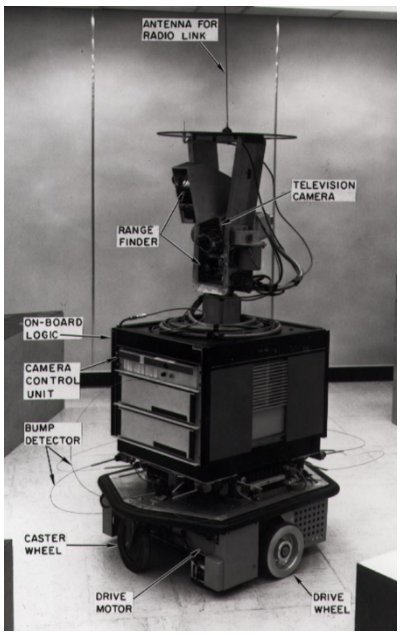
Cours 23 – Bases des graphes – Partie 4– Algorithme A*

Au dernier cours, nous avons introduit l'algorithme de Dijkstra, permettant la recherche du chemin optimal dans un graphe pondéré avec des poids positifs. Les hypothèses d'application de l'algorithme ***A**** (prononcer «A star») sont les mêmes que pour Dijskstra, avec en plus (plus restrictif) la présence d'une heuristique, voir la suite.

A / Objectif de l'algorithme



L'algorithme est ***publié en 1968 par Nils Nilsonn et al*** par des chercheurs travaillant à l'optimisation de la trajectoire d'un robot (*Shakey,* en photo ci-contre) dans une pièce comportant des obstacles.

Cet algorithme s'appuie sur celui de *Dijkstra*, en apportant une modification majeure qui le fait rentrer dans la catégorie de ***l'intelligence artificielle*** :

- *Dijkstra* va suivre une logique de découverte de graphe (donc une cartographie du terrain) en largeur, c'est-à-dire de façon concentrique par rapport au point de départ du robot.
- L'algorithme *A** propose de s'appuyer sur une intuition pour guider le robot, afin de prioriser des itinéraires qui semblent plus prometteurs, et donc de parcourir une portion moindre du graphe. Cette « intuition » s'appelle une ***heuristique***. Elle doit être construite par les chercheurs en amont du lancement du programme, soit par recherche et connaissance du milieu, soit par une intelligence artificielle.

Par la suite, les avancées techniques permettront à un robot de construire par apprentissage sa propre heuristique, c'est-à-dire dans notre exemple cartographier une zone pour converger plus rapidement vers une solution idéale la prochaine fois, on voit ici apparaître la notion de ***machine learning***, d'où la classification de cet algorithme dans les pères fondateurs de l'intelligence artificielle.

Parmi les applications classiques de cet algorithme (ou de ses variantes), on peut citer notamment :

- Les recherches de meilleure trajectoire : GPS, itinéraire d'un bot dans un jeu vidéo …
- L'identification d'analyse syntaxique utilisant des structures grammaticales stochastiques :

  Traducteurs automatiques, recherche de structures dans l'ADN/ARN (lecture génomes…)

B / Heuristique

Tout comme l'algorithme de *Dijkstra*, l'algorithme *A** a besoin de connaître le nœud initial N_D (celui d'où initialise l'algorithme) dans le graphe, mais en plus, il faut lui communiquer nœud d'arrivée N_A.

L'heuristique est une application h: $\begin{cases} S \to \mathbb{R}_+ \\ S_i \to h(S_i) \end{cases}$ qui à tout sommet du graphe associe une estimation du coût restant minimal permettant de passer du sommet S_i au sommet d'arrivée N_A. Déterminer l'heuristique est l'une des parties les plus complexes de la modélisation.

Exemples : On a toujours $h(N_A) = 0$

1 - On dispose d'une carte où les nœuds sont des points de la carte. Une heuristique classique est de prendre $h(S_i) = \|\overrightarrow{S_iN_A}\|$ la norme Euclidienne entre chaque nœud et le nœud d'arrivée.

2 – Si les nœuds sont des mots (ou des structures d'ADN), l'heuristique peut être une estimation du coût qui permet de passer du mot S_i au mot N_A (ex : coût = +1 à chaque modification, suppression ou ajout d'une lettre)

C / Entre Dijkstra et A* : différence sur la priorisation des nœuds

Dans l'algorithme de *Dijkstra*, nous avions introduit une application $D:\begin{cases} S \to \mathbb{R}_+ \\ S_i \to D(S_i) \end{cases}$, tenue à jour au fur et à mesure de l'algorithme, qui à tout sommet S_i du graphe associait la longueur du plus court chemin alors connu, allant du nœud initial N_D au sommet S_i.

→ A chaque itération, le nœud visité par *Dijkstra* est, parmi ceux déjà ajoutés à la liste des nœuds `A_visiter` :

le nœud le plus proche de N_D : $\text{Argmin}(D)$

Or ceci ne présume absolument pas que le nœud choisi soit le plus proche de N_A. L'idée de *A** est de choisir le nœud le plus proche de N_A. Pour cela, on pose $f:\begin{cases} S \to \mathbb{R}_+ \\ S_i \to D(S_i) + h(S_i) \end{cases}$

longueur chemin le + court $N_D \to S_i$ (connu actuellement) modifié au cours de l'algo.

longueur estimée $S_i \to N_A$ (chemin restant) fixé avant l'algo. non modifié pendant

→ A chaque itération, le nœud exploré sera alors celui, parmi ceux déjà ajoutés à la liste des nœuds `A_visiter` :

estimé le plus proche de N_A : $\text{Argmin}(f)$

Remarque: par habitude, pour l'algorithme *A**, l'application D est plutôt notée g, et on note alors $f = g + h$. Cette fonction f est appelée ***fonction d'évaluation***. Pour un sommet $S_i \in S$, $f(S_i)$ est une estimation du coût total du meilleur chemin allant de N_D à N_A en passant par S_i. Le coût réel n'est pas connu, puisque c'est ce qu'on cherche.

Lorsqu'on visite un nœud, il passe de la liste `A_visiter` (appelée liste `Open` dans la littérature anglo-saxonne) vers la liste `Deja_vus` (appelée liste `Close`). Les deux algorithmes s'appuient sur un **parcours en largeur,** c'est-à-dire qu'on attend d'avoir traité tous les voisins (ajout à `Deja_vus`, relaxation des arcs, …) avant de passer au nœud suivant.

Une différence importante est que pour *Dijkstra,* on a vu que lorsqu'un S_i nœud est sélectionné pour être visité, on peut figer $D(S_i)$ comme étant la plus courte distance entre N_D et S_i.
En revanche, dans le cas général de *A** (càd sauf cas particulier d'une heuristique **cohérente**, voir la suite), ceci n'est pas vrai. On peut donc avoir à modifier $g(S_i)$ alors que le nœud S_i fait partie de `Deja_vus`.

D / Relaxation d'un arc / d'une arête

Soit un nœud $S_{\text{pp}} \in S$ sélectionné comme étant $\text{Argmin}(f)$, dont on souhaite relâcher les arcs voisins. Soit $S_V \in S$ un autre nœud voisin de S_{pp}. Le relâchement de l'arête $(S_{\text{pp}}, S_V) \in A$ ressemble énormément à celui de l'algorithme de *Dijkstra*.

La seule différence réside dans le fait que pour *Dijkstra* une initialisation de D a été nécessaire (on a initialisé, pour tous les sommets du graphe, $\begin{cases} D(N_D) = 0 \\ \forall S_i \neq N_D,\ D(S_i) = \infty \end{cases}$. En revanche pour *A** l'initialisation n'est pas nécessaire, ainsi pour les sommets $S_{\text{pas vu}}$ pas encore vus lors du parcours, $g(S_{\text{pas vu}})$ n'est pas défini.

La distance entre N_D et S_{pp} alors connue à cette itération de l'algorithme est : $g(S_{\text{pp}})$.

$w(S_{\text{pp}} \to S_V)$ est le poids de l'arc ou de l'arête menant de S_{pp} vers S_V.

On découvre un nouveau chemin menant de N_D à S_V, de longueur $g_{\text{new}} = g(S_{\text{pp}}) + w(S_{\text{pp}} \to S_V)$

- Si $g(S_V)$ n'est pas défini, alors on pose $g(S_V) \longleftarrow g_{\text{new}}$ et le parent $P(S_V) \longleftarrow S_{\text{pp}}$
- Sinon, alors on compare si $g_{\text{new}} \leq g(S_V)$ # Si non, on ne fait rien, ancien chemin plus optimal !
  - auquel cas : on écrase $g(S_V) \longleftarrow g_{\text{new}}$ (et on met à jour le parent $P(S_V) \longleftarrow S_{\text{pp}}$)

E / Illustration avec un cas général – heuristique non cohérente

Crédits : l'exemple ci-dessous, ainsi que la plupart des définitions de ce cours, sont tirés de la capsule vidéo de *Hugo Larochelle* et *Froduald Kabanza* – professeurs à l'Université de Sherbrooke, Canada.

À chaque itération, on prend pour convention de trier la liste `A_visiter` par ordre décroissant de $f(S_i)$, et de visiter l'élément le plus à droite à l'itération suivante.

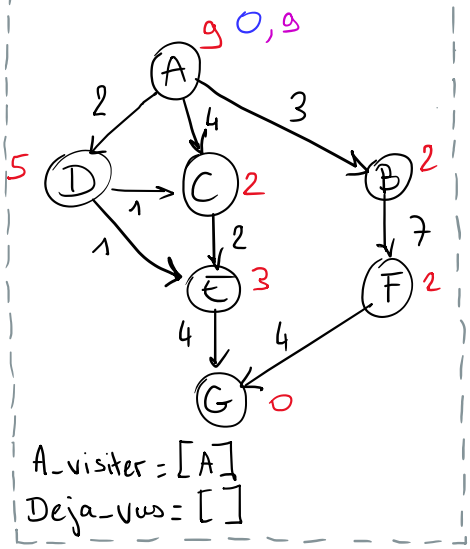
Considérons le graphe ci-dessous, avec ***A*** comme nœud initial, et ***G*** le nœud d'arrivée.

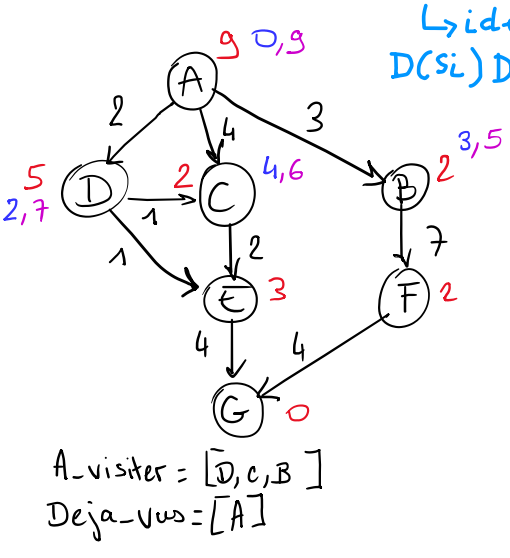
Code couleur : pour chaque nœud S_i , on étiquette

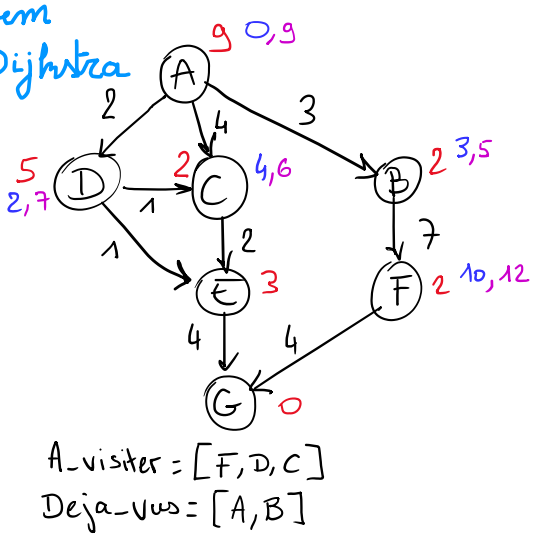
$f(S_i) = g(S_i) + h(S_i)$ — heuristique déjà donnée

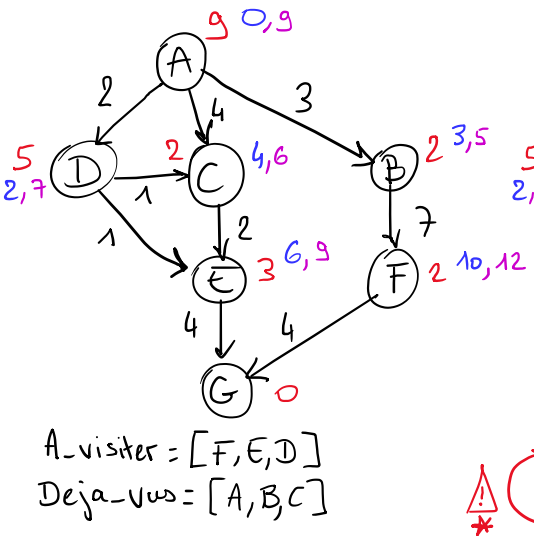
↳ idem D(Si) Dijkstra

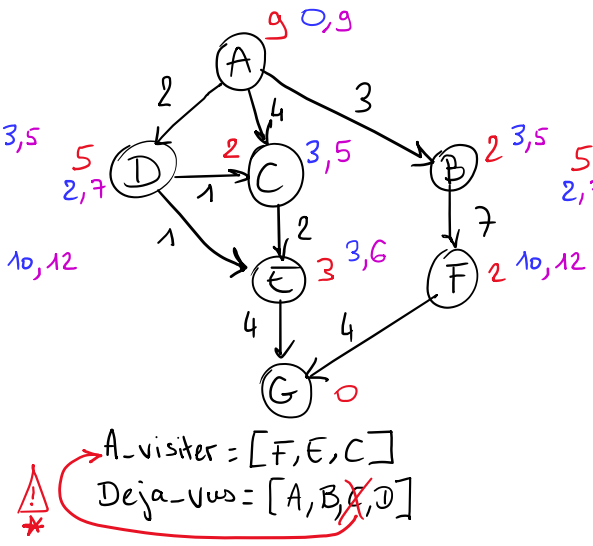
Initialisation

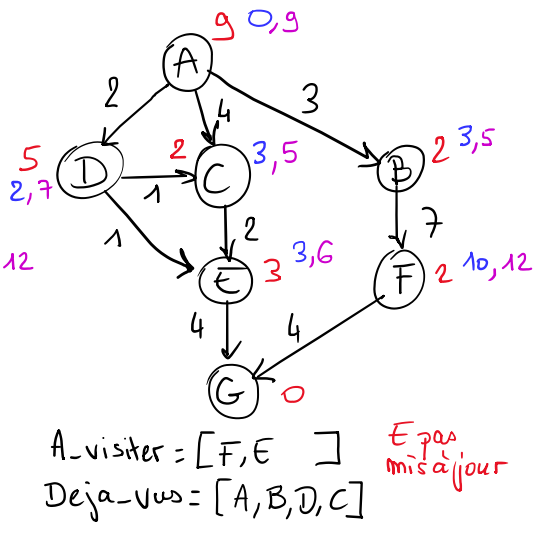


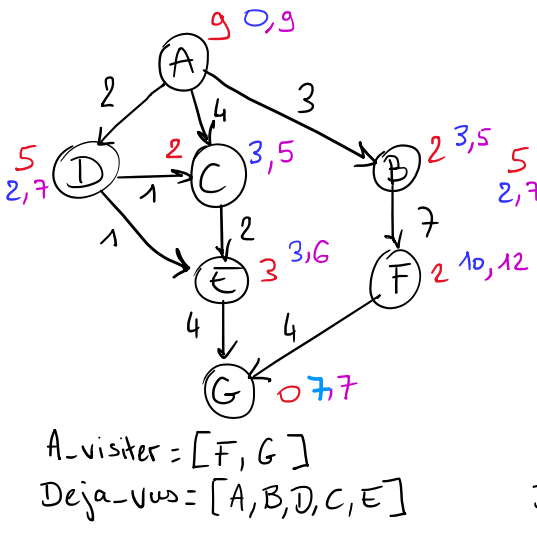


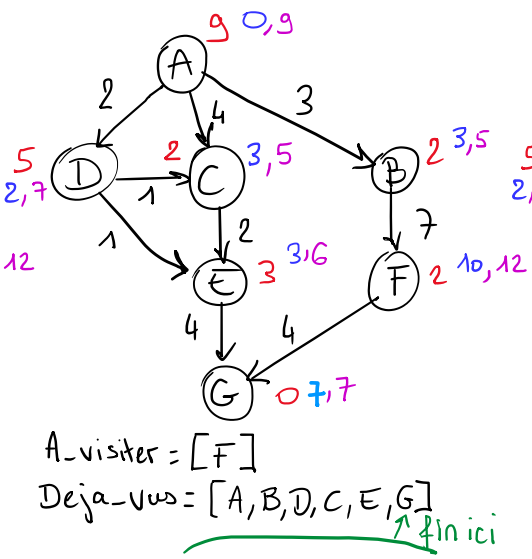


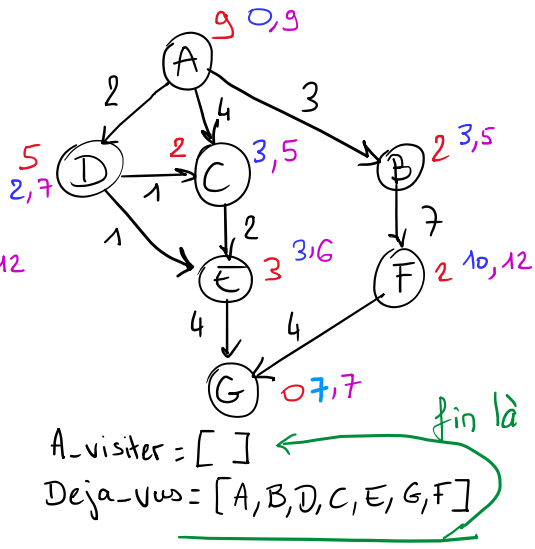












Remarque: pour certaines heuristiques, arrêt algorithme dès que N_A est visité, sinon cas général, dès que `A_visiter` vide.

F / Deux propriétés importantes de l'heuristique

Pour un graphe fini, pondéré avec des poids positifs, on peut montrer que l'algorithme termine toujours.

F. 1 / Heuristique admissible

On dit qu'une heuristique h est **admissible** ssi pour tout nœud S_i, $h(S_i)$ est un estimé $\leq w(S_i, N_A)$

Longueur du chemin le plus court qu'on va trouver en réalité

C'est à dire si l'heuristique a tendance à sous-évaluer la distance restante.

→ C'est notamment le cas de la norme Euclidienne, puisque quel que soit le chemin que l'on empruntera pour se rendre de $S_i \to N_A$, $\|\overrightarrow{S_iN_A}\| = h(S_i) \leq \sum_{\text{chemin}} \|\overrightarrow{S_iS_{i+1}}\| = \widehat{S_iN_A}$

Propriété : si l'heuristique h est **admissible**, l'algorithme *A** donne le chemin optimal (le plus court)

Remarque: parfois, il peut être difficile de trouver une heuristique admissible, ou on peut préférer une heuristique non admissible qui permette de faire converger l'algorithme beaucoup plus rapidement. Il faut alors étudier « à quel point » la solution proposée par l'algorithme est proche du chemin optimal (compromis entre différence de temps de calcul, et différence de temps de parcours du trajet par le robot, par exemple).

F. 2 / Heuristique cohérente

On dit qu'une heuristique h est **cohérente** (on dit aussi **monotone**, mais rigoureusement c'est f qui est monotone) ssi pour tout nœud $S_i \in S$ et tout nœud $S_V \in S$ voisin de S_i , $h(S_i) \leq h(S_V) + w(S_i, S_V)$

→ C'est notamment le cas de la norme Euclidienne, puisque $\|\overrightarrow{S_iN_A}\| \leq \|\overrightarrow{S_iS_V}\| + \|\overrightarrow{S_VN_A}\|$

Propriété : si l'heuristique h est **cohérente**, alors :

- h est aussi admissible
- lorsque l'algorithme *A** sélectionne un nœud de la liste `A_visiter` (donc celui de f le plus petit), alors il n'existe pas de chemin plus court menant à ce nœud.
  → on peut donc être certains qu'un élément passé dans `Déjà_vus` n'aura jamais à être supprimé de cette liste pour être replacé dans `A_visiter` (*cf.* exemple vu en partie **E**).
- l'algorithme se termine dès que l'on s'apprête à visiter l'objectif N_A

Remarque: dans le TP (et, selon moi, aux concours), on se placera sous cette hypothèse d'une heuristique **cohérente**, qui sera donnée.

G / Remarques d'ouverture

G.1 / Dijkstra peut être vu comme un cas particulier de *A**

Si l'on prend comme heuristique $\forall S_i \in S$, $h(S_i) = 0$, alors $f(S_i) = g(S_i)$ → la priorité ne se fait que sur la distance à N_D, et l'on retrouve *Dijkstra*. D'ailleurs, si on prend comme pondération entre tout nœud S_i et ses voisins S_V : $w(S_i, S_V) = 1$ → alors on retrouve le parcours en largeur (plus court chemin en nombre d'arêtes parcourues).

G.2 / Pondération de *f* : algorithme *WA** (Weighted *A**)

Posons une pondération x tell que $f(S_i) = xg(S_i) + (1-x)h(S_i)$.
L'algorithme ainsi obtenu est dit *WA**. On peut alors « doser » le poids relatif donné à *g* et à *h*. Voyons deux extrêmes :

- $x = 1$ c'est-à-dire $f(S_i) = g(S_i)$: l'algorithme est alors équivalent à *Dijkstra* (on privilégie les parcours concentriques)
  → beaucoup de nœuds visités (lent), mais converge toujours vers la solution optimale.
- $x = 0$ c'est-à-dire $f(S_i) = h(S_i)$: l'algorithme devient de type *glouton*.
  → converge très rapidement vers une solution, qui n'est pas forcément optimale (grande importance de h).