

Les Graphes

Introduction

Nicolas Delestre

Plan...

- 1 Introduction
- 2 TAD Graphe
- 3 Conception détaillée
 - Représentation par matrice
 - Représentation par liste
 - Représentation des étiquettes et valeurs
- 4 Quelques algorithmes sur les graphes
 - Parcours
 - Tri topologique
 - Chemin le plus courts : Dijkstra
 - A*
- 5 Conclusion

La structure de graphe 1 / 3

Définition

Un graphe G est composé de deux ensembles S et A :

- S est un ensemble fini d'éléments, appelés sommets (ou aussi nœuds)
- A est un ensemble fini d'éléments, appelés arcs

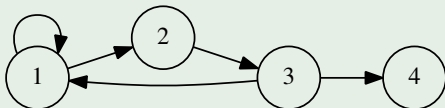
Notion d'arc

- un arc a est un élément du produit cartésien des sommets $S \times S$:
 - il est noté $a = (i, j)$ où i et j sont dans S
- A traduit une organisation relationnelle des sommets entre eux :
 - les organisations linéaires et hiérarchiques sont des cas particuliers

La structure de graphe 2 / 3

Concept d'orientation

- graphe non orienté : $(i, j) = (j, i)$
- graphe orienté : $(i, j) \neq (j, i)$



$G = (S, A) :$

- $S = \{1, 2, 3, 4\}$
- $A = \{(1, 1), (1, 2), (2, 3), (3, 1), (3, 4)\}$

Notation, terminologie 1 / 3

Notation

- si un graphe a n sommets, les sommets sont notés $1, 2, \dots, n$
- un arc orienté est noté (i, j) , où i et j sont pris dans l'intervalle $1..n$

Terminologie

Sommets adjacents Si (a, b) est un arc alors a et b sont des sommets adjacents

Arc incident Si (a, b) est un arc, alors :

- (a, b) est un arc incident extérieurement à a
- (a, b) est un arc incident intérieurement à b

Degrès d'un sommet nombre d'arcs incidents au sommet

Terminologie (suite)

Chemin séquence de sommets $(s_0, s_1, s_2, \dots, s_{p-1}, s_p)$ où $(s_0, s_1), (s_1, s_2), \dots, (s_{p-1}, s_p)$ sont des arcs

- la longueur d'un chemin (s_0, s_1, \dots, s_p) vaut p
- un chemin simple est un chemin qui ne comporte pas plusieurs fois le même arc
- un chemin élémentaire est un chemin qui ne passe pas plus d'une fois par le même sommet
- un circuit (ou cycle) est un chemin de la forme $(s_0, s_1, s_2, \dots, s_{p-1}, s_0)$

Graphe étiqueté graphe où une information est associée à chaque sommet

Graphe valué graphe où une valuation est associée à chaque arc

Terminologie (fin)

Graphe connexe Quelque soit s_1 et s_2 de S , il existe un chemin allant de s_1 à s_2

Graphe complet Quelque soit s_i de S , il existe un arc le reliant aux autres s_j ($j \neq i$) de S

Clique Ensemble de sommets deux à deux adjacents. Le sous graphe engendré est complet

Les TAD Graphe

- Il n'y a pas un TAD Graphe mais des TAD Graphe fonction :
 - de l'étiquetage ou pas des sommets
 - de la valuation ou pas des arcs
 - de l'orientation ou pas des arcs
- Il y a donc en tout 8 TAD Graphe possibles
- Nous allons en définir deux (orienté ou non orienté) que l'on pourra « réduire » en fonction de l'utilisation

TAD Graphe

Nom:	Graphe
Paramètre:	Sommet, Etiquette, Valeur
Utilise:	Booleen , Liste
Opérations:	<p>graphe: → Graphe</p> <p>ajouterSommet: Graphe × Sommet → Graphe</p> <p>ajouterArc: Graphe × Sommet × Sommet × → Graphe</p> <p>sommetPresent: Graphe × Sommet → Booleen</p> <p>arcPresent: Graphe × Sommet × Sommet → Booleen</p> <p>supprimerSommet: Graphe × Sommet → Graphe</p> <p>supprimerArc: Graphe × Sommet × Sommet → Graphe</p> <p>obtenirSommets: Graphe → Liste<Sommet></p> <p>obtenirSommetsAdjacents: Graphe × Sommet → Liste<Sommet></p> <p>possedeEtiquette: Graphe × Sommet → Booleen</p> <p>obtenirEtiquette: Graphe × Sommet → Etiquette</p> <p>fixerEtiquette: Graphe × Sommet × Etiquette → Graphe</p> <p>possedeValeur: Graphe × Sommet × Sommet → Booleen</p> <p>obtenirValeur: Graphe × Sommet × Sommet → Valeur</p> <p>fixerValeur: Graphe × Sommet × Sommet × Valeur → Graphe</p>
Préconditions:	<p>ajouterSommet(g,s): non sommetPresent(g,s)</p> <p>ajouterArc(g,s1,s2): sommetPresent(g,s1) et sommetPresent(g,s2) et non arcPresent(g,s1,s2)</p> <p>supprimerSommet(g,s): sommetPresent(g,s)</p> <p>supprimerArc(g,s1,s2): arcPresent(g,s1,s2)</p> <p>obtenirSommetsAdjacents(g,s): sommetPresent(g,s)</p> <p>possedeEtiquette(g,s): sommetPresent(g,s)</p> <p>obtenirEtiquette(g,s): sommetPresent(g,s) et possedeEtiquette(g,s)</p> <p>fixerEtiquette(g,s,l): sommetPresent(g,s)</p> <p>possedeValeur(g,s1,s2): arcPresent(g,s1,s2)</p> <p>obtenirValeur(g,s1,s2): arcPresent(g,s1,s2) et possedeValeur(g,s1,s2)</p> <p>fixerValeur(g,s1,s2,l): arcPresent(g,s1,s2)</p>

TAD GrapheOriente

Nom:	GrapheOriente
Paramètre:	Sommet, Etiquette, Valeur
Utilise:	Booleen , Liste
Opérations:	<p>grapheOriente: → GrapheOriente</p> <p>ajouterSommet: GrapheOriente × Sommet → GrapheOriente</p> <p>ajouterArc: GrapheOriente × Sommet × Sommet × → GrapheOriente</p> <p>sommetPresent: GrapheOriente × Sommet → Booleen</p> <p>arcPresent: GrapheOriente × Sommet × Sommet → Booleen</p> <p>supprimerSommet: GrapheOriente × Sommet → GrapheOriente</p> <p>supprimerArc: GrapheOriente × Sommet × Sommet → GrapheOriente</p> <p>obtenirSommets: GrapheOriente → Liste<Sommet></p> <p>obtenirPredecesseurs: GrapheOriente × Sommet → Liste<Sommet></p> <p>obtenirSuccesseurs: GrapheOriente × Sommet → Liste<Sommet></p> <p>possedeEtiquette: GrapheOriente × Sommet → Booleen</p> <p>obtenirEtiquette: GrapheOriente × Sommet → Etiquette</p> <p>fixerEtiquette: GrapheOriente × Sommet × Etiquette → GrapheOriente</p> <p>possedeValeur: GrapheOriente × Sommet × Sommet → Booleen</p> <p>obtenirValeur: GrapheOriente × Sommet × Sommet → Valeur</p> <p>fixerValeur: GrapheOriente × Sommet × Sommet × Valeur → GrapheOriente</p>
Préconditions:	<p>ajouterSommet(g,s): non sommetPresent(g,s)</p> <p>ajouterArc(g,s1,s2): non arcPresent(g,s)</p> <p>supprimerSommet(g,s): sommetPresent(g,s)</p> <p>supprimerArc(g,s1,s2): arcPresent(g,s1,s2)</p> <p>obtenirPredecesseurs(g,s): sommetPresent(g,s)</p> <p>obtenirSuccesseurs(g,s): sommetPresent(g,s)</p> <p>possedeEtiquette(g,s): sommetPresent(g,s)</p> <p>obtenirEtiquette(g,s): sommetPresent(g,s)</p> <p>fixerEtiquette(g,s,l): sommetPresent(g,s)</p> <p>possedeValeur(g,s1,s2): arcPresent(g,s1,s2)</p> <p>obtenirValeur(g,s1,s2): arcPresent(g,s1,s2)</p> <p>fixerValeur(g,s1,s2,l): arcPresent(g,s1,s2)</p>

Utilisation

Adapter ces deux TAD au besoin

- Besoin d'un graphe non orienté simple (pas d'étiquette, pas de valuation) dont les sommets seront identifiés par des naturels non nuls (interdiction d'utiliser les opérations obtenir/fixer Etiquettes/Valeurs) :
 - Graphe<Sommet=**NaturelNonNul**>
 - Graphe<**NaturelNonNul**>
- Besoin d'un graphe non orienté dont les sommets sont identifiés par des naturels non nuls et les arcs ont des valeurs de type réel (interdiction d'utiliser les opérations obtenir/fixer Etiquettes) :
 - Graphe<Sommet=**NaturelNonNul**,Valeur=**Reel**>
 - Graphe<**NaturelNonNul**,,Reel>

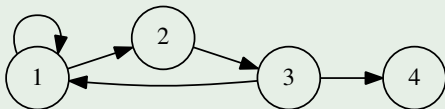
Représentation par une matrice d'adjacence 1 / 2

- Soit G un graphe à n sommets notés $1, 2, \dots, n$
- G est représenté par une matrice A $n \times n$ de booléens, tel que :
 - si (i, j) est un arc de G alors
 $A[i, j] \leftarrow \text{VRAI}$ (ou 1)
 - sinon
 $A[i, j] \leftarrow \text{FAUX}$ (ou 0)
- fini

Remarque

- Si la graphe est non orienté, la matrice est symétrique

Représentation par une matrice d'adjacence 2 / 2

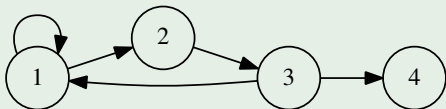


$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \end{matrix}$$

Représentation par une matrice d'incidence 1 / 2

- Soit G un graphe à n sommets et de p arcs
- G est représenté par une matrice A $n \times p$ de d'entiers, tel que :
 - pour tout arc $v_k = (i, j)$ avec $i \neq j$:
 - Si le graphe est orienté : $A[i, k] = 1$ et $A[j, k] = -1$
 - Si le graphe est non orienté : $A[i, k] = 1$ et $A[j, k] = 1$
 - pour tout arc $v_k = (i, i)$: $A[i, k] = 2$
 - 0 sinon

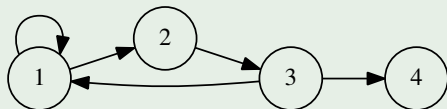
Représentation par une matrice d'incidence 2 / 2



$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{pmatrix} 2 & 1 & 0 & -1 & 0 \\ 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & -1 \end{pmatrix} \end{matrix}$$

Représentation par une liste d'adjacence

- G est représenté par une liste de sommets $ls = \langle s_1, \dots, s_n \rangle$
- Chaque élément s_i de la liste ls est une liste qui contient les sommets successeurs du sommet i dans le graphe G



$ls = ((1,2), (3), (1,4), ())$

Représentation des étiquettes et valeurs

- Les étiquettes à l'aide d'un dictionnaire dont les clés sont les sommets
- Les valeurs à l'aide d'un dictionnaire dont les clés sont des couples de sommets

Parcours d'un graphe 1 / 4

Parcours de tous les sommets

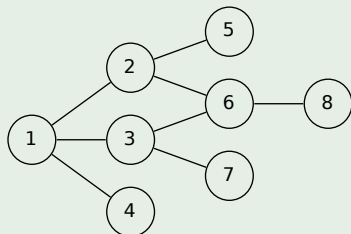
- visiter chaque sommet du graphe **une seule fois**
- appliquer un même traitement en chaque sommet

Parcours à partir d'un sommet s

- 1 parcours en profondeur
 - le principe consiste à descendre le plus "profond" dans le graphe à partir de s , en suivant un certain ordre, avant de revenir pour prendre une autre direction
- 2 parcours en largeur
 - le principe consiste à visiter les sommets situés à une distance 1 de s , puis ceux situés à une distance 2 de s , etc...

Parcours d'un graphe 2 / 4

Exemple



Parcours en profondeur à partir du sommet 1 :

- Parcours en profondeur : 1, 2, 5, 6, 8, 3, 7, 4
- Parcours en largeur : 1, 2, 3, 4, 5, 6, 7, 8

Parcours d'un graphe 3 / 4

Le type de parcours est fonction du TAD utilisé pour stocker les sommets à traiter :

- Pile \Rightarrow Parcours en profondeur
- File \Rightarrow Parcours en largeur

Souvent besoin d'ajouter une opération à ces TAD : `estPresent`

Dans tous les cas il faut un mécanisme pour éviter de boucler indéfiniment :

- Marquer les nœuds
- Lister les nœuds traités

Parcours d'un graphe 4 / 4

Exemple : Parcours en largeur

procédure parcoursEnLargeurIteratif (**E** g : Graphe<Sommet>, s : Sommet, traiter :
procédure(**E** Graphe<Sommet>,Sommet))

|**précondition(s)** sommetPresent(g,s)

Déclaration f : File<Sommet>
sCourant : Sommet

debut

f ← file()

enfiler(f,s)

tant que non estVide(f) **faire**

sCourant ← obtenirElement(f)

defiler(f)

marquer(sCourant)

traiter(g,sCourant)

pour chaque s' **de** obtenirSommetsAdjacents(g, sCourant)

si non estMarque(s') et non estPresent(f,s') **alors**

enfiler(f,s')

finsi

finpour

fintantque

fin

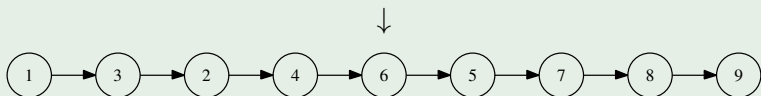
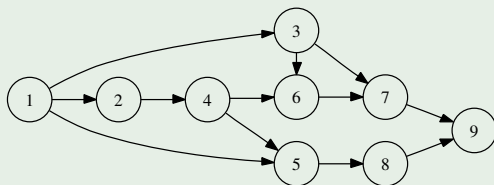
Tri topologique 1 / 2

Objectif

Un **graphe orienté acyclique avec une seule racine^a** peut représenter un ordre partiel entre les sommets.

Le tri topologique a pour but de créer un ordre total (une liste)

a. nœud qui ne possède pas de prédécesseur et qui peut atteindre tous les autres nœuds du graphe



Tri topologique 2 / 2

Algorithme

fonction triTopologique ($g : \text{GrapheOriente}\langle\text{Sommet}\rangle$, $s : \text{Sommet}$) : Liste<Sommet>

 | **précondition(s)** sommetPresent(g,s) et estVide(obtenirPredecesseurs(g,s))

Déclaration $l : \text{Liste}\langle\text{Sommet}\rangle$

$f : \text{File}\langle\text{Sommet}\rangle$

$sC : \text{Sommet}$

debut

$l \leftarrow \text{liste}()$

$f \leftarrow \text{file}()$

 enfiler(f,s)

tant que non estVide(f) **faire**

$sC \leftarrow \text{obtenirElement}(f)$

 defiler(f)

 insérer($l, \text{longueur}(l)+1, sC$)

 marquer(sC)

pour chaque s' **de** obtenirSuccesseurs(g,sC)

si non estMarque(s') et non estPresent(f,s') et tousPredecesseursMarques(g,s') **alors**

 enfiler(f,s')

fin

fin

fin

retourner l

fin

Exercices

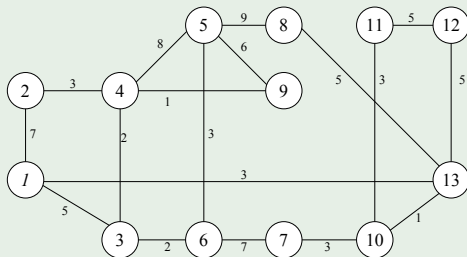
Donnez l'algorithme de la fonction tousPredecesseursMarques

Chemin le plus cours

Objectif

Soit un graphe étiqueté et valué par des nombres. Quel est le chemin le plus court pour aller d'un nœud A à un nœud B ?

Plusieurs algorithmes : **Dijkstra**, Floyd, Bellman-Ford, etc.



On suppose posséder la fonction :

- **fonction** obtenirCout (g : Graphe, s1,s2 : Sommet) : ReelPositif

Algorithme de Dijkstra 1 / 6

Principe ^a

a. Extrait de Wikipédia

Soit s_{deb} le nœud (ou sommet) source et s_{fin} le nœud destination du graphe G . L'algorithme fonctionne en construisant un sous-graphe P tel que la distance entre un sommet s de P depuis s_{deb} est connue pour être un minimum dans G . Initialement P contient simplement le nœud s_{deb} isolé, et la distance de s_{deb} à lui-même vaut 0. Des arcs sont ajoutés à P à chaque étape :

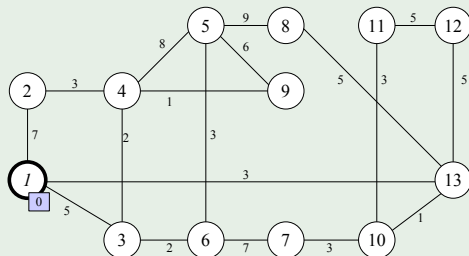
- 1 en identifiant **toutes les arêtes** $a_j \in A$ tel que $A = \{(s_{i1}, s_{i2}) \in P \times G - P\}$
- 2 en choisissant **l'arête** $a_j \in A$ avec $a_j = (s_{j1}, s_{j2})$ qui donne la distance minimum de s_{deb} à s_{j2} .

L'algorithme se termine soit quand P devient un arbre couvrant de G , soit quand $s_{fin} \in P$.

Ne fonctionne qu'avec des coûts à **valeurs positives**

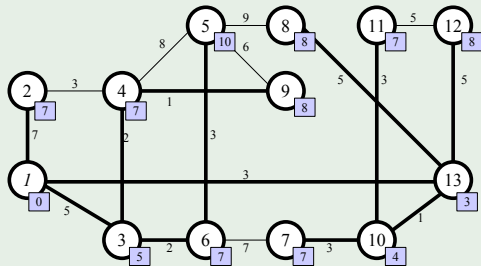
Algorithme de Dijkstra 2 / 6

Exemple avec $s_{deb} = 1$



Algorithme de Dijkstra 3 / 6

Exemple



Algorithme de Dijkstra 4 / 6

On suppose posséder les fonctions et procédures suivantes

- fonction** `arbreInitial` (`s` : Sommet) : `Arbre<Sommet>`
qui crée un arbre possédant uniquement le noeud `s`
- fonction** `arcsEntreArbreEtGraphe` (`a` : `Arbre<Sommet>`, `g` : `Graphe<Sommet,,ReelPositif>`) : `Liste<Liste<Sommet>>`
qui retourne la liste des arcs (liste de deux sommets) dont le premier sommet appartient à `a` et le second sommet appartient à `g` et n'appartient pas à `a`
- fonction** `arcMinimal` (`g` : `Graphe<Sommet,,ReelPositif>`, `arcs` : `Liste<Liste<Sommet>>`, `cout` : `Dictionnaire<Sommet, ReelPositif>`) : `Sommet, Sommet, ReelPositif`

\lfloor **précondition**(`s`) `non estVide(arcs) et $\forall i \in 1..longueur(arcs)$`
`estPresent(cout,obtenirElement(obtenirElement(arcs,i),1))`

qui retourne, parmi les arcs, l'arc (sommet source, sommet destination) dont le sommet destination est le plus proche (au sens du dictionnaire de cout) des sommets de `a` ainsi que le coût supplémentaire pour l'atteindre
- procédure** `ajouterCommeFils` (`E/S a` : `Arbre<Sommet>`, `E sommetPere`, `sommetFils` : `Sommet`)

\lfloor **précondition**(`s`) `estPresent(a, sommetPere)`

qui ajoute un nouveau noeud, à l'arbre `a`, contenant `sommetFils` qui sera fils du noeud contenant `sommetPere`

Algorithme de Dijkstra 5 / 6

Algorithme (étude complète du graphe)

fonction dijkstra (g : Graphe<Sommet,,ReelPositif>, s : Sommet) : Arbre<Sommet>, Dictionnaire<Sommet,ReelPositif>

| **précondition(s)** sommetPresent(g,s)

Déclaration arbreRecouvrant : Arbre<Sommet>, cout : Dictionnaire<Sommet,ReelPositif>
 l : Liste<Liste<Sommet>>, c : ReelPositif
 sommetDeA, sommetAAjouter : Sommet

debut

arbreRecouvrant ← arbreInitial(s)

cout ← dictionnaire()

ajouter(cout,s,0)

l ← arcsEntreArbreEtGraphe(g,arbreRecouvrant)

tant que non estVide(l) **faire**

 sommetDeA,sommetAAjouter,c ← arcMinimal(g,l,cout)

 ajouter(cout,

 sommetAAjouter,

 obtenirValeur(cout,sommetDeA)+c

)

 ajouterCommeFils(arbreRecouvrant,sommetDeA,sommetAAjouter)

 l ← arcsEntreArbreEtGraphe(g,arbreRecouvrant)

fin tant que

retourner arbreRecouvrant, cout

fin

Algorithme de Dijkstra 6 / 6

Exercices

Donnez les algorithmes des fonctions/procédures suivantes :

- **fonction** arbreInitial (s : Sommet) : Arbre<Sommet>
- **fonction** arcsEntreArbreEtGraphe (a : Arbre<Sommet>, g : Graphe<Sommet,,**ReelPositif**>) : Liste<Liste<Sommet>>
- **fonction** arcMinimal (g : Graphe<Sommet,,**ReelPositif**>, arcs : Liste<Liste<Sommet>>, cout : Dictionnaire<Sommet, **ReelPositif**>) : Sommet, Sommet, **ReelPositif**
 - | **précondition(s)** non estVide(arcs) et $\forall i \in 1..longueur(arcs)$
estPresent(cout,obtenirElement(obtenirElement(arcs,i),1))
- **procédure** ajouterCommeFils (**E/S** a : Arbre<Sommet>, **E** sommetPere, sommetFils : Sommet)
 - | **précondition(s)** estPresent(a , sommetPere)

A* 1 / 6

Constat

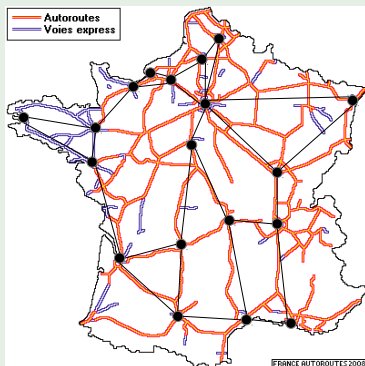
- L'algorithme de Dijkstra va donner le chemin le plus court car il explore tous les sommets du graphe (où jusqu'à trouver le sommet recherché)
- La complexité de cet algorithme est naïvement en $O(n^2)$, mais il peut être réduit à du $n + m$ avec des structures de données performantes (cf. articles de wikipédia et interstice^a)
- Lorsque le nombre de sommets et d'arcs grandit (par exemple dans les logiciels d'aide à la conduite), ne peut-on pas éviter de tout explorer ?

a. <https://interstices.info/le-plus-court-chemin/>

A* 2 / 6

Exemple

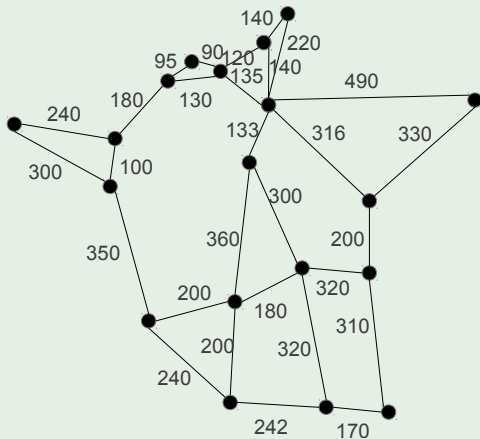
- Quel est le plus court chemin entre Rouen et Montpellier ?



<http://www.lecartographe.fr>

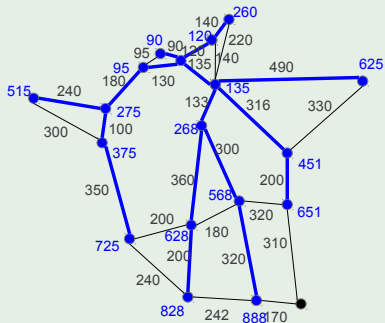
A* 3 / 6

Le graphe correspondant



A* 4 / 6

Utilisation de l'algorithme de Dijkstra



Remarque

Pratiquement tout le graphe est exploré, même des destinations que l'on « sait » non utiles (nord/ouest/est de la France)

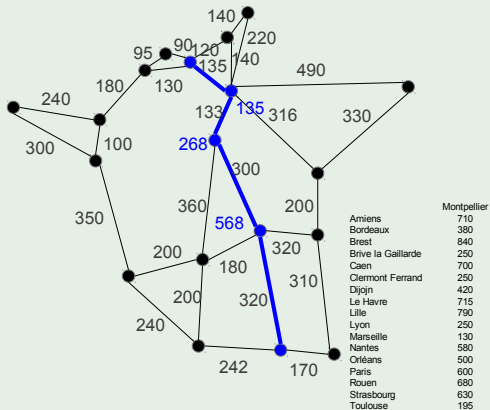
A* 5 / 6

Principe de l'algorithme A*

- Utiliser une fonction heuristique h qui estime le coût d'un sommet, c'est-à-dire à quelle distance il est de la solution
 - Dans l'exemple précédent, cela pourrait être la distance à vol d'oiseau entre une ville et Montpellier
- Modifier la fonction `arcMinimal` de façon en prendre en compte le coût depuis la source mais aussi l'heuristique h
 - **fonction** `arcMinimal` (g : Graphe<Sommet,,**ReelPositif**>, arcs : Liste<Liste<Sommet>>, cout : Dictionnaire<Sommet, **ReelPositif**>, h : fonction(Sommet) : **ReelPositif**) : Sommet, Sommet, **ReelPositif**
 - [**précondition(s)** non estVide(arcs) et $\forall i \in 1..longueur(arcs)$ estPresent(cout,obtenirElement(obtenirElement(arcs,i),1))
 - Si $h(s) = 0$ quelque soit s , alors l'algorithme A* est équivalent à l'algorithme de Dijkstra
 - Si h est mal choisie, il se peut que A* ne donne pas la meilleure solution

A* 6 / 6

Utilisation de l'algorithme A*



Conclusion

Le graphe est une structure de données très utilisée :

- Il permet de représenter beaucoup d'objets de la vie réelle, avec des problèmes d'optimisation (Cf. le cours de RO, ITI4)
 - par exemple représentation d'une carte routière, d'un réseau électrique, d'Internet
- Il est utilisé dans beaucoup de domaines scientifiques :
 - Théorie des langages (Cf. le cours sur la compilation, ITI 3.2)
 - Le Web sémantique
 - *Machine learning* (Réseaux de neurones, Réseaux bayésien)
- Il existe beaucoup d'algorithmes à étudier non présentés dans cette introduction : découverte de clique, algorithmes sur les flux, isomorphisme de graphe, distance entre graphes, etc.