3)\}$

Terminologie

- l'*ordre* d'un graphe est le nombre de ses sommets,
- une *boucle* est un arc ou une arête reliant un sommet à lui-même,
- un graphe non-orienté est dit *simple* s'il ne comporte pas de boucle, et s'il ne comporte jamais plus d'une arête entre deux sommets,
- un graphe orienté est dit *élémentaire* s'il ne contient pas de boucle,
- un graphe orienté est un *p-graphe* s'il comporte au plus p arcs entre deux sommets, le plus souvent on étudiera des 1-graphes,

Terminologie (suite)

- un *graphe partiel* d'un graphe orienté ou non est le graphe obtenu en supprimant certains arcs ou arêtes,
- un *sous-graphe* d'un graphe orienté ou non est le graphe obtenu en supprimant certains sommets et tous les arcs ou arêtes incidents aux sommets supprimés,
- un graphe orienté (resp. non-orienté) est dit *complet* s'il comporte un arc (s_i, s_j) et un arc (s_j, s_i) (resp. une arête (n_i, n_j)) pour tout couple (resp. toute paire) de sommets différents $s_i, s_j \in V^2$ (resp. $n_i, n_j \in N^2$).

Notion d'adjacence

Graphe non orienté : un sommet s_i est dit adjacent à un autre sommet s_j s'il existe une arête entre s_i et s_j. L'ensemble des sommets adjacents à un sommet s_i est défini par :

$$adj(s_i) = \{s_j / (s_i, s_j) \in E\}$$

Graphe orienté : on distingue les sommets successeurs des sommets prédécesseurs :

$$\begin{aligned} succ(s_i) &= \{s_j / (s_i, s_j) \in A\} \\ pred(s_i) &= \{s_j / (s_j, s_i) \in A\} \end{aligned} \tag{1}$$

Notion de degré d'un sommet

Graphe non orienté : le degré d'un sommet est le nombre d'arêtes incidentes à ce sommet (dans le cas d'un graphe simple, on aura $d^\circ(s_i) = |adj(s_i)|$).

Graphe orienté :

- le demi-degré extérieur d'un sommet s_i, noté $d^{\circ+}(s_i)$, est le nombre d'arcs partant de s_i (dans le cas d'un 1-graphe, on aura $d^{\circ+}(s_i) = |succ(s_i)|$),
- le demi-degré intérieur d'un sommet s_i, noté $d^{\circ-}(s_i)$, est le nombre d'arcs arrivant à s_i (dans le cas d'un 1-graphe, on aura $d^{\circ-}(s_i) = |pred(s_i)|$).

Plan

Représentation par matrice d'adjacence

Soit le graphe $G = (S, A)$. On suppose que les sommets de S sont numérotés de 1 à n, avec $n = |S|$.

Représentation de G : matrice booléenne M de taille $n \times n$ telle que $M[i][j] = 1$ si $(i, j) \in A$, et $M[i][j] = 0$ sinon.

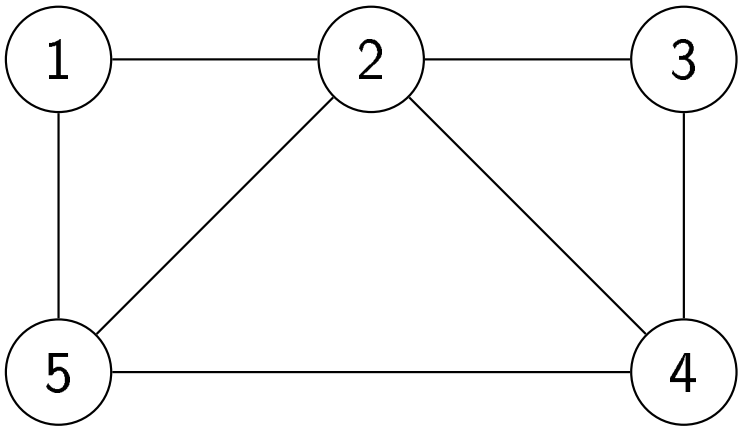
Si le graphe est valué (par exemple, si des distances sont associées aux arcs), on peut utiliser une matrice d'entiers, de telle sorte que :

- $M[i][j]$ soit égal à la valuation de l'arc (i, j) si $(i, j) \in A$,
- $M[i][j]$ soit égal à une valeur particulière (par exemple 0 ou $-\infty$) sinon (il n'existe pas d'arc entre 2 sommets i et j)

Cette représentation nécessite n^2 emplacements mémoire. Le test de l'existence d'un arc ou d'une arête est immédiat, en revanche le parcours de l'ensemble des arcs/arêtes prendra un temps de l'ordre de n^2.

Représentation par matrice d'adjacence

Exemple



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Représentation par listes d'adjacence

Représentation de G : tableau T de n listes, une pour chaque sommet de S. Pour chaque sommet $s_i \in S$, la liste d'adjacence $T[s_i]$ est une liste chainée de tous les sommets s_j tels qu'il existe un arc ou une arête $(s_i, s_j) \in A$.

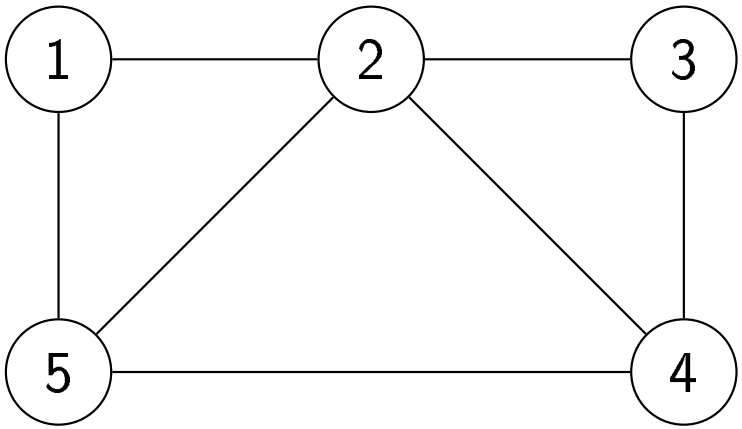
Si le graphe est valué (par exemple, si les arêtes représentent des distances), on peut stocker dans les listes d'adjacence, en plus du numéro de sommet, la valuation de l'arête.

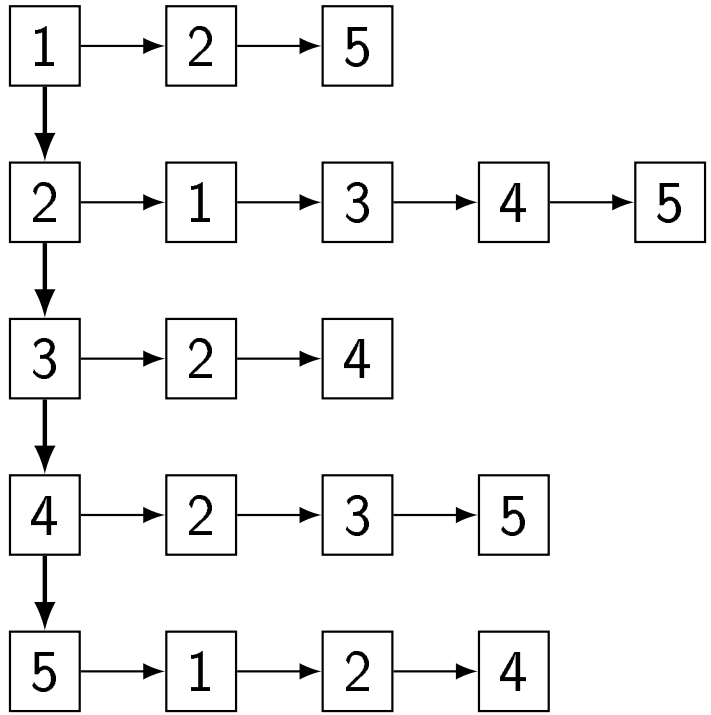
La mémoire nécessaire à cette représentation est proportionnelle au nombre m d'arcs (resp. à deux fois le nombre d'arêtes) si le graphe est orienté (resp. non orienté) et au nombre n de sommets soit $O(n + m)$.

Le test d'existence d'un arc (ou d'une arête) est moins direct, en revanche le calcul du degré d'un sommet ou l'accès à tous les successeurs d'un sommet est plus efficace.

Représentation par listes d'adjacence

Exemple





Remarque : dans le cas de graphes *non orientés*, pour chaque arête (s_i, s_j), s_j appartient à la liste chainée de $\mathtt{T}[s_i]$, et s_i appartient à la liste chainée de $\mathtt{T}[s_j]$.

Plan

Chemin (chaîne) et circuit (cycle)

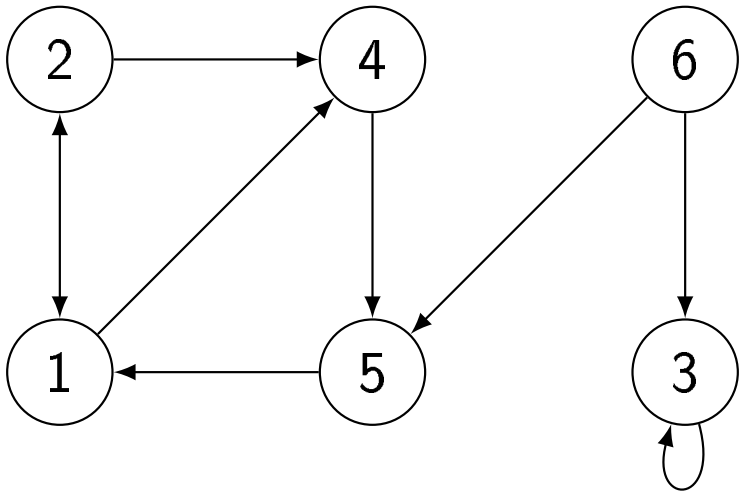
Dans un graphe orienté (resp. non orienté), un chemin (resp. chaîne) d'un sommet u vers un sommet v est une séquence $< s_0, s_1, \ldots, s_k >$ de sommets tels que $u = s_0$, $v = s_k$, et $(s_{i-1}, s_i) \in A$ pour $i \in [1, k]$. La longueur du chemin est le nombre d'arcs dans le chemin, c'est-à-dire k.

S'il existe un chemin de u à v, on dira que v est *accessible* à partir de u. Un chemin est *élémentaire* si les sommets qu'il contient sont tous distincts.

Un chemin $< s_0, s_1, \ldots, s_k >$ forme un circuit (resp. cycle) si $s_0 = s_k$ et si le chemin comporte au moins un arc ($k \geq 1$). Ce circuit est élémentaire si en plus les sommets $s_1, s_2, \ldots, s_k$ sont tous distincts. Une boucle est un circuit de longueur 1.

Chemin (chaîne) et circuit (cycle)

Exemple



Un chemin élémentaire dans ce graphe est $< 2, 1, 4, 5 >$.
Un chemin non élémentaire dans ce graphe est $< 6, 3, 3, 3 >$.
Un circuit élémentaire dans ce graphe est $< 2, 4, 5, 1, 2 >$.
Un circuit non élémentaire dans ce graphe est $< 2, 4, 5, 1, 4, 5, 1, 2 >$.

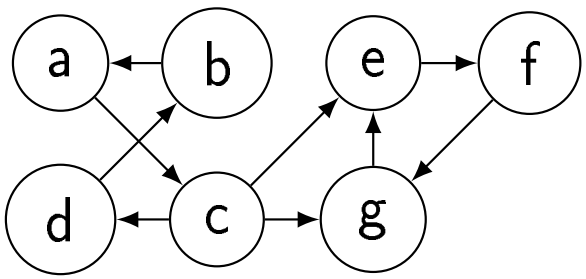
Fermeture transitive

La fermeture transitive d'un graphe $G = (S, A)$ est le graphe $G^f = (S, A^f)$ tel que pour tout couple de sommets $(s_i, s_j) \in S^2$, l'arc/arête (s_i, s_j) appartient à A^f si et seulement s'il existe un chemin/chaine de s_i vers s_j.

Le calcul de la fermeture transitive d'un graphe peut se faire en additionnant les « puissances » successives de sa matrice d'adjacence. Si le graphe est représenté par sa matrice d'adjacence, alors M^k (la matrice obtenue en multipliant M par elle même k fois successivement) est la matrice des chemins de longueur k.

Fermeture transitive

Exemple



	a	b	c	d	e	f	g
a	0	0	1	0	0	0	0
b	1	0	0	0	0	0	0
c	0	0	0	1	1	0	1
d	0	1	0	0	0	0	0
e	0	0	0	0	0	1	0
f	0	0	0	0	0	0	1
g	0	0	0	0	1	0	0

Existe-t-il un chemin de longueur 2 entre a et d ? Oui, car il existe un chemin à la même position horizontalement pour a et verticalement pour d.

Fermeture transitive

Exemple (suite)

	a	b	c	d	e	f	g
a	0	0	0	1	1	0	1
b	0	0	1	0	0	0	0
c	0	1	0	0	1	1	0
d	1	0	0	0	0	0	0
e	0	0	0	0	0	0	1
f	0	0	0	0	1	0	0
g	0	0	0	0	0	1	0

M^2

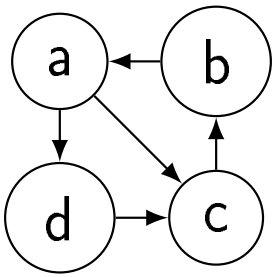
	a	b	c	d	e	f	g
a	1	1	1	1	1	1	1
b	1	1	1	1	1	1	1
c	1	1	1	1	1	1	1
d	1	1	1	1	1	1	1
e	0	0	0	0	1	1	1
f	0	0	0	0	1	1	1
g	0	0	0	0	1	1	1

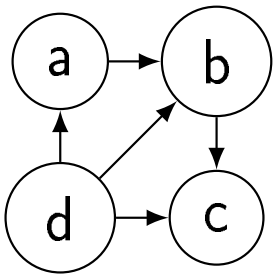
$M + M^2 + M^3 + M^4 + M^5$

$M + M^2 + M^3 + M^4 + M^5$ est la matrice des chemins de longueur inférieure ou égale à 5. Si on poursuit le calcul, on constate que la matrice ne change plus donc c'est la matrice d'adjacence de la fermeture transitive G^f du graphe G de départ.

Connexité

Un graphe non orienté (resp. orienté) est connexe (resp. fortement connexe) si chaque sommet est accessible à partir de *n'importe quel* autre. Autrement dit, si pour tout couple de sommets distincts $(si, sj) \in S2$, il existe une chaine entre si et sj. Exemple :





Une composante connexe (resp. composante fortement connexe) d'un graphe non orienté (resp. orienté) G est un sous-graphe G' de G qui est connexe (resp. fortement connexe) et maximal (c'est à dire qu'aucun autre sous-graphe connexe (resp. fortement connexe) de G ne contient G').

Graphe eulérien

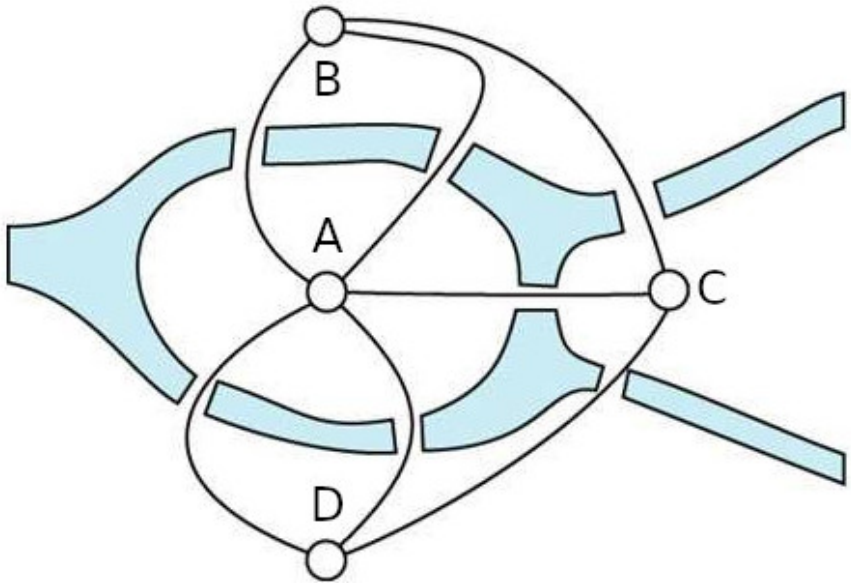
Dans un graphe non orienté, une chaine (resp. cycle) eulérienne est une chaine (resp. cycle) qui emprunte une et une seule fois chaque arête du graphe. Un graphe comportant une chaine ou un cycle eulérien est appelé graphe eulérien.

Théorème d'existence d'un cycle eulérien : un graphe (simple ou multiple) connexe admet un cycle eulérien si et seulement s'il n'a pas de sommet de degré *impair*.

Théorème d'existence d'une chaîne eulérienne : un graphe (simple ou multiple) connexe admet une chaine eulérienne entre deux sommets u et v si et seulement si le degré de u et le degré de v sont *impairs*, et les degrés de *tous les autres* sommets du graphe sont *pairs*.

Graphe eulérien

Exemple



Dans ce graphe (le fameux pont de Königsberg), les degrés des sommets A, B, C et D sont respectivement 5, 3, 3 et 3. On a donc 4 sommets de degré impair, et il n'y a pas de chaîne eulérienne, et encore moins de cycle eulérien.

Graphe eulérien

Théorème d'existence d'un circuit eulérien : un multigraphe orienté fortement connexe admet un circuit eulérien si et seulement si $d^{\circ+}(s_i) = d^{\circ-}(s_i)$ pour tout sommet $s_i \in S$.

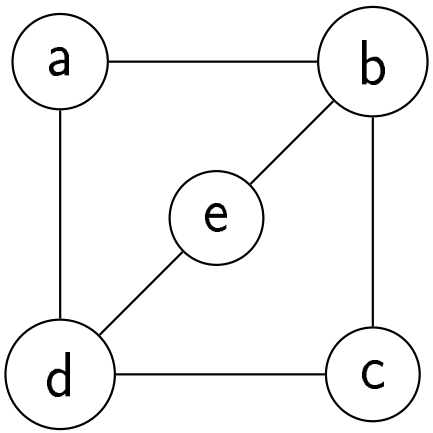
Théorème d'existence d'un chemin eulérien : un multigraphe orienté connexe admet un chemin eulérien de u vers v si et seulement si

- $d^{\circ+}(u) = d^{\circ-}(u) + 1$,
- $d^{\circ+}(v) = d^{\circ-}(v) - 1$,
- $d^{\circ+}(s_i) = d^{\circ-}(s_i)$ pour tout autre sommet $s_i \in S - \{u, v\}$.

Graphe hamiltonien

Dans un graphe simple non orienté comportant n sommets, une chaîne hamiltonienne est une chaîne élémentaire de longueur $n-1$. Autrement dit, une chaîne hamiltonienne passe une et une seule fois par chacun des n sommets du graphe.

Un cycle hamiltonien est un cycle élémentaire de longueur n. Un graphe possédant un cycle ou une chaîne hamiltonien sera dit graphe hamiltonien. Le graphe suivant ne possède pas de cycle hamiltonien, mais a une chaîne hamiltonienne.



Plan

Deux stratégies

Beaucoup de problèmes sur les graphes nécessitent que l'on parcourt l'ensemble des sommets et des arcs/arêtes d'un graphe.

Il y a deux principales stratégies d'exploration :

- le parcours en largeur consiste à explorer les sommets du graphe niveau par niveau, à partir d'un sommet donné,
- le parcours en profondeur consiste, à partir d'un sommet donné, à suivre un chemin le plus loin possible (jusqu'à un cul-de-sac ou un cycle), puis à faire des retours en arrière pour reprendre tous les chemins ignorés précédemment.

La différence fondamentale entre les deux parcours provient de la façon de gérer une liste d'attente : le parcours en largeur utilise une file, quand le parcours en profondeur utilise une pile.

Arbre couvrant

Arbre inclus dans un graphe (non orienté et connexe) et qui connecte tous les sommets du graphe.

(suite : voir cours du 1er semestre)

Parcours en largeur

voir cours du 1er semestre

Parcours en profondeur

voir cours du 1er semestre

Plan

Plus courts chemins

Application directe : recherche du plus court chemin en distance pour un trajet donné.

Un parcours en largeur du graphe associé ne resoudra pas ce problème : il permet de trouver l'itinéraire comportant le moins d'étapes mais ce n'est pas nécessairement le plus court en distance.

Une solution naïve consiste à énumérer tous les chemins, d'additionner les distances pour chacun d'eux et de choisir le plus court.

Pour résoudre ce type de problème, on met en œuvre des graphes orientés valués.

Définition

Soit $G = (S, A)$ un 1-graphe orienté valué tel que la fonction `cout : A → R` associe à chaque arc (s_i, s_j) de A un coût réel $cout(s_i, s_j)$.

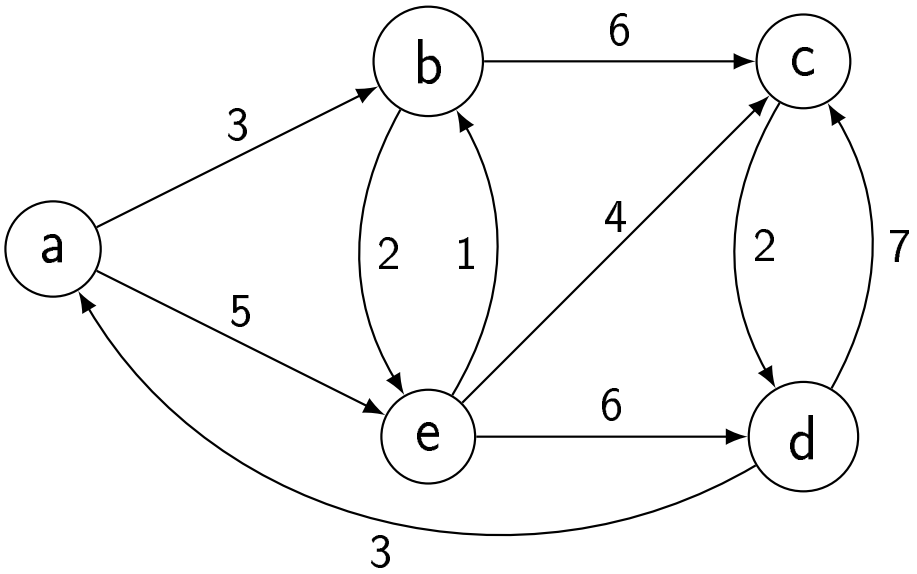
Le coût d'un chemin (ou poids du chemin) $p =< s_0, s_1, \ldots, s_k >$ est égal à la somme des coûts des arcs composant le chemin :

$$cout(p) = \sum_{i=1}^{k} cout(s_{i-1}, s_i)$$

Coût (ou poids) $\delta(s_i, s_j)$ d'un plus court chemin entre deux sommets s_i et s_j :

$$\begin{aligned} \delta(s_i, s_j) &= +\infty & si\ \nexists p \\ \delta(s_i, s_j) &= min\{cout(p)\} & sinon \end{aligned}$$

Exemple



$\delta(a, b)$?

$\delta(a, c)$?

$\delta(a, d)$?

$\delta(a, e)$?

Condition d'existence

S'il existe un chemin entre deux sommets u et v contenant un circuit de coût négatif, alors $\delta(u, v) = -\infty$, et il n'existe pas de plus court chemin entre u et v.

Autrement dit, à chaque fois que l'on passe dans un tel circuit, on diminue le coût total du chemin.

Un circuit négatif est appelé un circuit absorbant.

Origine unique

Soit $G = (S, A)$ un 1-graphe orienté valué, une fonction coût $\mathtt{A} \rightarrow \mathtt{R}$ et un sommet unique $s_0 \in S$.

C'est le calcul pour chaque sommet $s_j \in S$ du coût $\delta(s_0, s_j)$ du plus court chemin de s_0 à s_j (on supposera que le graphe G ne comporte pas de circuit absorbant).

Cette méthode n'est pas la plus efficace :

- si la destination est unique et si l'on dispose d'une borne minimale de la longueur, par exemple la distance euclidienne (on lui préfèrera l'algorithme A*),
- si on souhaite calculer tous les plus courts chemins entre tous les couples de sommets possibles (on lui préfèrera l'algorithme de Floyd-Warshall).

Arborescence

Calcul des coûts des plus courts chemins, mais aussi identification des sommets présents sur ces plus courts chemins.

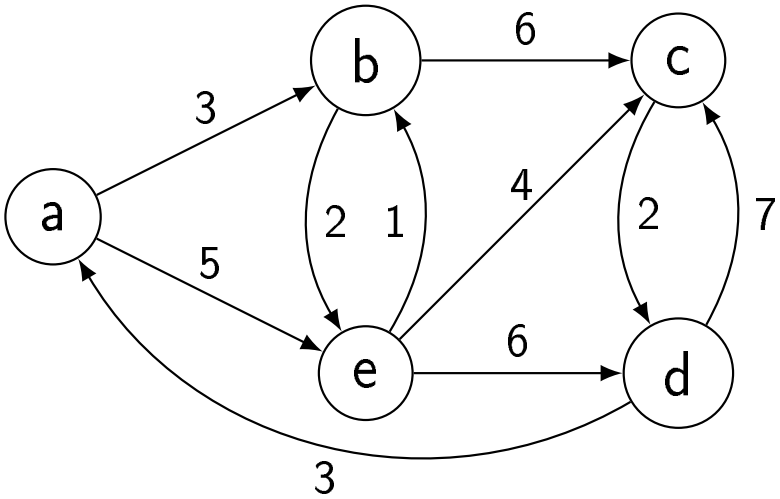
Représentation utilisée : la même que celle utilisée pour les arborescences couvrantes calculées lors d'un parcours en largeur ou en profondeur d'un graphe, c.-à-d. un tableau T tel que :

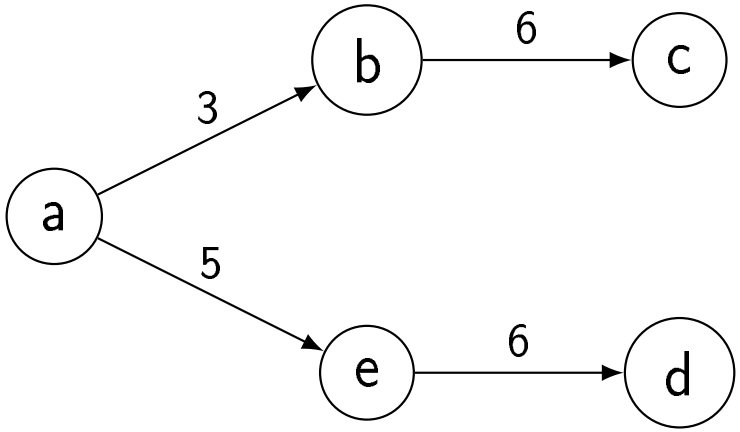
- $T[s_0] = \emptyset$,
- $T[s_j] = s_i$ si $s_i \rightarrow s_j$ est un arc de l'arborescence.

Pour connaitre le plus court chemin entre s_0 et un sommet s_k donné, il faudra alors « remonter » de s_k jusque s_0 en utilisant T.

Arborescence

Exemple





L'**une** des arborescences des plus courts chemins possible dont l'origine est a. Elle est représentée par le tableau T tel que $T[a] = \emptyset$, $T[b] = a$, $T[c] = b$, $T[d] = e$ et $T[e] = a$.

Algorithmes

On distingue les algorithmes qui résolvent le problème quand tous les coûts sont positifs ou nuls (Dijkstra) de ceux qui résolvent le problème quand les coûts sont positifs, négatifs ou nuls (Ford-Bellman), sous réserve qu'il n'y ait pas de circuit absorbant.

Ces algorithmes fonctionnent suivant un stratégie dite « gloutonne ». On associe à chaque sommet $s_i \in S$ une valeur $d[s_i]$ qui représente une **borne maximale du coût du plus court chemin** entre s_0 et s_i.

Initialisation : $d[s_0] = 0$ et $d[s_i] = +\infty$ pour tout sommet $s_i \neq s_0$.

Fonctionnement : diminution progressive des valeurs $d[s_i]$ jusqu'à ce qu'on ne puisse plus les diminuer, on a alors $d[s_i] = \delta(s_0, s_i)$.

Relâchement d'arc

Diminution de $d[s_j]$ en examinant itérativement chaque arc $s_i \rightarrow s_j$ du graphe.

Algorithme relacher(s_i, s_j)

données
s_i, s_j : sommets d'un graphe
début
si $d[s_j] > d[s_i] + cout(s_i, s_j)$ **alors**
▷ *il vaut mieux passer par* s_i *pour aller à* s_j
$d[s_j] \leftarrow d[s_i] + cout(s_i, s_j)$
$T[s_j] \leftarrow s_i$ ▷ *arc de l'arborescence*
finsi
fin

Algorithme de Dijkstra (1959)

Soient deux ensembles disjoints E et F tels que $E \cup F = S$, l'ensemble F contient chaque sommet s_i pour lequel on connaît un plus court chemin depuis s_0, l'ensemble E contient tous les autres sommets.

A chaque itération de l'algorithme, on choisit le sommet s_i dans E pour lequel la valeur $d[s_i]$ est *minimale*, on le rajoute dans F, et on *relâche* tous les arcs partant de ce sommet s_i.

Algorithme Dijkstra(G, s_0)

données
  s_0 sommet initial du graphe G
début
  pour chaque sommet $s_i \in S$ **faire**
    $d[s_i] \leftarrow +\infty$
    $T[s_i] \leftarrow \emptyset$
  finpour
  $d[s_0] \leftarrow 0$
  $E \leftarrow S$
  $F \leftarrow \emptyset$
  tant que $E \neq \emptyset$ **faire**
    $s_i \leftarrow$ sommet de F tel que $d[s_i]$ soit minimale
    $E \leftarrow E - \{s_i\}$
    $F \leftarrow F \cup \{s_i\}$
    $\forall s_j \in succ(s_i)$ `relacher(`s_i`,` s_j`)` $\triangleright$ *maj* $d[s_j]$ *et* $T[s_j]$
  fintq
  retourner T
fin

Correction

Il faut montrer qu'à chaque fois qu'un sommet s_i entre dans l'ensemble F, c'est le sommet le plus proche de s_0 de l'ensemble E c.-à-d. $d[s_i] = \delta(s_0, s_i)$.

Le premier sommet à entrer dans F est s_0, pour lequel $d[s_0] = \delta(s_0, s_0) = 0$.

À chaque itération, on fait entrer dans F un sommet $s_i \in F$ tel que $d[s_i]$ est minimal. S'il existe un autre chemin allant de s_0 à s_i :

- il passe par un sommet $s_j \in E$ tel que $d[s_j] > d[s_i]$ (puisque $s_j \notin F$),
- il passe par un sommet $s_j \in F$ tel que $d[s_j] \leq d[s_i]$.

Complexité

On suppose que le graphe possède n sommets et p arcs. La complexité de cet algorithme dépend de l'implémentation du graphe et de la façon de gérer l'ensemble E.

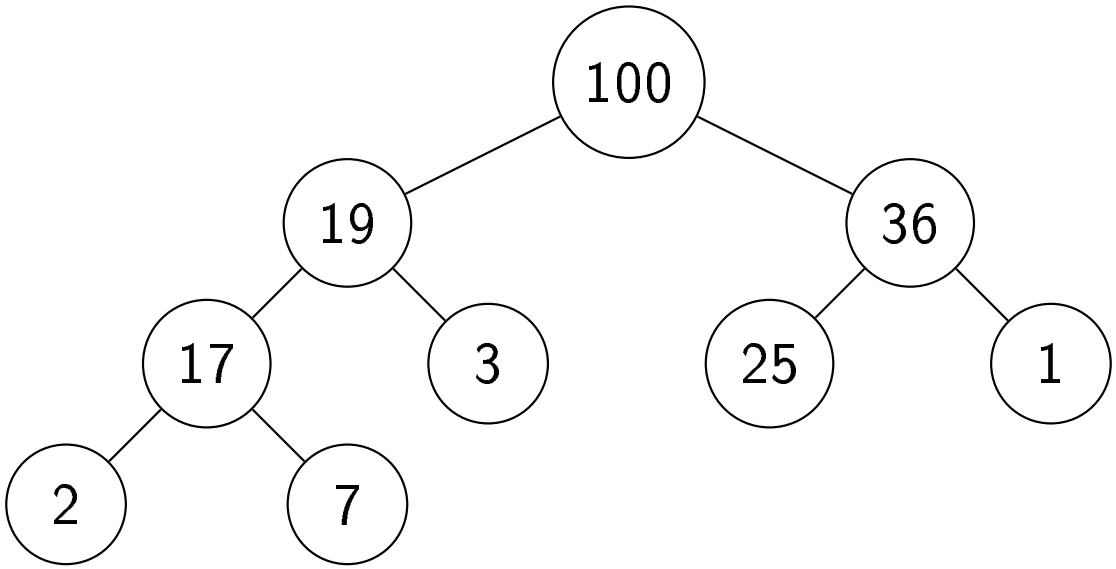
Si on utilise une *matrice* d'adjacence, l'algorithme sera en $O(n^2)$. En revanche, si on utilise une *liste* d'adjacence, la complexité dépend de son implantation :

- par une liste (ou un tableau) : la recherche du sommet de E ayant la plus petite valeur de d nécessite $n + (n-1) + \cdots + 1$ opérations et les opérations de relâchement de l'ordre de p opérations (chaque arc est relâché une seule fois), soit une complexité en $O(n^2)$,
- par un tas binaire : il permet de trouver plus rapidement la plus petite valeur de E (en temps constant), mais l'ajout, la suppression ou la modification d'un élément prend de l'ordre de $\log_2(n)$ opérations, soit une complexité en $O(p \log_2(n))$.

Un mot sur les tas

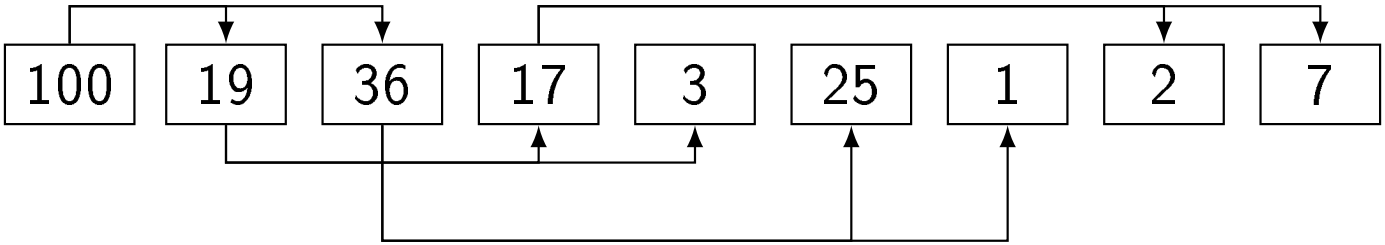
Un tas (*heap*) est un arbre binaire presque complet ordonné :

- presque complet : si tous les niveaux sont remplis, sauf éventuellement le dernier,
- ordonné : la clé d'un nœud parent a une plus haute priorité que les clés de ses enfants.



Implantation des tas

Une simple liste suffit.



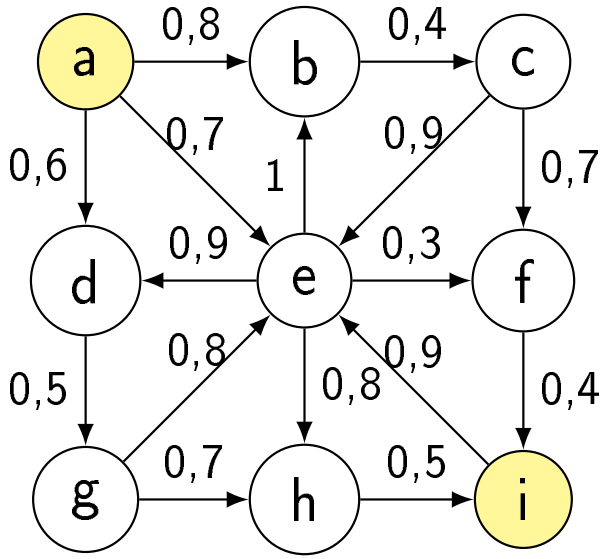
Les indices des fils d'un nœud d'indice i suivent la relation algébrique $2i + 1$ et $2i + 2$.

La gestion d'un tas se fait par les opérations `défiler` (retirer l'élément de priorité maximale) et `enfiler` (ajout d'un élément) en maintenant dans les deux cas la priorité du tas (par *percolation* et de façon itérative).

Exercice

Soit un réseau de télécommunication, composé d'émetteurs/récepteurs pouvant s'envoyer des messages, avec une certaine fiabilité de communication (une probabilité de non interruption).

Le problème est modélisé à l'aide du graphe orienté valué suivant, où la valuation d'un arc est une valeur réelle comprise entre 0 et 1 indiquant la probabilité pour que la communication soit fiable.



Adapter l'algorithme de Dijkstra pour déterminer le chemin le plus fiable pour envoyer un message de a vers i.

Algorithme A* (1968)

Algorithme de recherche de chemin dans un graphe, entre un nœud initial et un nœud final, conçu pour que la première solution trouvée soit l'une des meilleures, en privilégiant la *vitesse* de calcul sur l'*exactitude* des résultats.

A* utilise une *heuristique* basée sur la somme d'un coût associé à chaque nœud (0 pour le nœud initial) et la distance qui sépare ce nœud du nœud final. Le nœud est alors ajouté à une file d'attente prioritaire (*open list*).

L'algorithme retire le premier nœud de la file d'attente prioritaire (si elle est vide, il n'y a aucun chemin) :

- si c'est le nœud d'arrivée, on reconstruit le chemin complet,
- sinon de nouveaux nœuds sont créés pour tous les nœuds contigus admissibles (avec leur coût).

Algorithme A*

Le coût d'un nœud est calculé à partir de la somme du coût de son ancêtre et du coût de l'opération pour atteindre ce nouveau nœud, auquel est ajouté la distance du nouveau nœud au nœud d'arrivée. Ce nœud est alors ajouté à la file d'attente prioritaire, à moins qu'il ne soit déjà présent avec un coût inférieur ou égal.

L'algorithme maintient également la liste de nœuds qui ont été vérifiés (*closed list*). Si un nœud nouvellement produit est déjà dans cette liste avec un coût égal ou inférieur, aucune opération n'est faite sur ce nœud.

Une fois les trois étapes réalisées pour chaque nœud contigu au nœud pris de la file d'attente prioritaire, ce dernier est ajouté à la liste des nœuds vérifiés. Le prochain nœud est alors retiré de la file d'attente prioritaire et le processus recommence.

Algorithme A*(G, s_0, s_f)

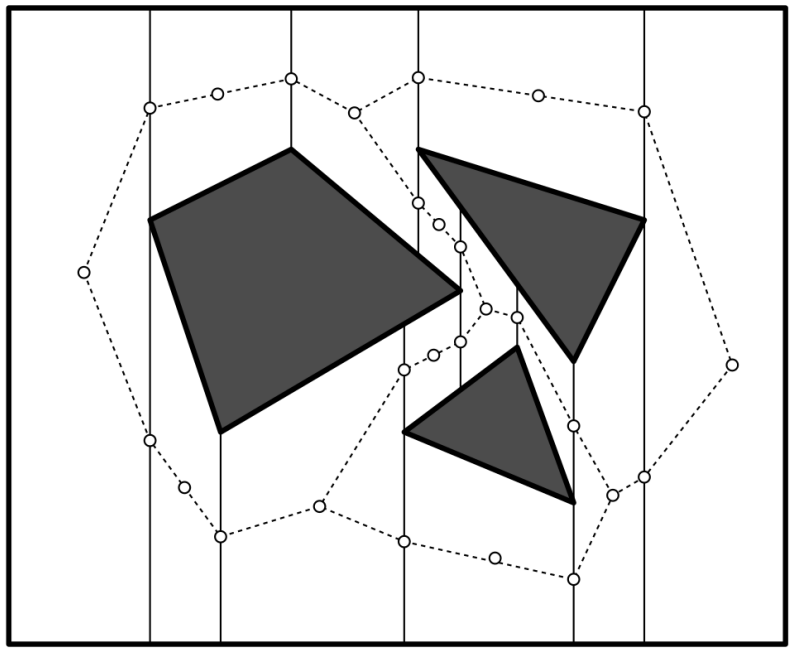
données
  s_0 (resp. s_f) sommet initial (resp. final) du graphe G
début
  closed_list ← File()
  open_list ← FilePrioritaire(s_0)
  tant que open_list n'est pas vide **faire**
    u ← open_list.defiler()
    si u = ∅ **alors**
      retourner pas de solution
    sinon si u = s_f **alors**
      retourner chemin(u)
    sinon
      pour chaque voisin v de u dans G **faire**
        si non (v ∈ closed_list **ou** v ∈ open_list avec un coût inférieur) **alors**
          v.cout ← u.cout + 1 + distance(v, s_f)
          open_list.ajouter(v) ▷ *substituer s'il existe*
      finpour
      closed_list.ajouter(u)
  fintq
fin

Plan

L'un des objectifs de la robotique est de parvenir à l'autonomie des déplacements (obtenue pour les bras robotiques).

Un préalable est qu'un robot doit avoir une connaissance de son environnement (obstacles statiques tels que les murs mais aussi dynamiques tels que les personnes).

Simplifications : plan 2D, obstacles statiques et de forme polygonale, robot réduit à un point.



Espace de travail, espace de configuration

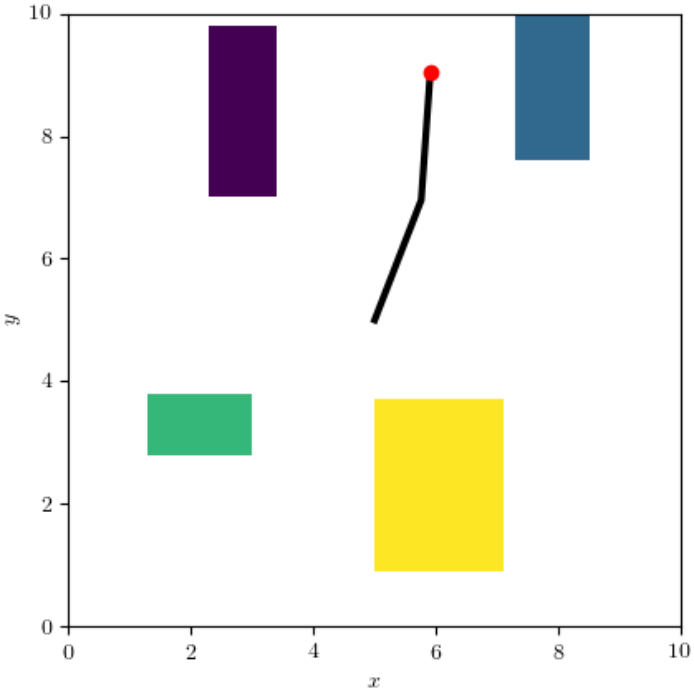
L'espace de travail (*work space* ou W) représente l'espace réel, l'espace de configuration (*configuration space* ou $\mathcal{C}$) représente l'espace paramétrique propre au robot. À un point $p \in \mathcal{C}$ correspond un certain placement $\mathcal{R}(p) \in W$.

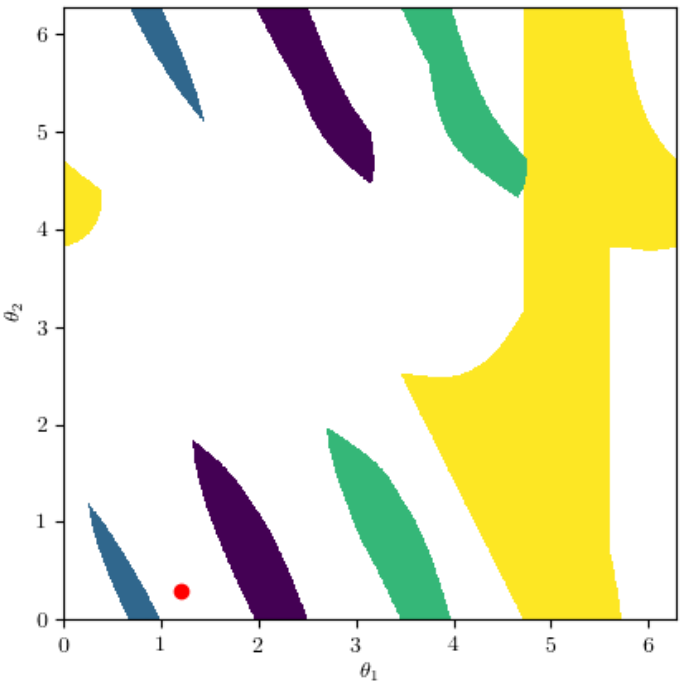
Dans le cas d'un robot capable de se déplacer en ligne droite et de tourner sur lui-même, $\mathcal{C}$ est l'espace $\mathbb{R}^2 \times [0, 360]$, ce qui correspond à une topologie équivalente à un cylindre. Un vecteur de cet espace correspond au placement $\mathcal{R}(x, y, \theta)$ dans W.

Les points de $\mathcal{C}$ qui correspondent à une intersection entre le robot et un obstacle doivent être exclus. Soit S l'ensemble des obstacles, on défini l'espace de configuration interdit $\mathcal{C}_p(\mathcal{R}, S)$ et l'espace de configuration libre $\mathcal{C}_f(\mathcal{R}, S)$.

Chemin

Les positions successives le long du chemin sont représentées par les points d'une courbe dans $\mathcal{C}$ et vice versa. Un chemin sans collision est une courbe de $\mathcal{C}_f$.





Robot ramené à un point

Dans le cas d'un robot ponctuel, les deux espaces (travail et configuration) sont identiques.

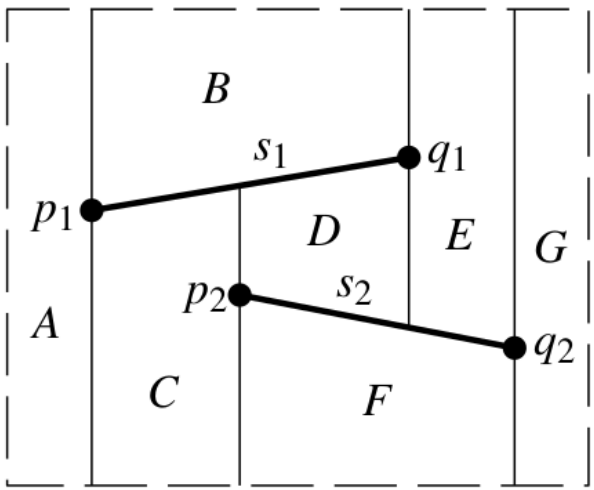
Plutôt que de représenter le chemin explicitement, on construit une représentation de l'espace de configuration libre $\mathcal{C}_f$, ce qui est efficace si l'espace de travail $\mathcal{C}$ ne change pas et si plusieurs chemins doivent être planifiés.

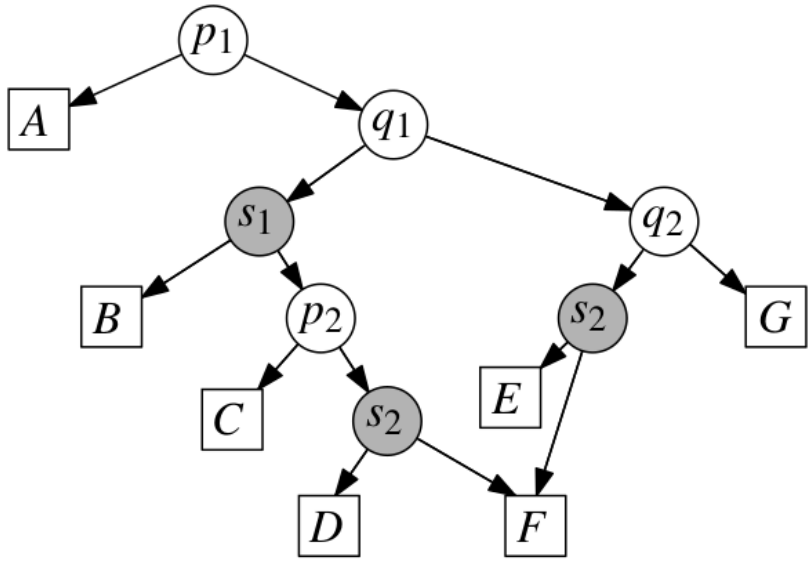
On restreint l'espace des solutions à une boîte englobante de l'ensemble des obstacles (ce qui revient à créer un obstacle infini à l'extérieur).

On peut construire une représentation de cet espace par une carte trapézoïdale.

Carte trapézoïdale

Exemple de deux segments s_1 et s_2 et l'arbre de recherche de la carte trapézoïdale. Les labels des sous-arbres droits (en blanc) sont les sommets terminaux, et ceux des sous-arbres gauches (en gris) réfèrent les segments. Les feuilles sont les trapèzes.



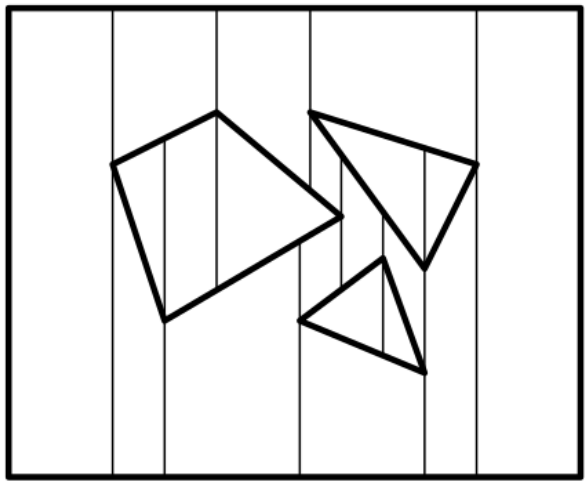


Représentation de l'espace de configuration libre

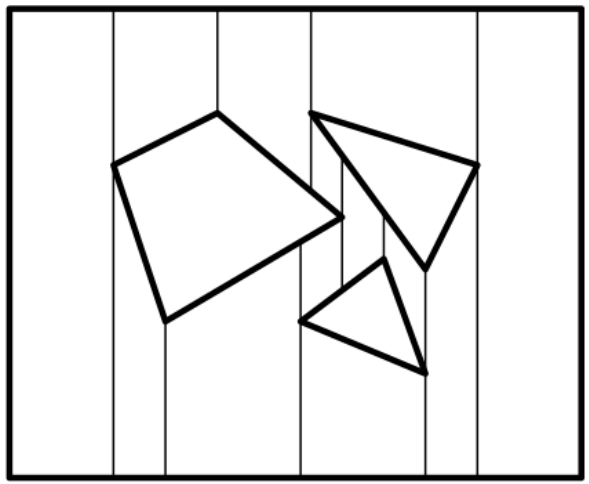
S est l'espace des obstacles (un ensemble de polygones *disjoints*).
On construit l'ensemble E des arêtes des polygones de S.

Une fois la carte trapézoïdale de ces segments construites, on supprime les trapèzes inscrits dans les polygones de S.

(a)



(b)

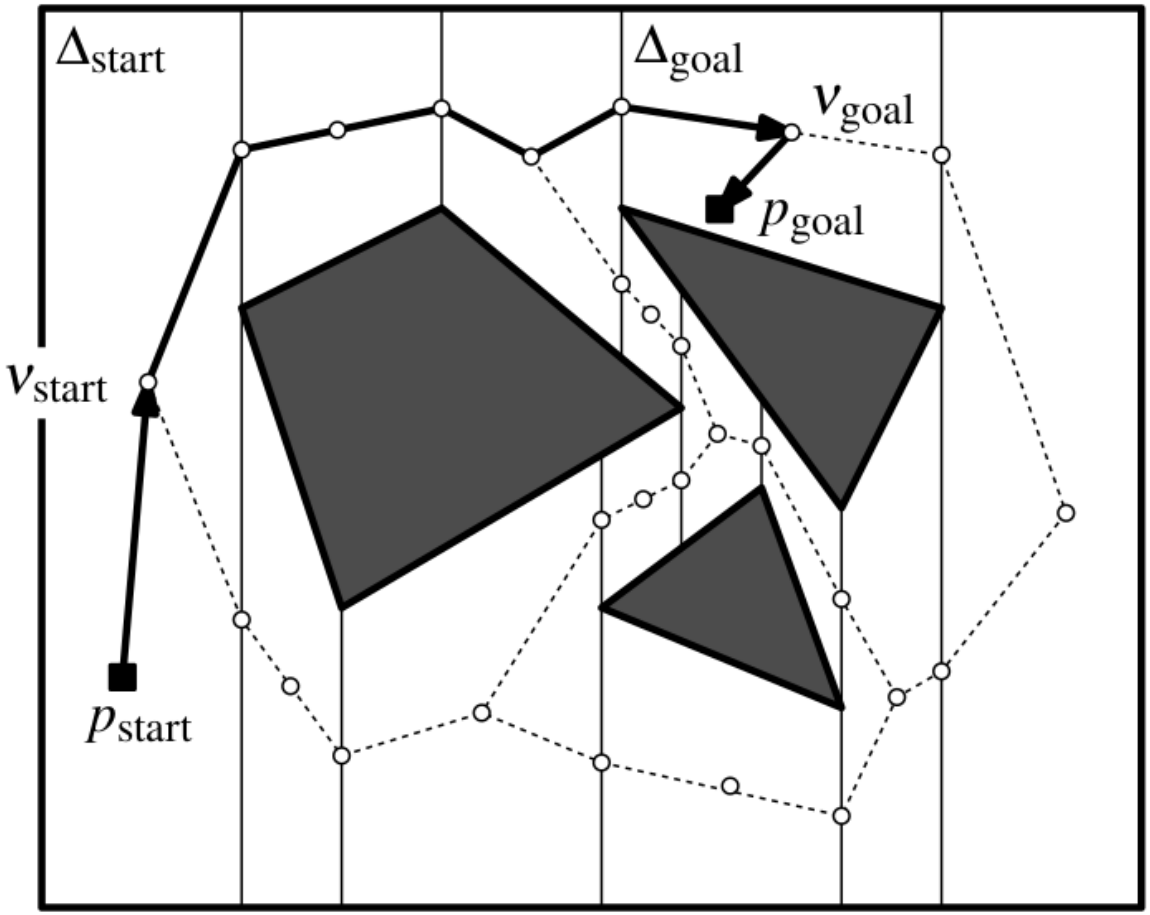


Cette carte permet de construire un parcours sous la forme d'un graphe planaire. À l'exception des trajets initiaux et finaux, la trajectoire suit ce graphe.

Construction :

1. on place un nœud n_t au centre de chaque trapèze,
2. on place un nœud n_v au centre de chaque extension verticale (segment commun à deux trapèzes),
3. on construit les arcs du chemin en appariant n_t et n_v s'ils appartiennent au même trapèze.

Ce graphe se construit en $O(n)$ en parcourant une liste double chaînée représentant la carte trapézoïdale.


Δ_{start}
Δ_{goal}
v_{goal}
p_{goal}
v_{start}
p_{start}