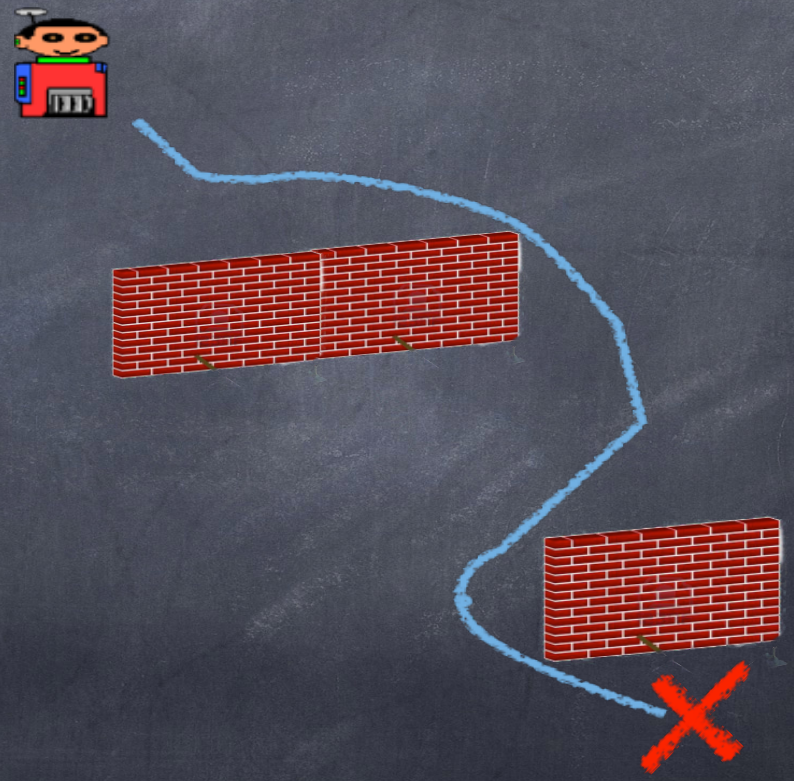
Algorithmes d'exploration (search) pour un agent

Algos: Best-first, Dijkstra, A*

Jacques Ferber
LIRMM - Université de Montpellier II

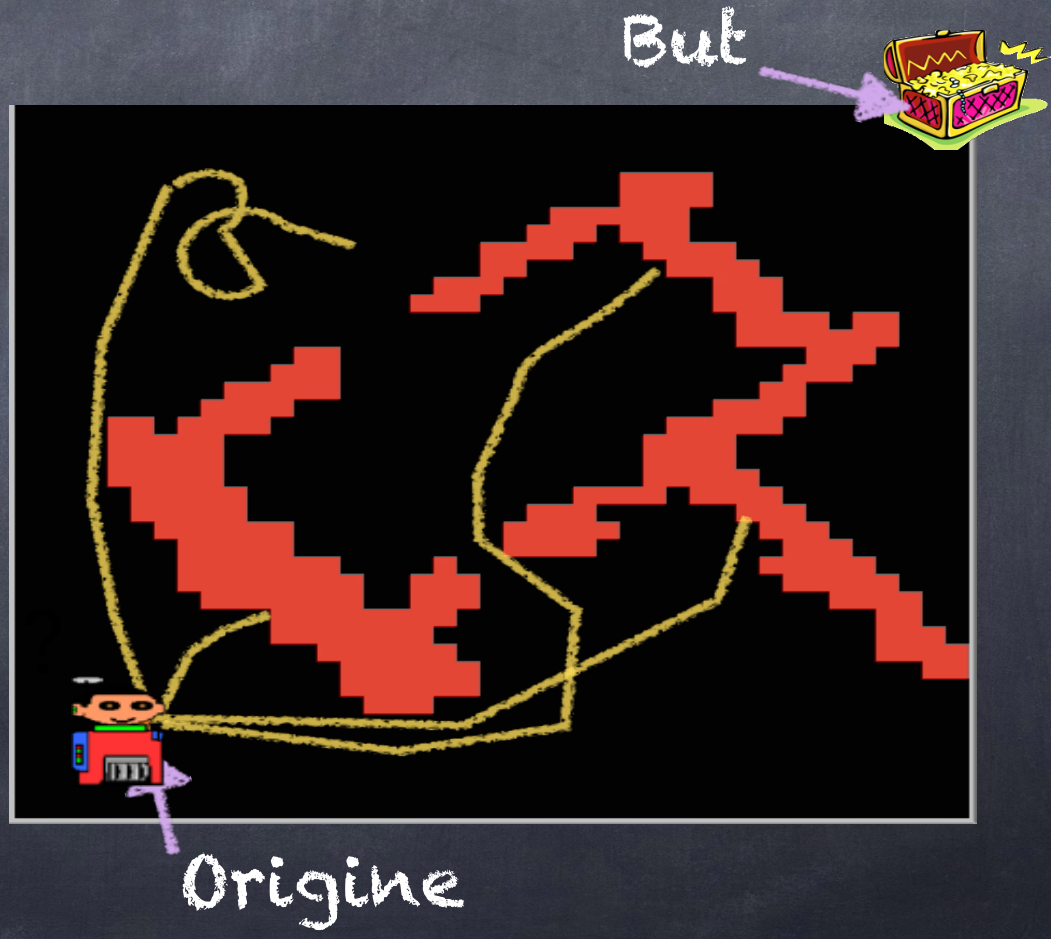
Problème: comment aller d'un point à un autre?

- Quel algorithme utiliser pour planifier un chemin pour aller d'un point à un autre?
  - Différent de la situation réactive (agents situés) où l'agent avance et trouve son chemin au fur et à mesure
  - On prendra en compte un environnement défini sous forme de grille



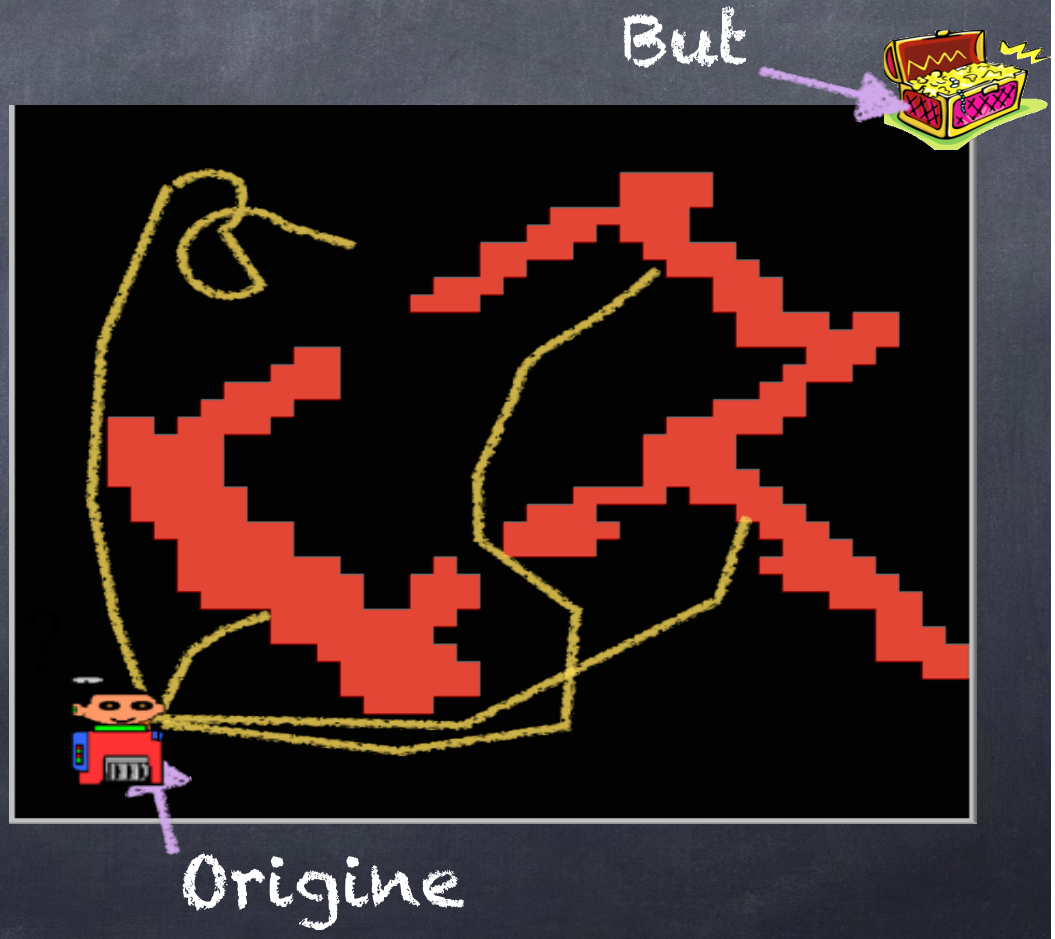
Aller d'un point à un autre

- L'agent (triangle rouge en bas à gauche) cherche à aller au but (carré jaune en haut à droite)
- Comment trouver sa route?
- Comment trouver le «meilleur» chemin ou tout du moins un «bon» chemin



Les difficultés

1. Trouver le but le plus rapidement possible
2. Ne pas rester bloqué par les obstacles
3. Optimiser le chemin une fois trouvé le but

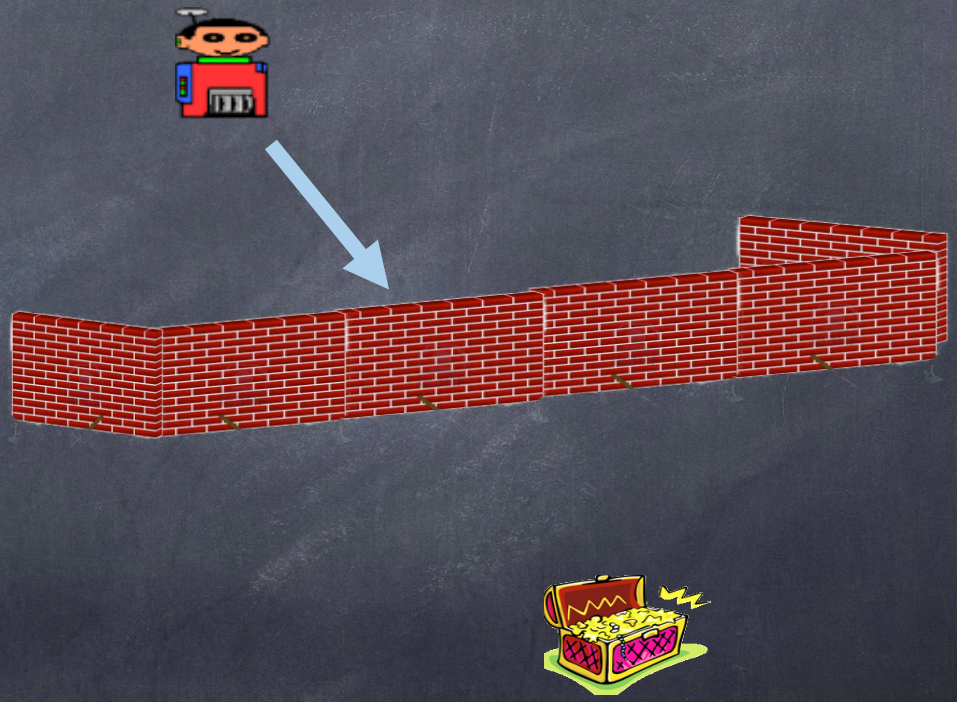


1: Trouver le but

- Idée: aller dans la direction où l'on se rapproche du but.
- 👉 Heuristique
  - 👉 Comme si on utilisait une boussole
- Sinon on explore jusqu'à ce qu'on trouve

2: Ne pas rester bloqué

- En allant directement vers le but, l'agent est bloqué
- Il peut être coincé dans un minimum local



3. Optimiser le chemin à parcourir

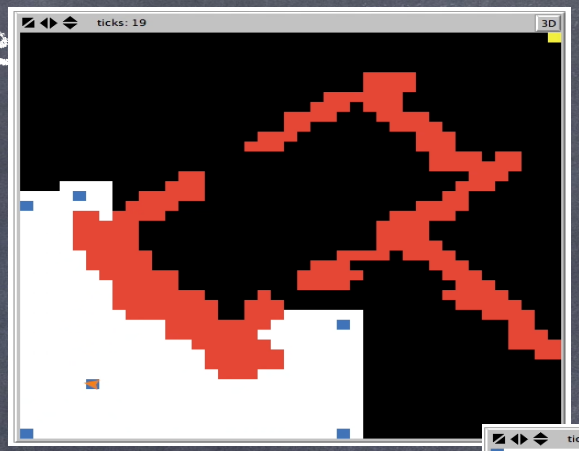
- Eviter de refaire tous les détours inutiles
  - Evidemment par rapport aux lieux que l'on a visités

Trois possibilités

- Explorer l'espace et essayer de trouver le «meilleur» chemin à partir de ce que l'on a exploré
  - Algorithme de Dijkstra
- Essayer d'aller vers le but en utilisant une heuristique (se rapprocher du but)
  - Algorithme de «meilleur d'abord» (best-first)
- Essayer de combiner ce qu'il y de mieux dans ces deux approches
  - Algorithme A*

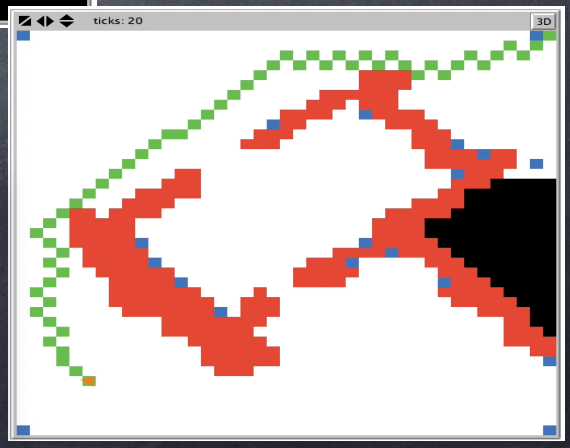
Algorithme de Dijkstra

- On ne sait pas où se trouve le but (on sait juste quand on y est)
- Il n'y a plus qu'à tout parcourir.... 😊
- Mais comme on a tout parcouru (ou presque) on est sûr d'obtenir le « meilleur » chemin pour arriver



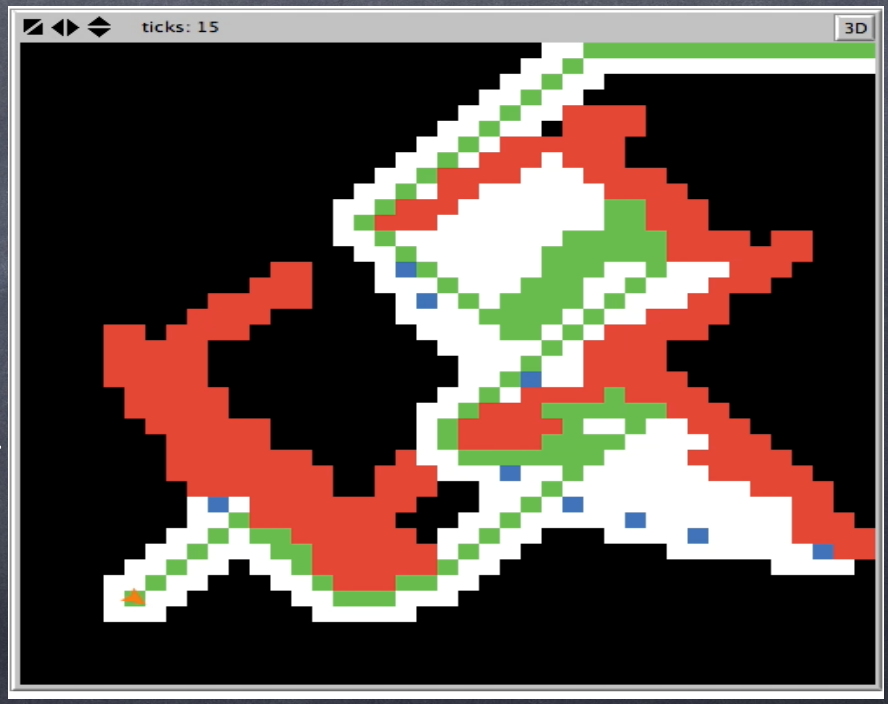
(a)

(b)



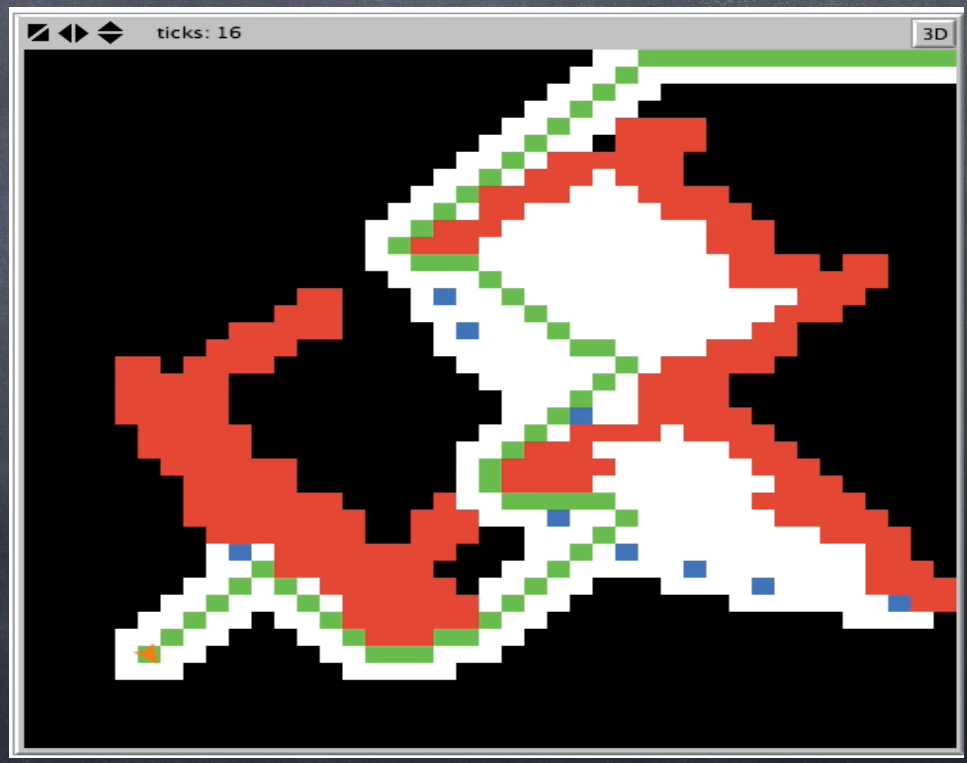
Le meilleur d'abord (best first)

- On utilise l'heuristique qui consiste à aller globalement dans la bonne direction (la direction qui diminue la distance vis à vis du but)
  - Risque de blocages
  - Dans ce cas, on explore autour
- Au retour, on redonne le chemin que l'on a parcouru
  - Pas du tout optimal !!! 😟



A*: prendre les bonnes idées de l'un et de l'autre

- On va vers le but en utilisant une heuristique, comme avec le meilleur d'abord
- On optimise le chemin parcouru en tenant compte de la distance à l'origine, comme avec Dijkstra
- « Bons » résultats
- Le plus utilisé dans les jeux et en I.A.!!! 😉



Les trois algorithmes

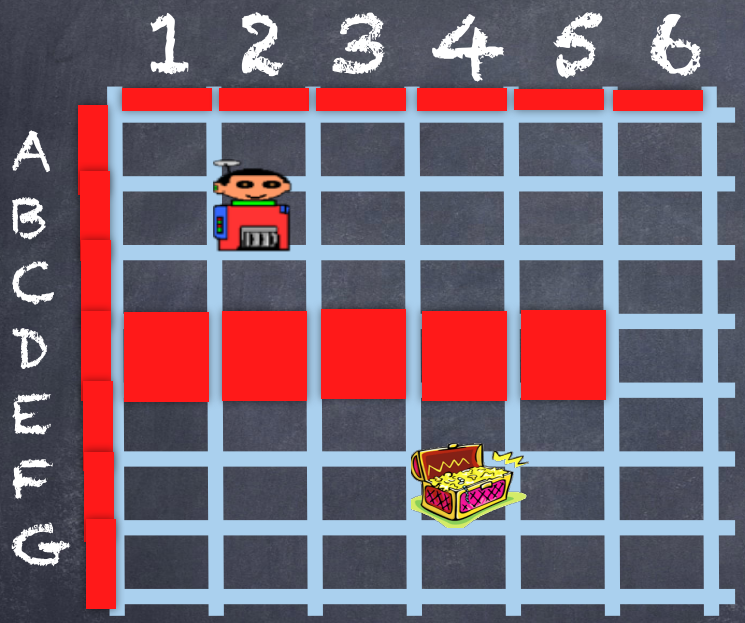
- C'est le même « moteur » qui permet de faire fonctionner ces différents algorithmes
- Ne diffèrent que par quelques points

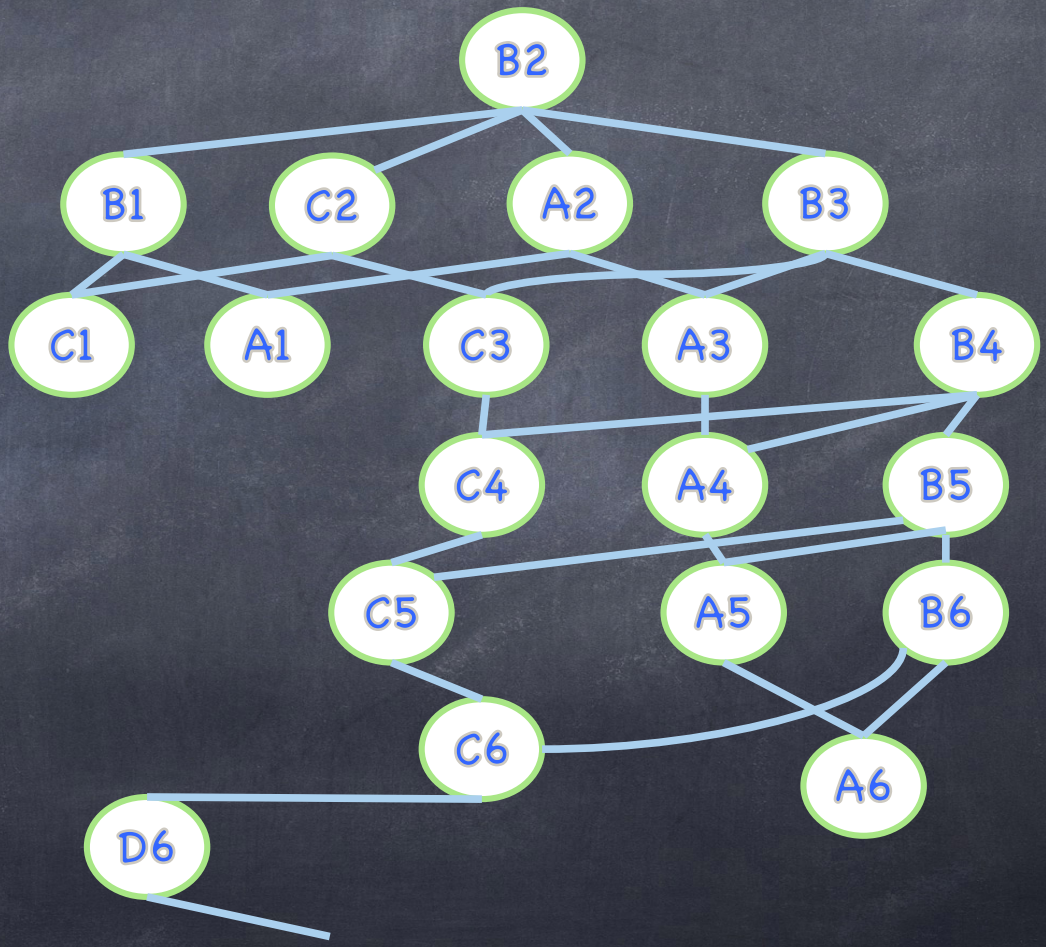
	Best-first	Dijkstra	A*
Choix du nœud à considérer	Plus petit h (heuristique distance au but)	Plus petit g(n) (cout du chemin parcouru)	Plus petit h (heuristique de distance au but)
Tri de la liste	h(n)	g(n)	f(n)=g(n)+h(n)
Calcul du « parent »	Nœud courant	Parent du nœud n identique au courant si g(n)<g(courant)	Parent du nœud n identique au courant si f(n)<f(courant)
Remarque	Calcul rapide, mais chemin tortueux	Calcul lent mais possibilité d'avoir le « meilleur » chemin	Calcul rapide et obtention d'un « bon » chemin (parfois le meilleur)

Le moteur de recherche

- Principe: on parcours une suite de nœuds dans un graphe
- Current = nœud courant que l'on analyse
- In-nodes = liste des nœuds à analyser, ordonnée suivant un critère (fonction heuristique ou coût de déplacement)
- Out-nodes = liste des nœuds analysés (pour éviter de boucler)

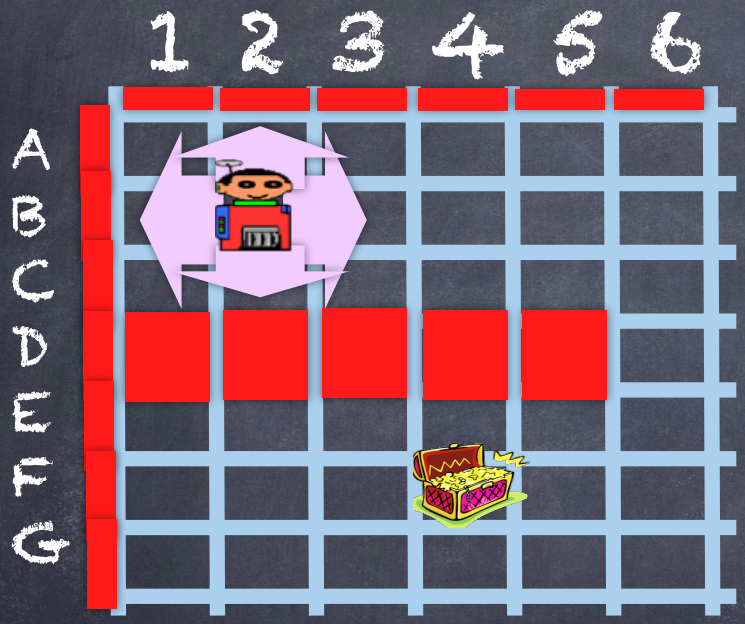
Le graphe





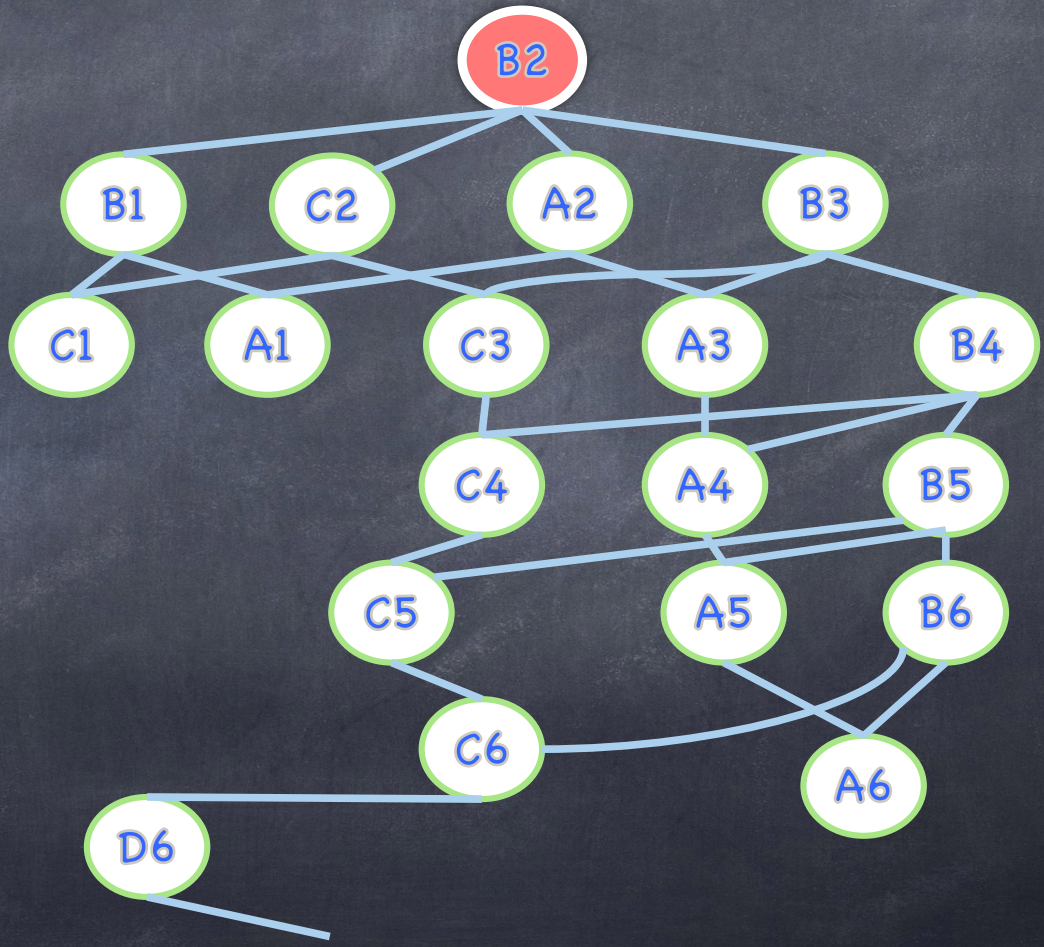
- Origine: Case B2
- But: case F4
- Obstacles en rouge

Le fonctionnement #1

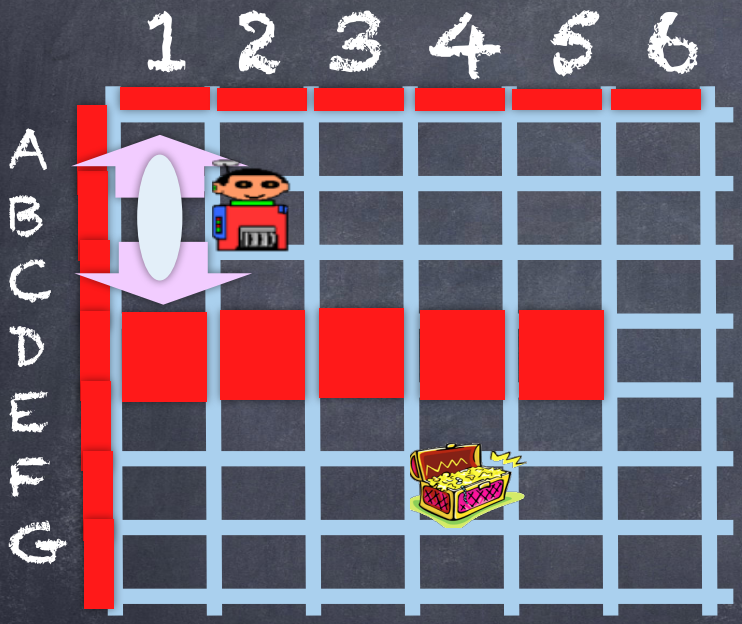


in: [B2]

out: [ ]

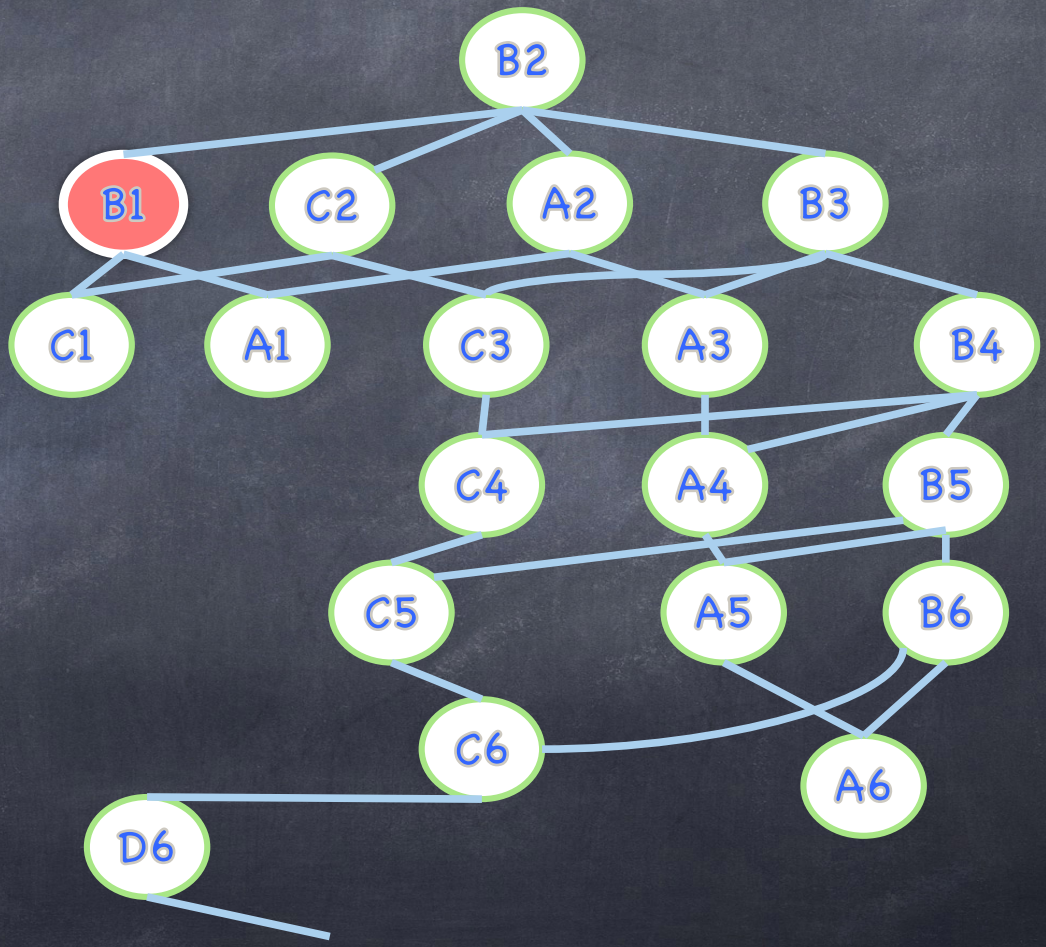


Le fonctionnement #2

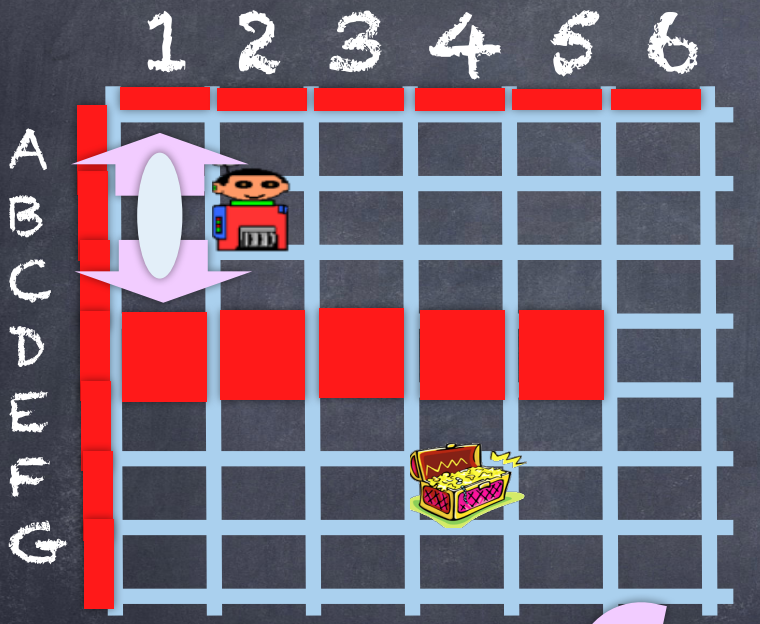


in: [B1 C2 A2 B3]

out: [B2]



Le fonctionnement #3

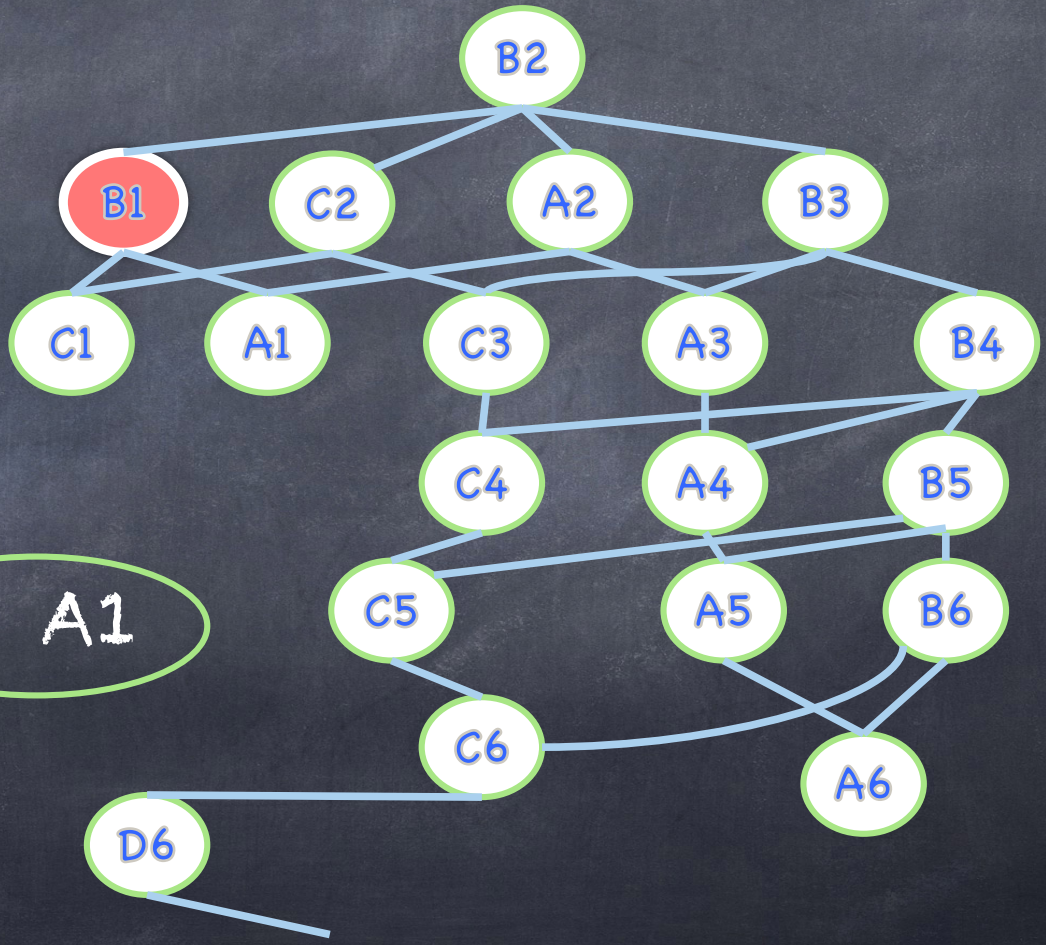


in: [C2 A2 B3]

out: [B2 B1]

?

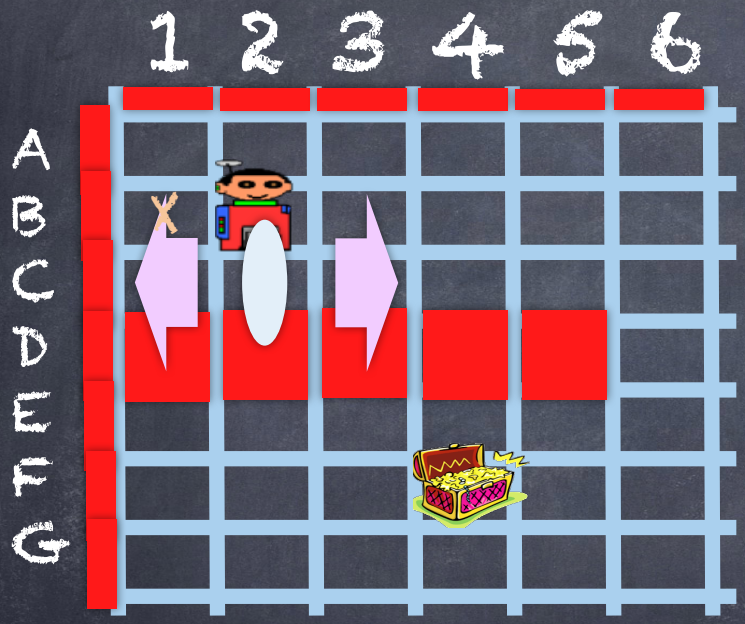
C1 A1



Dijkstra

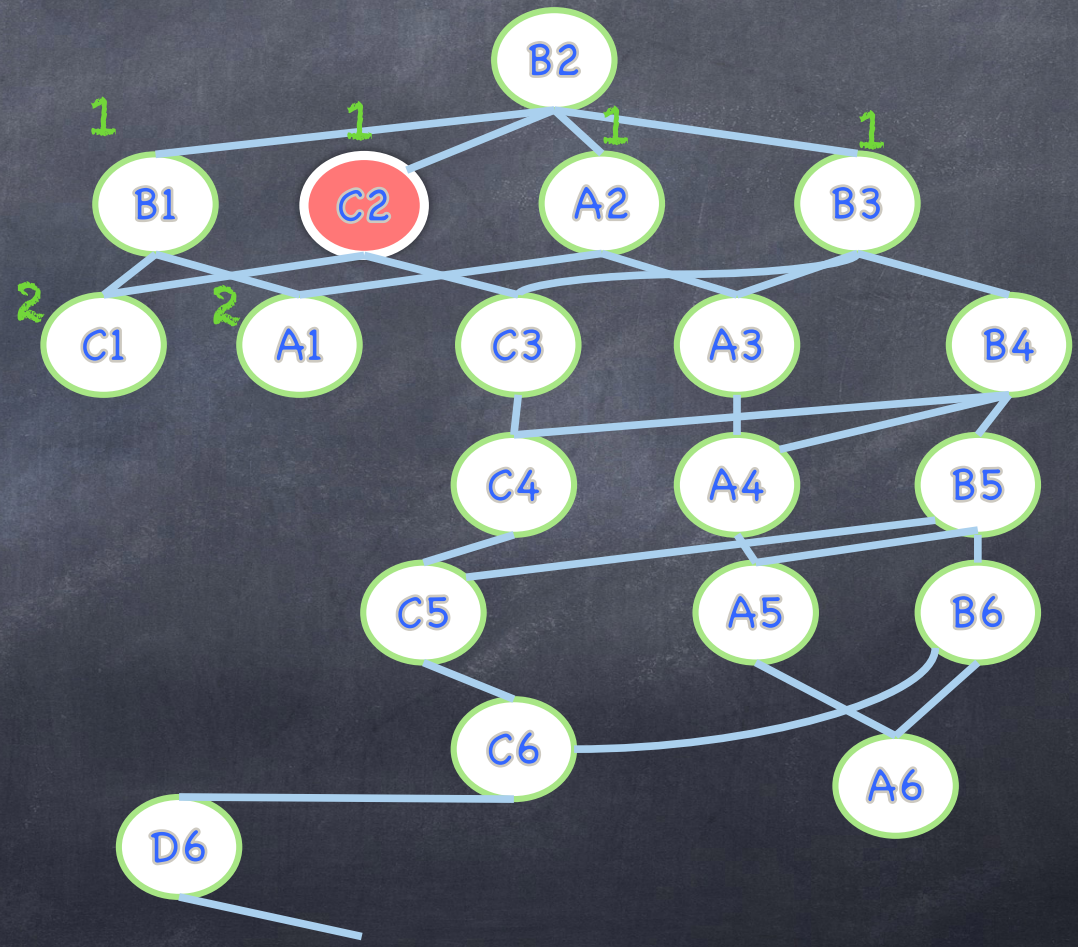
- On ordonne les nœuds par rapport à la distance à l'origine
- G(n) = distance(origine, n)

Le fonctionnement #4

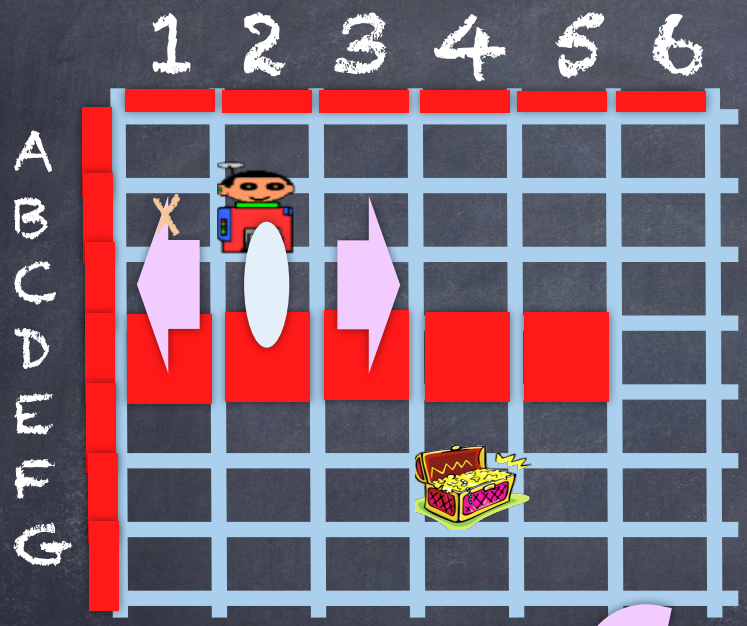


in: [A2 B3 C1 A1]

out: [C2 B2 B1]



Le fonctionnement #5

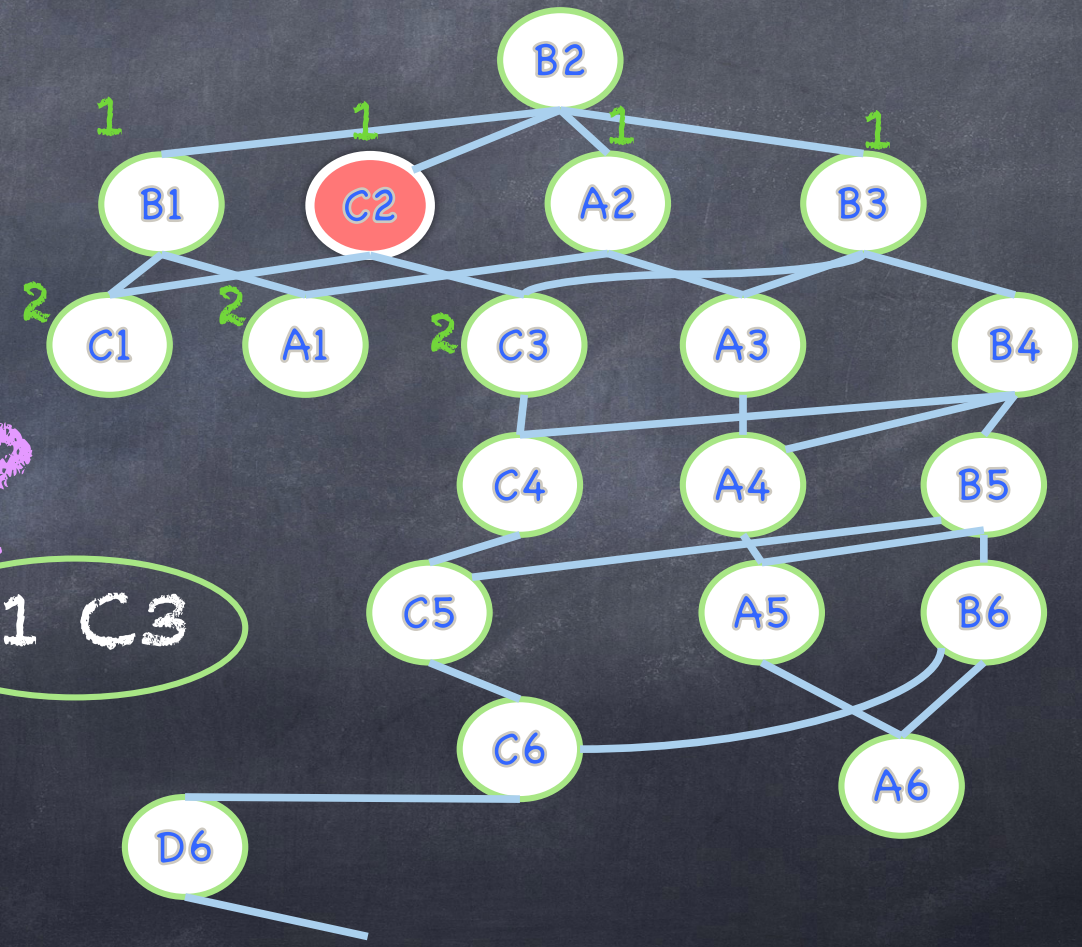


in: [A2 B3 C1 A1]

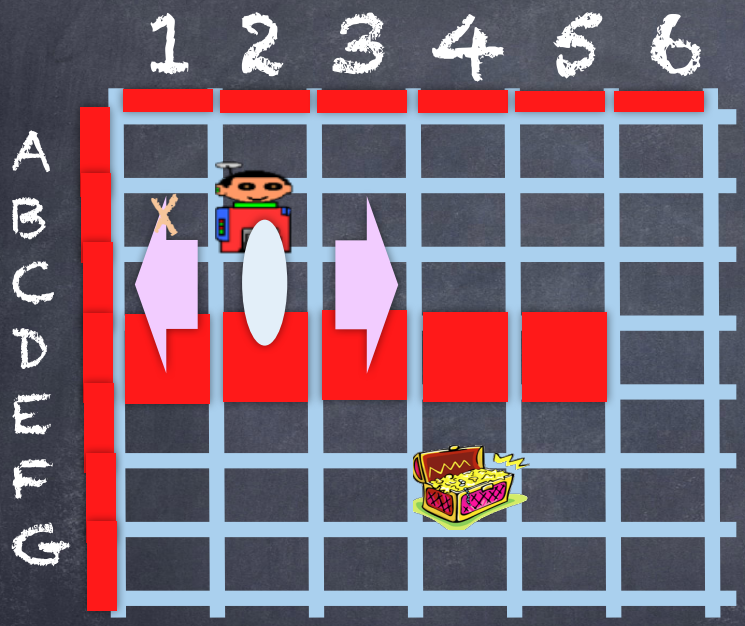
out: [C2 B2 B1]

?

C1 C3

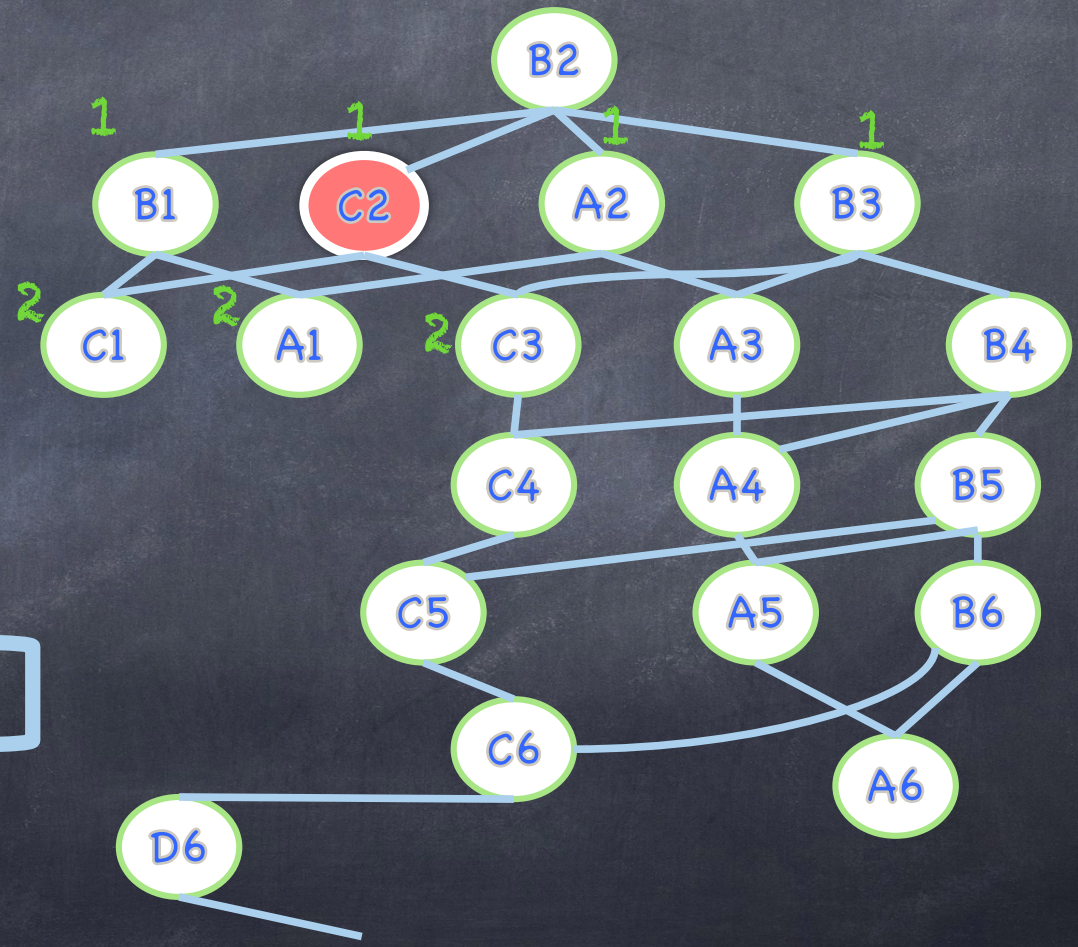


Le fonctionnement #6

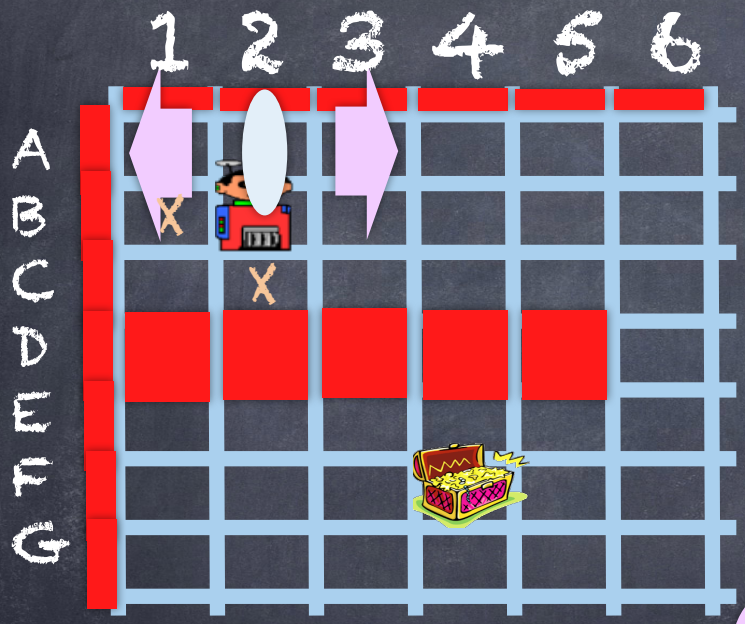


in: [A2 B3 C1 A1 C3]

out: [C2 B2 B1]



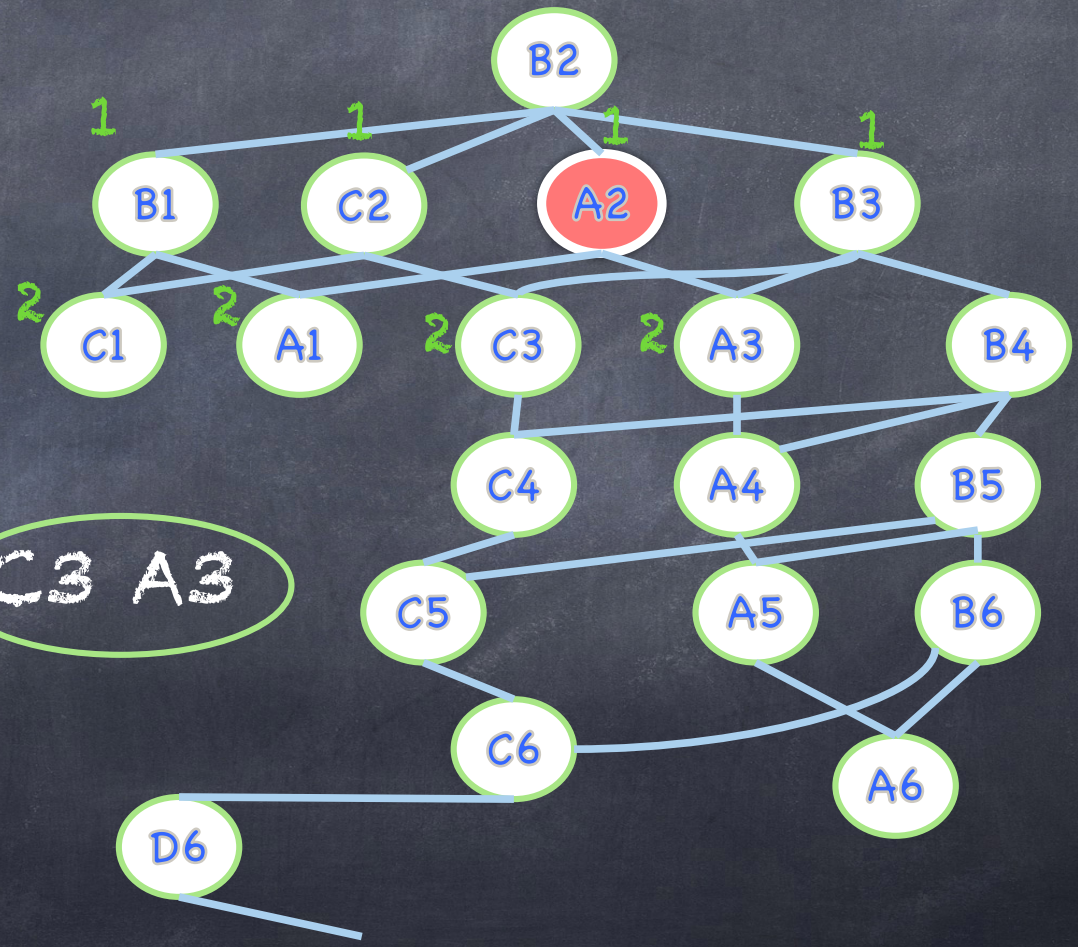
Le fonctionnement #7



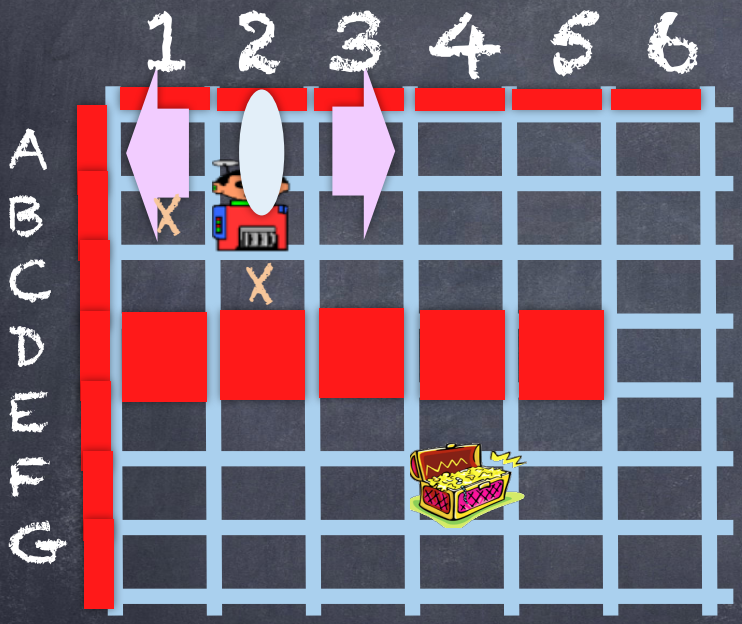
C3 A3

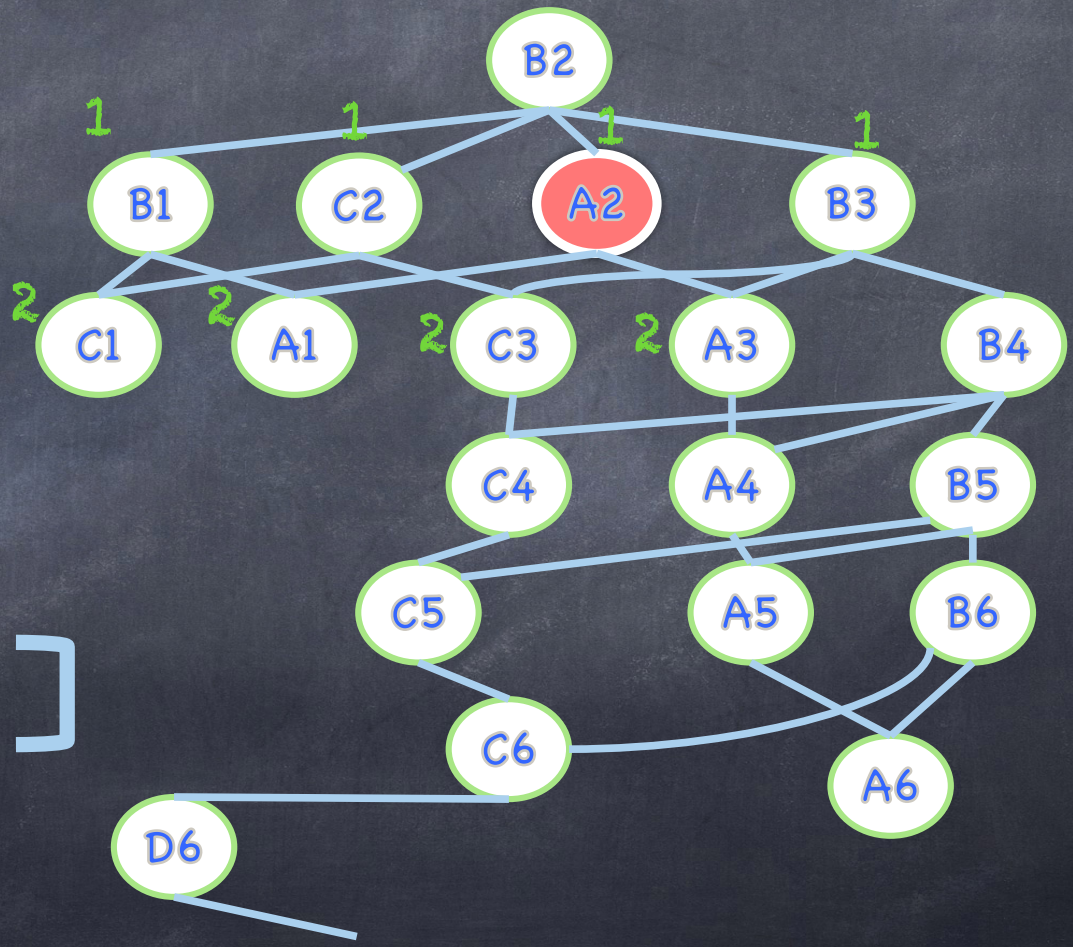
in: [B3 C1 A1 C3]

out: [A2 C2 B2 B1]



Le fonctionnement #8

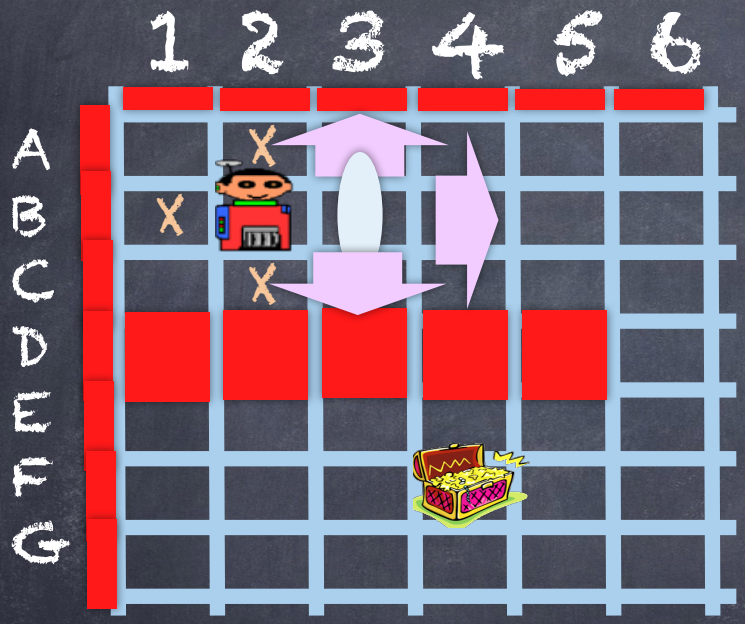


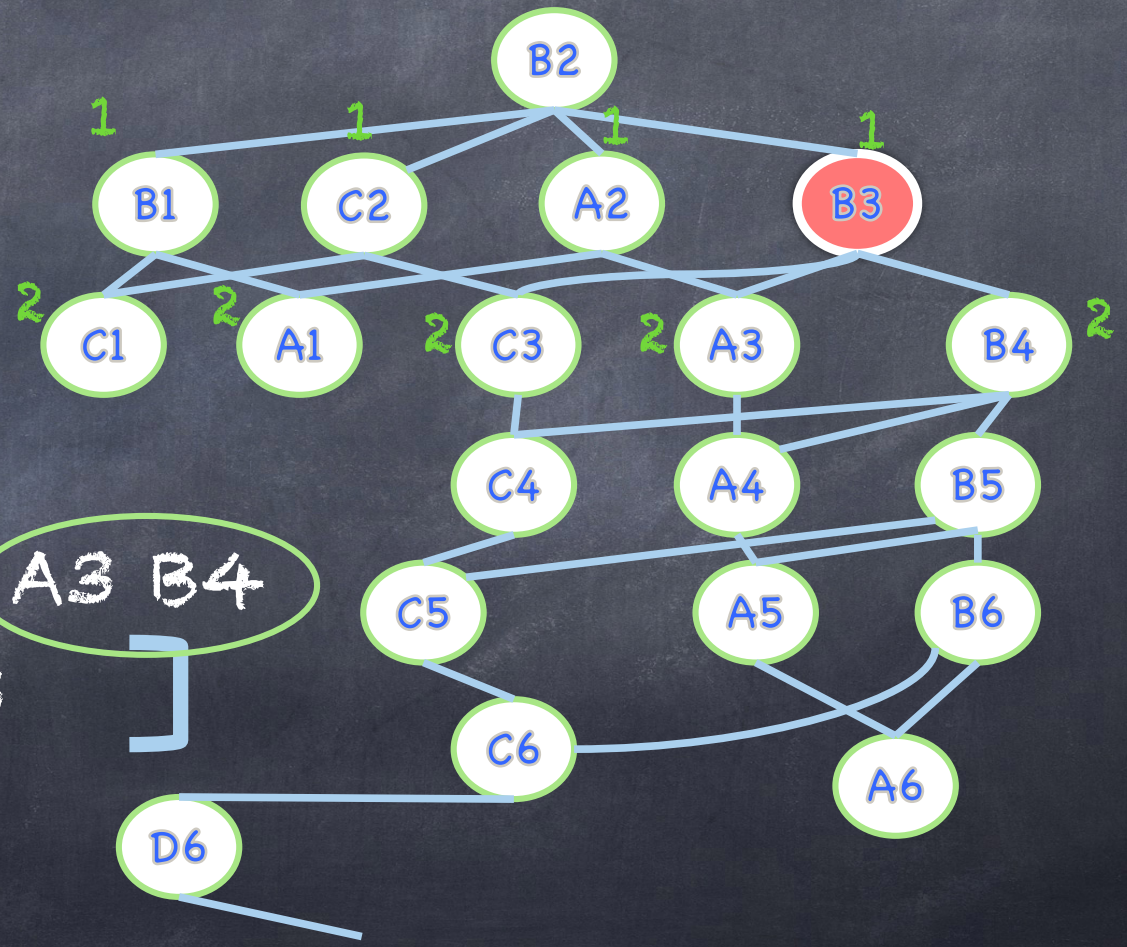


in: [B3 C1 A1 C3 A3]

out: [A2 C2 B2 B1]

Le fonctionnement #9



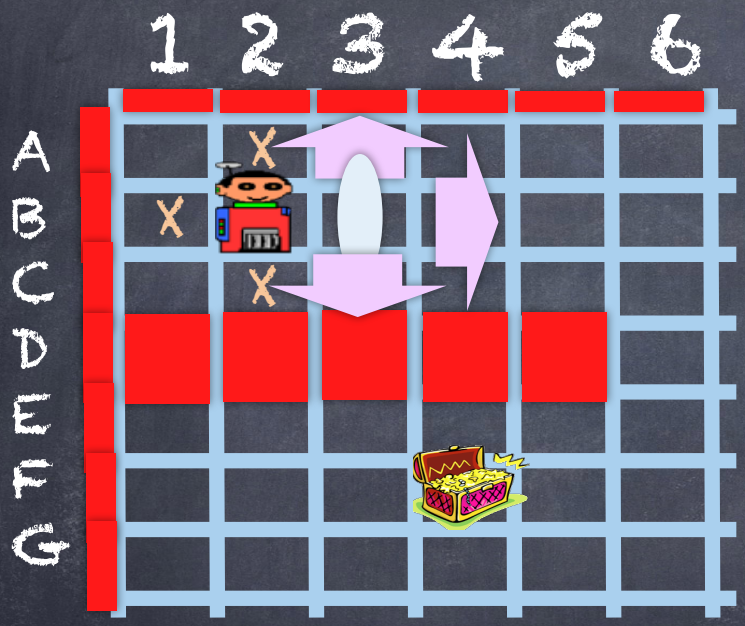


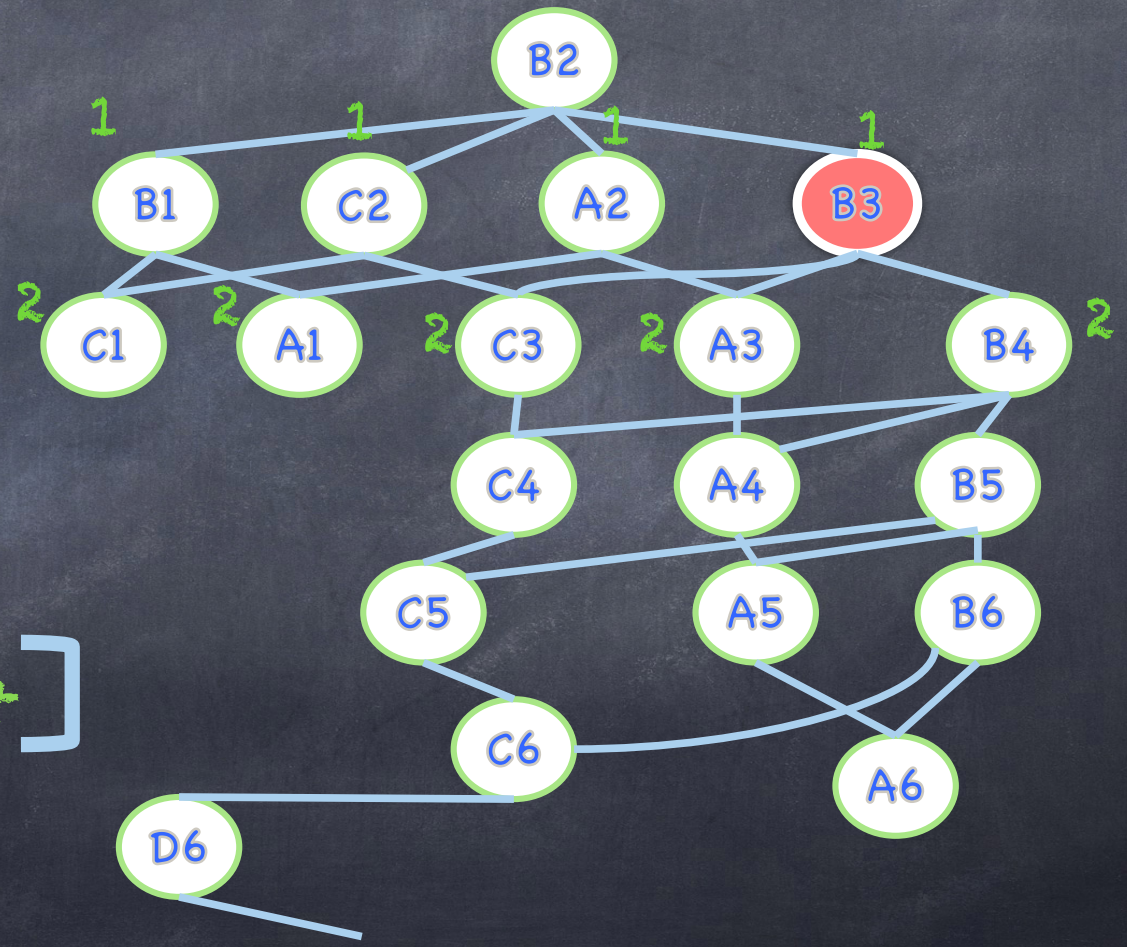
A3 B4

in: [B3 C1 A1 C3 A3]

out: [A2 C2 B2 B1]

Le fonctionnement #10

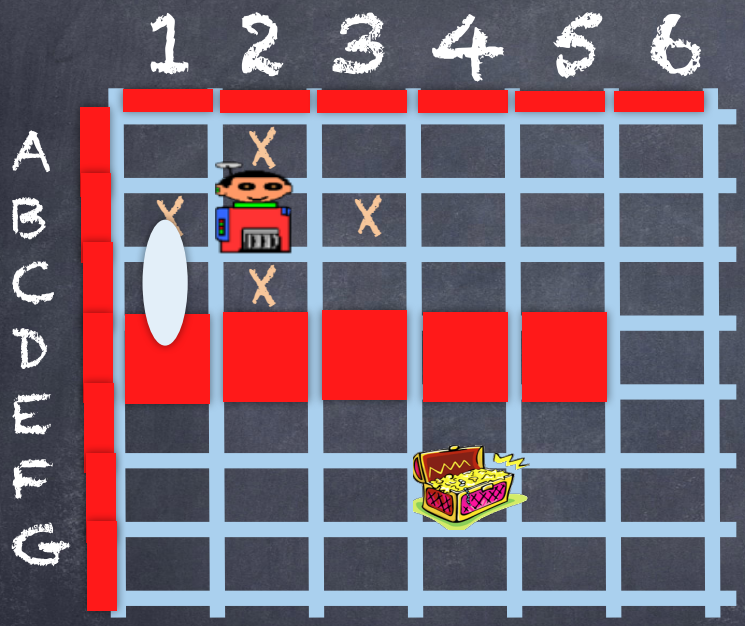


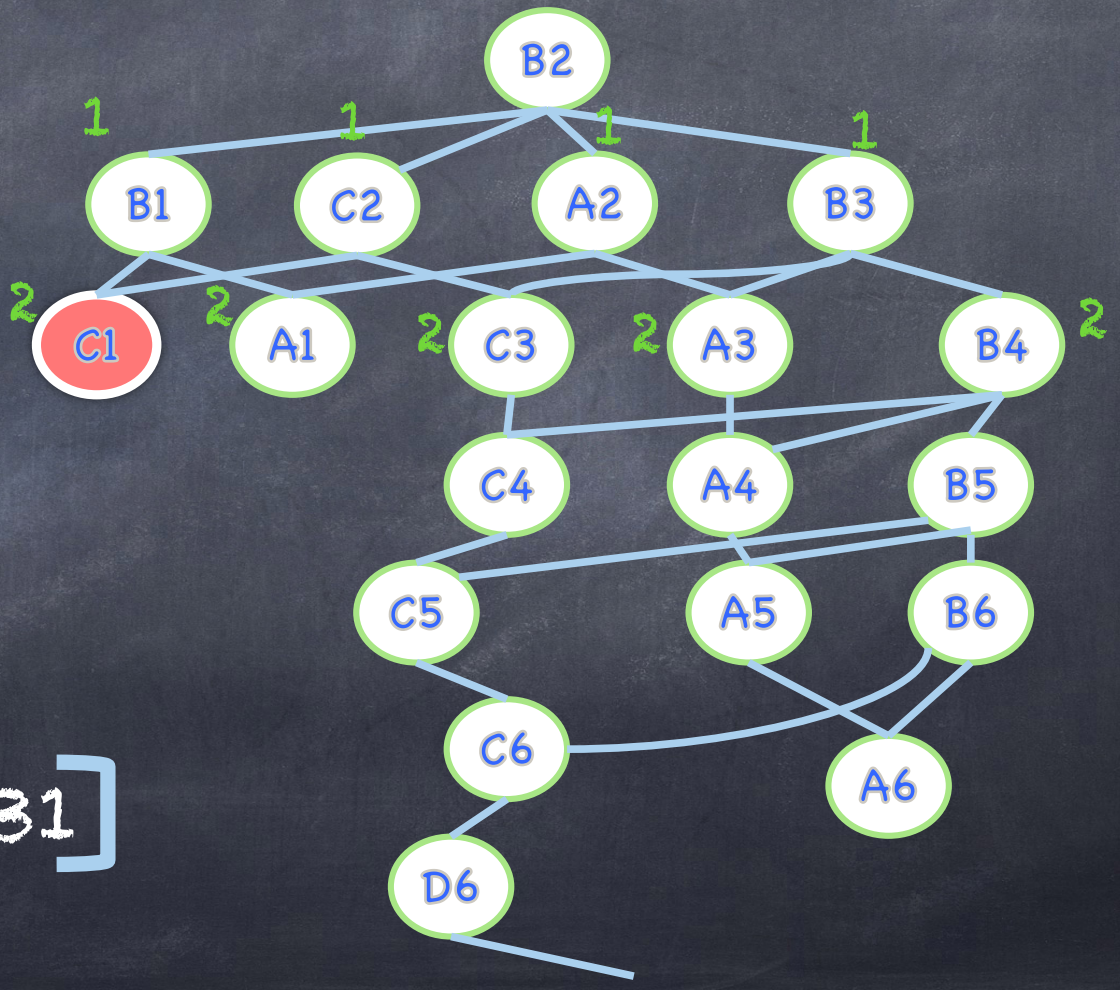


in: [C1 A1 C3 A3 B4]

out: [B3 A2 C2 B2 B1]

Le fonctionnement #11

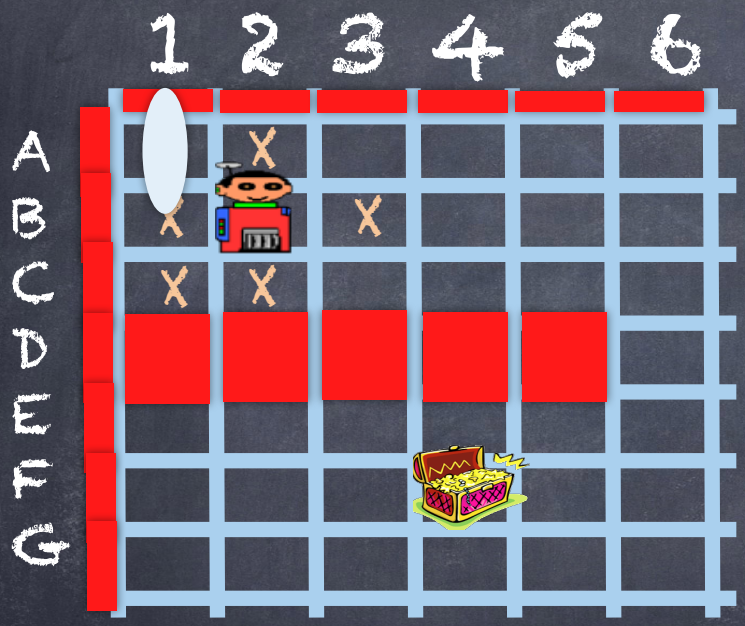


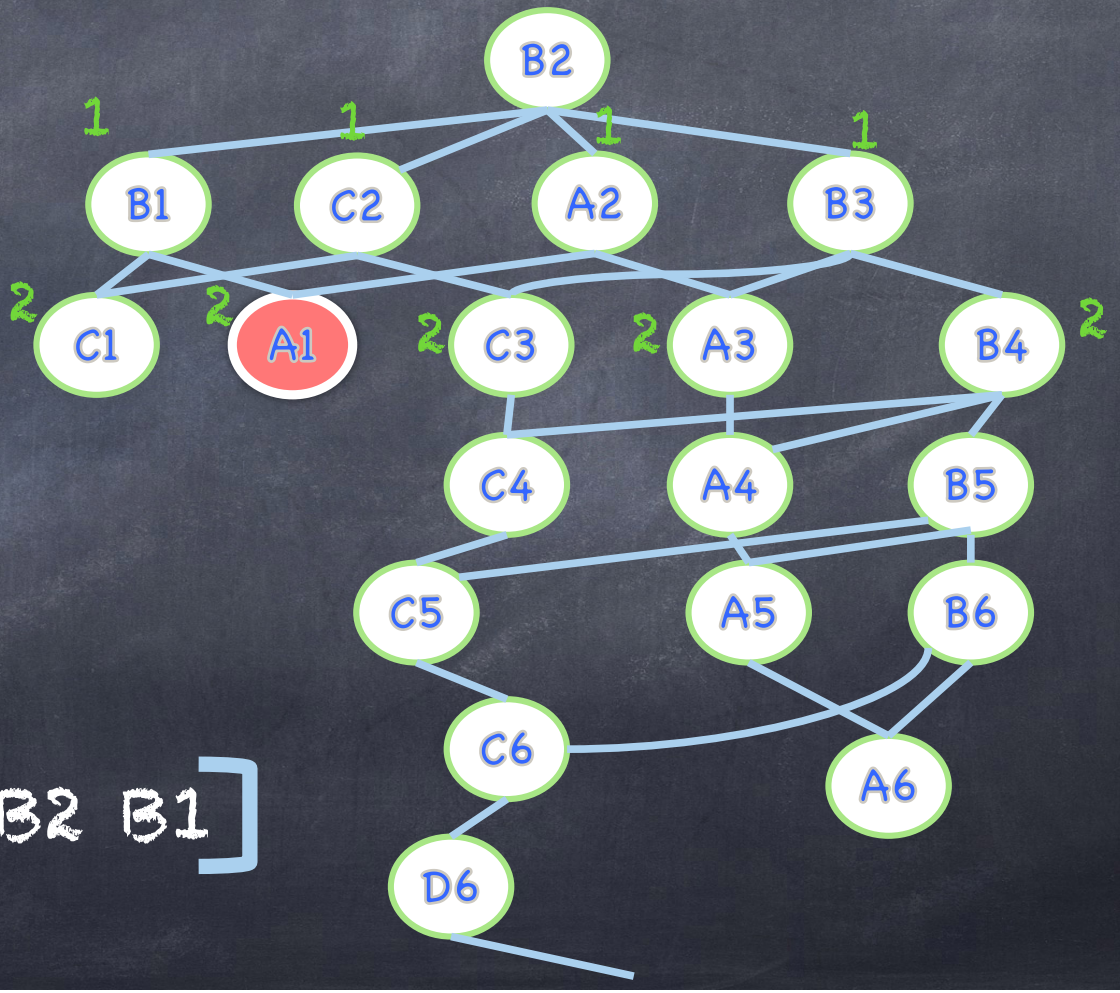


in: [A1 C3 A3 B4]

out: [C1 B3 A2 C2 B2 B1]

Le fonctionnement #12

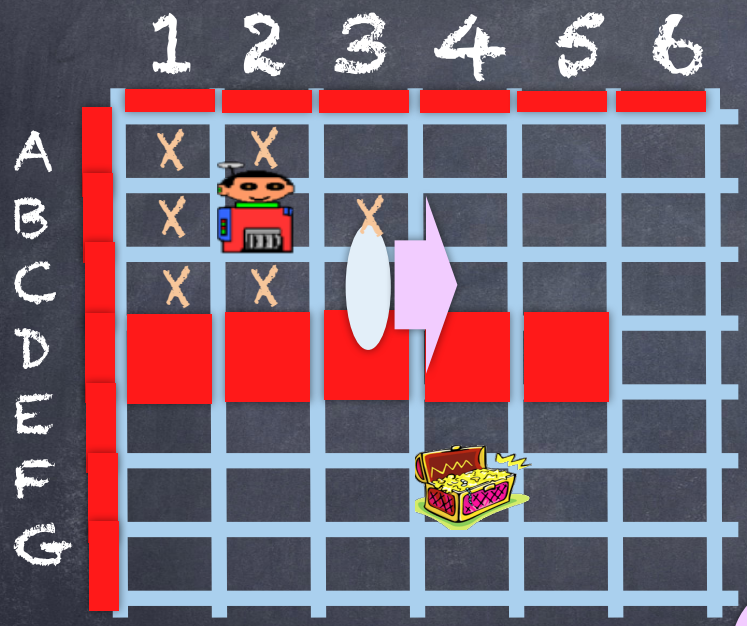


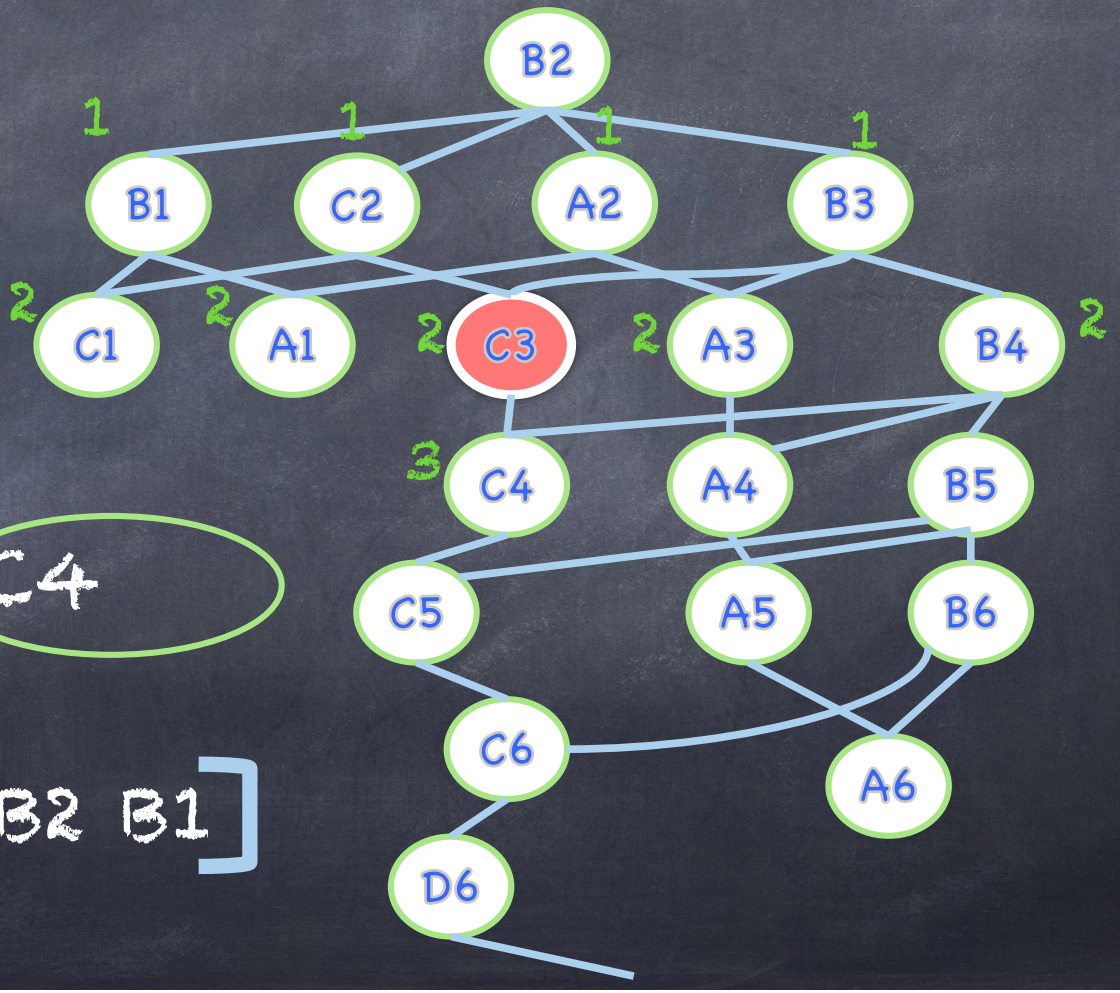


in: [C3 A3 B4]

out: [A1 C1 B3 A2 C2 B2 B1]

Le fonctionnement #13



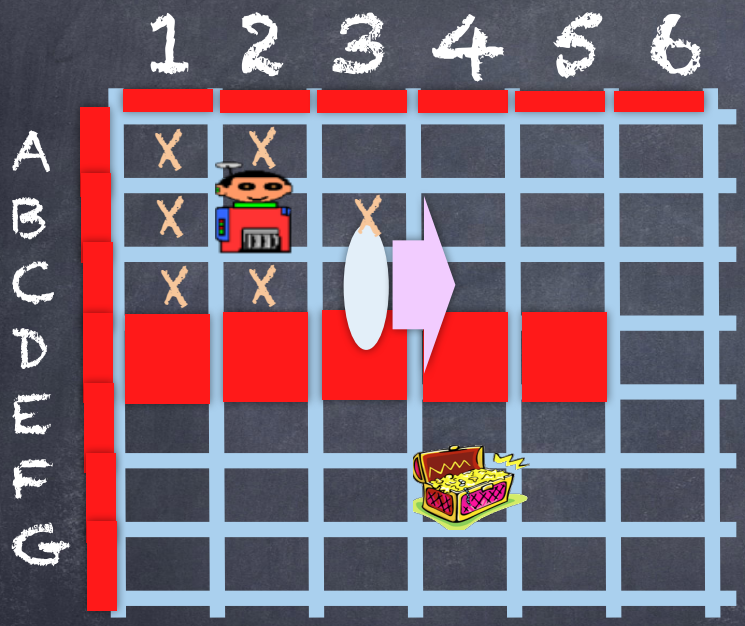


C4

in: [C3 A3 B4]

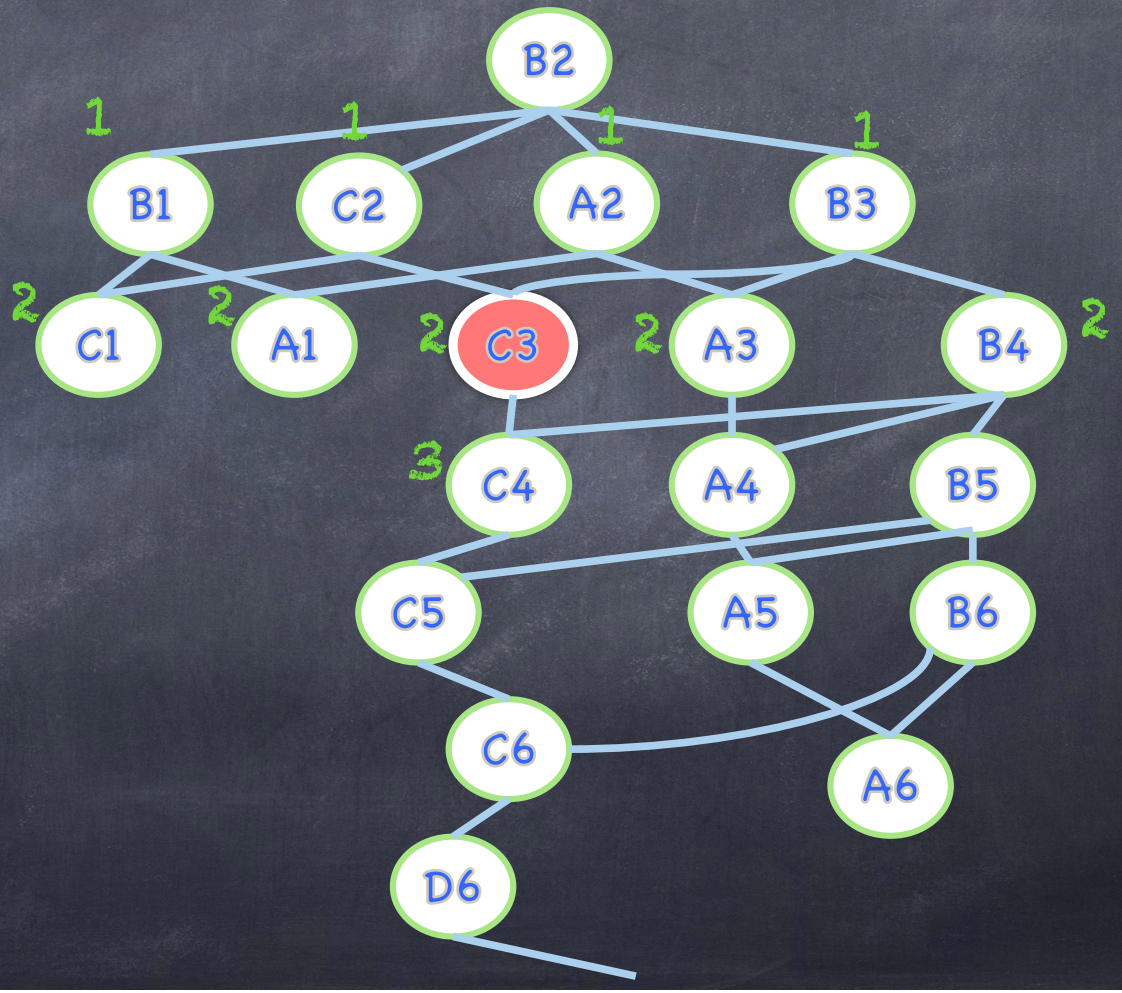
out: [A1 C1 B3 A2 C2 B2 B1]

Le fonctionnement #14

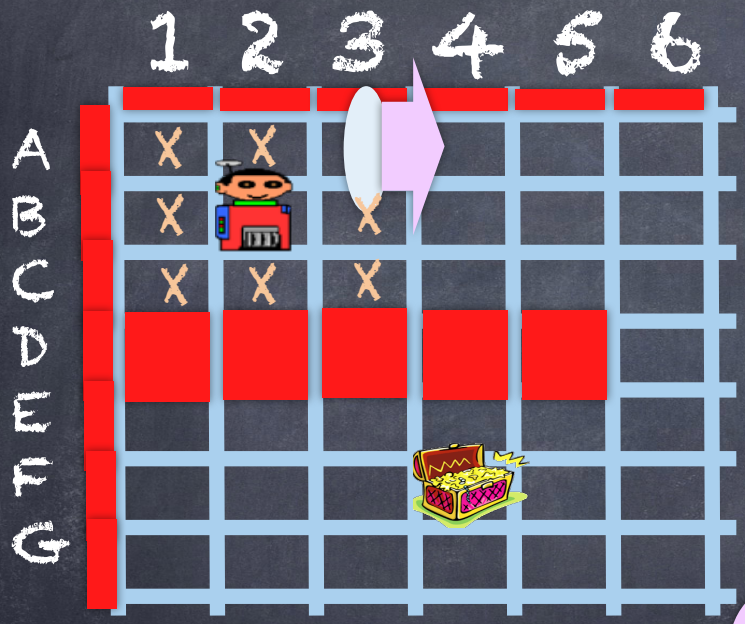


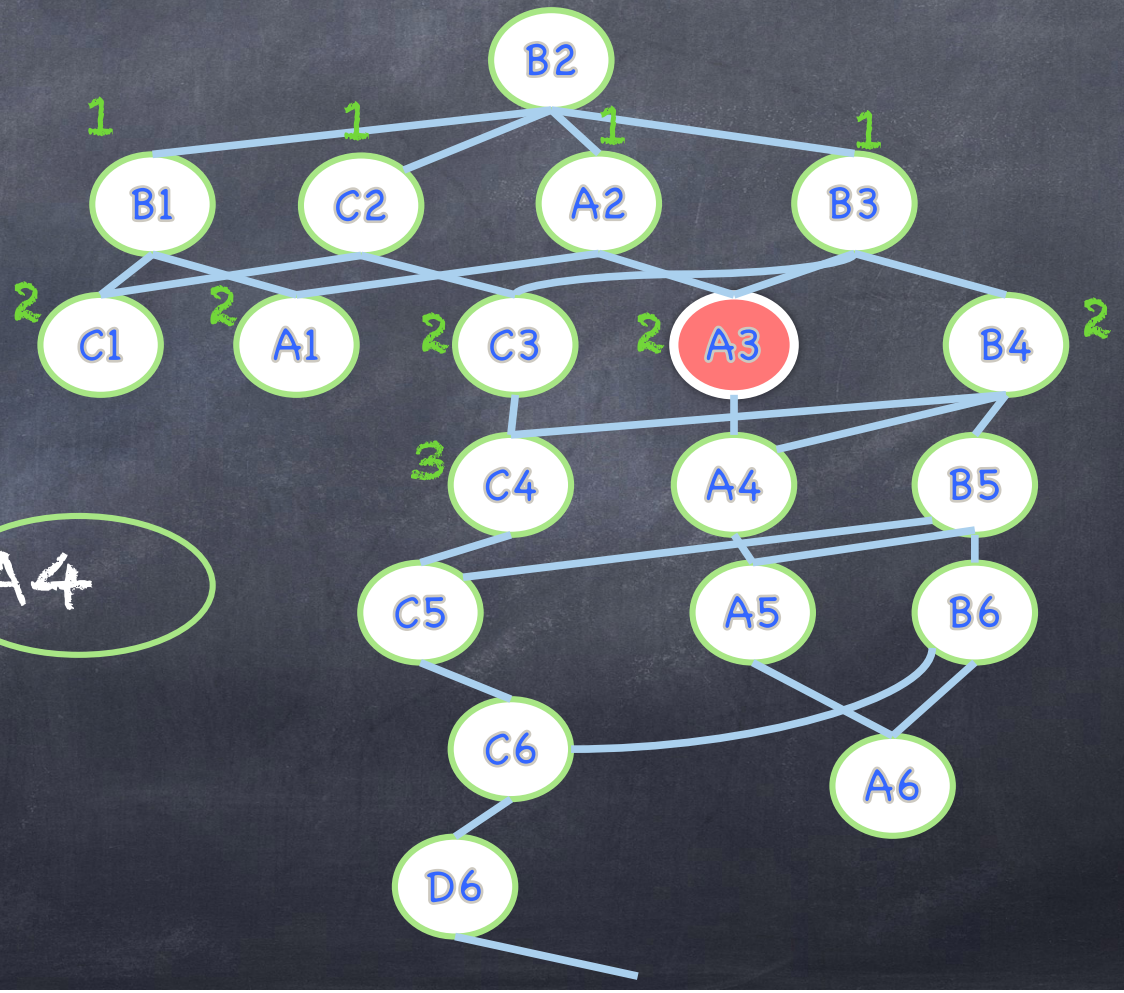
in: [A3 B4 C4]

out: [A1 C1 B3 A2
C2 B2 B1]



Le fonctionnement #15



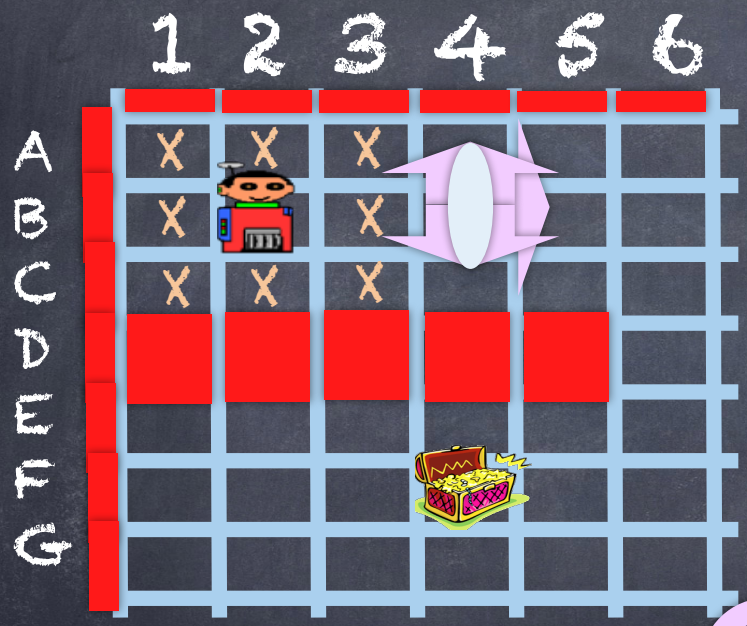


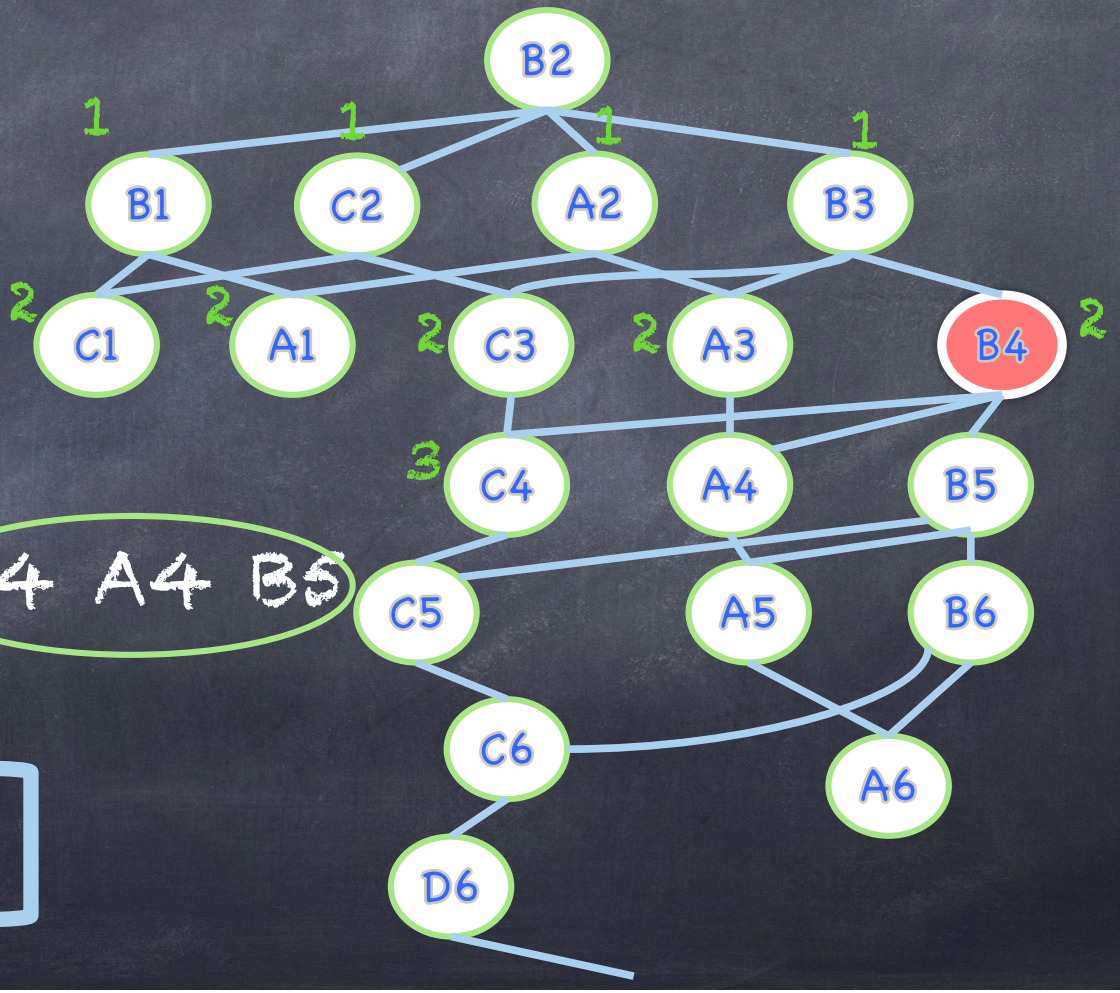
A4

in: [B4 C4]

out: [A3 A1 C1 B3
A2 C2 B2 B1]

Le fonctionnement #15



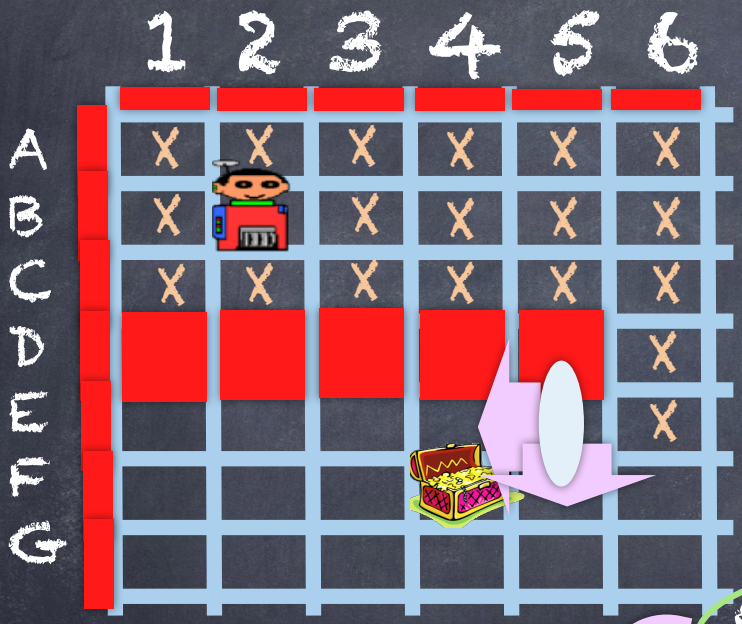


C4 A4 B5

in: [C4 A4]

out: [B4 A3 A1 C1 B3 A2 C2 B2 B1]

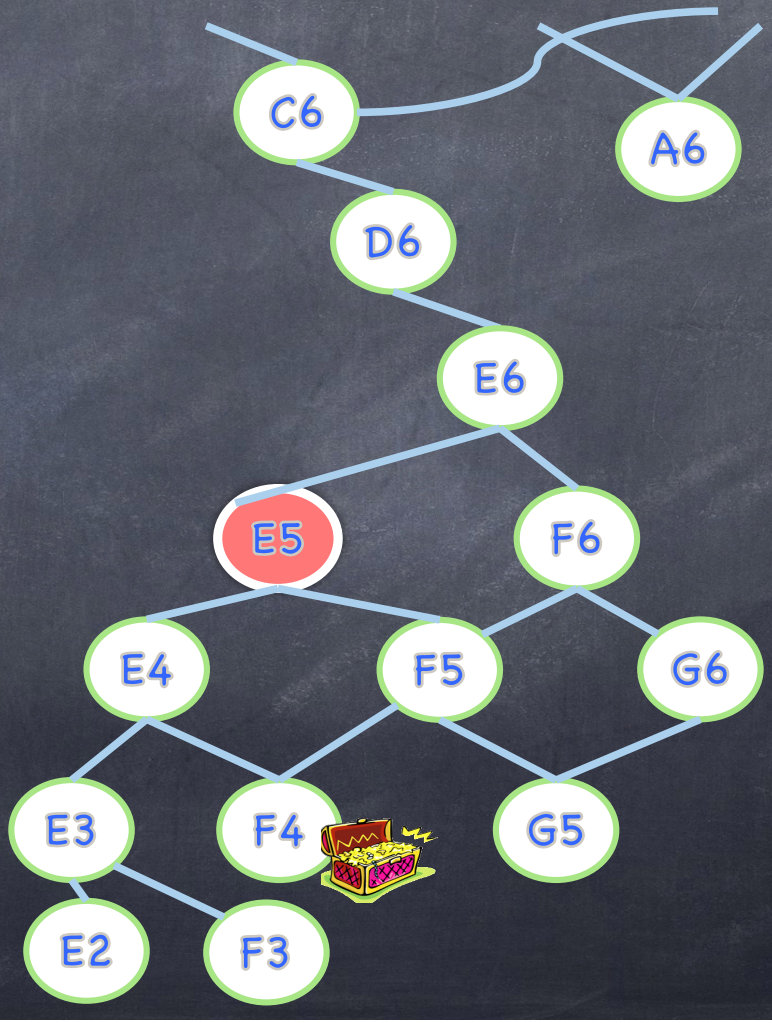
Le fonctionnement #fin1



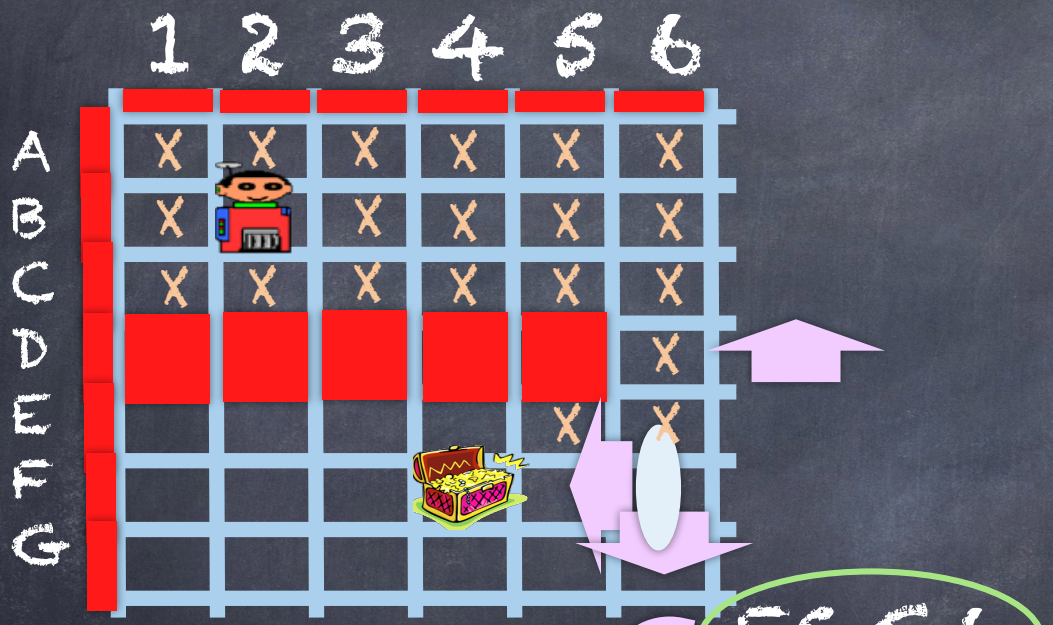
E4 F5

in: [F6]

out: [E5 E6 D6 ....]



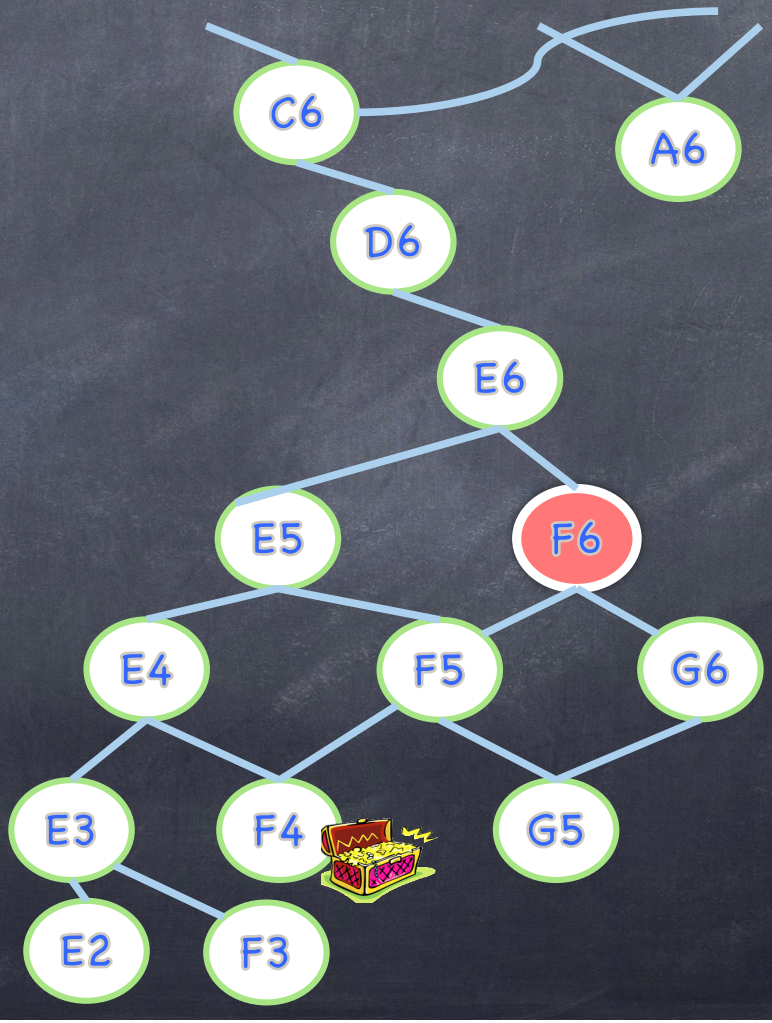
Le fonctionnement #fin2



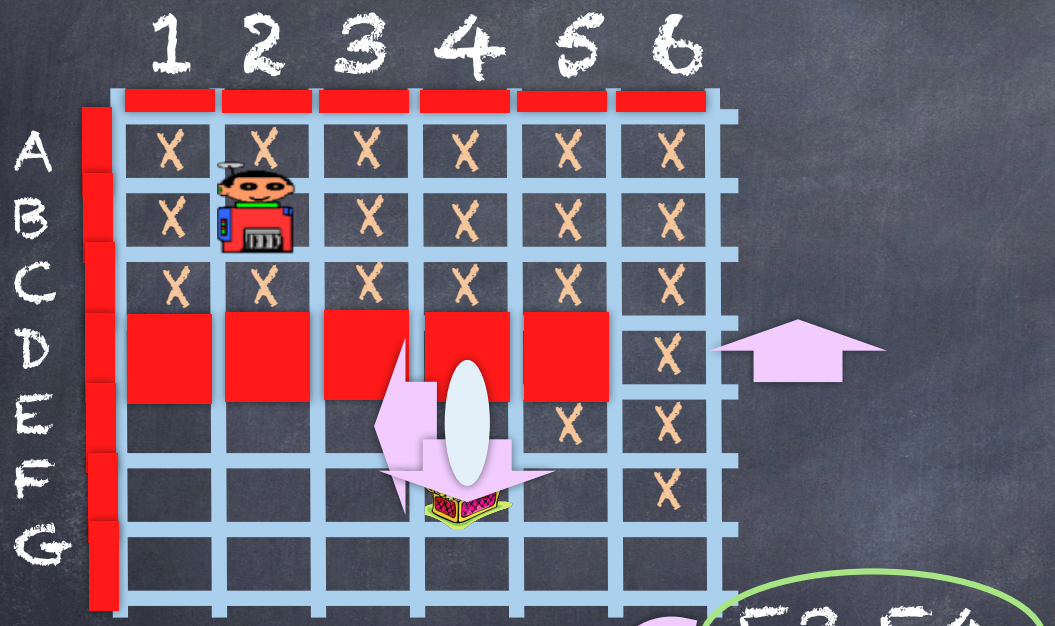
F5 G6

in: [E4 F5]

out: [F6 E5 E6 D6 ....]



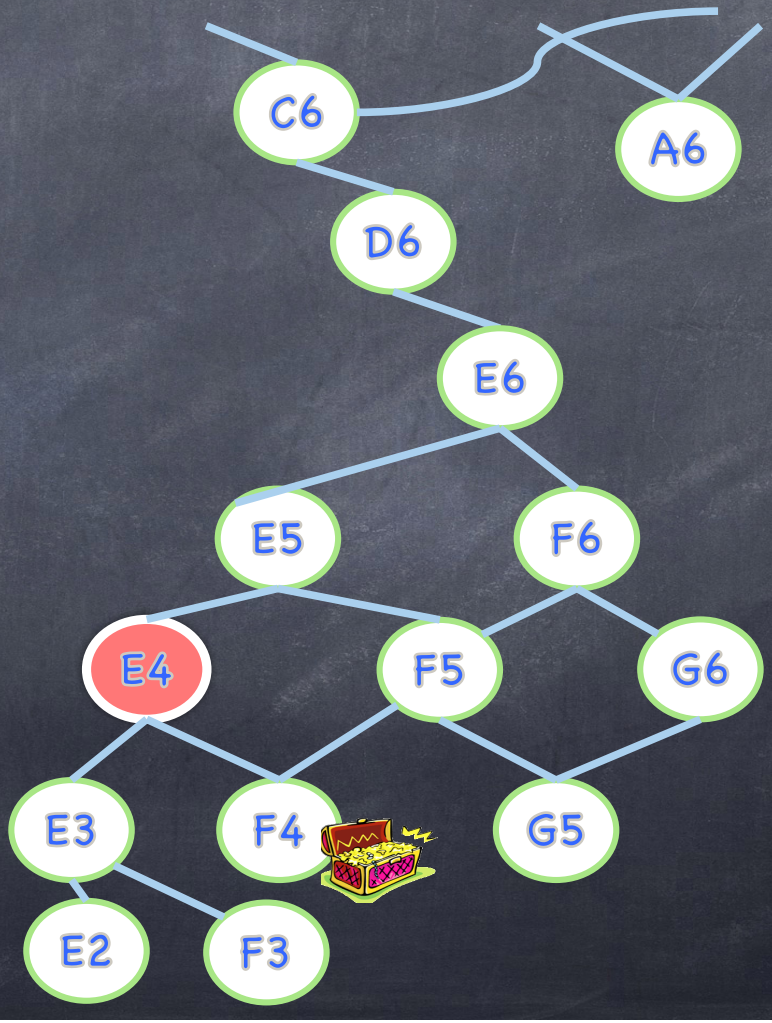
Le fonctionnement #fin3



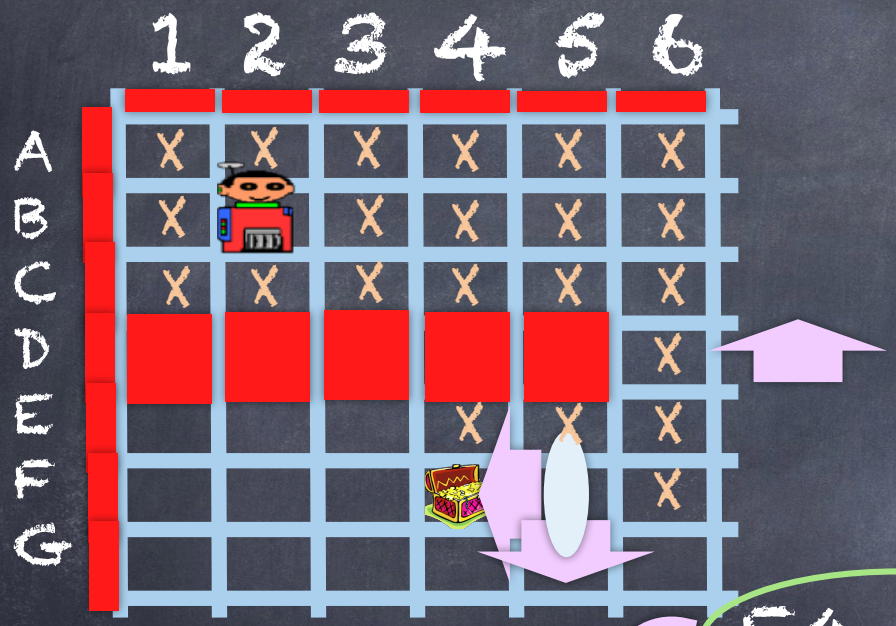
E3 F4

in: [F5 G6]

out: [E4 F6 E5 E6 D6 ....]



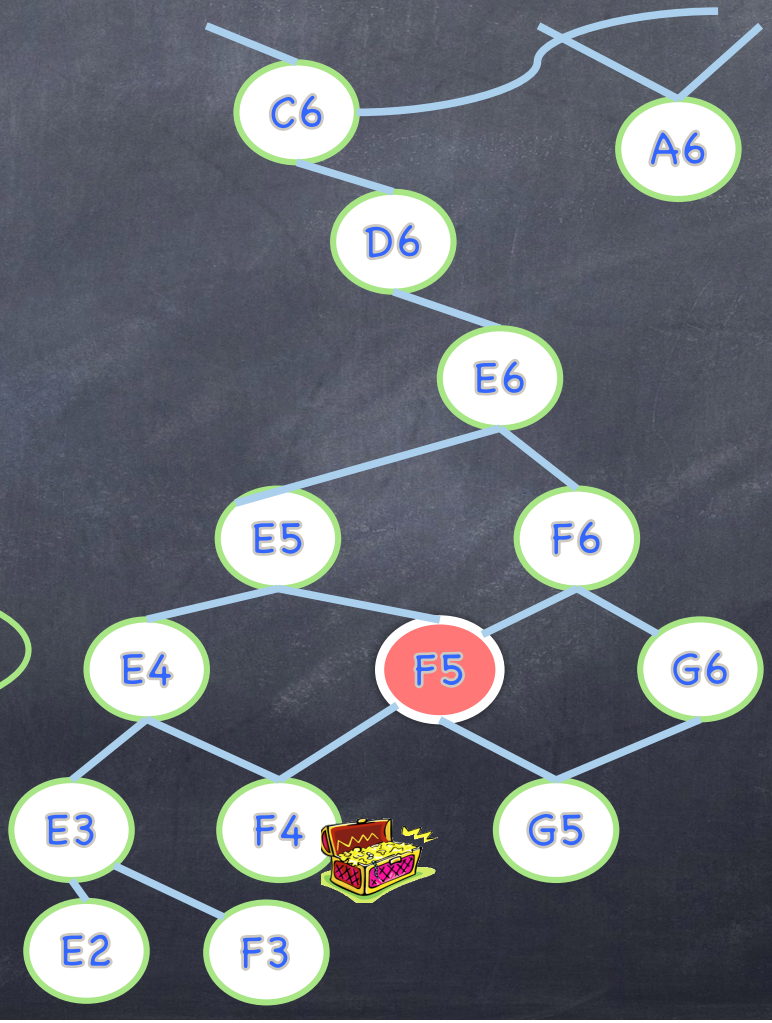
Le fonctionnement #fin4



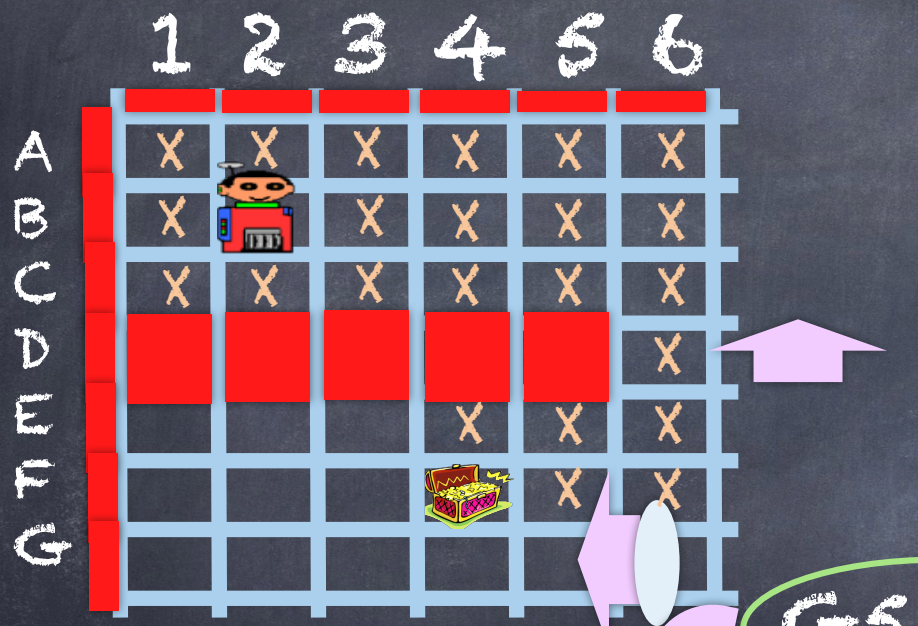
F4 G5

in: [G6 E3 F4]

out: [F5 E4 F6 E5 E6 ....]



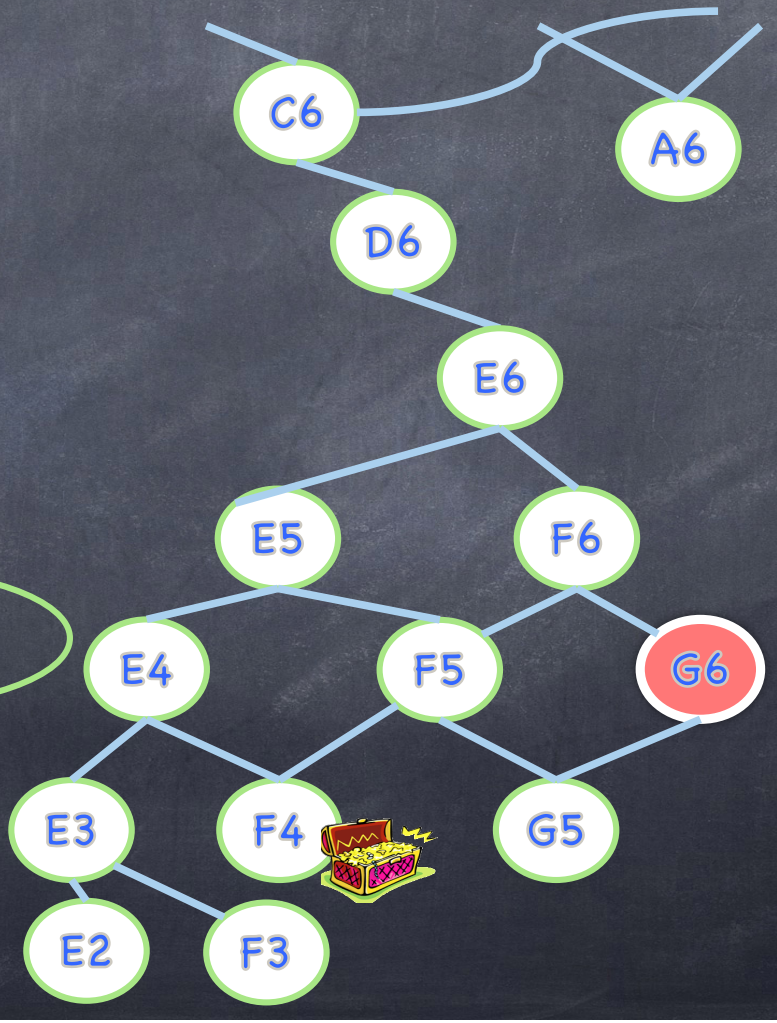
Le fonctionnement #fins



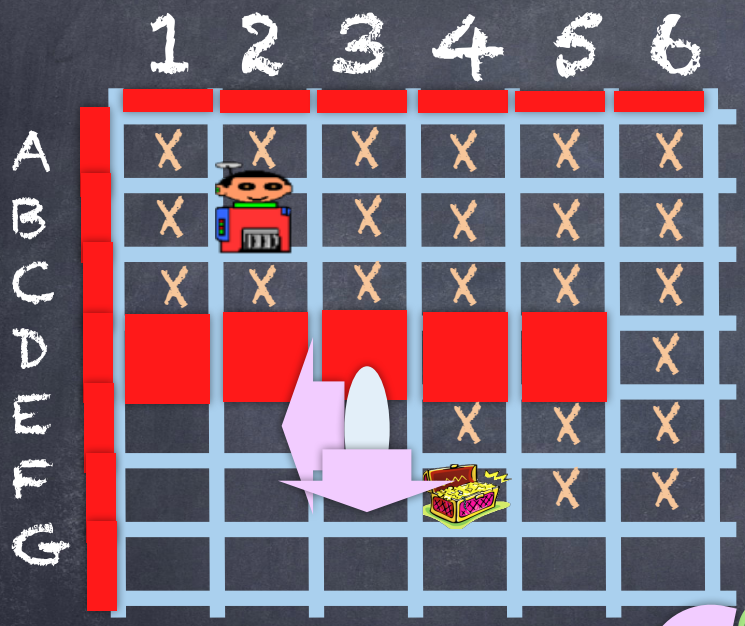
G5

in: [E3 F4 G5]

out: [G6 F5 E4 F6 E5 ...]



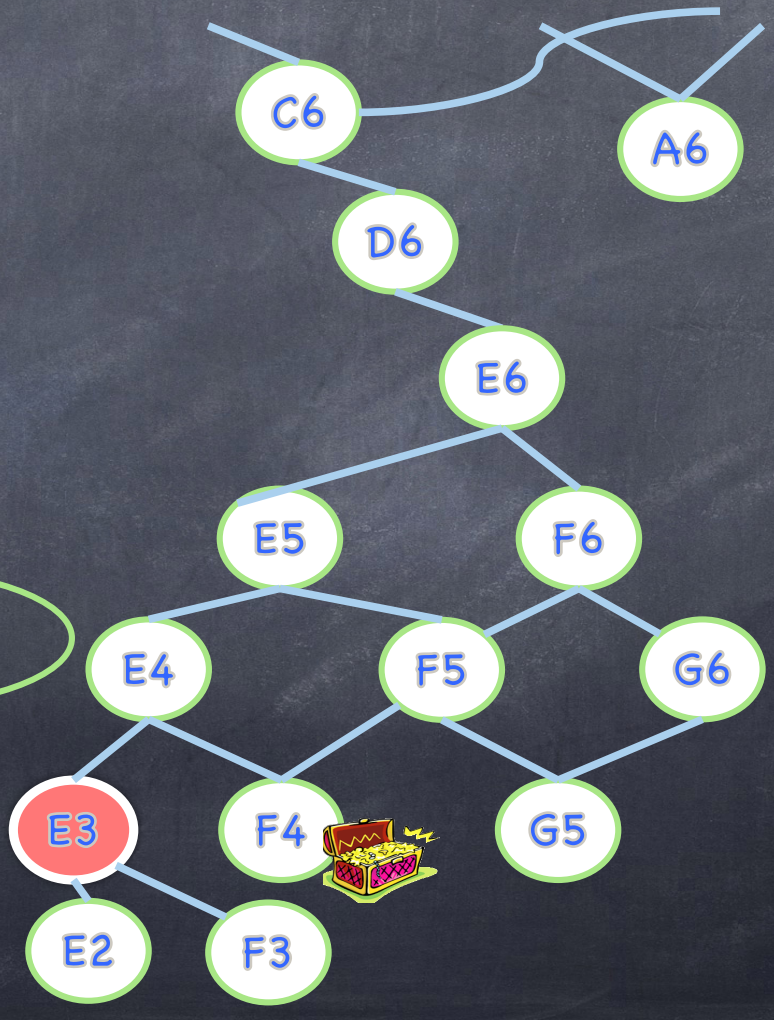
Le fonctionnement #fin6



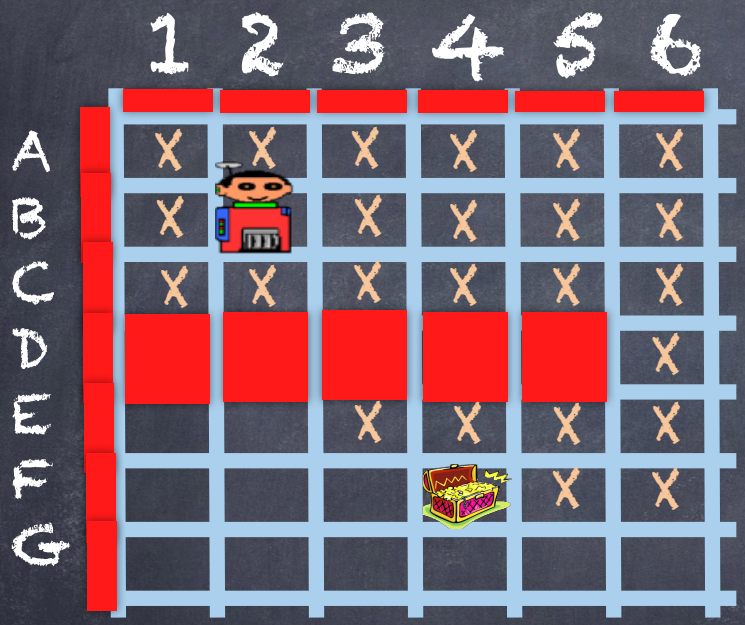
E2 F3

in: [F4 G5]

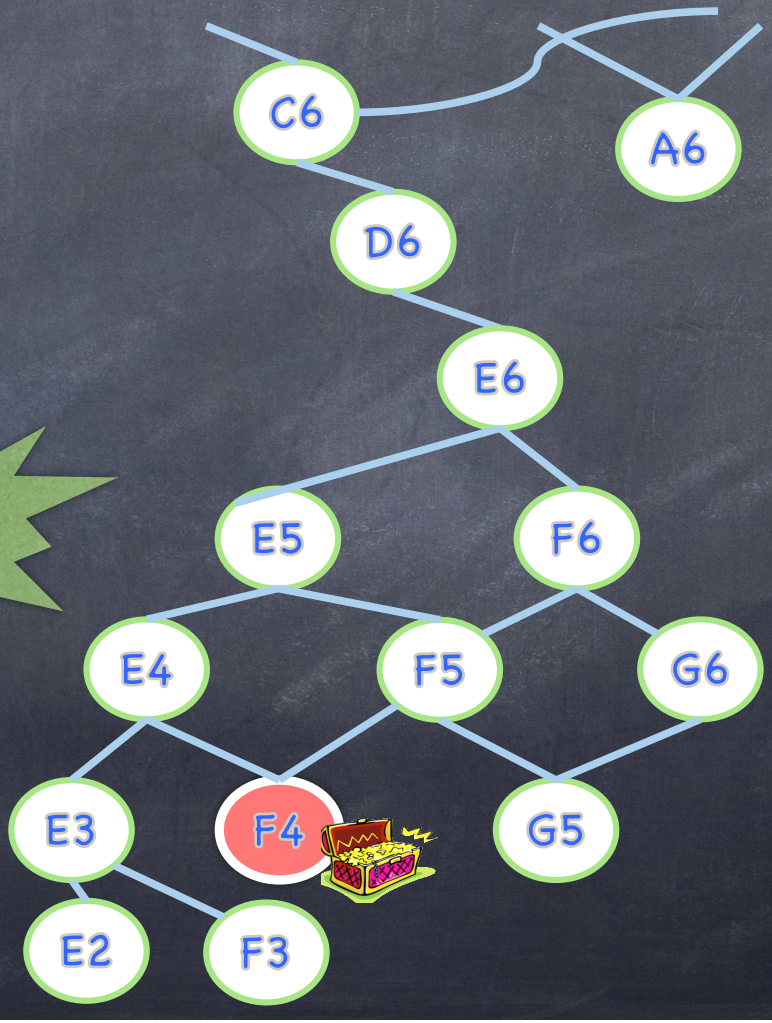
out: [E3 G6 F5 E4 F6 ...]



Le fonctionnement #fin6



Trouvé



in: [G5 E2 F3]

out: [F4 E3 G6 F5 E4 ..]

Le moteur de recherche générique

```
to search
  current <- init-curent()
  tanta que pas trouvé {
    si current = goal alors succès ;; et récupérer le chemin
    in-nodes <- merge-nodes (generate (current), in-nodes )
    si in-nodes est vide, alors échec ;; pas trouvé le chemin
    tmp <- first in-nodes ;; et le supprime de in-nodes
    si tmp est dans out-nodes, continuer ;; pour ne pas boucler
    current <- tmp
    out-nodes <- add(out-nodes, current) ;; pour ne pas boucler
  }
```

Ce qui fait la différence:

- La structure des nœuds et les fonctions d'évaluation
- La gestion du nœud parent
- Le tri des nœuds dans merge-node

La structure d'un noeud

Un nœud comporte plusieurs informations:

- **p** : Le contenu, qui est le lieu même. Dans une carte routière, cela sera la ville. Dans notre cas, c'est la case (le patch)
- **h** : La valeur de ce nœud pour l'heuristique h(n) de choix de chemin à trouver à la « descente » (ou « l'aller » quand on va vers le but).
- **c** : Le coût ou valeur de ce nœud comme « meilleur » nœud pour reconstituer le chemin à la « remontée » (ou au « retour » après avoir trouvé le but). Fonction f(n)
- **p** : Le nœud parent

Note: pour Best-first et Dijsktra, les valeurs **h** et **f** sont confondues. Ce n'est que pour A* que l'on fait la différence

make-node (p, h, c, parent)

Le nœud parent

- Le nœud parent est initialement le nœud qui a généré le nœud,
- Correspond au lieu précédent par lequel on est passé pour arriver à cet endroit
- Vrai pour Best-First
- Mais pour Dijkstra et A* on reconstruit le nœud parent pour reprendre, au retour,

Gestion du nœud parent (merge-nodes)

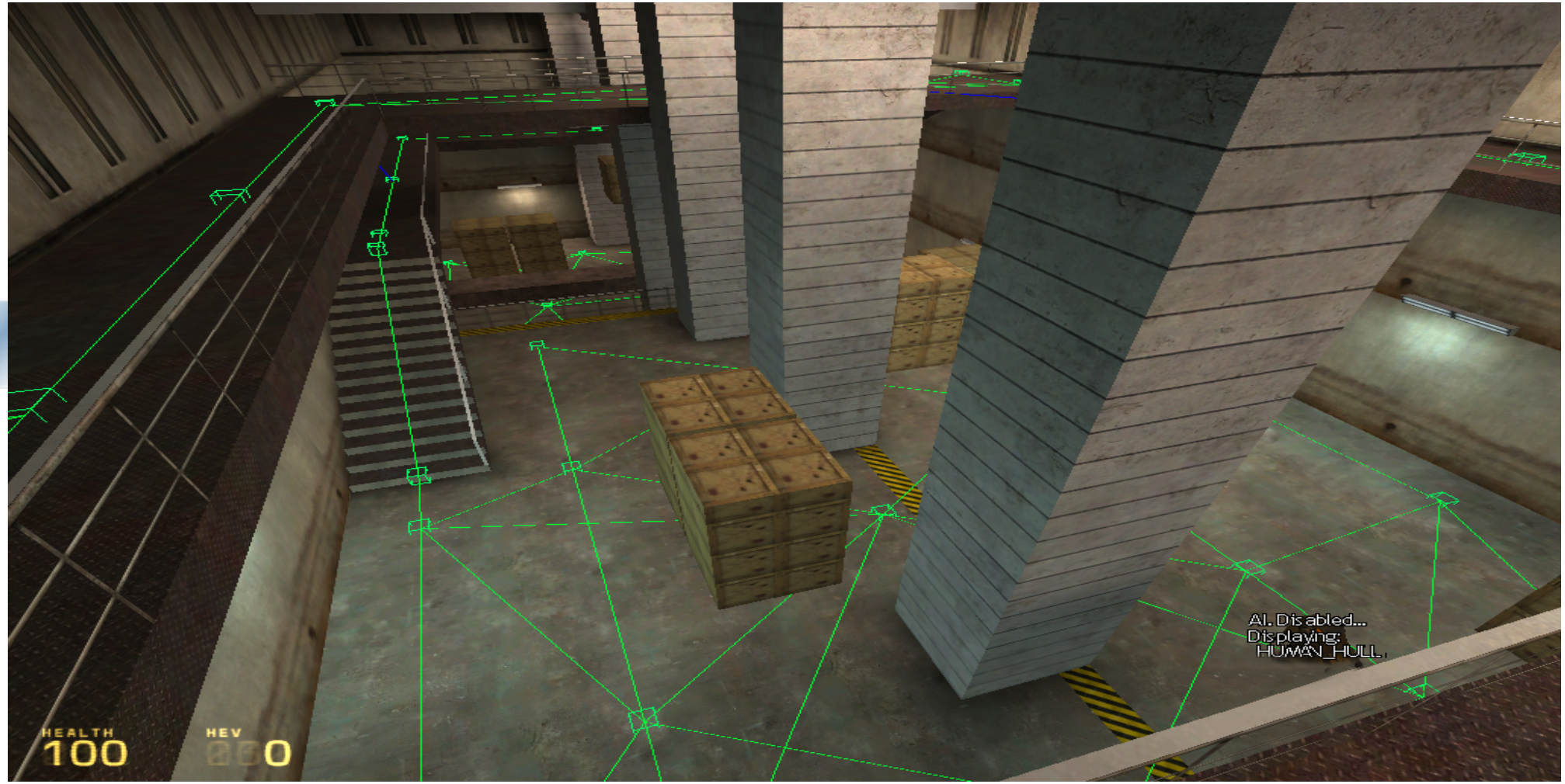
- Principe: pour tout n des nœuds générés par le nœud courant,

- S'il existe x un nœud de in-node de même nom (même case), tel que
  le cout de n < cout de x alors
  val(x) <- val(n),
  cout (x) <- val(n),
  parent(x) <- parent (n)
  s'il n'existe pas de nœud de in-node avec le même nom (même case), alors on ajouter n à in-nodes.

Movement dans les jeux : waypoints

- Waypoint
  - Position sur une carte qui est utilisée pour la navigation
  - Généralement placée par un concepteur de jeu
- Link
  - Connection entre deux waypoints
  - Souvent annoté avec le type de navigation (sauter, nager, grimper)
  - Pour un personnage donné, 2 waypoints sont reliés si:
    - Le personnage a suffisamment de place pour aller d'un noeud à l'autre sans entrer dans le décors
    - Le personnage possède les capacité de navigation requises pour passer d'un point à l'autre.
- Graphe de waypoints
  - Structure de donnée comprenant tous les waypoints ainsi que les liens.
  - Généré manuellement par un concepteur au créé par programme et annoté par un concepteur.

Movement: Node Graph



Conclusion

- Ces trois algorithmes utilisent le même moteur d'exploration
- Ne diffèrent que par quelques points
- Utiliser Dijkstra quand la qualité du résultat est important
  - Très utilisé pour le calcul d'itinéraire dans Mappy ou Google Map
- Utiliser A* quand on a besoin d'avoir un bon résultat rapidement
  - Très utilisé en robotique ou dans les jeux videos