Module Master M2

Systèmes complexes
Chapitre Zéro : Rappels de quelques notions

Présenté par : Prof. Kholladi Mohamed-Khireddine
Département d'Informatique
Facultés des Sciences Exactes
Université Echahid Hamma Lakhdar d'El Oued
Tél. 0770314924
Email. kholladi@univ-eloued.dz et kholladi@yahoo.fr
Site Web. www.univ-eloued.dz
http://kholladi.doomby.com/ et http://kholladi.e-monsite.com/

0 – Rappels de quelques notions

0.0 - Sommaire

1. Rappel 01 - Sur les notions de graphe, de réseau et de flot
2. Rappel 02 - Algorithme de Dijkstra
3. Rappel 03 - Résolution des problèmes par certaines méthodes d'exploration
4. Rappel 04 – Sur la notion d'agents intelligents
5. Rappel 05 – Exploration par le parcours d'un graphe
6. Rappel 06 – Exploration par le parcours en largeur d'un graphe
7. Rappel 07 – Recherche des plus courts chemins dans un graphe

0.1 – Introduction

Avant d'aborder le cours sur les systèmes complexes et même de raisonnement et décisions, il est nécessaire que les étudiants connaissance des de théorie de graphe et des systèmes multi-agents. Il est évident qu'ils puissent aussi connaître l'algorithme de Dijkstra et la résolution des problèmes par certaines méthodes d'exploration avant d'aborder les méthodes d'exploration des systèmes complexes.

0.2 – Rappel 01 sur les notions de graphe, de réseau et de flot

0.2.1 - Qu'est-ce qu'un graphe ?

Un graphe est un ensemble de sommets (ou de nœuds) et un ensemble d'arêtes (ou d'arcs) comme sur la figure 0.1.

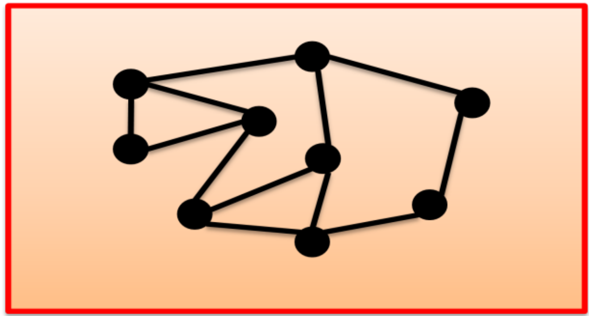


Figure 0.1 – Graphe

On note qu'il y a zéro ou une arête entre chaque paire de sommets.

0.2.2 - Dessin d'un graphe ?

Soient les sommets suivants : A, B, C, D et les arêtes suivantes : AD, BD, BC, CD. On va dessiner trois graphes comme sur la figure 0.2.

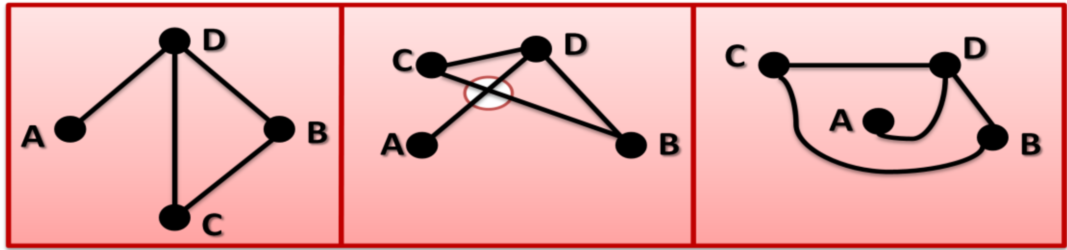


Figure 0.2 – Dessin d'un graphe

On note que l'on a toujours le même graphe sur les trois dessins.

0.2.3 - À quoi sert un graphe ?

Un graphe sert à la représentation des relations entre des éléments comme la figure 0.3. Cela peut être par exemple un réseau social d'amitié, de coopération, de relations scientifiques, de relations culturelles, de relations religieuses, de relations artistiques, etc. Les relations peuvent être professionnels réels ou virtuels.

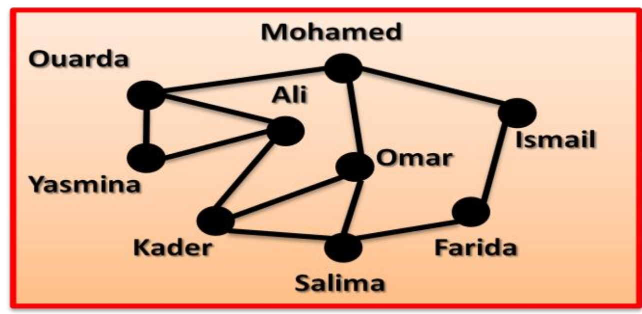


Figure 0.3 – Exemple de relations d'amitié entre des éléments

On note que Mohamed a un lien d'amitié avec Ouarda et que Kader n'a aucun lien d'amitié ace Yasmina. Cela peut être aussi par exemple un réseau de transport Aérien, maritime ou fluvial, routier ou ferroviaire comme sur la figure 0.4.

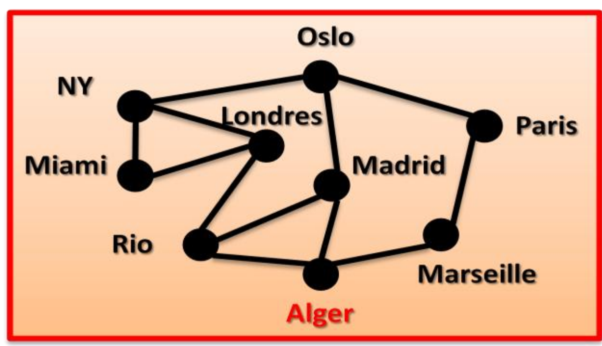


Figure 0.4 – Exemple de réseau de transport

On note qu'Alger a une liaison maritime directe avec Marseille et Alger n'a pas de liaison maritime directe avec Paris. Cela peut être aussi par exemple un réseau informatique qui relie des machines physiques. Un des plus grands réseaux mondiaux, qui relie les serveurs informatiques du monde entier, c'est bien l'Internet comme sur la figure 0.5.

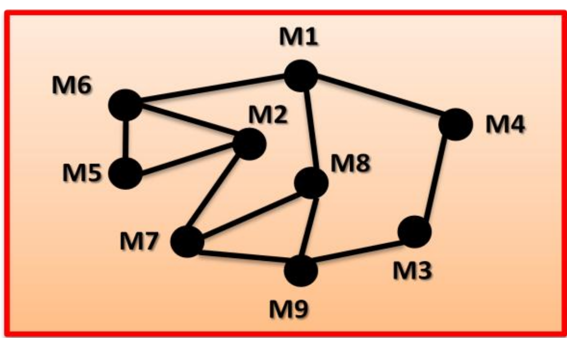


Figure 0.5 – Exemple d'un réseau de machines informatiques

Les applications de la théorie des graphes sont aujourd'hui énormes et phénoménales tant au plan civil que militaire, qui sont :

- Aide à la décision,
- Stratégie (militaire, jeux, économique, management, etc.)
- Optimisation (plus court chemin, GPS, coût minimal),
- Réseaux de transports :
  - Chemins de fer,
  - Métropolitain,
  - Lignes aériennes, maritimes, ferroviaires et routière),

- électricité, gaz, oléoducs (transport de l'énergie), eau,
- Internet (réseau de l'information),
- Ports, aéroports, Chantiers navales,
- Ordonnancement des tâches,
- Etc.

0.2.4 – Où sont les graphes ?

On les trouve dans :

- Les réseaux
  - Sociaux
  - Routiers, ferroviaires, aériens, maritimes et fluviaux
  - De distribution de biens, de médicaments
  - D'énergie (électricité, gaz, pétrole, et eau)
- Les mathématiques
- L'informatique
  - Relations entre des données, des objets
  - Routage dans les réseaux
- Etc.

On note aussi qu'il y a beaucoup de problèmes qui ne sont pas encore résolus.

0.2.5 - Graphes extrêmes

Ce sont des graphes complets comme sur la figure 0.6.

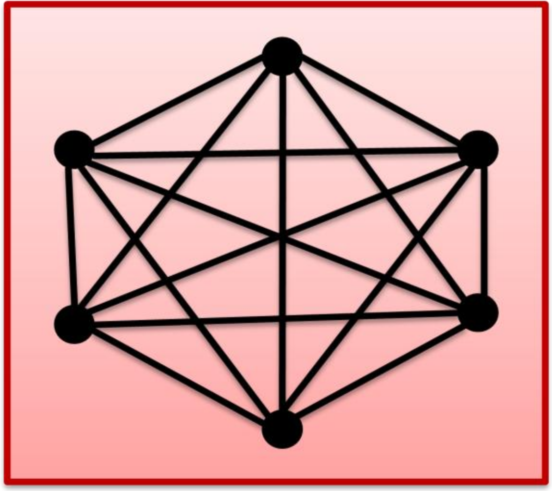


Figure 0.6 – Graphe complet

On met toutes les arêtes possibles entre les sommets. C'est-à-dire si on a un graphe avec n sommets, on aura (n-1) arêtes partantes d'un sommet quelconque du graphe et ce pour tous les sommets du graphe.

0.2.6 - Notion de graphe symétrique ou antisymétrique

Ci-dessous sur la figure 0.7, on peut parler de l'arête (A, B) mais pas de (B, A). Lorsque (A, B) ∈ U entraîne (B, A) ∈ U, le graphe est dit symétrique. Un graphe non orienté s'interprète comme un graphe symétrique. Lorsque (A, B) ∉ U entraîne (B, A) ∈ U, le graphe est dit antisymétrique.

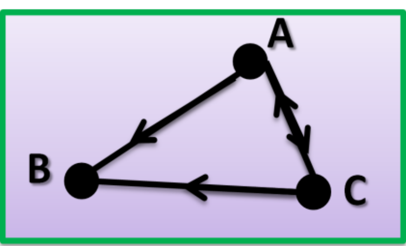
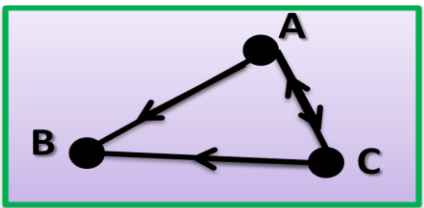


Figure 0.7 – le graphe G(S, U)

Dans un graphe défini comme étant orienté, on doit faire apparaître les sens de parcours sur chaque arête. Sauf indication contraire, un graphe sera considéré comme non orienté et les arêtes peuvent être parcourues dans les deux sens. Ce type de graphe peut se présenter dans des problèmes élémentaires de type combinatoire. La plupart des applications de la théorie graphes, (problèmes d'optimisation, communications, trafics aériens, etc.) conduisent à des graphes orientés. Dans le cas d'un graphe orienté, le parcours entre deux sommets peut être à double sens. On pourrait coder ce double sens comme sur le graphe en bas et parler de l'arête "AC", d'origine "A", d'extrémité "C" en convenant de distinguer l'arc "AC" de l'arc "CA" (comme pour les couples ou les vecteurs) mais il est d'usage d'utiliser deux arcs afin d'éviter toute ambigüité à l'instar du graphe de la figure 0.8.



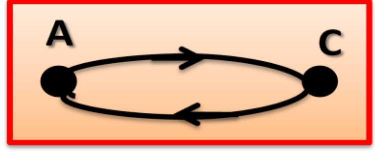


Figure 0.8 – Graphes avec arêtes à double sens

Ce qui s'avère d'ailleurs indispensable par exemple dans le cas concret d'un trajet aller-retour sur deux routes différentes. Il peut aussi s'agir de débits aller-retour dans un pipe-line, etc.

0.2.7 - Notion de graphe complet

Exercice : Parmi les graphes suivants de la figure 0.9, lesquels sont complets et lesquels sont non complets?

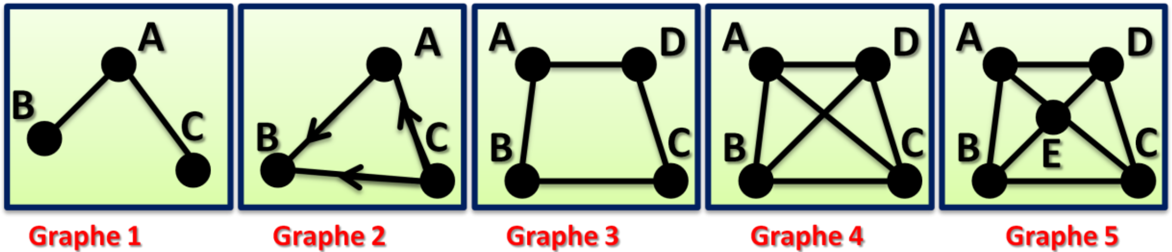


Figure 0.9 – Graphes complets et graphes non complets

- Les graphes 1, 3 et 5 ne sont pas complets.
- Les graphes 2 et 4 sont complets.

0.2.8 - Notions de distance et de diamètre

On appelle distance entre deux sommets distincts la longueur de la plus courte chaîne qui les relie. La plus grande distance entre deux sommets est le diamètre du graphe. Dans le graphe de la figure 0.10, la distance entre les sommes "A" et "E" est égale à 2. Le diamètre est égal à 4.

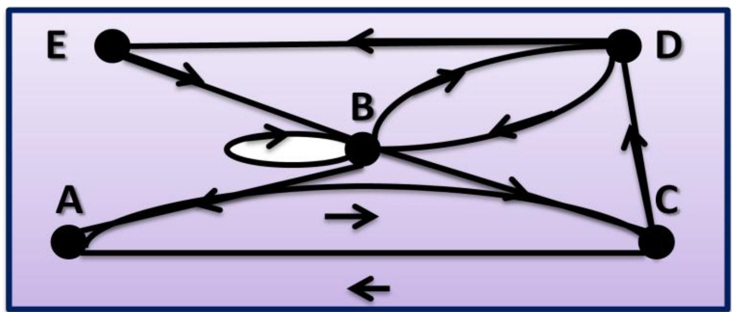


Figure 0.10 – Notion de distance et de diamètre

0.2.9 - Graphe ou pas un graphe

On peut avoir zéro ou une arête entre chaque paire de sommets comme sur la figure 0.11.

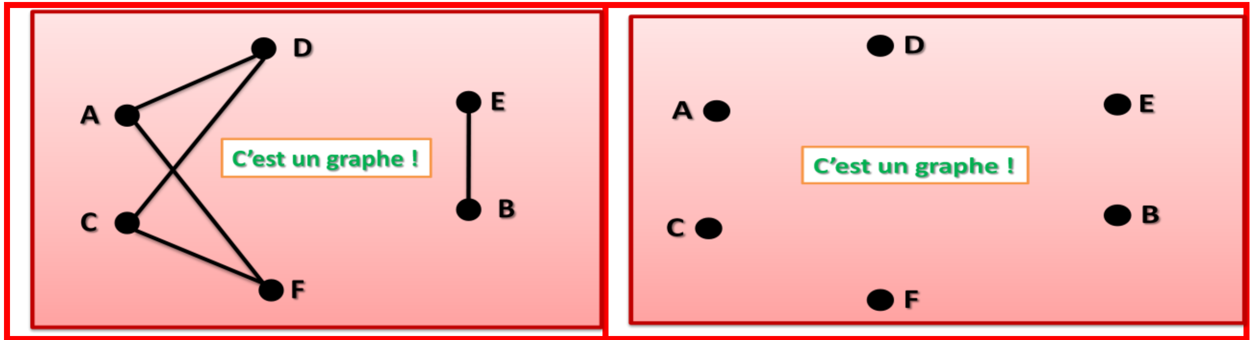


Figure 0.11 – C'est un graphe

0.2.10 – Notions de voisins et de degré

Les figures 0.12 et 0.13 illustrent les notions de voisins et de degré dans un graphe.

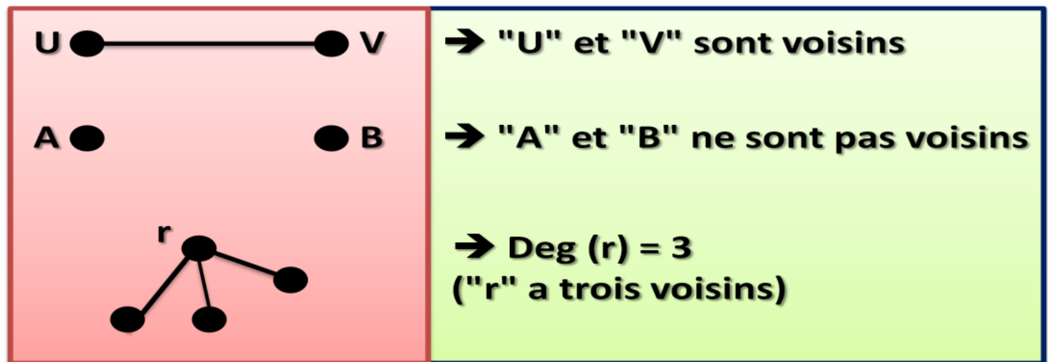


Figure 0.12 – Voisins et degré

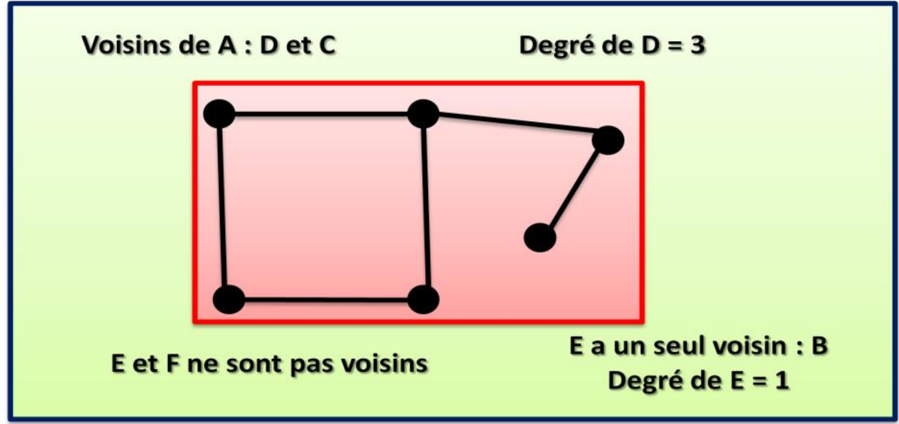


Figure 0.13 – Exemples de voisins et de degré

0.2.11 – Notions de chemin et de cycle

C'est une suite d'arêtes reliant les sommets entre eux dans notre cas A et B comme sur la figure 0.14. La longueur d'un chemin est son nombre d'arêtes.

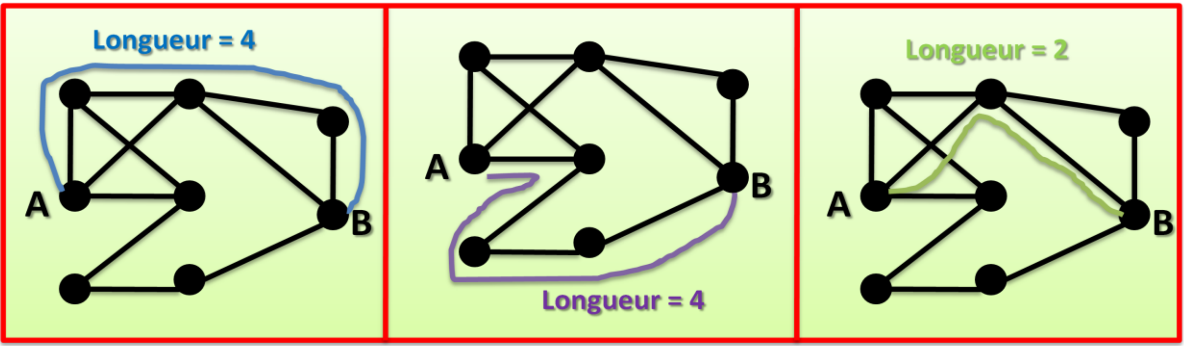


Figure 0.14 – Chemin dans un graphe

Un cycle est un chemin dont les extrémités sont reliées comme sur la figure 0.15. La longueur d'un cycle est son nombre d'arêtes.

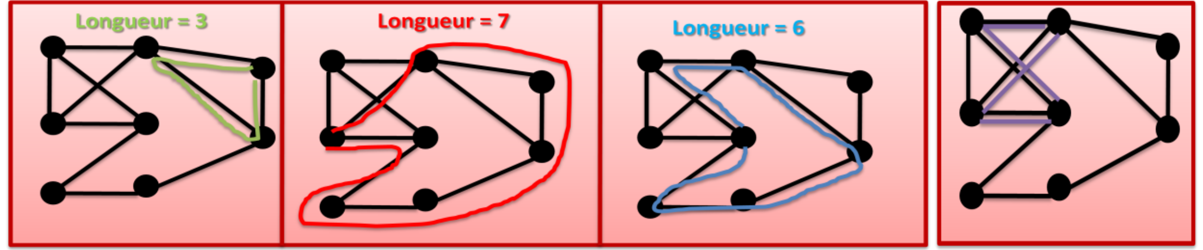


Figure 0.15 – Cycle dans un graphe

0.2.12 – Notion de graphe connexe et de graphe non connexe

Dans un graphe G (U, V) connexe est un graphe pour tout "u" et "v", il contient un chemin entre "u" et "v". Un graphe est dit fortement connexe si, pour toute paire de sommets "u" et "v", il existe un chemin entre "u" et "v" et un chemin de "v" à "u". Si le graphe n'est pas fortement connexe, il admet plusieurs composantes fortement connexes (voir la figure 0.16).

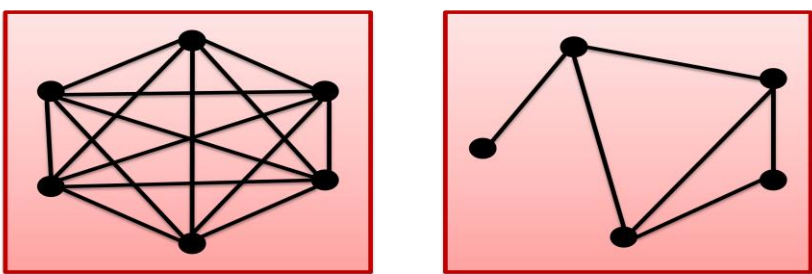


Figure 0.16 – Graphes connexes

On voit sur la figure 0.17, que les sommets "D" et "E" n'ont de liaisons. Comme aussi pour les sommets "F" et "B". Mais on a deux composantes connexes du graphe.

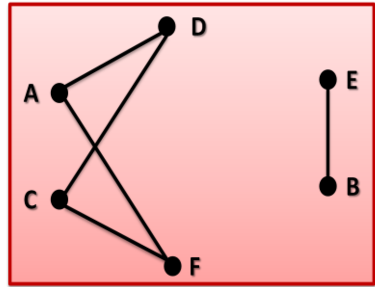


Figure 0.17 – Graphe non connexe

0.2.13 – Notion d'arbre

La figure 0.18 illustre la nation d'arbre. Un arbre est un graphe connexe et sans cycle.

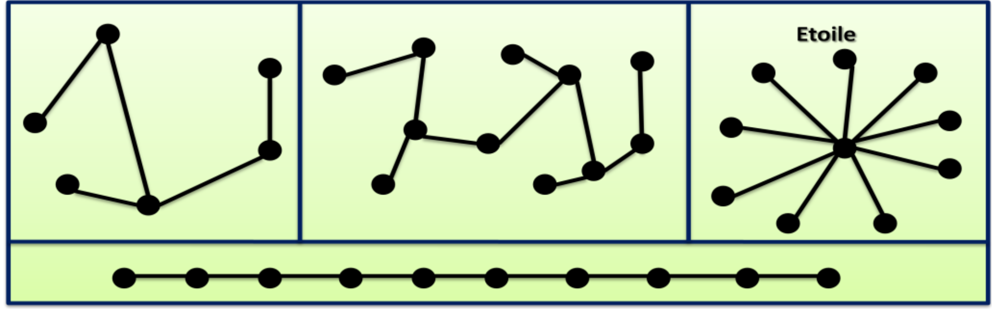


Figure 0.18 – Notion d'arbre

Soit T = (V, E) un graphe à n sommets et m arêtes alors on a : T est un arbre ⇔ T est connexe, sans cycle ⇔ T est connexe et m = n – 1 ⇔ T est sans cycle et m = n – 1 ⇔ T est connexe minimal ⇔ T est sans maximal ⇔ Pour tout "u" et "v", T contient un unique chemin entre "u" et "v" (voir la figure 0.19).

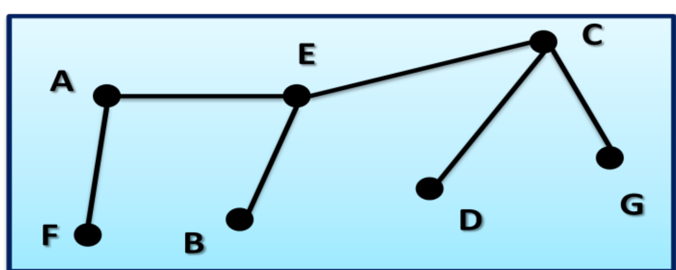


Figure 0.19 – Exemple d'arbre

Comment compte-t-on les arbres à n sommets? La figure 0.20 illustre un exemple de graphes qui ne sont pas des arbres, car dans le premier on a un cycle et dans le deuxième il n'est pas connexe.

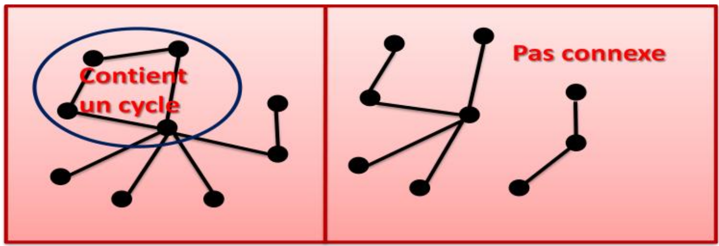


Figure 0.20 – Notion de non arbre

0.2.14 – Notion de forêt

Une forêt est un ensemble disjoint d'arbres comme sur la figure 0.21.

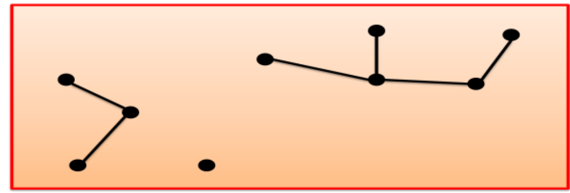


Figure 0.21 – Notion de forêt

0.2.14 – Notion d'arbre couvrant

Un graphe connexe est un graphe qui contient un arbre couvrant comme sur la figure 0.22. En supprimant certaines arêtes du graphe de haut à gauche de la figure 0.22, on obtient un arbre couvrant en bas à gauche de la figure 0.22 (graphe en rose). Soit un graphe G = (V, E), on a G est connexe ⇔ G contient un arbre couvrant.

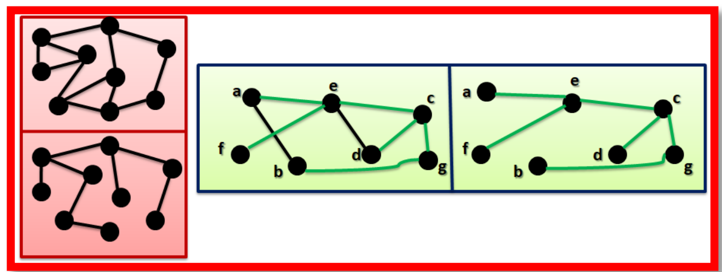


Figure 0.22 – Notion d'arbre couvrant

Comment construire un arbre couvrant un graphe? Eh bien, on peut le construire par l'algorithme DFS (parcours en profondeur) ou l'algorithme BFS (parcours en largeur) qui respecte les distances.

0.2.14 – Notion de graphe pondéré

Soit un graphe G = (V, E) est un graphe pondéré, si on a G est connexe ⇔ G contient un arbre couvrant. Comment construire un arbre couvrant d'un graphe de poids min? Eh bien,

on peut le construire par l'algorithme de PRIM ou l'algorithme de Kruskal. Comment construire un arbre couvrant d'un graphe respectant les distances pondérées à partir d'un sommet "r"? Eh bien, on peut le construire par l'algorithme de Dijkstra.

0.2.15 – Notion de somme des degrés

Dans la figure 0.23, comment calcul la somme des degrés du graphe ? C'est-à-dire :

deg(A) +deg(B) + deg(C) + deg(D)+ deg(E) +deg(F) = ?

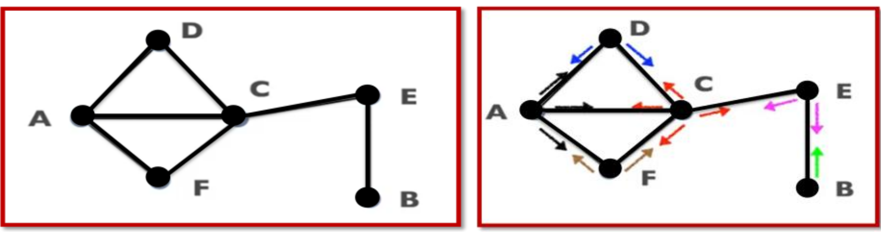


Figure 0.23 – Somme des degrés

Le calcul est le suivant (voir le graphe de droite de la figure 0.23) :

[deg(A) = 3]+[deg(B) =1] + [deg(C) =4] + [deg(D) =2]+ [deg(E) = 2]+[deg(F) = 2]= 14

On remarque que la somme des degrés d'un graphe est égale à deux fois le nombre d'arêtes.

Pour résumer les notions vues sur les graphes, on a vu :

1. Sommets
2. Arêtes = relations entre deux sommets
3. Voisins, degrés
4. Chemins, cycles (longueurs)
5. Graphe connexe
6. Arbre
7. Arbre couvrant d'un graphe connexe
8. Une arborescence est un arbre orienté.
9. Une boucle est une relation d'un sommet avec lui-même.

0.2.15 - Notions d'arête et d'arc

Figure 0.24 résume les notions d'arête, d'arc, d'arête pondérée et d'arc pondéré, ainsi que les notions de multi-arête et d'hyper-arête.

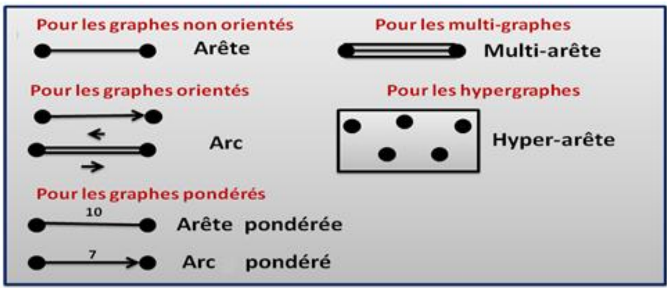


Figure 0.24 – Notions d'arêtes et d'arcs

0.2.16 – Notion de graphe

Un graphe est un ensemble de sommets ou des nœuds. Un graphe peut comporter des arêtes ou des arcs. Dans la figure 0.25, on a le graphe G = {1, 2, 3, 4} suivant :

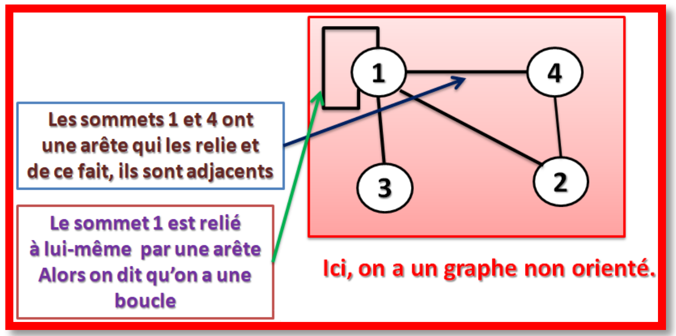


Figure 0.25 – Un graphe non orienté

0.2.17 – Notions de graphe non orienté et de graphe orienté

Un graphe non orienté est un graphe, dont les arêtes ne sont pas orientées. Un graphe orienté est un graphe, dont les arêtes ou plutôt arcs sont orientées, comme sur la figure 0.26.

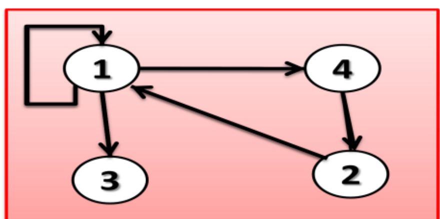


Figure 0.26 – Graphe orienté

0.2.18 – Notion d'ordre d'un graphe

L'ordre d'un graphe c'est le nombre de ses sommets. Soit le graphe G = {1, 2, 3, 4} de la figure 0.25, on a l'ordre du graphe "G" est égal à 4. Un graphe est dit régulier si les degrés de tous les sommets sont égaux comme sur la figure 0.27.

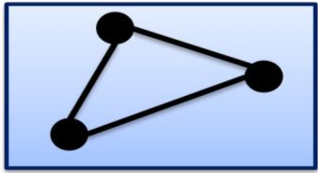


Figure 0.27 – Un graphe régulier

0.2.19 – Notion de degré d'un sommet d'un graphe

Le degré d'un sommet d'un graphe c'est le nombre d'arêtes dont le sommet en question est l'extrémité. Soit le graphe G = {1, 2, 3, 4} de la figure 0.25, on a le degré du sommet 4 du graphe "G" est égal à 2. Le tableau suivant résume les degrés des quatre sommets du graphe "G".

Sommets	1	2	3	4
degrés	5	2	1	2

0.2.20 – Notions de graphe complet et de sous -graphe

Un graphe complet non orienté est un graphe où chaque couple de sommets est relié par une arête et pas de boucle. Un graphe complet orienté est un graphe où tous les sommets sont adjacents et pas de boucle. Un sous-graphe est un sous ensemble de sommets d'un graphe. Soit notre graphe G = {1, 2, 3, 4} de la figure 0.28, on peut avoir le sous graphe G1 = {1, 2, 4} en vert.

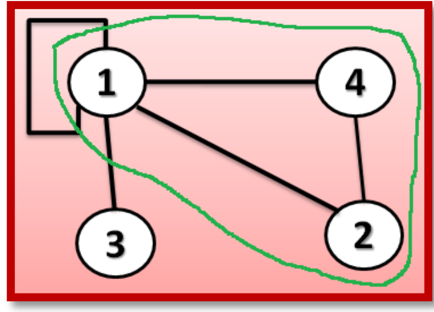


Figure 0.28 – Notion de sous-graphe

0.2.21 - Notion de réseau

Un réseau est un graphe orienté composé de sommets internes et de deux sommets particuliers : sommet source "s" et sommet puits "t" comme sur la figure 0.29. En suppose qu'il contient aussi des arcs. On prend comme hypothèses pour simplifier, que le sommet source "s" n'a pas d'arcs entrants et le sommet puits "t" n'a pas d'arcs sortants

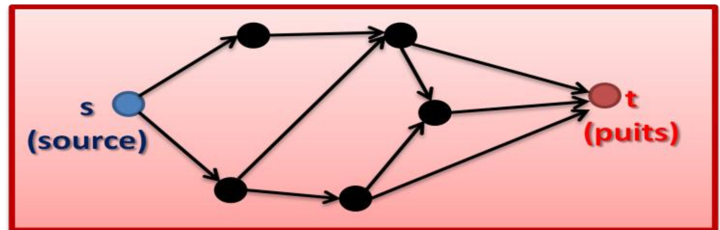


Figure 0.29 – Notion de réseau

Un réseau est un graphe valué ou pondéré. Il a des sommets et des arcs avec des valeurs. Un chemin est une succession d'arcs adjacents bien orientés. Une chaîne est une succession d'arcs adjacents dont les arcs ne sont pas nécessairement bien orientés. Tout chemin est une chaîne. Toute chaîne n'est pas nécessairement un chemin. Les arcs bien orientés sont appelés des arcs directs. Les arcs mal orientés sont appelés des arcs inverses. Un circuit est un chemin qui boucle. A la suite de notre cours, on va étudier exclusivement les graphes sans circuit avec une entrée et une sortie. La valeur de l'arc est appelée capacité. Les flux circulant dans un arc est positif et inférieur ou égal à la capacité de l'arc. Un chemin ou Chaîne sera dit(e) saturé(e) si au moins un arc qui le compose est saturé. Un arc direct sera dit saturé si le flux qui y circule sera égal à la capacité de l'arc. Cela veut dire qu'on ne peut rien ajouter. Un arc inverse sera dit saturé si le flux qui y circule est nul. Cela veut dire qu'on ne peut rien enlever. Maintenant sur chaque arc, on met une valeur d'étiquette qu'on appelle la capacité de l'arc comme sur la figure 0.30.

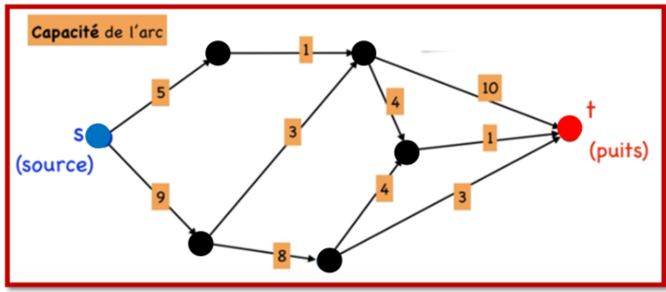


Figure 0.30 – Capacité de l'arc

On suppose que la capacité d'un arc est le diamètre du tuyau de l'arc pour pouvoir faire passer un flux à l'intérieur. On a un réseau à flots (ou un réseau de transport).

0.2.22 - Notions de flot, de la valeur d'un flot et de maximisation de sa valeur

La figure 0.31 illustre toutes les notions qui touchent la notion de flot.

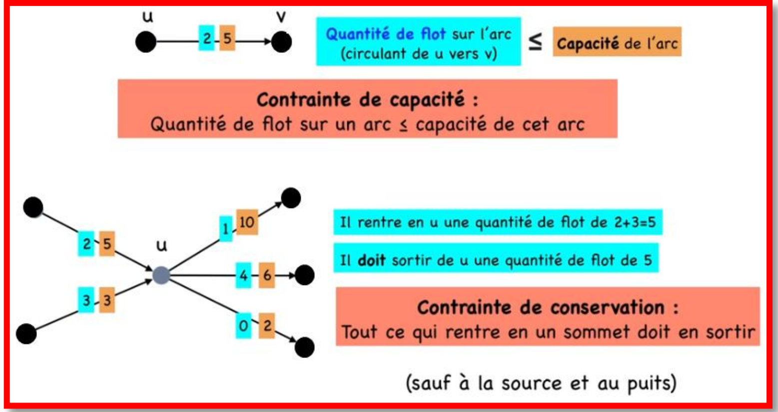


Figure 0.31 – Notion de flot

Le flot de la figure 0.32 indique comment on fait les calcul de la valeur du flot.

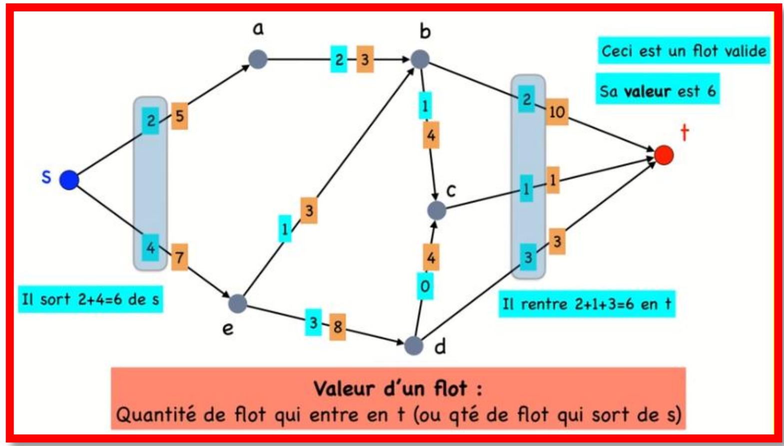


Figure 0.32 – Notion de la valeur d'un flot

La figure 0.33 illustre sur un exemple le calcul de la valeur maximale d'un flot.

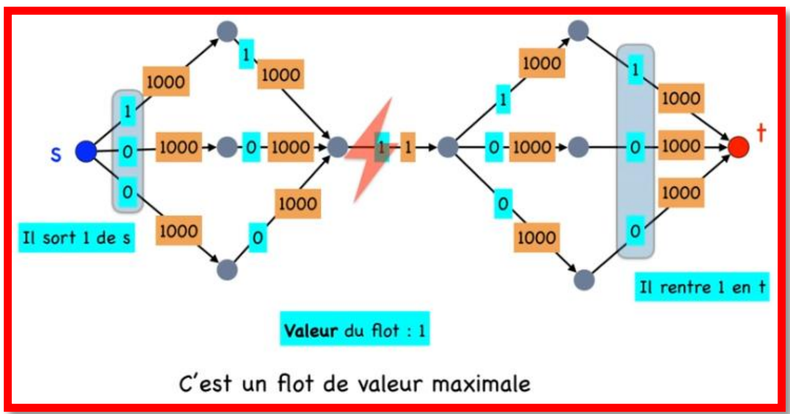


Figure 0.33 – Notion de maximisation de la valeur d'un flot

La figure 0.34 illustre le calcul de la capacité d'une coupe.

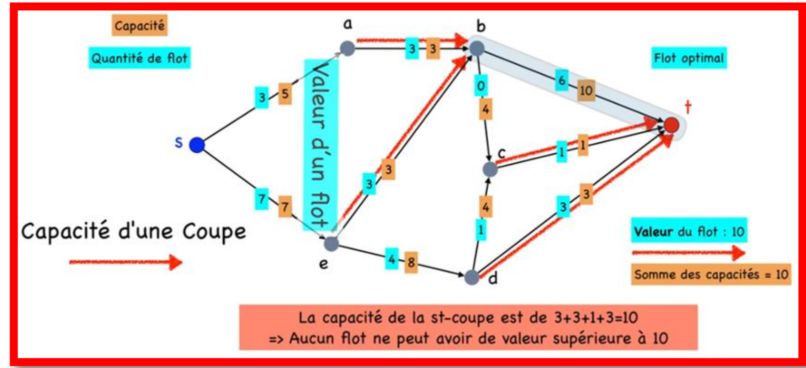


Figure 0.34 – Calcul de la capacité d'une coupe

0.2.23 - Algorithme de Ford-Fulkerson

C'est un algorithme pour construire un flot max dans un graphe. Il fonctionne à l'aide de chaîne améliorante comme représentée par la ligne rouge du réseau de la figure 0.35.

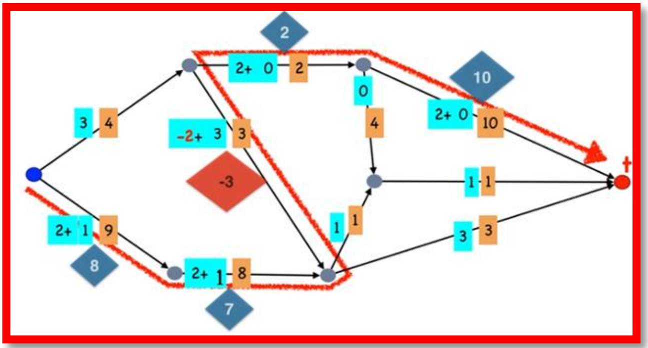


Figure 0.35 – Principe de l'algorithme de Ford-Fullkerson

La première étape, c'est de prendre un flot initial nul de valeur 0 comme sur la figure 0.36.

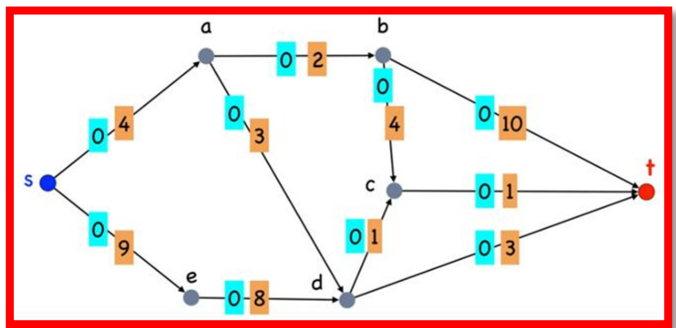


Figure 0.36 – La première étape de l'algorithme

Ce réseau on va l'améliorer étape par étape. La deuxième étape, c'est de prendre une chaîne améliorante comme la ligne rouge de l'exemple de la figure 0.7.

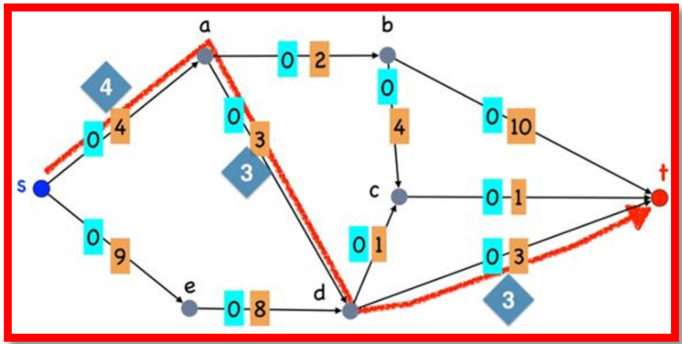


Figure 0.37 – Prise d'une chaîne améliorante

On prend la capacité max de chaque arc. On sélection la valeur de l'arc min, ici est égal à 3, dans le flot on mettre 3 dans chaque arc de la ligne rouge. En faisant passer le flot de 3 dans arc de la ligne, on obtient le flot 1 de valeur égale à 3 comme sur la figure 0.38.

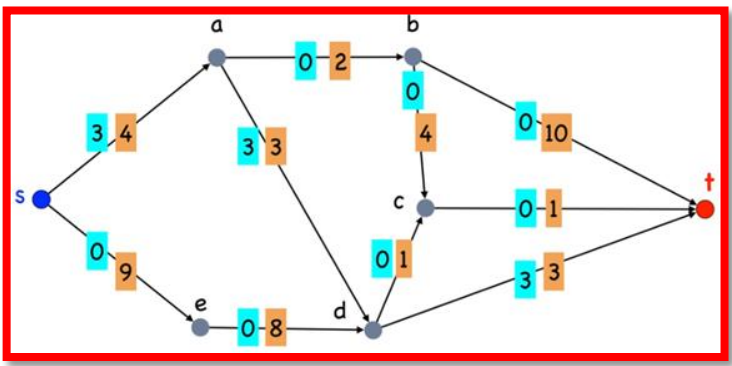


Figure 0.38 – Le flot 1 de valeur 3

L'algorithme dit tant que l'on peut trouver des chaînes améliorantes, on doit améliorer. On prend par exemple la nouvelle chaîne améliorante (la nouvelle ligne rouge sur la figure 0.39).

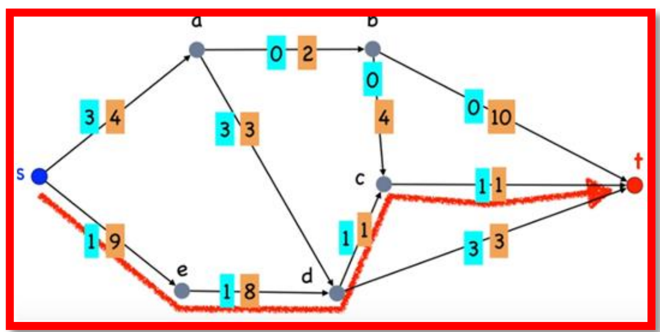


Figure 0.39 – La nouvelle ligne rouge de l'algorithme

On va procéder de la même manière que précédemment. On obtient un flot 2 de valeur égale à 4 comme sur la figure 0.40.

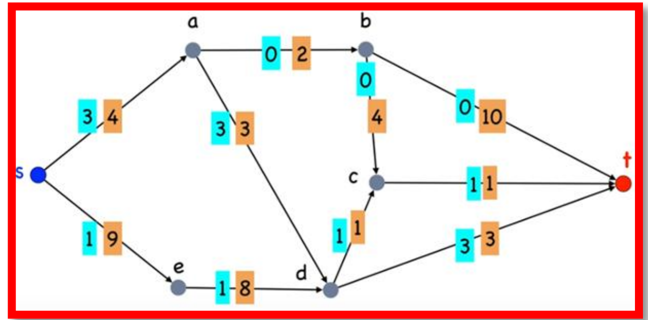


Figure 0.40 – Le flot 2 de valeur 4

On va maintenant prendre une nouvelle chaîne améliorante. On prend par exemple la nouvelle chaîne améliorante (la nouvelle ligne rouge sur la figure 0.41, où l'arc est parcouru à l'envers).

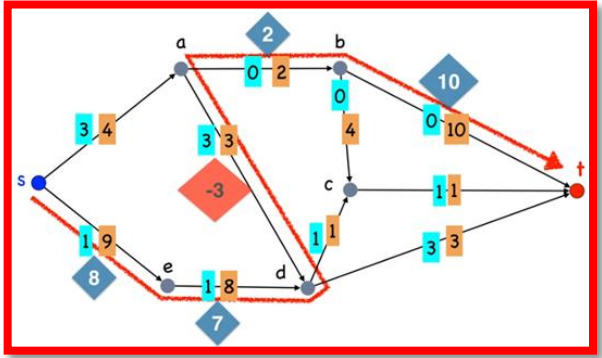


Figure 0.41 – La nouvelle chaîne améliorante

On va augmenter les arcs par la valeur min de la ligne rouge et c'est égal à 2. En fait ici, on a la valeur des valeurs des arcs. La valeur la plus petite est égale à 2. Donc, on va augmenter de plus 2 chaque arc de la ligne rouge comme sur la figure 0.42.

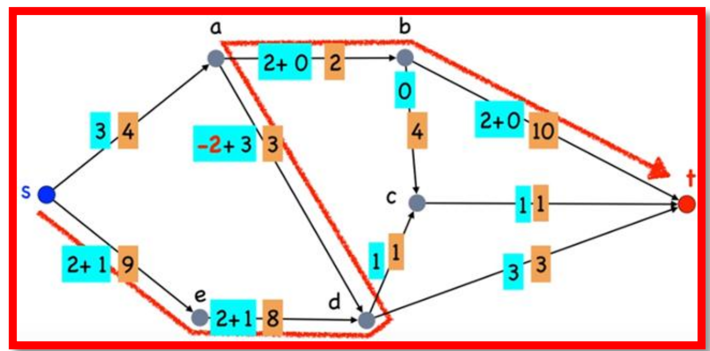


Figure 0.42 – Augmentation de plus 2 pour chaque arc de la ligne rouge

On va obtenir un nouveau flot valide. Donc le nouveau flot 3 à pour valeur égale à 6 comme sur la figure 0.43.

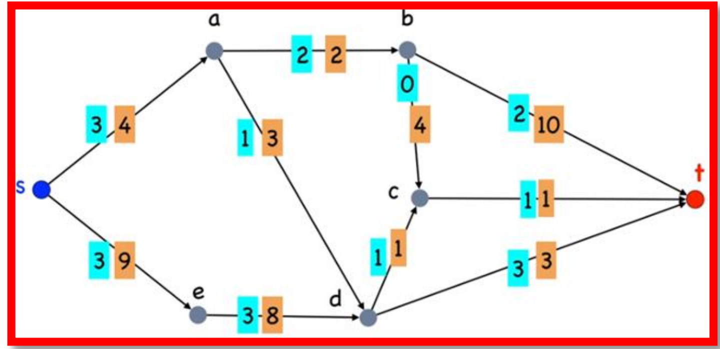


Figure 0.43 – Le flot de valeur 6

En continuant, on va se trouver dans des conditions difficiles à voir. Donc le nouveau flot 3 a pour valeur égale à 6 comme sur la figure 0.44.

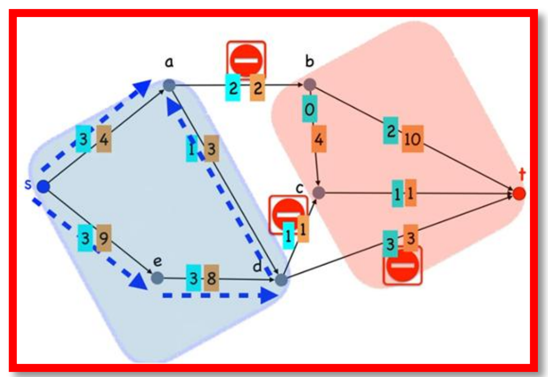


Figure 0.44 – Le nouveau flot 3 a pour valeur 6

On constate que certains arcs sont saturés et de ce fait pour chercher une chaîne améliorante, cela devient très dur. En fait, on se retrouve avec un graphe scindé en deux comme sur la figure 0.44. C'est la dernière étape de l'algorithme. On voit bien qu'on ne peut pas faire mieux que 6 comme sur figure 0.45. C'est la fin de l'algorithme.

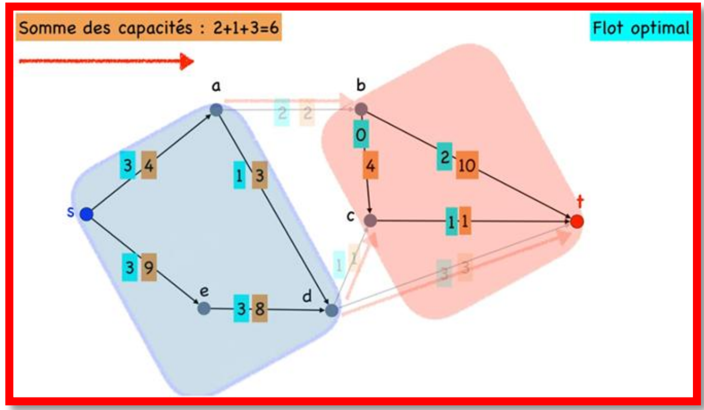


Figure 0.45 – La fin de l'algorithme

En résumé de l'algorithme de Ford-Fulkerson

1ère succession d'étapes de l'algorithme : saturer tous les chemins

- À partir du flot nul
- Obtention d'un flot complet.

2ème succession d'étape de l'algorithme : saturer toutes chaînes

À partir d'un flot complet

Obtention d'un flot maximal.

- On peut avoir plusieurs flots complets avec des valeurs différentes, mais on a un seul flot maximal avec une seule valeur.
- Pour respecter la loi de conservation des flux, il faut ajouter aux arcs directs et enlever aux arcs inverses.

0.3 – Rappel 02 – Algorithme de Dijkstra

0.3.1 - Notion de distance pondérée

Soit un graphe G formé des sommets {a, b, c, d, e} formé d'arêtes pondérées comme sur le schéma du graphe de la figure 0.46. On va chercher les plus courts chemin entre, par exemple, les sommets "a" et "b" en faisant la somme des différents arcs reliant le sommet "a" et le sommet "b" du graphe.

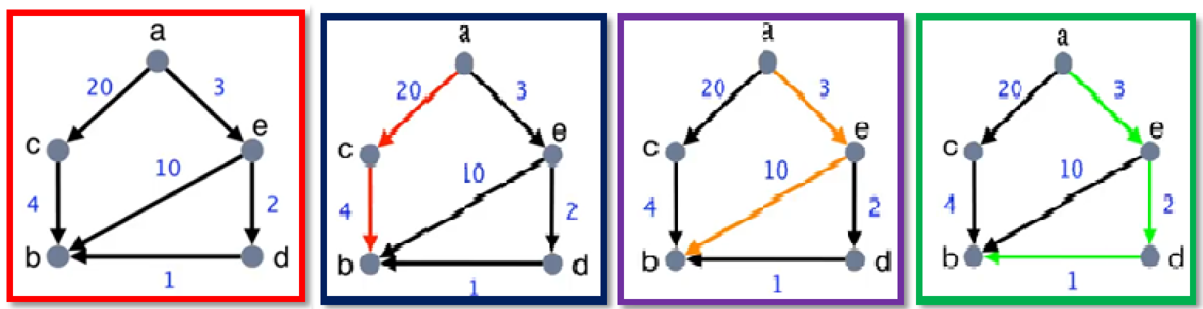


Figure 0.46 – Calcul de la longueur d'un chemin

La longueur du chemin entre les sommets "a" et "b" en passant par "c" est de 24 (flèches rouges). La longueur du chemin entre les sommets "a" et "b" en passant par "e" est de 13 (flèches oranges). La longueur du chemin entre les sommets "a" et "b" en passant par "e" et "d" est de 06 (flèches vertes). Attention ! Le nombre d'arcs et la longueur d'un chemin ce n'est pas la même chose. Cet algorithme prend en entrée un graphe pondéré ou non comme celui de notre schéma à gauche de la figure 0.47 et un sommet de départ du calcul du chemin par exemple dans notre cas on prendra le sommet "a". La première des choses que fait

l'algorithme c'est une initialisation assez pessimiste. Il suppose que la distance entre le sommet "a" et les autres sommets est égale à l'infini et la distance le sommet "a" et lui-même est égale à zéro comme sur le schéma à droite de la figure 0.47.

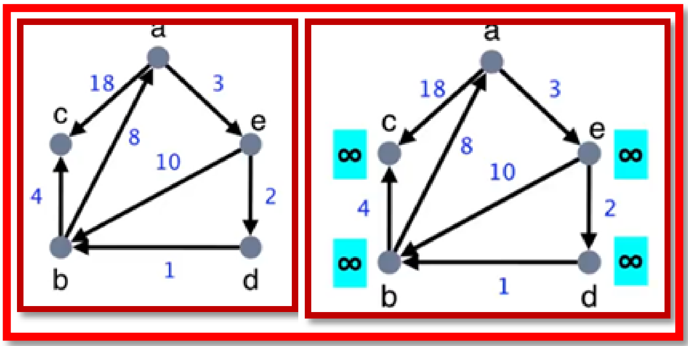


Figure 0.47 – Graphe pondéré et la distance d'un sommet a

Maintenant, on passe au traitement. D'abord, on va traiter le sommet de départ (dans notre cas, c'est le sommet "a" et on l'entoure en rouge comme sur le schéma de gauche de la figure 0.48. On va peindre en vert le sommet "a" pour dire qu'il a été déjà traité et on va relâcher les arcs sortant du sommet "a" dans un ordre quelconque comme sur le schéma de droite de la figure 0.48. On prend par exemple, l'arc "ac" (en vert) de longueur 18 du sommet "a" vers le sommet "c". On modifie l'étiquette du sommet "c" de valeur infini par la valeur 18. Ensuite, on prend le deuxième arc "ae" (en vert) de longueur 3 du sommet "a" vers le sommet "e". On modifie l'étiquette du sommet "e" de valeur infini par la valeur 3. Fin de traitement du sommet "a".

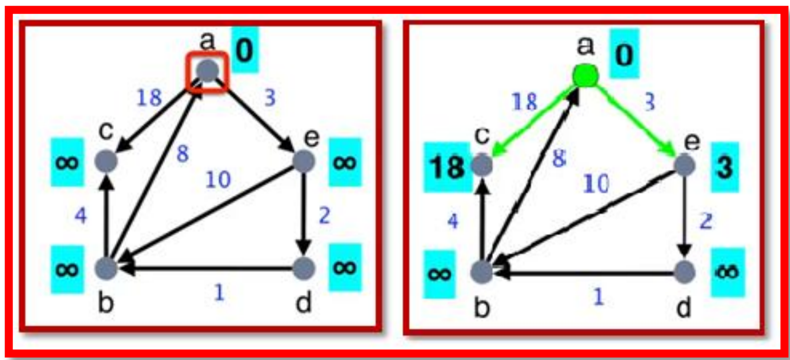


Figure 0.48 – On peint en vert le sommet "a"

Maintenant, on passe au traitement du prochain sommet. L'algorithme de Dijkstra dit que le prochain sommet est celui qui a la plus petite valeur d'étiquette. Dans notre cas, c'est le sommet "e" et on va l'entourer de rouge comme sur le schéma de gauche de la figure 0.49. On va peindre en vert le sommet "e" pour dire qu'il a été déjà traité et on va relâcher les arcs

sortant du sommet "e" dans un ordre quelconque comme sur le schéma de gauche en bas. On prend par exemple, l'arc "ad" (en vert) de longueur 2 du sommet "e" vers le sommet "d". on modifie l'étiquette du sommet "d" de valeur infini par la valeur 3 + 2 = 5. Ensuite, on prend le deuxième arc "eb" (en vert) de longueur 10 du sommet "e" vers le sommet "b". On modifie l'étiquette du sommet "b" de valeur infini par la valeur 3 + 10 = 13. Fin de traitement du sommet "e".

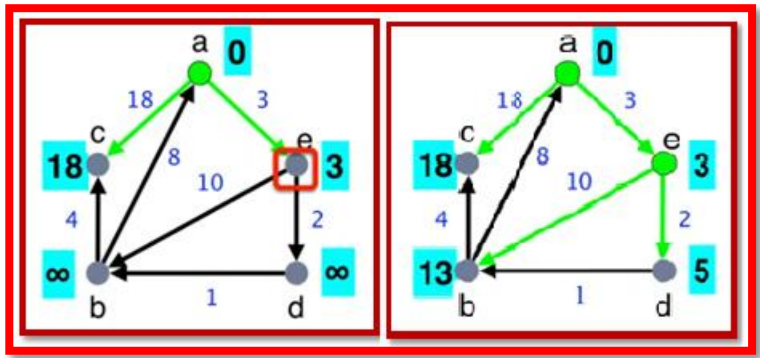


Figure 0.49 – On peint en vert le sommet "e"

Maintenant, on passe au traitement du prochain sommet. L'algorithme de Dijkstra dit que le prochain sommet est celui qui a la plus petite valeur d'étiquette. Dans notre cas, c'est le sommet "d" et on va l'entourer de rouge comme sur le schéma de gauche de la figure 0.50. On va peindre en vert le sommet "d" pour dire qu'il a été déjà traité et on va relâcher les arcs sortant du sommet "d" dans un ordre quelconque comme sur le schéma de droite de la figure 0.50. Dans notre cas, on a un seul arc "db". On prend cet arc "ad" (en vert) de longueur 1 du sommet "d" vers le sommet "b". On modifie l'étiquette du sommet "d" de valeur 13 par la valeur 3 + 2 + 1 = 6 et on supprime la couleur verte de l'arc "eb**".** Fin de traitement du sommet "d".

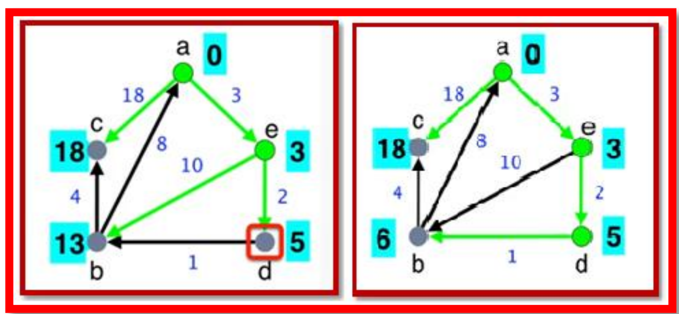


Figure 0.50 – On peint en vert le sommet "d"

Maintenant, il reste deux sommets non traités le sommet "b" et le sommet "c". On prend le sommet qui a la plus petite étiquette comme sur le schéma de gauche de la figure 0.51.

Dans notre cas, c'est le sommet "b". On va peindre en vert le sommet "b" pour dire qu'il a été déjà traité et on va relâcher les arcs sortant du sommet "b" dans un ordre quelconque comme sur le schéma de droite de la figure 0.51. Dans notre cas, on deux arcs "ba" et "bc". On prend l'arc "ba" (en noir) de longueur 8 du sommet "b" vers le sommet "a". On calcule la nouvelle valeur de l'étiquette du sommet a, on 3 + 2 + 1 + 8 = 14. On voit la nouvelle étiquette est supérieure à l'ancienne égale à 0 donc on supprime la couleur verte de l'arc "ba". Ainsi, il nous reste à exploré l'arc "bc". Le calcul de la nouvelle étiquette du sommet "c", nous donne la valeur de 3 + 2 + 1 + 4 = 10 est meilleure que l'ancienne valeur de 18. Donc, on modifie la valeur de l'étiquette du sommet "c". Fin de traitement du sommet "b".

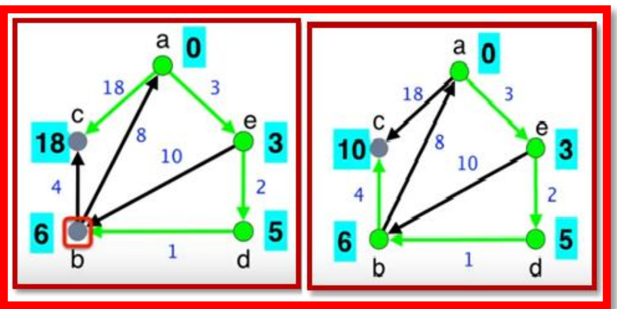
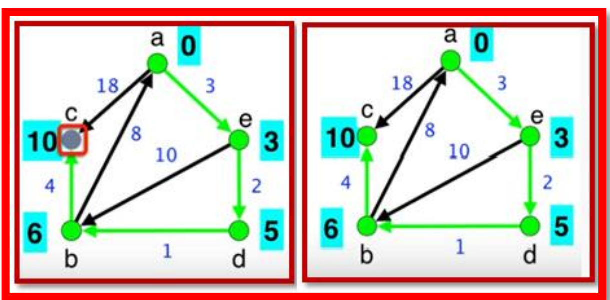


Figure 0.51 – On peint en vert le sommet "b"

L'algorithme n'est complètement terminé, il nous reste un sommet non traité. Dans notre cas, c'est le sommet "c". Alors on le traite comme sur le schéma de gauche de la figure 0.52. Donc, on va peindre en vert le sommet "c" pour dire qu'il a été déjà traité et on va relâcher les arcs sortant du sommet "c". Et comme, il n'y a aucun sortant, dans cas particulier, rien ne change. Comme, il n'y a aucun sommet non traité. C'est la fin de l'algorithme comme sur la schéma de la figure 0.52.



Fichier 0.52 – On peint en vert le sommet "c"

On peut s'amuser à dérouler l'algorithme de Dijkstra, en choisissant n'importe quel sommet de départ par "b", "c", "d" ou "e". Maintenant, si on oublie les étapes intermédiaires de l'algorithme, on a le graphe de départ en gauche de la figure 0.53, et le graphe résultat du

milieu de la figure 0.53. Le graphe à droite de la figure 0.53 illustre la suppression des arcs non sélectionnés en noir. On obtient une sorte d'arborescence enracinée en sommet "a" et donne le chemin le plus court vers n'importe quel sommet du graphe en faisant la somme des étiquettes.

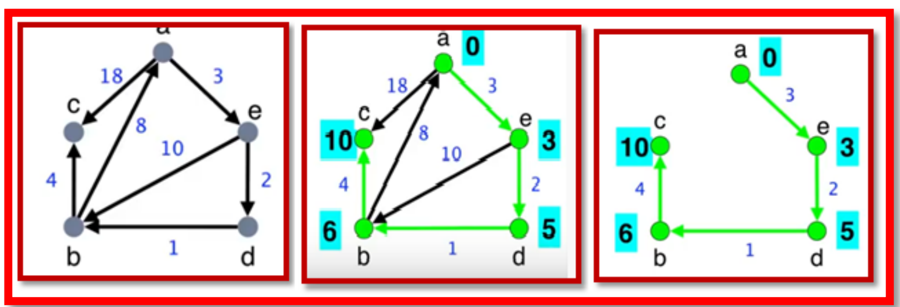


Figure 0.53 – Algorithme de Dijkstra

0.4 – Rappel 03 – Résolution des problèmes par certaines méthodes d'exploration

0.4.1 - La méthode d'exploration heuristique A* - Motivations

0.4.10 - Objectifs

Résolution de problème par l'exploration dans un graphe par l'algorithme **A***.

Comprendre **A***

Notion d'heuristique

Propriétés théoriques

Implémenter et simuler **A***

0.4.1.1 – Motivations

Par exemple trouver le chemin dans la ville, c'est-à-dire trouver un chemin de la 10ème ave et 50ème rue à la 3ème ave et 51ème rue comme sur la figure 0.54. Dans la figure 0.55, on la solution au problème.

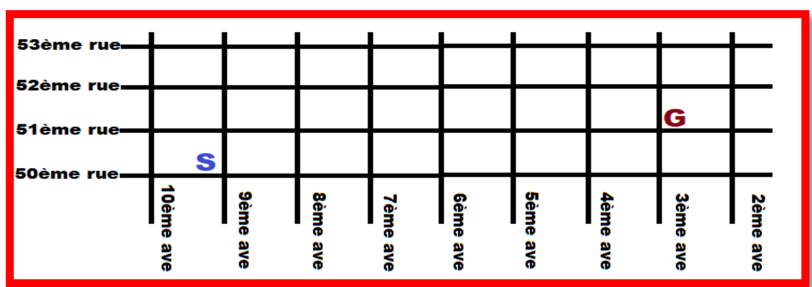


Figure 0.54 – recherche d'un chemin

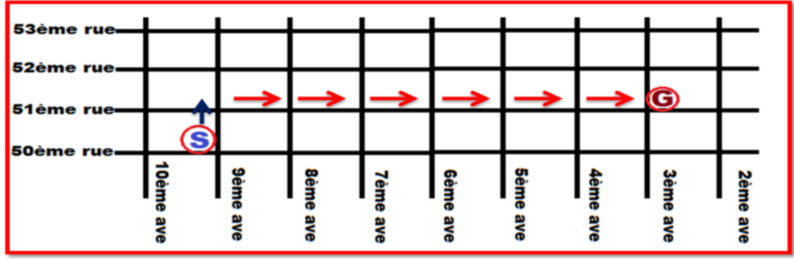


Figure 0.55 – La solution du problème.

Par exemple en utilisant Google Maps, on cherche à trouver un chemin à l'aide de Google Maps pour aller de N'Gaous vers El-Khroub en Algérie comme sur la figure 0.56.

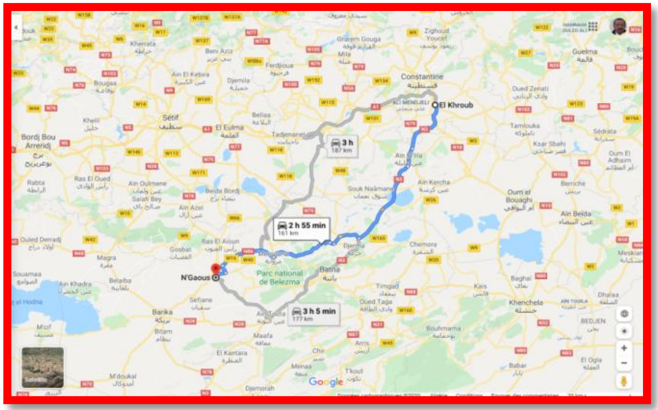


Figure 0.56 – Recherche d'un chemin sur Google Maps

Par exemple pour la livraison, il s'agit de trouver un chemin de livraison de colis pour un robot voir la figure 0.57.

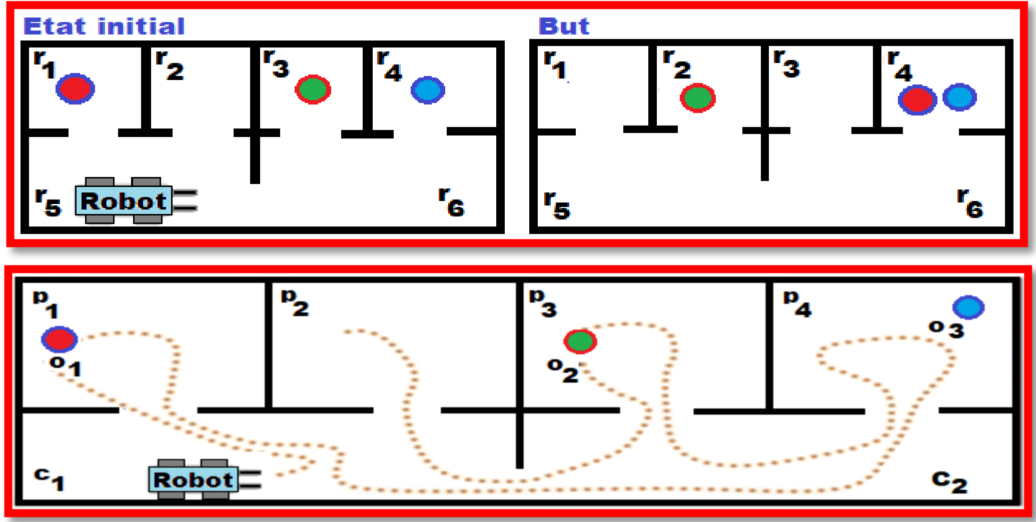


Figure 0.57 – Recherche d'un chemin de livraison pour un robot avec solution.

Par exemple pour la navigation d'un objet ou d'une chose, il faut trouver un chemin à l'aide de Google Maps pour déplacer un objet ou une chose d'un point rouge à autre comme sur la figure 0.58 en tenant compte des contraintes du terrain.

Figure 0.58 – La navigation d'un bjet ou d'une chose

Par exemple pour le jeu N-Puzzle, il faut trouver le moyen de passer d'une configuration initiale à la configuration finale comme sur la figure 0.59.

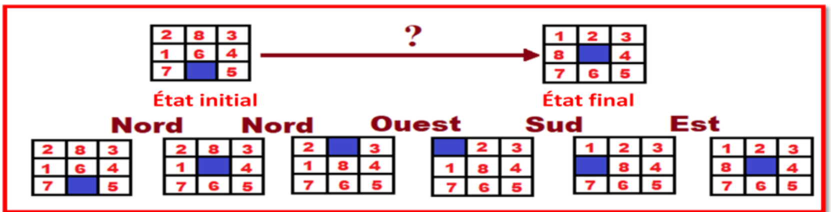


Figure 0.59 – Jeu de N-Puzzle

0.4.2 -- Résolution des problèmes

Les étapes intuitives pour un humain :

0 Modéliser la situation actuelle

1 Énumérer les solutions possibles

2 Évaluer la valeur des solutions

3 Retenir la meilleure option possible satisfaisant le but.

Mais comment parcourir efficacement la liste des solutions? La résolution de plusieurs problèmes peut être faite par une exploration dans un graphe :

1. Chaque nœud correspond à un état de l'environnement.
2. Chaque chemin à travers un graphe représente alors une suite d'actions prises par l'agent.

3. Pour résoudre le problème, il suffit de chercher le chemin qui satisfait le mieux la mesure de performance.

0.4.2.1 - Problème d'exploration dans un graphe

- Algorithme d'exploration dans un graphe
  - Entrées :
    - Un nœud initial.
    - Une fonction goal(n) qui retourne true si le but est atteint.
    - Une fonction de transition (n) qui retourne les nœuds successeurs de n.
    - Une fonction C(n', n) strictement positive, qui retourne le coût de passer de n à n' (permet de considérer le cas avec des coûts variables).
  - Sorties :
- Un chemin dans un graphe (séquence de nœuds/arêtes).
  - Le coût d'un chemin est la somme des coûts des arêtes dans le graphe.
  - Il peut y avoir plusieurs nœuds qui satisfont le but.
- Enjeux :
  - Trouver un chemin solution, ou
  - Trouver un chemin optimal, ou
  - Trouver rapidement un chemin (optimalité pas importante)

0.4.2.2 - Exemple : trouver le chemin entre deux villes (voir la figure 0.60)

- Villes : nœuds
- Chemins entre deux villes : arêtes
- Ville de départ : nœud (état) initial - n_0
- Routes entre les villes : transition (n_0) = (n_3, n_2, n_1)
- Distances entre les villes : C(n_0, n_2) = 4
- Ville de destination : Goal (n) - vrai si n_6 (où n_6 est le nœud de la ville de destination).

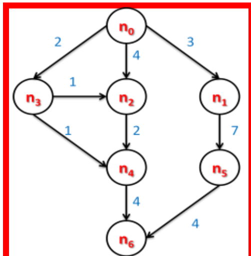


Figure 0.60 – Graphe d'exploration des villes

0.4.2.2 - Rappel sur les algorithmes d'exploration dans des graphes

- Exploration sans heuristique et coût uniforme :
  - Exploration en profondeur (Depht-First-Search) – pour un nœud donné, explore le premier enfant avant d'explorer un nœud frère ;
  - Exploration en largeur (Breadth-First-Search) – pour un nœud donné, explore les nœuds frères avant leurs enfants.
- Exploration sans heuristique et coût variable :
  - Algorithme de Dijkstra – Trouve le chemin le plus court entre un nœud source et tous les autres nœuds.
- Exploration avec heuristique et coût variable :
  - Best-First-Search ;
  - Greedy Best-First-Search ;
  - A*.

0.4.3 - Résolution de problème par une exploration heuristique dans un graphe

L'exploration heuristique est à la base de beaucoup d'approches en intelligence artificielle. Une heuristique est utilisée pour guider l'exploration comme les heuristiques exploitent les connaissances du domaine d'application. Le graphe est défini récursivement (plutôt qu'explicitement).

0.4.3.1 - Algorithme A*

A* est une extension de l'algorithme de Dijkstra. Il utilise pour trouver un chemin optimal dans un graphe via l'ajout d'une heuristique. Une heuristique h(n) est une fonction d'estimation du coût restant entre un nœud n d'un graphe et le but le nœud à atteindre). Les heuristiques sont à la base de beaucoup de travaux en intelligence artificielle tels que la recherche de meilleurs heuristiques et l'apprentissage automatique d'heuristiques. Pour décrire A*, il est pratique de décrire un algorithme générique très simple, dont A* est un cas particulier.

0.4.3.2 - Variables importantes : Open et Closed

Les variables importantes sont :

- Open contient les nœuds non encore traités, c'est-à-dire à la frontière de la partie du graphe explorée jusqu'à maintenant.
- Closed contient les nœuds déjà traités, c'est-à-dire à l'intérieur de la frontière délimitée par Open.

a - Insertion des nœuds dans Open

Les nœuds n dans Open sont triés selon l'estimation f(n) de leur valeur. On appelle f(n) une fonction d'évaluation. Pour chaque nœud n, f(n) est un nombre réel positif ou nul, estimant le coût du meilleur chemin partant du nœud initial passant par n et arrivant au but. Dans Open, les nœuds, se suivent en ordre croissant selon les valeurs de f(n) et le tri se fait par insertion pour qu'on s'assure que le nouveau nœud va au bon endroit. On explore donc les nœuds les plus prometteurs en premier.

b - Définition de la fonction d'évaluation f

La fonction d'évaluation f(n) tente d'estimer le coût du chemin optimal entre le nœud initial et le but, et qui passe par n. A tout moment, on connaît seulement le coût optimal pour la partie explorée entre le nœud initial et un nœud déjà exploré. Dans A*, on sépare le calcul de f(n) en deux parties :

- g(n) : coût du meilleur cheminement mené au nœud n depuis le nœud initial – c'est le coût du meilleur chemin trouvé jusqu'à maintenant qui se rend au nœud n ;
- h(n) : coût estimé du reste du chemin optimal partant du nœud n jusqu'au but. h(n) est la fonction heuristique – on suppose que h(n) est non négative et h(n) = 0 si le nœud n est le nœud but ;
- Et f(n) = g(n) + h(n).

c - **Exemples de fonction heuristique h(n)**

Le chemin entre deux villes est la distance euclidienne (à vol d'oiseau) entre la ville n et la ville de destination. Pour ce qui est du jeu N-Puzzle, on a le nombre de tuiles mal placées et la somme des distances des tuiles. La qualité de configuration d'un jeu par rapport à une configuration gagnante.

0.4.3.3 - Algorithme générique d’exploration dans un graphe

Algorithme RECHERCHE-DANS-GRAPHE(nœud Initial)

1. Déclarer deux nœuds : n, n’
2. Déclarer deux listes : open, closed // toutes les deux listes sont vides au départ
3. Insérer nœud initial dans open
4. Tant que (1) // la condition de sortie (exit) est déterminée dans la boucle
    1. Si open est vide, sortir de la boucle avec échec
    2. n = nœud au début de open
    3. Enlever n de open et l’ajouter dans closed
    4. Si n est le but (goal(n) est true), sortir de la boucle avec succès en retournant le chemin
    5. Pour chaque successeur n’ de n (chaque n’ appartenant à transitions(n))
        1. Initialiser la valeur g(n’) à g(n) + c(n, n’)
        2. Mettre le parent de n’ à n
        3. Si closed ou open contient un nœud n’’ égal à n’ avec f(n’) ≤ f(n’’)
            1. Enlever n’’ de closed ou open et insérer n’ dans open (ordre croissant selon f(n))
        4. Si n’ n’est ni dans open, ni dans closed
            1. Insérer n’ dans open (ordre croissant selon f(n))

0.4.3.4 - Exemple A* avec recherche d’un chemin entre deux villes

Soit le Graphe d’exploration du schéma de la figure 0.61 :

n_0 : ville de départ

n_6 : destination

h : distance à vol d’oiseau

c : distance réelle entre deux villes

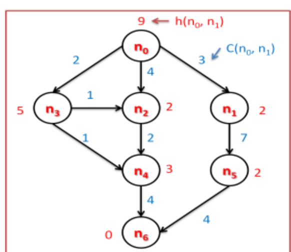


Figure 0.61 - Exemple A* avec recherche d’un chemin entre deux villes

Le tableau suivant donne, dans les différentes étapes de l'algorithme, le changement du de Open et de Closed.

Contenu de open à chaque itération (état, f, parent)	**Contenu de closed à chaque itération**
1. (n_0, 9, void)	1. vide
2. (n_1, 5, n_0), (n_2, 6, n_0), (n_3, 7, n_0)	2. (n_0, 9, void)
3. (n_2, 6, n_0), (n_3, 7, n_0), (n_5, 12, n_1)	3. (n_0, 9, void), (n_1, 5, n_0)
4. (n_3, 7, n_0), (n_4, 9, n_2), (n_5, 12, n_1)	4. (n_0, 9, void), (n_1, 5, n_0), (n_2, 6, n_0)
5. (n_2, 5, n_3), (n_4, 6, n_3), (n_5, 12, n_1)	5. (n_0, 9, void), (n_1, 5, n_0), (n_3, 7, n_0)
6. (n_4, 6, n_3), (n_5, 12, n_1)	6. (n_0, 9, void), (n_1, 5, n_0), (n_3, 7, n_0), (n_2, 5, n_3)
7. (n_6, 7, n_4), (n_5, 12, n_1)	7. (n_0, 9, void), (n_1, 5, n_0), (n_3, 7, n_0), (n_2, 5, n_3), (n_4, 6, n_3)

0.4.3.5 - D'autres algorithmes d'exploration heuristique

- Best-First-Search
  - Variante plus générale où f peut prendre une forme quelconque.
  - A* est un cas spécial de Best-First-Search, où

$$\mathbf{f(n) = g(n) + h(n)}$$

- Greedy Best-First-Search
  - C'est un Best-First-Search où f(n) = h(n)
  - N'est pas garanti de trouver un chemin optimal, mais marche parfois bien en pratique.

0.4.3.6 - Propriété théorique de A*

Si le graphe est fini, A* termine toujours. Si un chemin vers le but existe, A* va en trouver un. Si la fonction heuristique h retourne toujours une estimation inférieure ou égale au coût réel à venir, on dit que h est admissible. Dans ce cas, A* retourne toujours un chemin optimal. Parfois, on entend par A* la version de l'algorithme avec la condition additionnelle que h soit admissible. A* est alors un Best-First-Search où f(n) = g(n) + h(n) est admissible.

a -Propriété de A* : exploration en largeur

En utilisant des coûts des arcs uniformément égaux et strictement positifs (par exemple, tous égaux à 1) et h(n) retournant toujours zéro quelque soit le nœud n, ainsi A* devient une

exploration en largeur. Open devient une queue LILO (Last In, Last Out) en d'autres termes (Dernier Entré, Dernier Sortie).

b – **Propriété de A* : Dijkstra**

En utilisant une heuristique h(n) retournant toujours zéro, A* est équivalent à l'algorithme de Dijkstra. Cependant, A* s'arrête lorsque le chemin optimal vers le but a été trouvé. Dijkstra continuerait jusqu'à avoir trouvé le chemin optimal vers tous les nœuds.

c - Propriété de A*

Soit f*(n) le coût exact (pas une estimation) du chemin optimal du nœud initial au nœud but passant par n. Soit g*(n) le coût du chemin optimal du nœud n. Soit h*(n) le coût exact du chemin optimal du nœud n au nœud but. On a donc :

$$f^*(n) = g^*(n) + h^*(n).$$

Si l'heuristique est admissible, pour chaque nœud n exploré par A*, on peut montrer que l'on a toujours :

$$f(n) \leq f^*(n)$$

Si quelque soit un nœud n_1 et son successeur n_2, on a toujours :

$$h(n_1) \leq C(n_1, n_2) + h(n_2)$$

Où $C(n_1, n_2)$ est le coût de l'arc (n_1, n_2).

On dit alors que h est cohérent (on dit aussi parfois monotone – mais c'est en réalité f qui devient monotone). Dans ce ca : h est aussi admissible et chaque fois que A* choisit un nœud au début de Open, A* a alors trouvé le chemin optimal vers ce nœud .Le nœud ne sera plus revisité. Si on a deux heuristiques admissibles h_1 et h_2 tel que :

$$h_1(n) < h_2(n)$$

alors $h_2(n)$ conduit plus vite au but. avec h_2, A* explore moins ou autant de nœuds avant d'arriver au but qu'avec h_1. Si h n'est pas admissible, soit b la borne supérieure sur la surestimation du coût, c'est-à-dire, on a toujours :

$$h(n) \leq h^*(n) + b$$

A* retournera une solution dont le coût est au plus b de plus que le coût optimal, c'est-à-dire, A* ne se trompe pas plus que b sur l'optimalité. Si h(n) = h*(n) pour tout état n, l'optimalité de A* est garantie. Étant donné une fonction non admissible, l'algorithme A* donne toujours une solution lorsqu'elle existe, mais il n'y a pas de certitude qu'elle soit optimale.

d - Variation de A*

- Beam search
  - On met une limite sur le contenu de Open et Closed.
  - Recommandé lorsque pas assez d'espace mémoire.
- Iterative deepening
  - On met une limite sur la profondeur.
  - On lance A* jusqu'à la limite de profondeur spécifiée.
  - Si pas de solution, on augmente la profondeur et on recommence A*.
  - Ainsi de suite jusqu'à trouver une solution.
- Recursive Best-First-Search et Simplfied Memory-Bounded A* (SMA*)
  - Variantes de A* qui utilisent moins de mémoire mais peuvent être plus lentes..
- D* (inventé par Stenz et ses collègues)
  - A* dynamique, où le coût des arêtes peut changer durant l'exécution.
  - Évite de refaire certains calculs lorsqu'il est appelé plusieurs fois pour atteindre le même but, suit des changements de l'environnement..

0.4.4 - Application : Jeu d'énigme

8-Puzzle

- État : Configuration légale du jeu
- État initial : Configuration de départ
- État final (but) : Configuration gagnante
- Transition : (voir la figure 0.62)

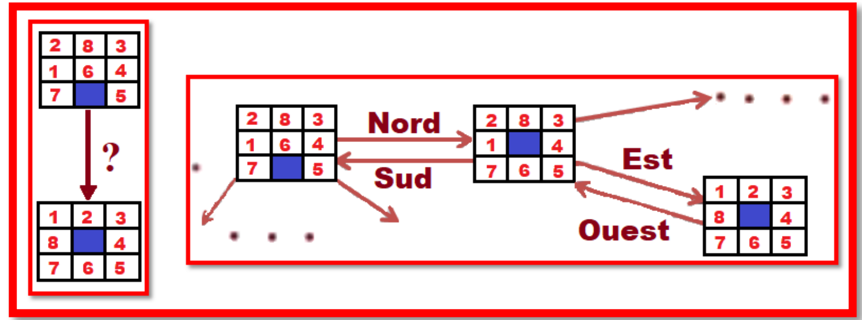


Figure 0.62 – Jeu d'énigme

0.4.4.1 – Application : Planification classique (A*)

Ici, on souhaite qu'un robot va exécuter certaines tâches pour déplacer des marchandises dans entrepôt. Chaque transition correspond à une exécution que peut effectuer notre robot (Goto (x, y) et Take(…)) comme sur la figure 0.63.

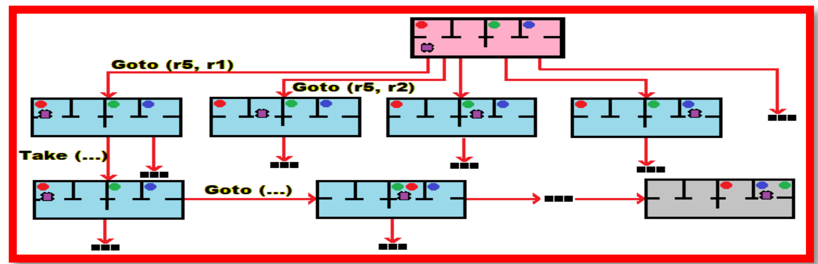
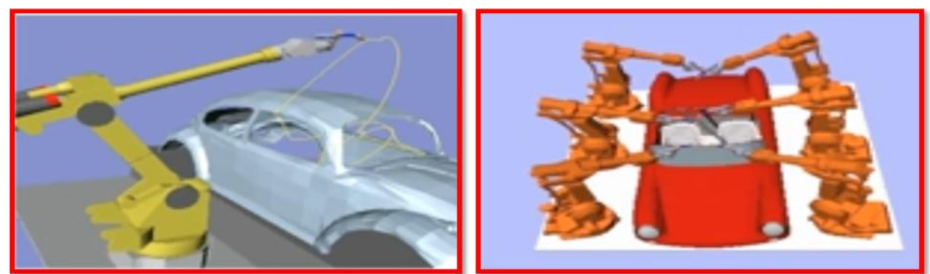


Figure 0.63 – Planification classique (A)*

0.4.4.2 - Application : Industrie automobile

Ici, le déplacement d'un ou des bras mécaniques pour effectuer des tâches de montage, de vissage ou de peinture, etc. sans entrer en collision entre eux, ni avec les pièces traitées via un graphe de transition de façon à optimiser les mouvements des bras mécaniques sans problème majeur comme sur la figure 0.64.



Démos de Motion Planning Kit (Jean-Claude Latombe)

Figure 0.64 – Industrie automobile

0.4.5- Énoncé du problème

Calculer la trajectoire géométrique d'un solide articulé sans collision avec des obstacles statiques via un algorithme **A*** exécuté sur un graphe de transitions pour optimiser les mouvements du robot comme sur la figure 0.65.

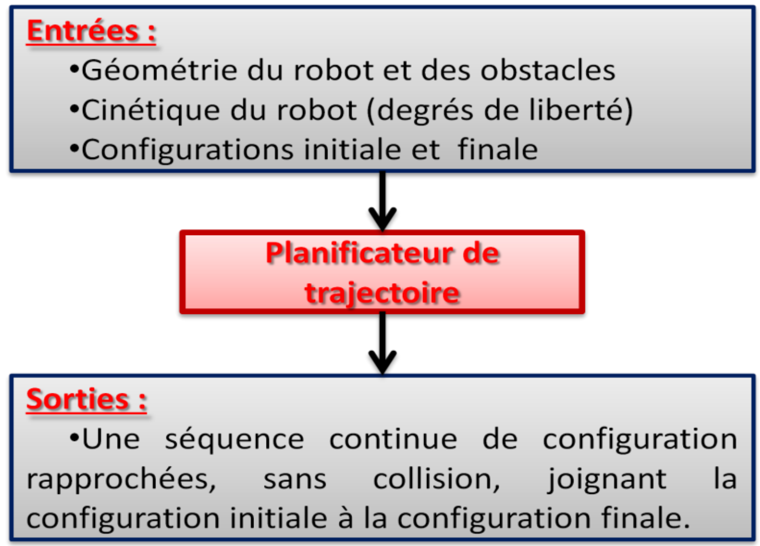


Figure 0.65 - Énoncé du problème

a - Cadre général de résolution des problèmes

Pour pouvoir effectuer des explorations heuristiques dans un graphe, il faut, dans le cas des problèmes continus, exécuter une opération de discrétisation pour pouvoir appliquer par exemple l'algorithme **A*** ou équivalent (voir la figure 0.66).

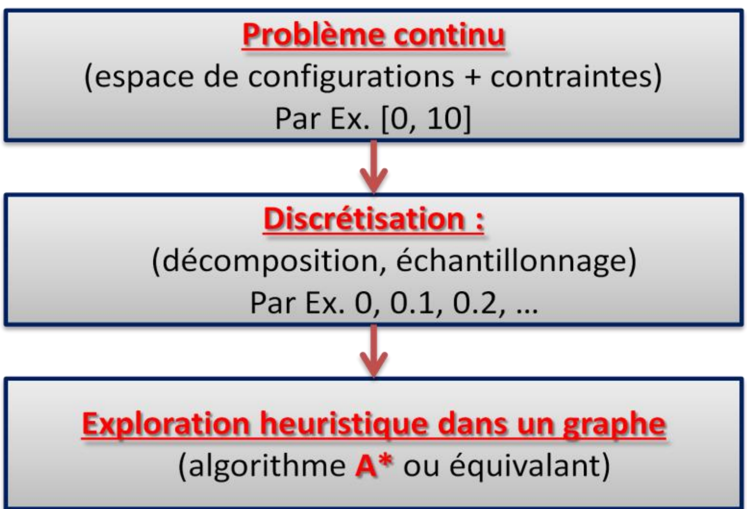


Figure 0.66 – Cadre général de résolution des problèmes

0.4.5.1 - Approche combinatoire par décomposition en cellules

Un autre exemple de problème, on prévoit de discrétiser l'espace ou la description de l'environnement de notre agent, c'est celui où l'on aurait un robot qui se déplacerait dans un environnement donné pour permettre au robot d'effectuer certaines tâches comme sur la figure 0.67.

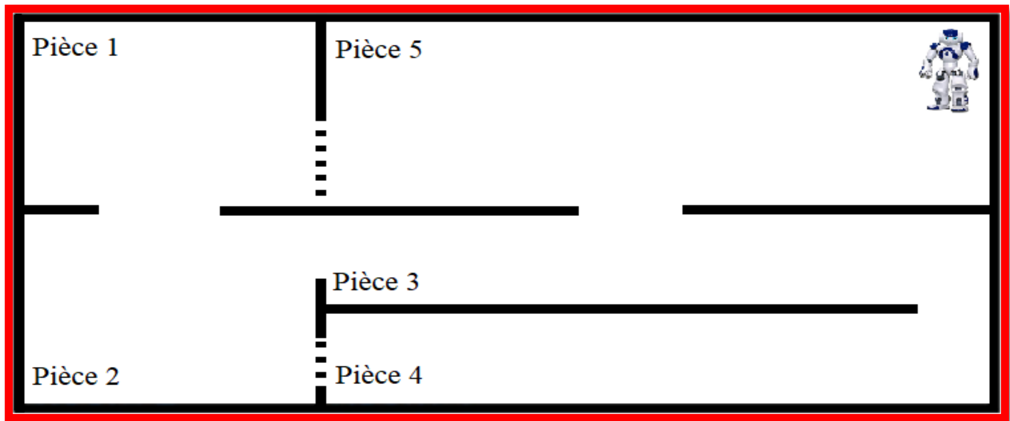


Figure 0.67 – Approche combinatoires par décomposition en cellules

Décomposer la carte en grille :

- 4-connected (illustré ici) ou 8-connected
- Nœud : case occupé par le robot + orientation du robot comme sur la figure 0.67.

Transitions :

- Tourner à gauche →
- Tourner à droite ←
- Aller tout droit

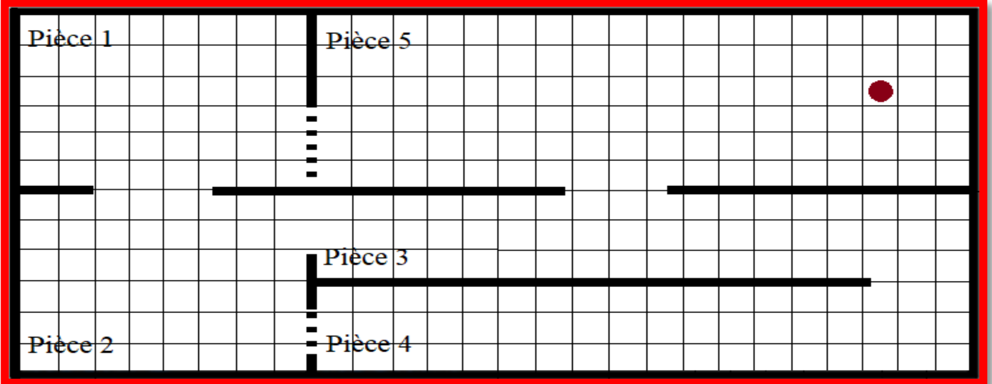


Figure 0.68 – Décomposition en cellules

0.4.6 - L'exploration locale - Objectifs

Il s'agit de comprendre la différence entre une exploration heuristique et une exploration locale, la méthode Hill Climbing, la méthode recuit simulé (simulated annealing) et les algorithmes génétiques.

0.4.6.1 - L'exploration locale - Motivations

Exemple : N-reines

Le problème consiste à placer N reines sur un échiquier de taille N*N de sorte que deux reines ne s'attaquent pas mutuellement : c'est-à-dire, jamais deux reines sur la même diagonale, la même ligne ou la même colonne. Dans notre exemple, on a la solution pour N = 8 comme la figure 0.69.

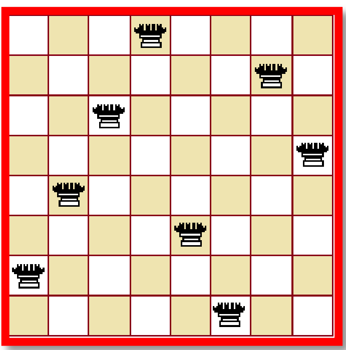


Figure 0.69 – Le jeu des N-reines

0.4.6.2 - L'exploration locale – Hill Climping

a - La méthode Hill Climping

- Entrées :
  - Nœud initial
  - Fonction objectif à optimiser (notée F(n) dans les algorithmes)
  - Fonction générant des nœuds successeurs (voisins)
- Méthode :
  - Le nœud courant est initialisé au nœud initial
  - Itérativement, le nœud courant est comparé à ses successeurs (voisins) immédiats :

- Le meilleur voisin n'ayant la plus grande valeur de F(n') tel que F(n') > F(n) devient le nœud courant;
- Si un tel voisin n'existe pas, on s'arrête et on retourne le nœud courant comme solution.

b - Algorithme Hill Climping

Algorithme HILL-CLIMBING(nœud initial) // cette variante maximise

1. déclarer deux nœuds : n, n'
2. n = nœud initial
3. Tant que (1) // la condition de sortie (exit) est déterminé dans la boucle
4. n'= nœud successeur de n ayant la plus grande valeur F(n')
5. Si F(n') ≤ F(n) // si on minimisait le test serait F(n') ≥ F(n)
6. Retourn n // on n'arrive pas à améliorer p/r à F(n)
7. n = n'

c - Illustration de l'algorithme Hill Climping

Imaginer ce que vous feriez pour arriver à trouver le sommet d'une colline donnée, en plein brouillard et soufrant d'amnésie comme sur la figure 0.70.

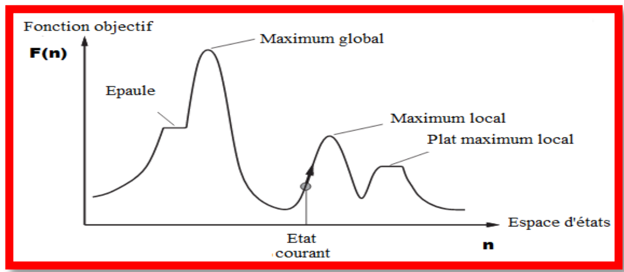


Figure 0.70 – La fonction objective et espace d'états

d - Exemple de simulation de Hill Climping

Soit la fonction objective suivante, définie pour les entiers de 1 à 16 :

n =	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
F(n) =	4	6	15	5	3	2	4	5	6	7	8	10	9	8	7	3

Quelle valeur de n trouverait la méthode Hill Climping si la valeur initiale de n était 6 et que les successeurs (voisins) utilisés étaient n - 1 (seulement si n > 1) et n + 1 (seulement si n < 16)? La réponse est la suite des valeurs de n parcourues : 6 → 7 → 8 → 9 → 10 → 11 → 12. C'est-à-dire que le Hill Climping termine et retourne n = 12.

e – Exemple du jeu des N-reines

Le problème consiste à placer **N** reines sur un échiquier de taille **N*N** de sorte que deux reines ne s'attaquent pas mutuellement : c'est-à-dire, jamais deux reines sur la même diagonale, sur la même ligne ou sur la même colonne comme sur la figure 0.71.

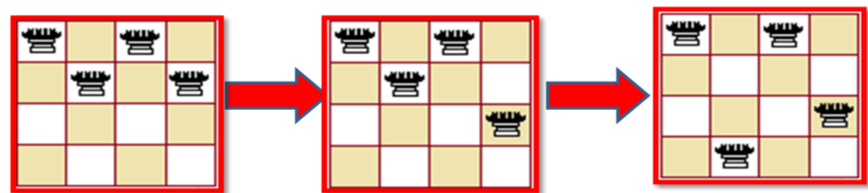


Figure 0.71 – Phases de transition anti attaque du jeu des N-reines

f - Hill Climping avec 8 reines (voir la figure 0.72)

- n : Configuration de l'échiquier avec N reines
- F(n) : nombre de paires de reines qui s'attaquent mutuellement directement ou indirectement dans la configuration n.
- On veut le minimiser F(n) pour l'état (nœud) affiché : 17
- Encadré : les meilleurs successeurs, si on bouge une reine dans sa colonne.

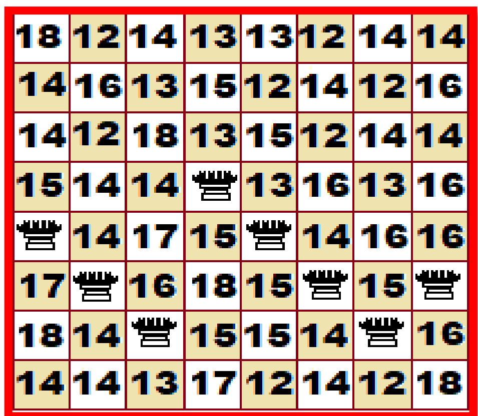


Figure 0.72 – La valeur de F(n) sur l'échiquier

Un exemple de minimum local avec **F(n) = 1**. Comme sur la figure 0.73.

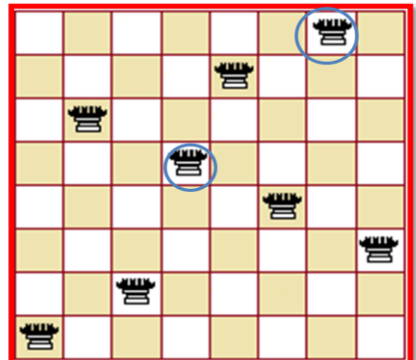


Figure 0.73 – Minimum local avec F(n)

0.4.7 - Méthode recuit simulé (simulated annealing)

C'est une amélioration de l'algorithme Hill Climping pour minimiser le risque d'être piégé dans des maxima/minima locaux. Au lieu de regarder le meilleur voisin immédiat du nœud courant, avec une certaine probabilité, on va regarder un moins bon voisin immédiat. On espère ainsi s'échapper des optima locaux. Au début de la recherche, la probabilité de prendre un moins bon voisin est plus élevée et diminue graduellement. Le nombre d'itérations et la diminution des probabilités sont définis à l'aide d'un schéma (Schedule) de températures, en ordre décroissant : Ex. Schéma de 100 itérations [2^{-0}, 2^{-1}, 2^{-2}, .., 2^{-99}] et la meilleure définition du schéma va varier d'un problème à l'autre.

a - Algorithme recuit simulé

Algorithme SIMULATED-ANNELING(nœudinitial, schema) // cette variante maximise

1. déclarer deux nœuds : n, n'
2. déclarer : t, T, ΔE
3. n = nœudinitial
4. Pour t = 1 … taille (schema)
5. T = schéma[t]
6. n' = successeur de n choisi au hasard
7. ΔE = F(n') - F(n) // si on minimisait, ΔE = F(n) – F(n')
8. Si ΔE > 0 alors assigner n = n' // amélioration p/r à n
9. Sinon assigner n = n' seulement avec la probabilité $e^{\Delta E/T}$

10. retourner n

On remarque que Plus T est petit, plus $e^{\Delta E/T}$ est petite

0.4.8 - D'autres améliorations :

recherche taboue (tabu search)

L'algorithme recuit simulé minimise le risque d'être piégé dans des optima locaux. Par contre, il n'élimine pas la possibilité d'osciller indéfiniment en revenant à un nœud antérieurement visité. L'idée, on pourrait enregistrer les nœuds visités. Ici, on revient à A* et approches similaires. Mais c'est impraticable si l'espace d'états est trop grand. L'algorithme de recherche taboue (tabu search) enregistre seulement les k derniers nœuds visités. L'ensemble tabou est l'ensemble contenant les k nœuds. Le paramètre k est choisi empiriquement. Cela n'élimine pas les oscillations, mais les réduits. Il existe en fait plusieurs autres pour construire l'ensemble tabou.

0.4.9 - D'autres améliorations : exploration par faisceau (beam search)

L'idée, plutôt que de maintenir un seul nœud solution n, on pourrait maintenir un ensemble de k nœuds différents. On commence avec un ensemble de k nœuds choisis aléatoirement. À chaque itération, tous les successeurs des k nœuds sont générés. On choisit les k meilleurs parmi ces nœuds et on recommence. Cet algorithme est appelé exploration locale par faisceau (local beam serach), à ne pas confondre avec la recherche taboue. Variante stochastique de la recherche par faisceau, plutôt que de prendre les k meilleurs, on assigne une probabilité de choisir chaque nœud, même s'il n'est pas parmi les k meilleurs (comme dans le recuit simulé.

0.4.10 - Algorithme génétique

Très similaire à l'exploration locale ou stochastique par faisceau (local ou stochastic beam search). Dans l'algorithme génétique, on commence aussi avec un ensemble de k nœuds choisis aléatoirement, cet ensemble est appelé une population. Un successeur est généré en combinant deux parents. Un nœud est représenté par un mot (chaîne) sur un alphabet : c'est le code générique du nœud. La fonction objectif est appelée fitness function (fonction d'adaptation). La prochaine génération est produite par (1) sélection, (2) croisement et (3) mutation. Inspiré du processus de l'évolution naturelle des espèces, après tout l'intelligence

humaine est le résultat d'un processus d'évolution sur des millions d'années : théorie de l'évolution (Darwin, 1858), théorie de la sélection naturelle (Weismann) et concepts de génétiques (Mendel). La simulation de l'évolution n'a pas besoin de durer des millions d'années sur un ordinateur.

a - Algorithme génétique

Algorithme ALGORITHME-GENETIQUE(k, nb_itérations) // cette variante maximise

1. population = ensemble (n_1, n_2, …, n_k) génère aléatoirement de k chromosomes
2. pour t = 1 … nb_itérations
3. Nouvelle_population = { }
4. pour i = 1 … k
5. n = chromosome pris dans population avec probabilité qui augmente selon F(n)
6. n' = chromosome différent pris dans population – {n} de la même façon
7. n* = résultat du croisement entre n et n'
8. avec petite probabilité, appliquer une mutation à n*
9. Ajouter n* à nouvelle_population
10. population = nouvelle_population

Retourner n dans population avec de F(n) la plus élevée

b – Croisement : exemple avec huit reines

La figure 0.74 illustre le croisement dans jeu des huit reines.

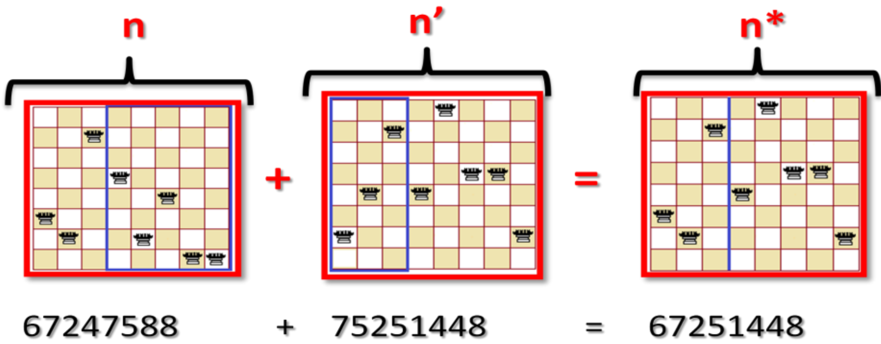


Figure 0.74 – Croisement dans le jeu de huit reines

Fonction d'adaptation : nombre de paires de reines qui ne s'attaquent pas (min = 0, max = 8 *7/2 = 28). La probabilité de sélection du premier chromosome est proportionnelle à l'adaptation : (voir la figure 0.75)

1. 24/(24+23+20+11) = 31%
2. 23/(24+23+20+11) = 29%
3. 20/(24+23+20+11) = 26%
4. 11/(24+23+20+11) = 14%

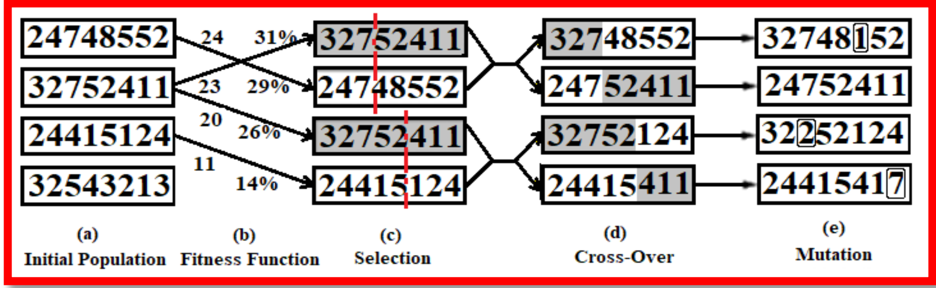


Figure 0.75 – Fonction d'adaptation

Plusieurs autres choix de processus de sélection seraient valides. Ex. on pourrait ne jamais sélectionner les chromosomes faisant parties 25% pires. L'important est que la probabilité qu'un chromosome n soit choisi augmente en fonction de sa valeur F(n).

0.5 - Notion d'agents intelligents

0.5.1 - Agents Intelligents – Motivations

- Définir la notion d'agent intelligent : Voir la notion d'agent rationnel.
- Comprendre l'analyse PEAS (Performance measure, Environment, Actuators and Sensors).
- Observer divers types d'environnements.
- Observer divers types d'agents

a - Deux branches de l'intelligence artificielle

La compréhension de l'intelligence est étudiée à travers deux branches suivantes :

1. Les neurosciences computationnelles avec le développement des modèles mathématiques du fonctionnement du cerveau humain au niveau neuronal.

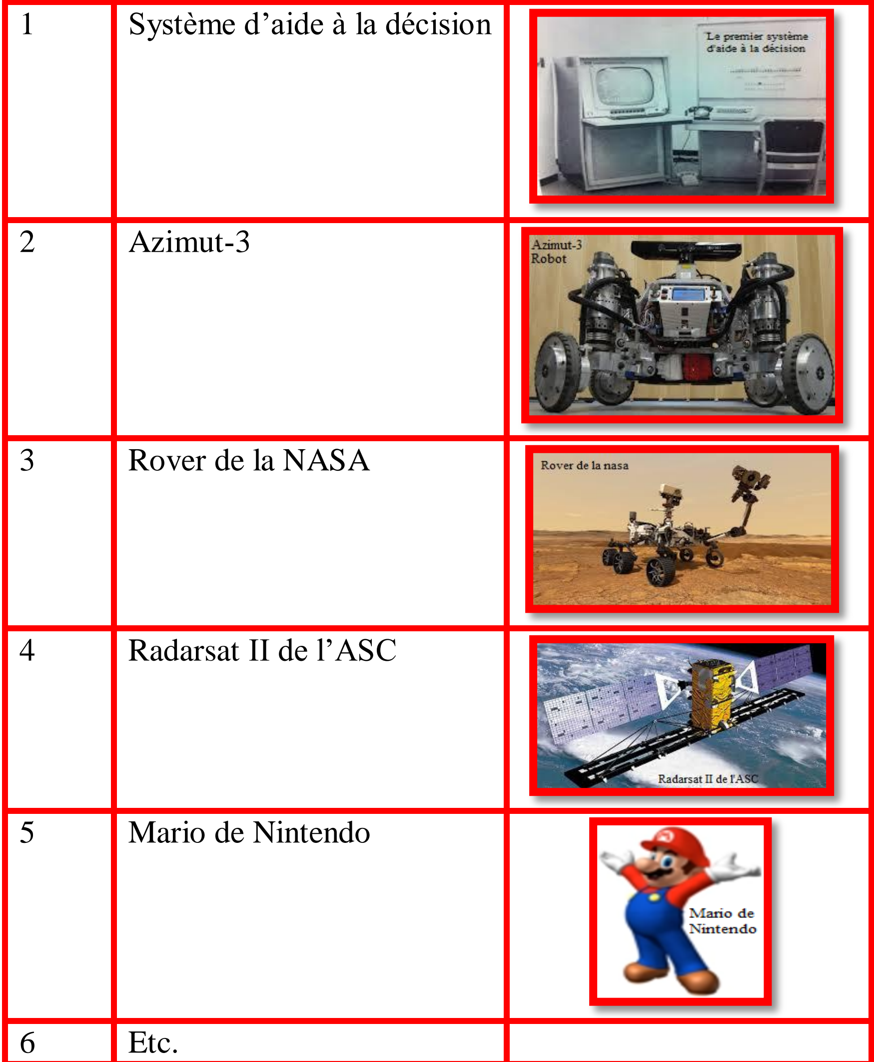
2. Et Les sciences cognitives et psychologie pour comprendre le raisonnement humain et prédire la performance d'un humain pour effectuer une tâche.

Exemple. L'architecture ACT-R pour évaluer le risque couru en parlant au téléphone lors de la conduite d'un véhicule (modèle multitasking chez l'homme).

0.5.2 - Agents intelligents

- Création d'agents intelligents
  - Capacités fondamentales : perception, représentation des connaissances (modélisation), apprentissage, raisonnement et prise de décision.

Exemples d'agents Intelligents comme sur le tableau suivant :



1	Système d'aide à la décision	
2	Azimut-3	
3	Rover de la NASA	
4	Radarsat II de l'ASC	
5	Mario de Nintendo	
6	Etc.	

a - Nécessité d'une intelligence artificielle

- Programmation d'actions voir décisions automatiques
- Programmation d'actions (scripts et machines à états finis)
- Décisions automatiques :
    - Les actions à exécuter ne sont ni scriptées, ni programmées à l'avance, etc.
    - L'agent décide lui-même de ses propres actions à partir d'un certain calcul ou raisonnement.
    - On donne à l'ordinateur la capacité de prendre des décisions intelligentes dans toutes les situations possibles.

b - Qu'est-ce qu'un agent?

Un agent est n'importe quelle entité qui perçoit son environnement par des capteurs ou des senseurs (sensors) et agit sur cet environnement par des actionneurs (actuators). Un agent humain possède des yeux, des oreilles et d'autres senseurs et des mains, des jambes, une bouche et d'autres actionneurs. Un agent logiciel possède un clavier, un accès lecture à un disque dure et autres capteurs, un écran et un accès écriture à un disque dur comme actionneurs.

c - Agent et environnement

Le processus agent "f" prend en entrée une séquence d'observations (perceptions) et retourne une action comme sur la figure 0.76.

$$F : p^* \rightarrow A$$

En pratique le processus est une implémentation par un programme sur une architecture matérielle particulière.

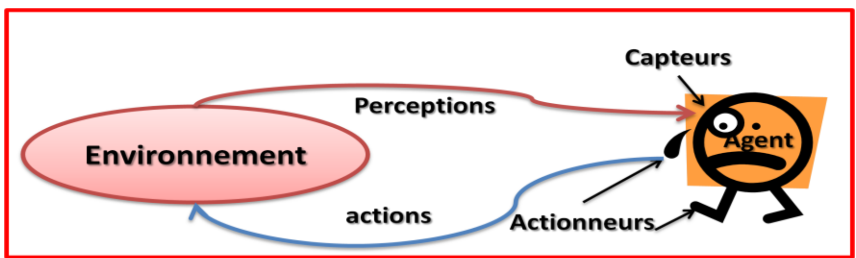


Figure 0.76 - Agent et son environnement

d - Ebauche de l'algorithme d'un Agent

Function SCELETON-AGENT (percept) return action

static memory, the agent's memory of the worlds

memory ← UPDATE-MEMORY(memory, percept)

action ← CHOOSE-BEST-ACTION (memory)

memory ← UPDATE-MEMORY(memory, action)

Return action

e - **Exemple d'agent : Aspirateur robotisé (voir la figure 0.77)**

Les observations (données sensorielles) : position et état des lieux. Par exemple : [A, Clean (ou propre)], [A, Dirty (ou sale)], [B, Clean (ou propre)], [B, Dirty (ou sale). Les actions : Left (ou Gauche), Right (ou Droite), Suck (ou Aspiré) et NoOp (ou pas aspiré et bougé)

"f" :

[A, Clean] →Right

[A, Dirty] → Suck

…

[A, Clean] [A, Clean] [A, Dirty] → Suck

[A, Clean] [A, Clean] [A, Clean] → Suck

…

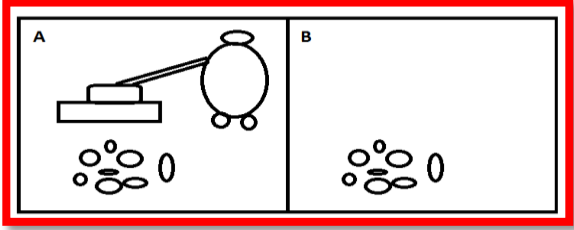


Figure 0.77 – Robot aspirateur

0.5.3 - Qu'est-ce qu'un agent rationnel?

Un agent rationnel doit agir correctement en fonction de ce qu'il perçoit et de ses capacités d'actions. L'action correcte est celle permettant à l'agent de réussir le mieux sa tâche. La mesure de performance est une fonction objective mesurant la qualité d'un comportement de l'agent. Par exemple, une mesure de performance pour le robot aspirateur pourrait être la qualité de déchets aspirés, la propreté des lieux, la durée de la tâche, le bruit

généré, etc. un agent rationnel consiste en une séquence d'observations (données sensorielles) et des connaissances propres, un agent rationnel devrait choisir une action qui maximise la mesure de performance.

La rationalité ne veut pas dire "qui sait tout" (par exemple, connaît tous les effets de ses actions). Elle ne veut pas dire aussi "parfait" :

1. La rationalité maximise la performance espérée;
2. La perfection maximise la performance réelle/actuelle;
3. Mais souvent on ne peut pas connaître la performance réelle avant l'action.

Un agent peut effectuer des actions, d'observations pour cueillir des informations nécessaires à sa tâche. Un agent est autonome s'il est capable d'adapter son comportement en fonction de son expérience et en fonction de sa capacité d'apprentissage et d'adaptation.

0.5.4 - Modèle PEAS

PEAS est un modèle de conception des agents par la spécification des composants majeurs comme les mesures de performance (Performance), les éléments de l'environnement (Environnement), les actions que l'agent peut effectuer (Actionneurs ou Actuators) et les séquences des observations ou perceptions de l'agent (capteurs ou Sensors).

PEAS = **P**erformance, **E**nvironnement, **A**ctuators, **S**ensors.

a -Modèle PEAS pour un robot Taxi

- Agent : Robot Taxi
- Mesures de performance : sécurité, vitesse, respect du code de la route, voyage confortable, maximisation des profits.
- Environnements : Route, trafic, piétons, clients.
- Actionneurs : Volant, changement de vitesse, accélérateur, frein, clignotant, klaxon, etc.
- Capteurs : caméras, sonar, compteur de vitesse, GPS, odomètre, témoins du moteur, etc.

b - Modèle PEAS pour un diagnostique médical automatisé

- Agent : Système de diagnostique médical
- Mesures de performance : Santé des patients, minimisation des coûts, satisfaction des patients.
- Environnements : Patients, hôpital, personnel soignant.

- Actionneurs : Moniteur pour afficher des questions, les résultats des tests ou de diagnostiques, le traitement, etc.
- Capteurs : Clavier et souris pour saisir les symptômes, les réponses aux questions, etc.

0.5.5 - Caractéristiques de l'environnement

Chaque problème aura son propre environnement avec ses caractéristiques particulières. Les caractéristiques que l'on peut distinguer sont complètement observables (voir partiellement observables), déterministe (voir stochastique), épisodique (voir dynamique), discret (voir continu) et agent unique (voir multi-agents).

a - Complètement observable (voir partiellement observable)

Grâce à ses capteurs, l'agent a accès à l'état complet de l'environnement à chaque instant. Le jeu des échecs est complètement observable, car on voit la position de toutes les pièces sur l'échiquier la même chose pour le jeu de dames. Le jeu du poker est partiellement observable, car on ne connaît pas les cartes dans les mains de l'adversaire la même chose pour le jeu des dominos.

b - Déterministe (voir stochastique)

L'état suivant de l'environnement est entièrement déterminé par l'état courant et l'action effectuée par le ou les agents. Le jeu des échecs est déterministe, car le déplacement d'une pièce donne toujours le même résultat. Le jeu du poker est stochastique, car la distribution des cartes est aléatoire. Notes importantes :

- On considère comme stochastique les phénomènes qui ne peuvent pas être prédits d'une manière parfaite.
- On ne tient pas compte des actions des autres agents pour déterminer si l'agent est déterministe ou non.

c - Episodique (voir séquentiel)

Les opérations / comportements de l'agent sont divisés en épisodes, car chaque épisode consiste à observer l'environnement et effectuer une seule action et celle-ci n'a pas d'influence sur l'environnement dans le prochain épisode. La reconnaissance de caractères est épisodique, car la prédiction d'un caractère du système n'influence pas le prochain caractère à reconnaître. Le jeu du poker est séquentiel, car il dépend de la décision de mise ou non d'un joueur, ce qui a un impact sur l'état suivant de la partie.

d - Statique (voir dynamique

L'environnement ne change pas lorsque le ou les agents n'agissent pas. Le jeu des échecs est statique, car l'état du jeu ne change pas si personne ne joue. Le jeu du ping-pong est dynamique, car la balle bouge même si on ne fait rien. Notes importantes : on ne tient pas compte des actions des autres agents pour déterminer si statique ou pas.

e - Discret (voir continu)

S'il a un nombre limité et clairement distincts de données sensorielles et d'actions. Le jeu des échecs est discret, car toutes les actions et les états du jeu peuvent être énumérés. Le conduite automatique d'une voiture est dans un environnement continu, car l'angle du volet est un nombre réel. Le jeu de Ping-pong (ou du Tennis) est dans un environnement continu, car la position de la balle est une paire (x, y) de nombres réels.

f - Agent unique (voir multi-agents)

Un agent opérant seul dans un environnement. Résoudre un Sudoku est à agent unique, car il n'y a aucun adversaire comme les jeux des mots croisés ou des mots fléchés. Le jeu des échecs est multi-agents, car il y a toujours un adversaire comme les jeux de cartes de dames ou de dominos.

Parfois, plus d'une caractéristique est appropriée à un environnement. Dans le jeu de Ping-pong (ou du Tennis), on a la position de la balle est plus simple à concevoir en nombres réels, par contre, sur un écran d'ordinateur, il y a un nombre fini de pixels, etc. Pour le déplacement d'un robot, on a si seul dans un environnement, ses déplacements sont théoriquement déterminés (la physique mécanique est déterministe), par contre, puisqu'un robot ne contrôle pas parfaitement ses mouvements, on préfère normalement le modéliser comme stochastique. On identifie souvent les caractéristiques d'un environnement en réfléchissant à comment on programmerait voir simulerait cet environnement ?

0.5.6 - Les différents types d'agents

On distingue quatre types d'agents :

1. Agents à reflexes simples (Simple reflex agents).
2. Agents à reflexes basés sur des modèles (Model-based reflex agents).
3. Agents basés sur les buts (Goal-based agents)
4. Agents basés sur les utilités (Utility-based agents).

a - Agents à reflexes simples (Simple reflex agents) voir la figure 0.78

Il faut connaître l'état de l'environnement. Étant donné l'état de connaissances, on peut agir en effectuant une action sur l'environnement. Il agit à partir de la perception actuelle en ignorant l'historique des observations précédentes.

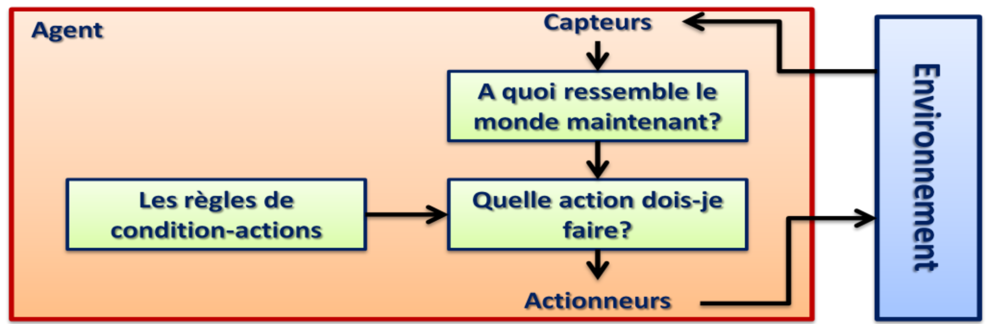


Figure 0.78 – Agents à reflexes simples

Algorithme de principe

Function SIMPLE-REFLEX-AGENT (percept) return an action

Persistent: rules, a set of condition-action rules

state ← INTERPRET-INPUT (percept)

rule ← RULE-MATCH (state, rules)

action ← rule-ACTION

Return action

b - Agents à reflexes basé sur des modèles (Model-based reflex agents)

L'agent accumule l'information dans le temps pour estimer l'état de l'environnement comme sur la figure 0.79.

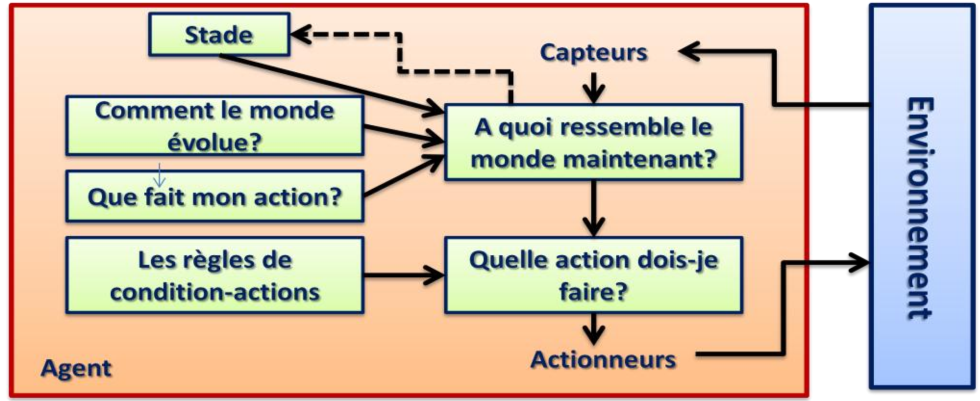


Figure 0.79 - Agents à reflexes basé sur des modèles

c - Agents basés sur les buts (Goal-based agents)

Plutôt que de spécifier une règle conditio-action explicitement, on ne fait que spécifier un but ou un objectif que l'on va pouvoir tenir en compte au futur. On verra la capacité de raisonnement de l'agent pour atteindre le but comme sur la figure 0.80.

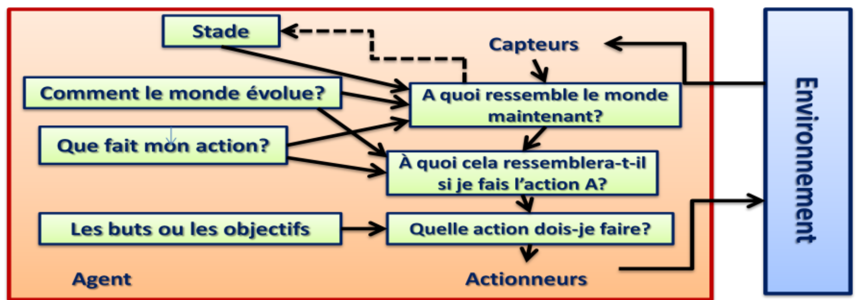


Figure 0.80 - Agents basés sur les buts

d -Agents basés sur les utilités (Utility-based agents)

On utilise une fonction d'utilité pour atteindre un but. On intègre la notion de préférence entre les différentes actions (Ex. action qui résout une tâche donnée le plus rapidement possible) comme sur la figure 0.81.

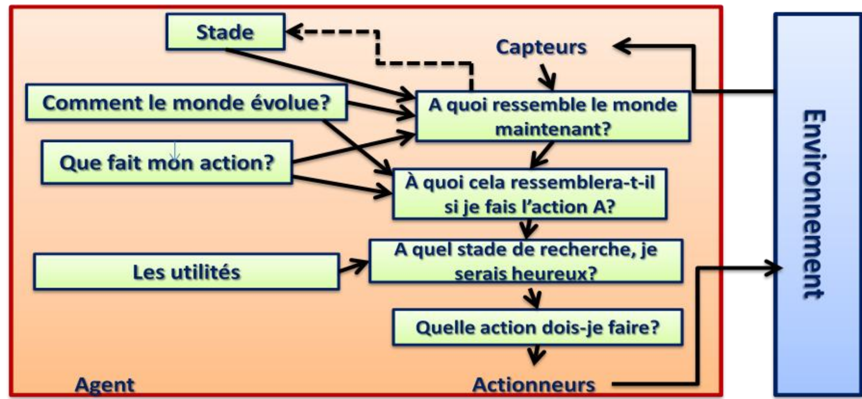


Figure 0.81 - Agents basés sur les utilités

e - Apprentissage chez un agent

Les quatre types d'agents varient dans la façon de prendre leur décision. À partir de quelles connaissances prendre ces décisions? La solution est d'apprendre et de maitriser ces connaissances. Il faut voir plusieurs façons de faire l'apprentissage et ce pour différents types d'agents.

0.6 - l'exploration par le parcours en profondeur dans un graphe : Algorithme DFS

0.6.1 – Le parcours en profondeur dans un graphe

Soit le graphe à gauche "G" suivant et sa copie à droite pour y dérouler le cheminement du parcours. On choisit un sommet au hasard "r" et on commence l'exploration du graphe. Ainsi, on commence l'exploration du graphe "G" par exemple par le sommet "r" et ensuite, on choisit une direction au hasard comme sur la figure 0.82. On n'oubliera pas de procéder à la mémorisation des sommets visités et à la mémorisation des arêtes traversées dans notre cas en couleur verte. Les sommets visités et les arêtes traversées sont colorés en vert au fur et à mesure et les autres restent en couleur noir.

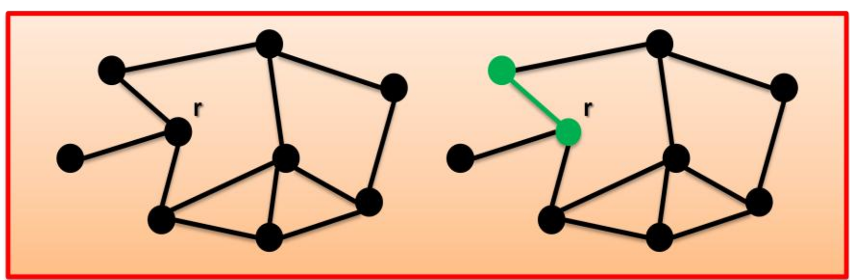


Figure 0.82 – Le graphe G du parcours point de départ "r"

A partir de ce nouveau sommet en vert, alors on entame la poursuite du parcours en profondeur de mon exploration en cherchant le prochain sommet non visité, et ainsi de suite jusqu'à tomber sur un sommet qui n'a pas de sommets suivants non visités et on est coincé comme sur la figure 0.83. Si, on est coincé, il faut revenir sur ses pas ou en arrière. On fait en quelque sorte un backtrack (retour arrière).

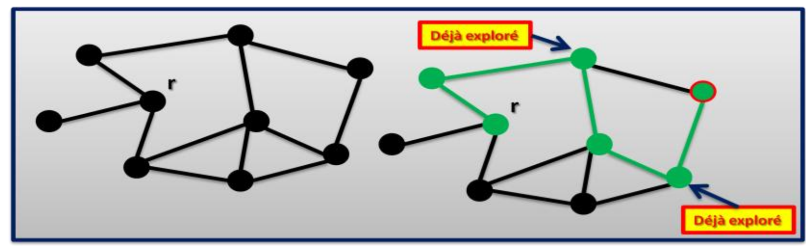


Figure 0.83 – Le fait que l'on soit coincé dans un sommet en rouge

Si on est coincé dans un sommet, on revient en arrière comme le montre la flèche rouge dans figure 0.84, et on entame la poursuite de l'exploration en profondeur à partir de ce nouveau sommet.

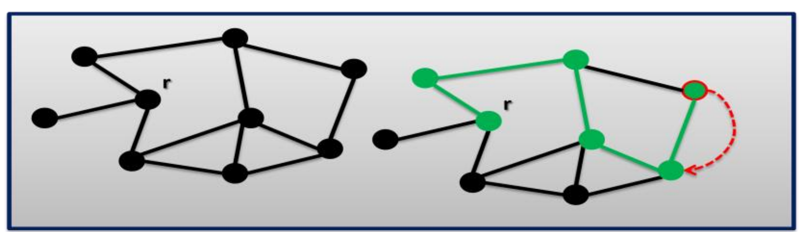


Figure 0.84 – La flèche de retour du sommet de blocage

A partir de ce nouveau sommet, on continue l'exploration comme sur la figure 0.85 et on essaie d'aller en profondeur.

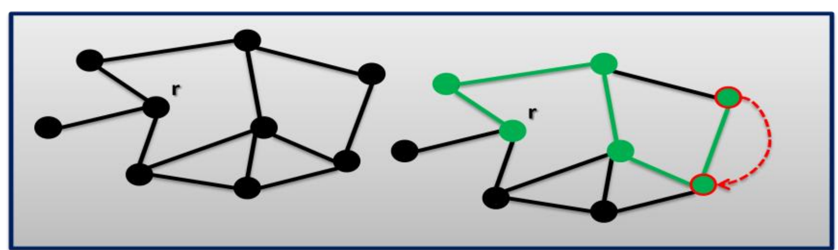


Figure 0.85 – Poursuite de l'exploration à partir du nouveau sommet

On continue l'exploration jusqu'à tomber sur un sommet, où l'on sera coincé parce qu'il n'y a pas de sommets non visités à explorer comme sur la figure 0.86. On est donc de nouveau coincé. Il va falloir revenir de nouveau en arrière pour trouver une solution.

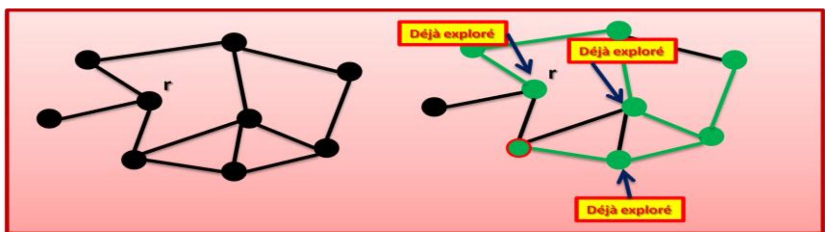


Figure 0.86 – Poursuite de l'exploration avec retour arrière

Revenir sur ses pas, cela veut revenir vers le sommet à partir duquel le sommet courant a été découvert. Il faut mémoriser cette information comme sur la figure 0.87. A chaque fois, qu'on est coincé, alors on revient en arrière.

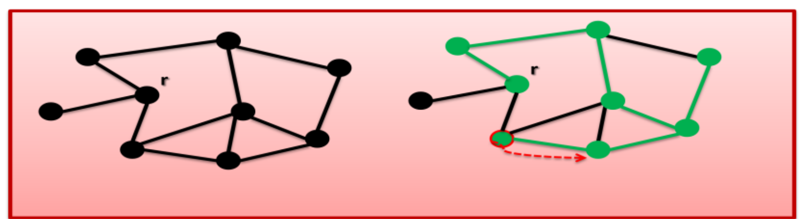


Figure 0.87 – Retour sur ses pas

Ainsi, on revient en arrière, jusqu'au moment où on arrive de nouveau sur le sommet de départ "r" comme la figure 0.88. A partir de sommet "r", on essaie de chercher des sommets non découverts. A partir de sommet "r", on essaie de chercher des sommets non découverts.

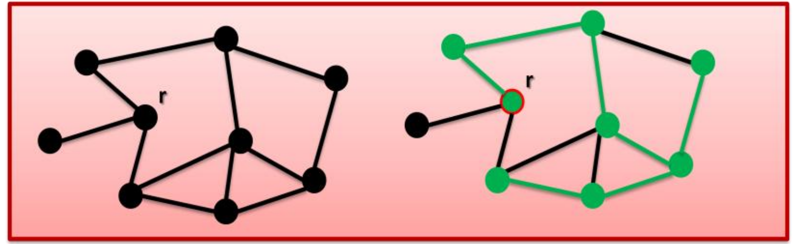


Figure 0.88 – Retour au sommet de départ "r"

Ainsi, de retour sur le sommet de départ "r", on remarque qu'il est impossible d'aller plus loin en profondeur comme sur la figure 0.89. On ne peut pas, découvrir de nouveaux sommets. En fait, on est arrivé à la fin de l'algorithme DFS.

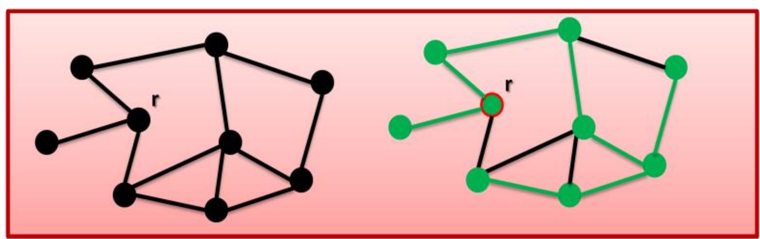


Figure 0.89 – Voir si on peut aller en profondeur à partir du sommet de départ

En résultat, on a parcouru tous les sommets et on est coincés au sommet de départ "r". En supprimant les arêtes non traversées, on obtient arbre couvrant le graphe "G" comme la figure 0.90. C'est un arbre couvrant le graphe "G" initial, si le graphe "G" est connexe.

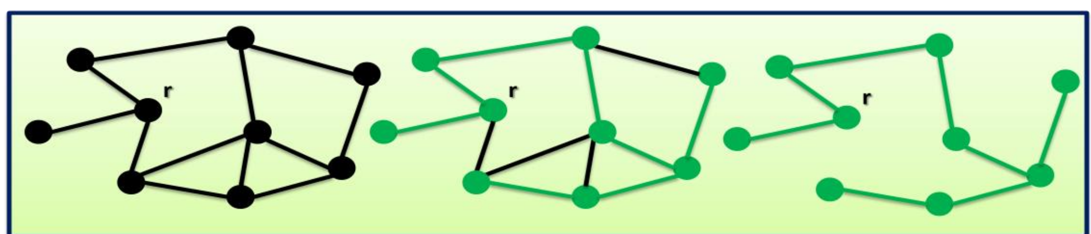


Figure 0.90 – Arbre couvrant du graphe G

0.6.2 – Comparaison avec le parcours en largeur

Ici, on fait juste une comparaison les résultats d'exploration du graphe "G" en effectuant le parcours en largeur (graphe de gauche) et le parcours en profondeur (graphe de droite) comme sur la figure 0.91.

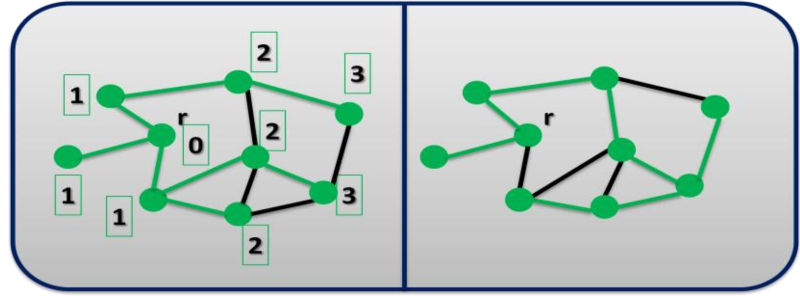


Figure 0.91 – Comparaison avec le parcours en largeur

Si on supprime les arêtes non traversées dans les deux graphes, on obtient vraiment des résultats différents comme sur la figure 0.92. On n'a pas les mêmes arbres couvrants.

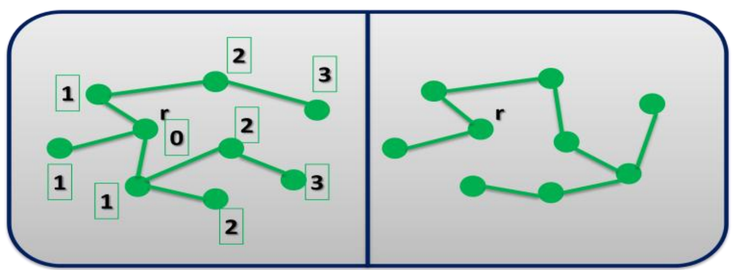
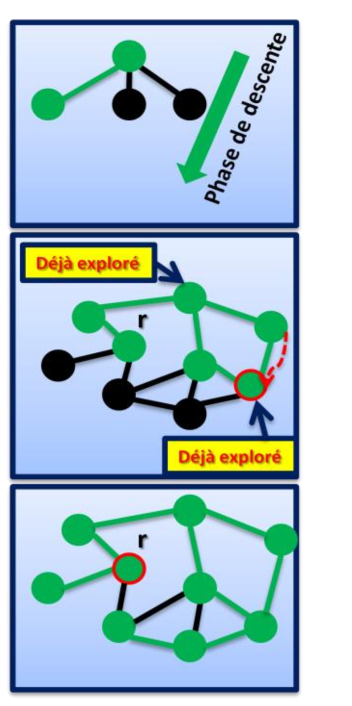


Figure 0.92 – On supprime les arêtes non traversées dans les deux graphes

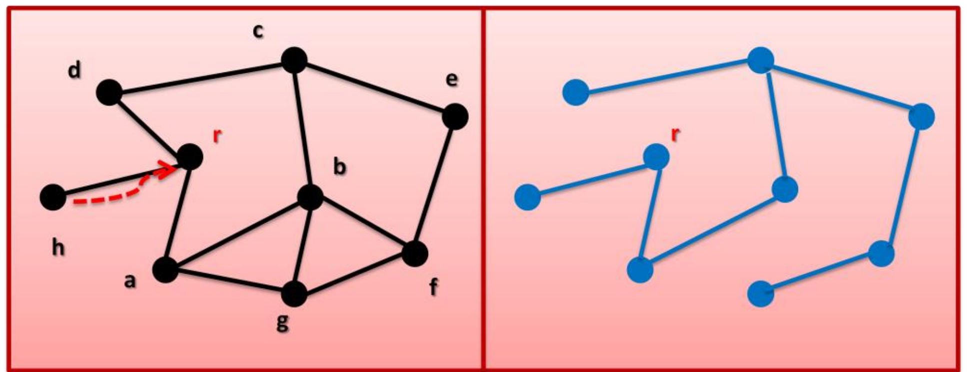
0.6.3 - Résumé des opérations

- Partir d'un sommet "r"
  - Sommet courant, le marquer en couleur vert,
  - Comme étant visité
- Tant que c'est possible :
  - Aller vers un voisin non visité du sommet courant
  - Marquer ce voisin (couleur verte)
  - Mémoriser l'arête par laquelle, on est arrivé
- Si tous les voisins du sommet courant sont déjà visités
  - Revenir en arrière vers le sommet par lequel il a été découvert (backtrack)
  - Phase de descente à partir du sommet courant, etc.
- Si coincé sur le sommet de départ "r"
- Fin de l'algorithme.



0.6.4 - Diversité des choix lors de la descente

Chaque cheminement nous donne un arbre couvrant différent du graphe "G" comme sur la figure 0.93.



0.93 – Les différents arbres couvrants

0.6.5 - Conclusion

L'algorithme DFS permet de construire un arbre couvrant du graphe "G" (si le graphe est connexe). Pour savoir si le graphe "G" est connexe? Il faut appliquer le parcours, si tous les sommets du graphe sont dans l'arbre couvrant alors le graphe est connexe, sinon, le graphe "G" n'est pas connexe. Beaucoup d'applications utilisent cet algorithme.

0.7 – Exploration par le parcours en largeur dans un graphe "G" : Algorithme BFS

C'est un algorithme classique de la théorie des graphes. Si on entre plus dans les détails des choses, il va falloir aborder quelques notions sur les graphes et on parlera un peu de structure des données. Comme les notions de connexité et d'arbre. Soit G = (V, E) un graphe, on dit "G" est connexe ⇔ "G" contient un arbre couvrant T = (V, ET) avec ET $\subseteq$ E comme sur la figure 0.94.

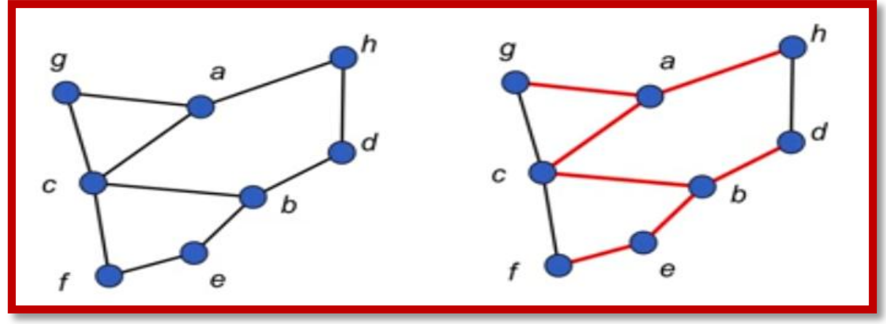


Figure 0.94 – Graphe connexe et arbre couvrant

0.7.1 – Notion de distance

Soit G = (V, E) un graphe, la distance dG (u, v) entre les sommets "u" et "v", c'est la longueur du plus court chemin entre les sommets "u" et "v" dans un graphe "G".

Soit le graphe de la figure 0.95, on a donc :

d (a, b) = 2

d (a, e) = 3

d (b, d) = 1

Etc.

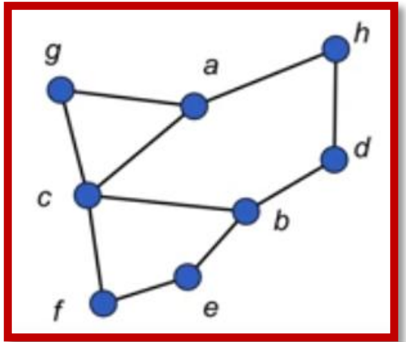


Figure 0.95 – Notion de distance

Si "u" et "v" ne sont pas dans la même composante connexe alors dG (u, v) est infini et la distance d (u, u) = 0 pour tout "u".

0.7.2 - Quels sont les objectifs du BFS?

- En entrée :
  - Un graphe G = (V, E)
  - Un sommet "r" du graphe "G "
- En sortie :
  - Les différentes distances entre le sommet "r" du graphe "G" et chaque sommet du graphe "G"
  - On aura aussi un arbre couvrant, si "G" est connexe,
  - Sinon, on aura un arbre qui va couvrir la composante fortement connexe du graphe "G".

0.7.3 – Quelques tableaux

Ici, on va voir quelques tableaux, qui nous sont nécessaires :

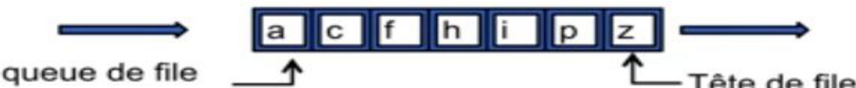
Dist [.], Père [.] et Couleur [.] : trois tableaux à |V| cases chacun.

À chaque sommet "u", on associe :

- Dist [u] : entier. À la fin Dist[u] = dG (r, u)
- Père [u] : sommet. À la fin, on pourra extraire un plus court chemin entre "r" et chaque sommet du graphe "G".
- Couleur[u] : {Blanc, Noir, Gris} Marque sur les sommets pour ne pas tourner en rond dans le graphe "G".

0.7.4 – Un outil : une file de sommets

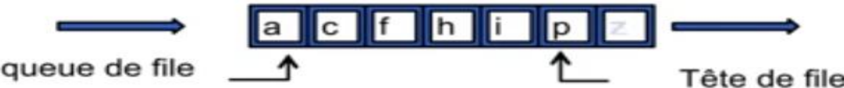
Soit une file "F" de sommets, comme suit :



On entre par la queue de la file et on sort par la tête de la file. On a les primitives d'accès à cette file "F" :

- Tête (F) : Retourne l'élément en tête de la file "F" sans le supprimer de la file "F".
- Défiler (F) : Retourne l'élément en tête de la file "F" en le supprimant de la file "F".

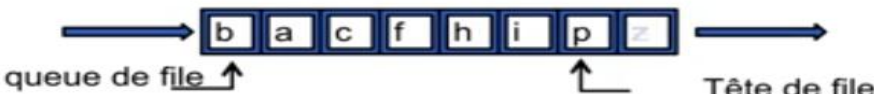
Si défile "z", on aura la file suivante :



Par convention, sur le schéma, si on fait Défiler (z), on va décaler la tête de file sur "p" et on mettre "z" en couleur invisible. On rajoute une autre primitive d'accès à cette file "F" :

- Enfiler (F, u) : Mettre l'élément "u" dans la file "F" (en queue de la file "F" bien sûr).

Si enfile "b", on aura la file suivante :



Si on fait Enfiler (F, b), on va ajouter "b" en queue de la file "F". On rajoute une autre primitive d'accès à cette file "F" :

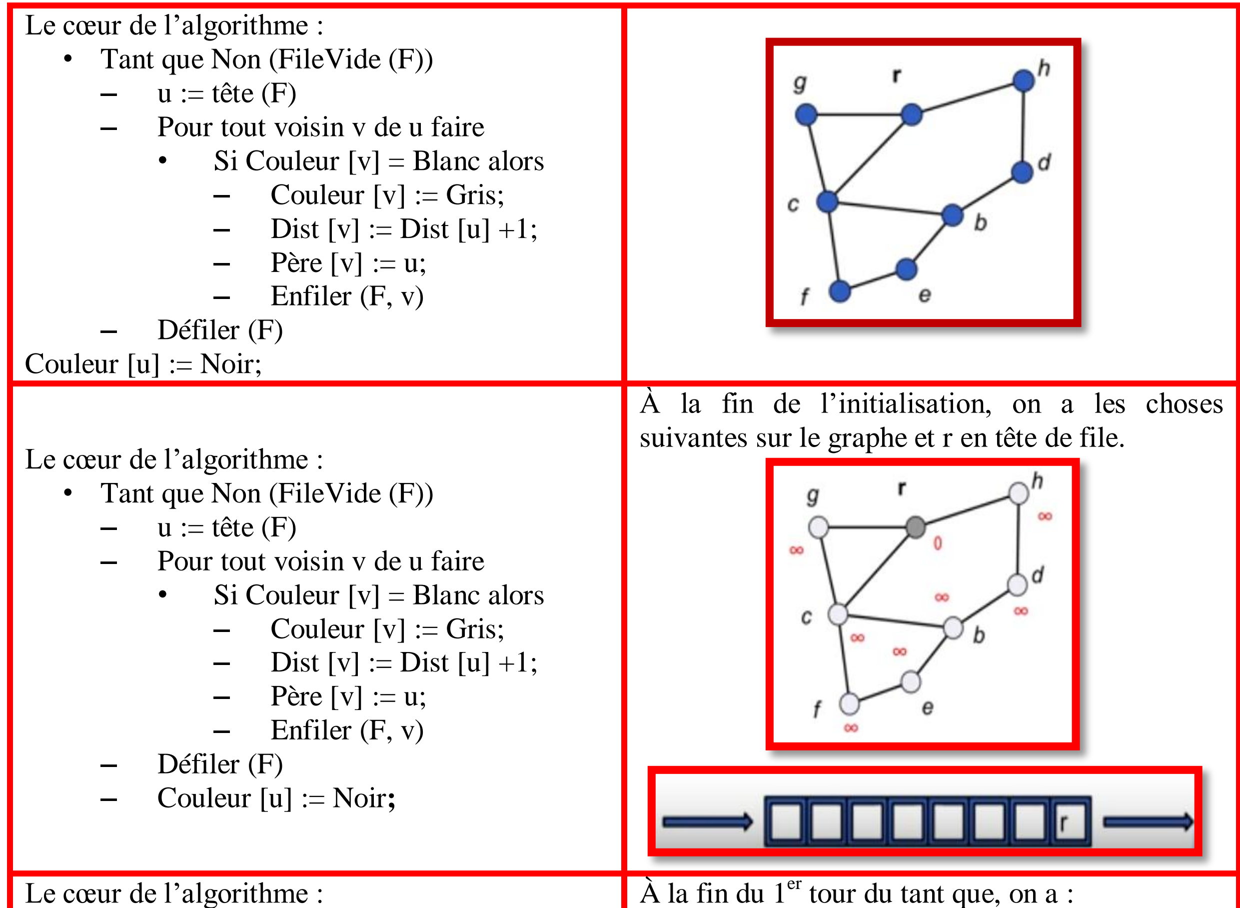
- FileVide (F) : Booléen (Vrai si seulement si la file "F" est vide).

0.7.5 - Le BFS (initialisations)

- En entrée : G = (V, E) et r
- Var : Dist [.], Père [.], Couleur [.], u, v, file F (vide)
- Initialisation :
- Pour tout u de V faire
    - Couleur [u] := Blanc; /* Pas encore visité */
    - Père [u] := NIL; /* Pas encore de chemin pour venir en u -/
    - Dist [u] := +infini /* en pratique |V| +1 par ex. */
- Enfiler (F, r) /* Le 1er sommet exploré va être r */
- Couleur [r] := Gris; /* r est en cours de traitement */
- Dsit [r] := 0 /* on sait que dG (r, r) = 0 */

0.7.6 - BFS en action

Le tableau suivant illustre le déroulement de l’algorithme BFS.



Le cœur de l’algorithme : • Tant que Non (FileVide (F)) – u := tête (F) – Pour tout voisin v de u faire • Si Couleur [v] = Blanc alors – Couleur [v] := Gris; – Dist [v] := Dist [u] +1; – Père [v] := u; – Enfiler (F, v) – Défiler (F) Couleur [u] := Noir;	g r h d c b f e
Le cœur de l’algorithme : • Tant que Non (FileVide (F)) – u := tête (F) – Pour tout voisin v de u faire • Si Couleur [v] = Blanc alors – Couleur [v] := Gris; – Dist [v] := Dist [u] +1; – Père [v] := u; – Enfiler (F, v) – Défiler (F) – Couleur [u] := Noir;	À la fin de l’initialisation, on a les choses suivantes sur le graphe et r en tête de file. g r h ∞ 0 ∞ d ∞ ∞ c b ∞ ∞ f e ∞ r
Le cœur de l’algorithme :	À la fin du 1er tour du tant que, on a :

- Tant que Non (FileVide (F))
  - u := tête (F)
  - Pour tout voisin v de u faire
    - Si Couleur [v] = Blanc alors
      - Couleur [v] := Gris;
      - Dist [v] := Dist [u] +1;
      - Père [v] := u;
      - Enfiler (F, v)
  - Défiler (F)

Couleur [u] := Noir;

"c", "g" et "h" passe à la couleur grise et "r" passe au noir et sort de la file.

Le cœur de l'algorithme :

- Tant que Non (FileVide (F))
  - u := tête (F)
  - Pour tout voisin v de u faire
    - Si Couleur [v] = Blanc alors
      - Couleur [v] := Gris;
      - Dist [v] := Dist [u] +1;
      - Père [v] := u;
      - Enfiler (F, v)
  - Défiler (F)

Couleur [u] := Noir;

À la fin du 2ème tour du tant que, on a :

Ici, l'examen de "g" n'a rien modifié (sauf sa couleur et il sort de la file).

Le cœur de l'algorithme :

- Tant que Non (FileVide (F))
  - u := tête (F)
  - Pour tout voisin v de u faire
    - Si Couleur [v] = Blanc alors
      - Couleur [v] := Gris;
      - Dist [v] := Dist [u] +1;
      - Père [v] := u;
      - Enfiler (F, v)
  - Défiler (F)

Couleur [u] := Noir;

À la fin du 3ème tour du tant que, on a::

L'examen de "h" permet d'atteindre "d".

Le cœur de l'algorithme :

- Tant que Non (FileVide (F))
  - u := tête (F)
  - Pour tout voisin v de u faire
    - Si Couleur [v] = Blanc alors
      - Couleur [v] := Gris;
      - Dist [v] := Dist [u] +1;
      - Père [v] := u;
      - Enfiler (F, v)
  - Défiler (F)

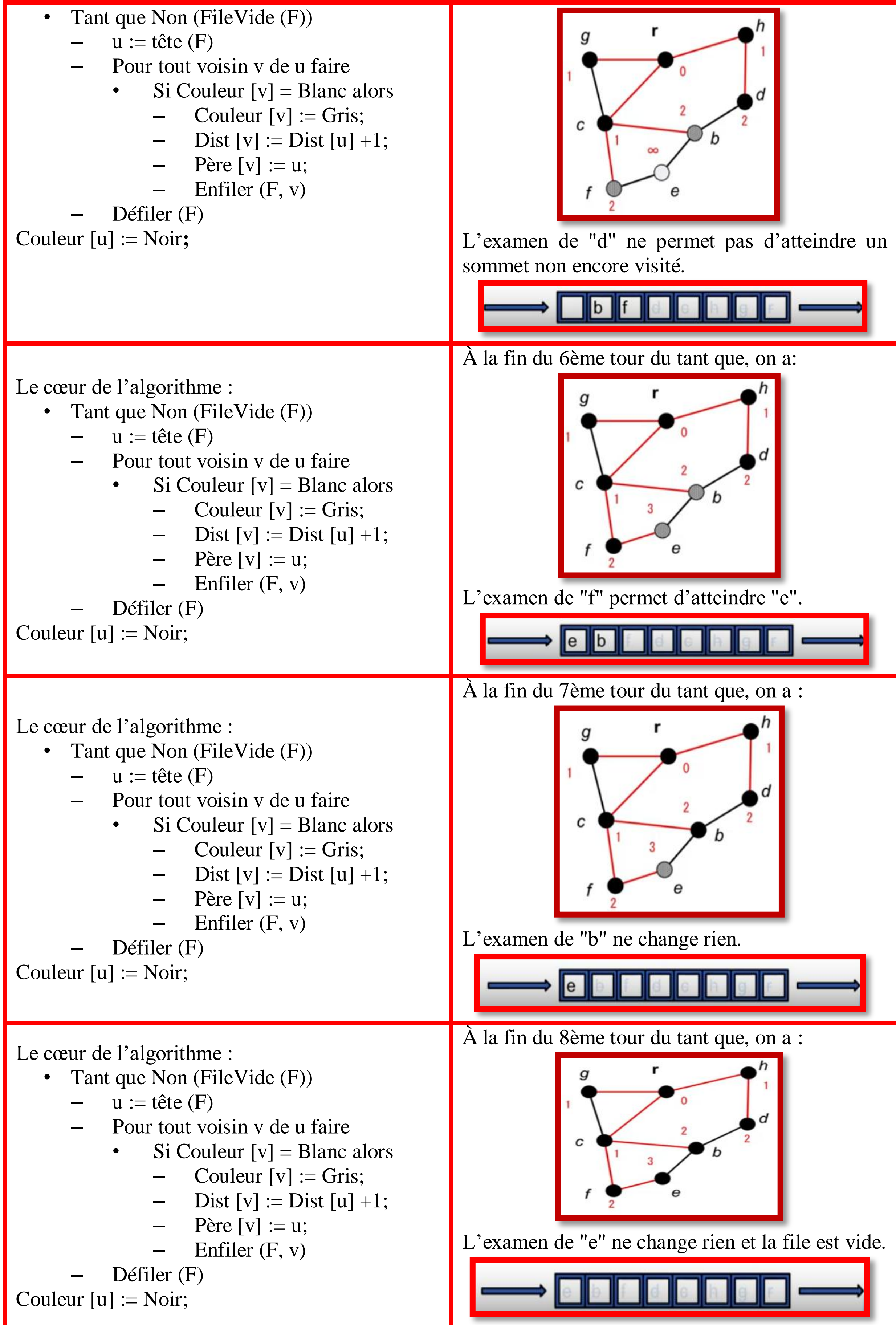
Couleur [u] := Noir;

À la fin du 4ème tour du tant que, on a :

L'examen de "c" permet d'atteindre "f" et "b".

Le cœur de l'algorithme :

À la fin du 5ème tour du tant que, on a :



- Tant que Non (FileVide (F))
  - u := tête (F)
  - Pour tout voisin v de u faire
    - Si Couleur [v] = Blanc alors
      - Couleur [v] := Gris;
      - Dist [v] := Dist [u] +1;
      - Père [v] := u;
      - Enfiler (F, v)
  - Défiler (F)

Couleur [u] := Noir;

L'examen de "d" ne permet pas d'atteindre un sommet non encore visité.

Le cœur de l'algorithme :

- Tant que Non (FileVide (F))
  - u := tête (F)
  - Pour tout voisin v de u faire
    - Si Couleur [v] = Blanc alors
      - Couleur [v] := Gris;
      - Dist [v] := Dist [u] +1;
      - Père [v] := u;
      - Enfiler (F, v)
  - Défiler (F)

Couleur [u] := Noir;

À la fin du 6ème tour du tant que, on a:

L'examen de "f" permet d'atteindre "e".

Le cœur de l'algorithme :

- Tant que Non (FileVide (F))
  - u := tête (F)
  - Pour tout voisin v de u faire
    - Si Couleur [v] = Blanc alors
      - Couleur [v] := Gris;
      - Dist [v] := Dist [u] +1;
      - Père [v] := u;
      - Enfiler (F, v)
  - Défiler (F)

Couleur [u] := Noir;

À la fin du 7ème tour du tant que, on a :

L'examen de "b" ne change rien.

Le cœur de l'algorithme :

- Tant que Non (FileVide (F))
  - u := tête (F)
  - Pour tout voisin v de u faire
    - Si Couleur [v] = Blanc alors
      - Couleur [v] := Gris;
      - Dist [v] := Dist [u] +1;
      - Père [v] := u;
      - Enfiler (F, v)
  - Défiler (F)

Couleur [u] := Noir;

À la fin du 8ème tour du tant que, on a :

L'examen de "e" ne change rien et la file est vide.

0.7.7 - BFS en action : résultat

Le tableau Père [.] est pour le codage d'un arbre "T" couvrant "G" ssi "G" est connexe, sinon un arbre couvrant de la composante connexe dans laquelle se trouve "r". La propriété est que pour tout "u" de "G", dG (r, u) = dT (r, u). Cette propriété est vraie quelque soit l'ordre de choix des voisins à chaque étape comme sur la figure 0.96.

Synthèse du résultat:

L'état des marques à la fin de l'algorithme:

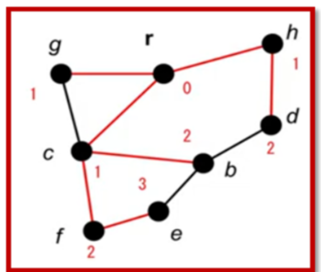


Figure 0.96 – BFS - Résulta

0.7.8 - BFS en action : synthèse

- Les sommets sont découverts niveau par niveau
  - Parcours en largeur
- Tableau Père :
  - Cailloux du petit Poucet
- Couleur :
  - Evite de créer des cycles
- L'ordre d'examen des voisins n'a pas d'importance
  - Arbres différents mais mêmes propriétés
- L'implémentation concrète de la file n'a pas d'importance
  - En langage C par exemple, on peut utiliser des listes chaînées, des pointeurs, des tableaux, etc.

0.7.9 - BFS en action : le produit

- À la fin du BFS, on peut retourner :
  - Le tableau Dist [.]
  - Le tableau Père [.]
  - Ou les deux

- Dans certaines applications (test de connexité), on peut vouloir le tableau Couleur [.], …
- Calculer toutes les distances à partir d’un sommet donné.
- Faire le test de connexité ("G" est connexe si et seulement si à la fin tous les sommets sont noirs.
- Construire un arbre couvrant ("G")
  - L’arbre respecte les distances
- Tester si le graphe est bipartie, c’est-à-dire s’il peut être coloré en deux couleurs.

0.8 - Recherche des plus courts chemins dans un graphe "G " et Notions de distance & de parcours en largeur

0.8.1 – Le déroulement de l’algorithme

Il peut y avoir plusieurs plus courts chemins dans un graphe. La question est comment trouver ces plus courts chemins ? Par exemple, si on cherche les plus courts chemins entre les sommets "r" et "A", on va avoir plusieurs chemins comme sur la figure 0.97.

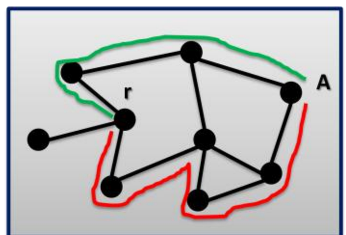
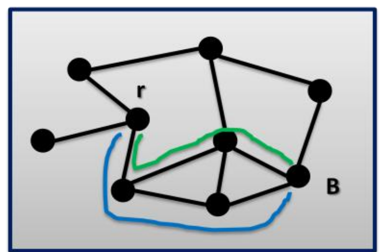


Figure 0.97 – Graphe pour la recherche du plus court chemin entre deux sommets

On a par exemple sur la figure 0.97 deux chemins pour aller du sommet "r" vers le sommet "A". Le chemin vert est plus intéressant que le chemin rouge, il est plus court. Il existe plusieurs courts chemins. Comment faire ? Et lequel choisir ? Il peut y avoir plusieurs plus courts chemins dans un graphe comme sur la figure 0.98. Par exemple, on a deux plus courts chemins entre les sommets "r" et "B" de longueur 03. La longueur d’un chemin est le nombre d’arêtes entre le sommet de départ et le sommet d’arrivée.



0.98 – Plusieurs plus cours chemins entre deux sommets

La question est de savoir comment trouver les plus courts chemins? Comment les calculer ? Comment les construire ? Pour ce faire, on va l'illustrer par un algorithme du parcours en largeur. On prendra le graphe "G" de gauche et le graphe "G" de droite pour y illustrer la marche de l'algorithme comme sur la figure 0.99. On prendra au hasard le sommet "r" comme point de départ et on exécutera le parcours dans le graphe de droite.

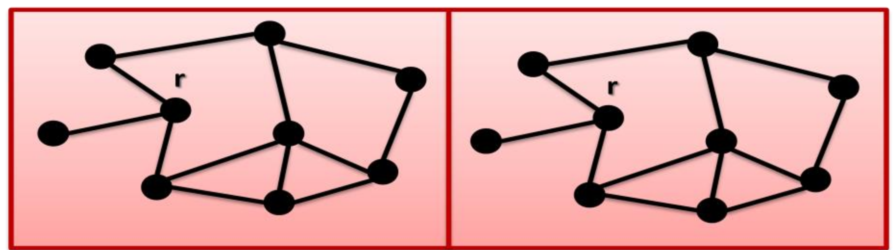


Figure 0.99 – Le graphe de départ à gauche et le graphe d'exécution à droite

On part du sommet "r" et il est à distance 0 de lui-même. On parcourt les sommets en largeurs comme sur la figure 0.100. On mémorise les arêtes traversées (en vert) et les sommets atteints (en vert aussi). Les premiers sommets plus proches sont à distance de 01 du sommet "r".

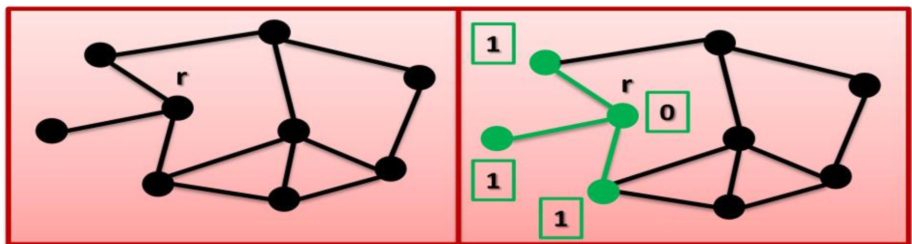


Figure 0.100 – Parcours des sommets en largeurs à partir du sommet "r"

A partir de ces trois sommets, on continue de rechercher les sommets voisins proches. On va trouver trois autres sommets et la distance sera de 02 comme sur la figure 0.101.

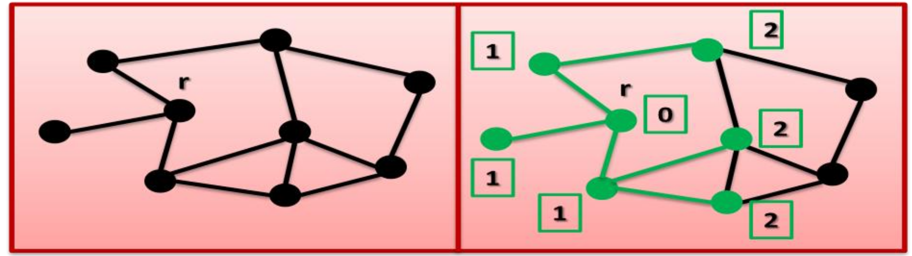


Figure 0.101 – Recherche des sommets voisins proches

On prend le sommet à distance 02 et on examine les voisins. On va chercher les sommets non atteints comme sur la figure 0.102.

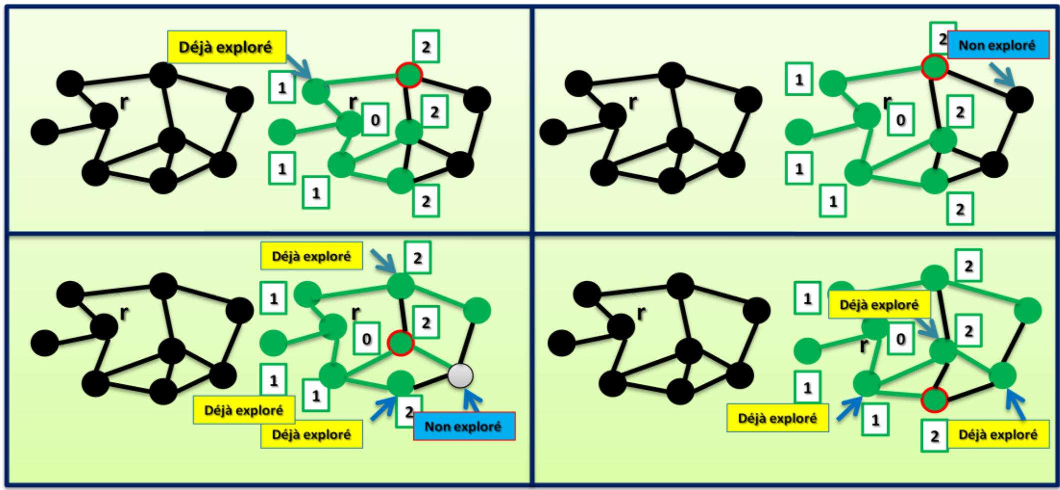


Figure 0.102 – Prise des sommets à distance égale à 2

Cela n’amène rien de nouveau. On va numéroter les sommets trouvés par la distance 03. On continue jusqu’à ce qu’il n’y a plus de sommets à explorer. Et c’est la fin de l’algorithme comme sur la figure 0.103.

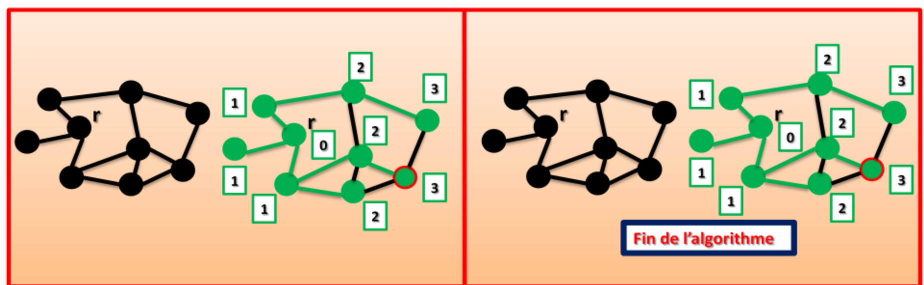


Figure 0.103 – Fin de l’algorithme

Maintenant, on va représenter ce résultat sous une autre forme. C’est le graphe de gauche qui est une représentation par niveau de distance comme sur la figure 0.104.

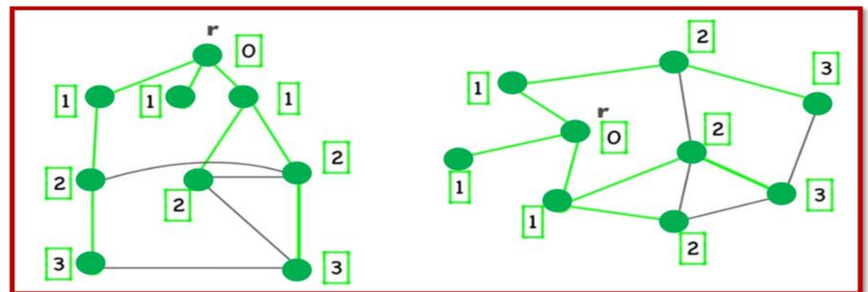


Figure 0.104 – Représentation par niveau de distance

Si on supprime les arêtes non traversées, on obtient un arbre couvant du graphe de départ comme sur la figure 0.105.

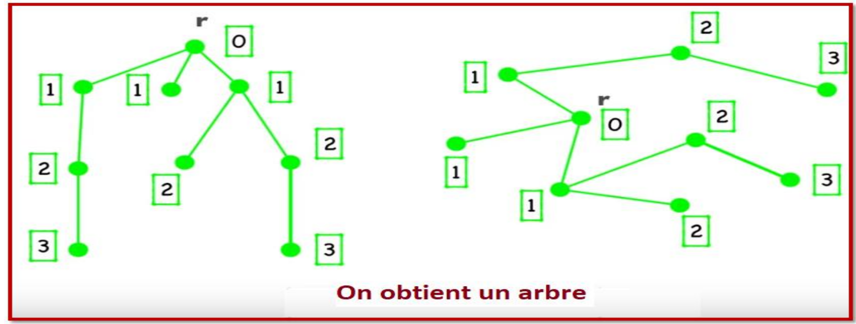


Figure 0.105 – Arbre couvrant

Si on fait un petit résumé des opérations. On part de du sommet r et on explore les sommets voisins par niveau jusqu'à ce qu'il n'y a plus rien à explorer comme sur la figure 0.106.

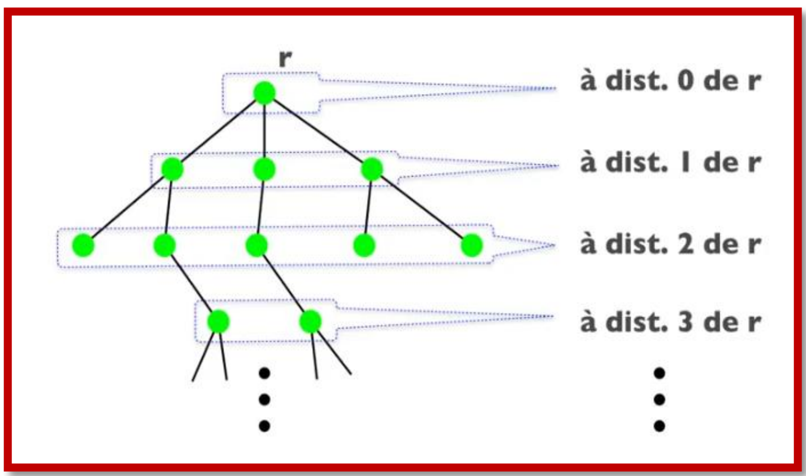


Figure 0.106 – Recherche des sommets à explorer par distance

0.8.2 - Résumé

- Distance entre u et v est la longueur du plus court chemin entre u et v.
- Parcours en largeur :
  - Construire un arbre couvrant (si et seulement si le graphe est connexe)
  - Arbre de plus courts chemins.
- Savoir si G est Connexe ?
  - Appliquer le parcours
  - Si tous les sommets sont dans l'arbre alors G est connexe, sinon G n'est connexe.

0.8.3 - Si les arêtes ne sont pas toutes de même longueurs ?

La figure 107 illustre la recherche du plus court chemin entre deux sommets d'un graphe où les arêtes n'ont pas les mêmes longueurs, d'où la notion des trajets hétérogènes.

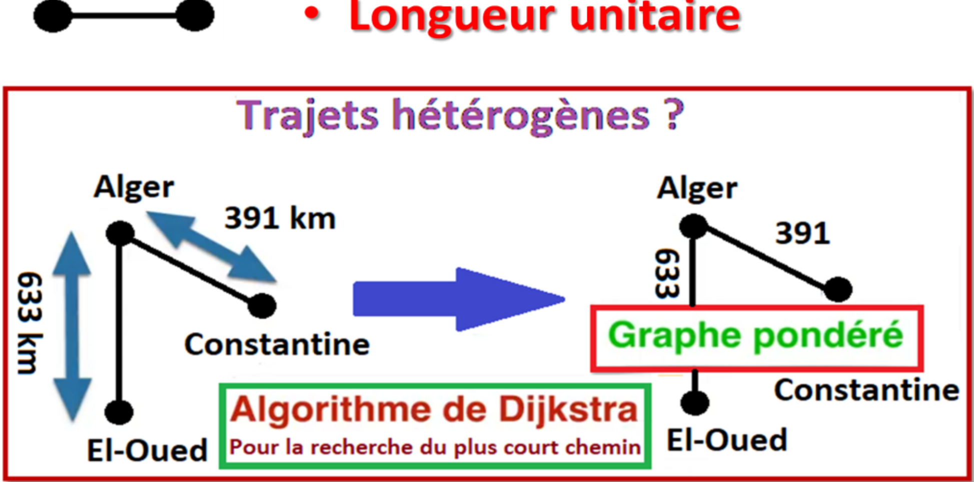


Figure 0.107 – Recherche du plus chemin dans des trajets hétérogènes