Intelligence artificielle : planification

Grégory Bonnet

Université de Caen Normandie - GREYC

http://www.gregory.bonnet.free.fr/

Plan du cours

Problèmes de planification

Rappel du contexte

Domaine

- variables d'état (ex. : PIÈCE_{13}, DALLE_COULÉE)
- domaines de variable booléens (ex. : VRAI, FAUX)
- domaines de variables multi-valués (ex. : CUISINE, SALON, SALLE_D'EAU)
- contraintes (ex. : $\text{PIÈCE}_{11} = \neg\emptyset \implies \text{TOITURE}$)

Problème de satisfaction de contraintes

À partir de la description des contraintes, trouver une affectation de valeurs aux variables telles que toutes les contraintes soient satisfaites.

Problème de planification

Disposant de connaissances sur des actions et un état à atteindre, trouver la séquence d'actions qui le permet.

Représentation des actions

Schéma d'action

Une action (ACT) est spécifiée en termes de préconditions (PRE) qui doivent être vraies pour qu'elle puisse être exécutée et en terme d'effets (EFF) qui s'ensuivent quand elle est exécutée.

Exemple

$\texttt{ACT}(\text{CASSER}_{14})$

- $\texttt{PRE} := \text{PIÈCE}_{14} = \neg\emptyset$
- $\texttt{EFF} := \text{PIÈCE}_{14} = \emptyset$

Exemple

$\texttt{ACT}(\text{ECHANGER}_{11,12})$

- $\texttt{PRE} := \text{PIÈCE}_{11} = \neg\emptyset \vee \text{PIÈCE}_{12} = \neg\emptyset$
- $\texttt{EFF} := \text{PIÈCE}_{11} = \text{PIÈCE}_{12} \wedge \text{PIÈCE}_{12} = \text{PIÈCE}_{11}$

Exécuter une action

Exécuter une action A dans l'état E conduit à un état E' identique exceptées les modifications provoquées par $\texttt{EFF}(A)$.

Hypothèses sur les actions

Redondance des effets

- Un effet positif ne modifie pas un littéral si ce dernier est déjà positif,
- Un effet négatif ne modifie pas un littéral si ce dernier est déjà négatif.

Problème du cadre

Que faire des littéraux qui ne sont pas spécifiés par l'action ?

- Les littéraux qui ne sont pas dans `PRE` sont ignorés : leur valeur importe peu,
- Les littéraux qui ne sont pas dans `EFF` sont ignorés : leur valeur reste inchangée.

Problème de l'applicabilité

Une action est applicable à chaque état qui satisfait les préconditions. Sinon l'action n'a pas d'effet.

Problème de qualification

L'exécution d'une action réussit toujours : il ne peut pas y avoir d'échec.

Qu'est-ce que la planification ?

Étant donné :

- une description du monde
- un état initial du monde
- une description du but
- un ensemble d'actions disponibles pour changer le monde

Planification

La planification consiste à calculer une séquence d'actions permettant de passer de l'état initial à un état qui satisfait le but.

Planification progressive ou chaînage avant

Partir de l'état initial et chercher à atteindre un état qui satisfait le but.

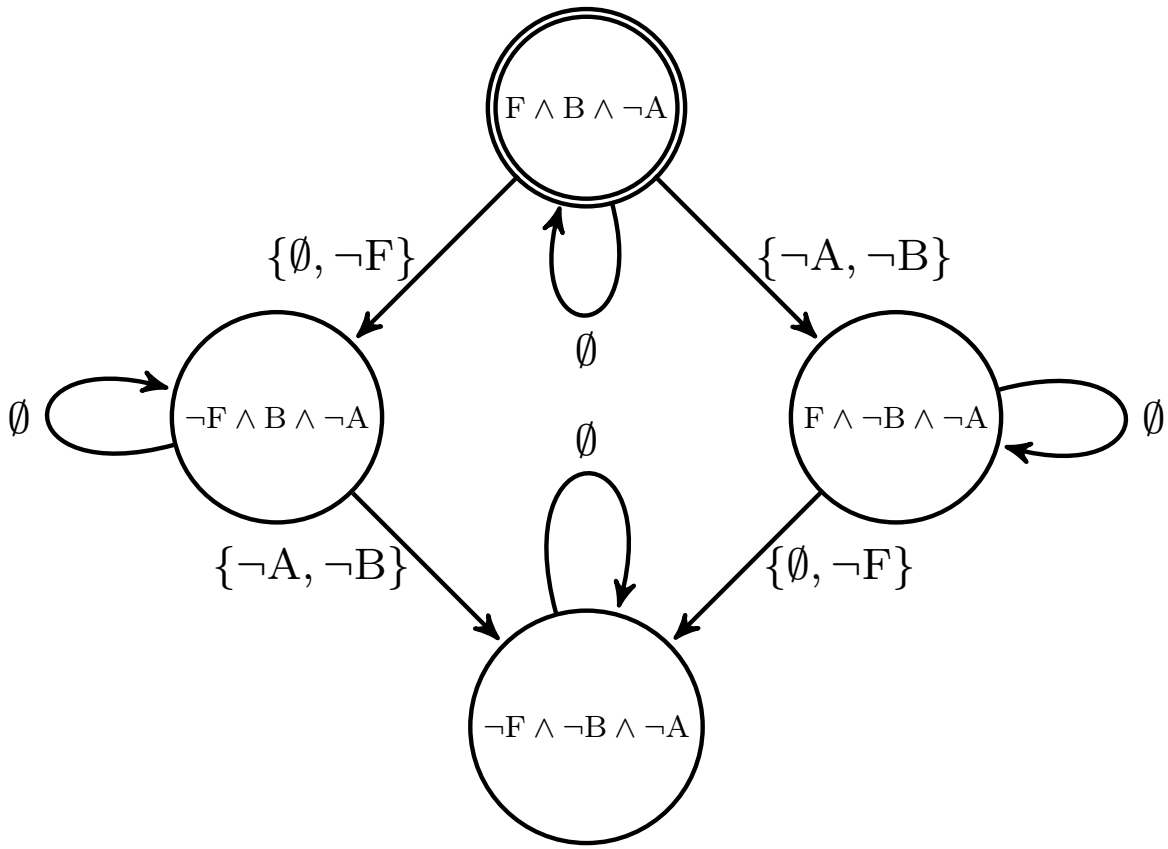
Planification régressive ou chaînage arrière

Part d'une spécification partielle du but et chercher à atteindre l'état initial.

Planification et graphes

Un problème de planification peut se représenter par un graphe

- Un nœud est un état du monde
- Un arc est l'application d'une action
- Un chemin entre l'état initial et un état satisfaisant le but est un plan d'actions



Que faut-il coder ?

satisfies(instantiation, goals)

1: **return** *goals* $\subset$ *instanciation*

is_applicable(action, instantiation)

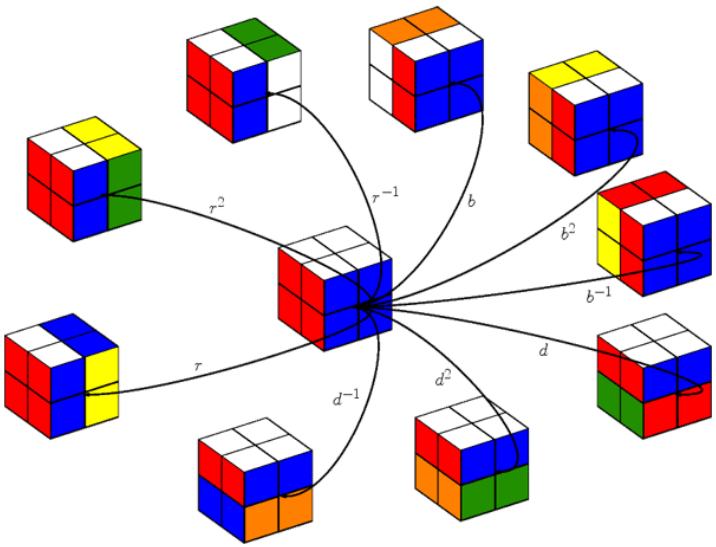
1: **return** *action.preconditions* $\subset$ *instanciation*

apply(action, instantiation)

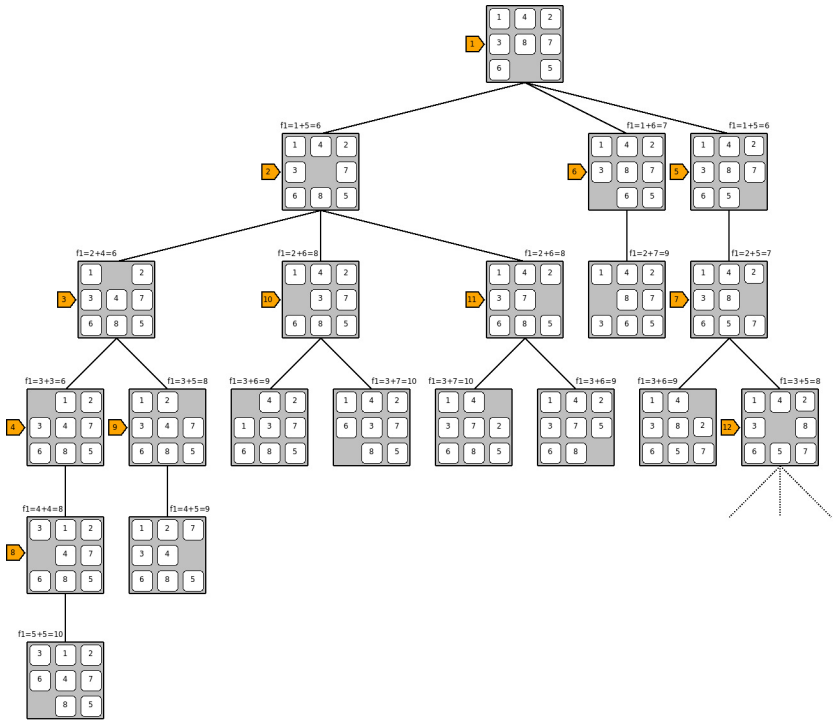
1: **if** is_applicable(*action*, *instantation*) **then**
2: $P \leftarrow$ *instantion.positive* $-$ *action.effects.negative* $+$ *action.effects.positive*
3: $N \leftarrow$ *instantion.negative* $-$ *action.effects.positive* $+$ *action.effects.negative*
4: **return** Instantiation(P, N)
5: **else**
6: **return** Instantiation(*instantiation.positive*, *instantiation.negative*)
7: **end if**

Application à d'autres domaines

Rubik's Cube



Taquin



Recherche non informée

Recherche en profondeur

DFS(problem, instantiation, plan, closed)

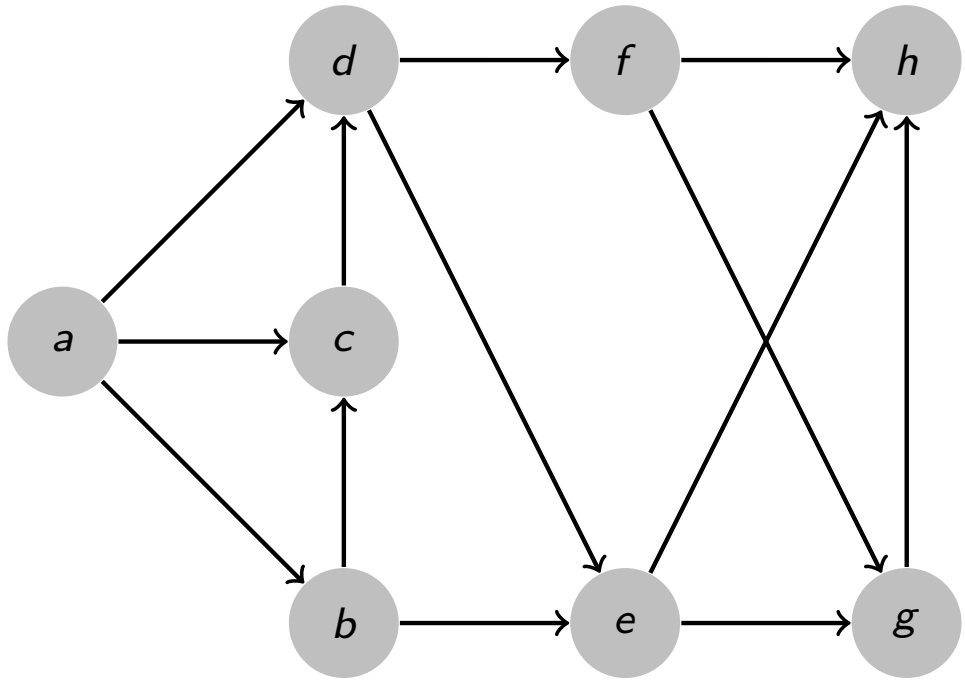
Require: instantiation $\leftarrow$ *initial_state*, plan $\leftarrow$ STACK({}), closed $\leftarrow$ {*initial_state*}

```
1:  if satisfies(instantiation, problem.goal) then
2:      return plan
3:  else
4:      for all action ∈ problem.actions do
5:          if is_applicable(action, instantiation) then
6:              next ← apply(action, instantiation)
7:              if not next ∈ closed then
8:                  push(plan, action)
9:                  closed ← closed ∪ next
10:                 subplan ← dfs(problem, next, plan, closed)
11:                 if not subplan = ∅ then
12:                     return subplan
13:                 else
14:                     pop(plan)
15:                 end if
16:             end if
17:         end if
18:     end for
19:     return ∅ {Can be implemented as a null object}
20: end if
```

Exercice : appliquez DFS

Hypothèses

- l'état initial est *a* et l'état but est *h*
- les arêtes sortantes indiquent l'action applicable dans un état
- les arêtes entrantes indiquent l'état résultant de l'application de l'action
- l'ordre des enfants d'un nœud est l'ordre alphabétique



Problèmes liés à la recherche en profondeur

Problème des solutions sous-optimales

- Une branche de l'arbre peut mener au but
- Rien ne dit qu'il existe une branche plus courte
- $\Longrightarrow$ il existe donc un plan plus parcimonieux !

Problème des arbres de grande profondeur

L'algorithme risque de développer une branche stérile sans que le mécanisme de retour en arrière puisse se déclencher.

Améliorations possibles : la recherche en profondeur itérative

- Profondeur maximale de récursion
- Pour X de 1 à N faire un `DFS` à profondeur X
- Complexité en temps : $1 + b + b^2 + \ldots + b^d = \mathcal{O}(b^d)$
- Complexité en espace : $\mathcal{O}(b \times d)$

Recherche en largeur

BFS(problem)

Require: *father* $\leftarrow$ MAP(*State*, *State*), *plan* $\leftarrow$ MAP(*State*, *Action*)

```
 1: closed ← {problem.initial_state}
 2: open ← QUEUE({problem.initial_state})
 3: father[problem.initial_state] ← ∅
 4: if satisfies(problem.initial_state, problem.goal) then
 5:     return {}
 6: end if
 7: while open ≠ {} do
 8:     instantiation ← dequeue(open)
 9:     closed ← closed ∪ instantiation
10:     for all action ∈ problem.actions do
11:         if is_applicable(action, instantiation) then
12:             next ← apply(action, instantiation)
13:             if not next ∈ closed and not next ∈ open then
14:                 father[next] ← instantiation
15:                 plan[next] ← action
16:                 if satisfies(next, problem.goal) then
17:                     return get-bfs-plan(father, plan, next)
18:                 else
19:                     enqueue(open, next)
20:                 end if
21:             end if
22:         end if
23:     end for
24: end while
25: return ∅ {Can be implemented as a null object}
```

Recherche en largeur

Reconstruction du plan

get-bfs-plan(father, plan, goal)

1: $BFS\_plan \leftarrow \texttt{QUEUE}(\{\})$
2: **while** $goal \neq \emptyset$ **do**
3: $\quad$ enqueue($BFS\_plan, plan[goal]$)
4: $\quad goal \leftarrow father[goal]$
5: **end while**
6: **return** reverse(BFS_plan)

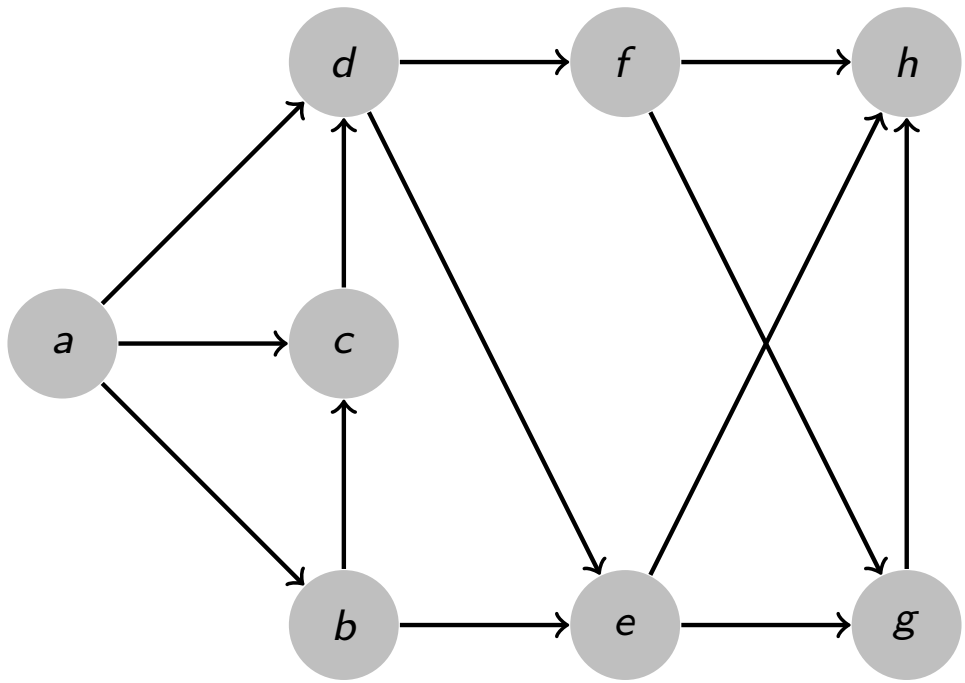
Propriétés

- L'algorithme trouve toujours le plan le plus court en premier
- Toutefois, il peut explorer inutilement des nœuds

Exercice : appliquez BFS

Hypothèses

- l'état initial est *a* et l'état but est *h*
- les arêtes sortantes indiquent l'action applicable dans un état
- les arêtes entrantes indiquent l'état résultant de l'application de l'action
- l'ordre des enfants d'un nœud est l'ordre alphabétique



Recherche heuristique

Introduction du problème des coûts

Le coût d'un plan n'est pas le nombre d'action

- Une action a un coût
- le coût d'un plan est la somme du coût des actions
- $\Longrightarrow$ algorithme de Dijkstra

Compléxité de la recherche aveugle

- $\mathcal{O}(b^d)$
- pouvons-nous éviter d'explorer les états inutiles ?
- $\Longrightarrow$ algorithme A^*

DIJKSTRA(problem)

Require: $plan \leftarrow \texttt{MAP}(State, Action)$, $distance \leftarrow \texttt{MAP}(State, \texttt{REAL})$, $father \leftarrow \texttt{MAP}(State, State)$

$goals \leftarrow \emptyset$, $father[problem.initial\_state] \leftarrow \emptyset$
$distance[problem.initial\_state] \leftarrow 0$, $open \leftarrow \{problem.initial\_state\}$
while $open \neq \emptyset$ **do**
    $instantiation \leftarrow \text{argmin}_{node \in open}(distance(node))$
    $open \leftarrow open \setminus \{instantiation\}$
    if $\text{satisfies}(instantiation, problem.goal)$ **then**
        $goals \leftarrow goals \cup \{instantiation\}$
    end if
    for all $action \in problem.actions$ **do**
        if $\text{is\_applicable}(action, instantiation)$ **then**
            $next \leftarrow \text{apply}(action, instantiation)$
            if not $next \in distance$ **then**
                $distance[next] \leftarrow +\infty$
            end if
            if $distance[next] > distance[instantiation] + \text{cost}(action)$ **then**
                $distance[next] \leftarrow distance[instantiation] + \text{cost}(action)$
                $father[next] \leftarrow instantiation$
                $plan[next] \leftarrow action$
                $open \leftarrow open \cup \{next\}$
            end if
        end if
    end for
end while
if $goal = \emptyset$ **then**
    return $\emptyset$ {Can be implemented as a null object}
else
    return $\text{get-dijkstra-plan}(father, plan, goals, distance)$
end if

Algorithme de Dijkstra

Reconstruction du plan

get-dijkstra-plan(father, plan, goals, distance)

1: $DIJ\_plan \leftarrow \texttt{QUEUE}(\{\})$
2: $goal \leftarrow \text{argmin}_{node \in goals}(distance[node])$
3: **while** $goal \neq \emptyset$ **do**
4: $\quad \text{enqueue}(DIJ\_plan, plan[goal])$
5: $\quad goal \leftarrow father[goal]$
6: **end while**
7: **return** $\text{reverse}(DIJ\_plan)$

Hypothèses derrière l'algorithme de Dijkstra

- Les coûts sont non négatifs
- Un sous-ensemble du plan de coût minimal est aussi de coût minimal

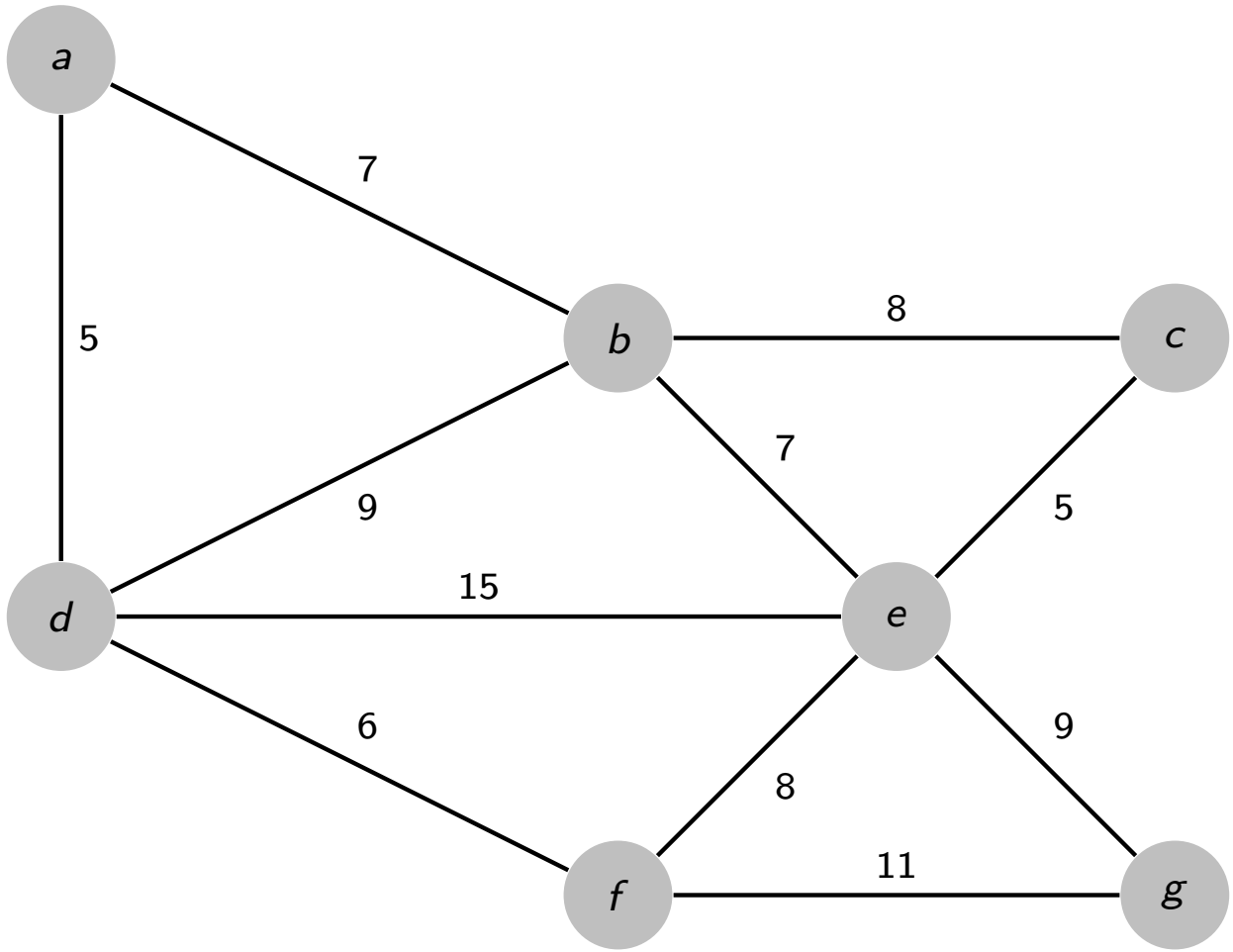
Propriétés

- L'algorithme retourne le plan de coût minimal
- Toutefois, il peut toujours explorer inutilement des nœuds

Exercice : appliquez l'algorithme de Dijkstra

Hypothèses

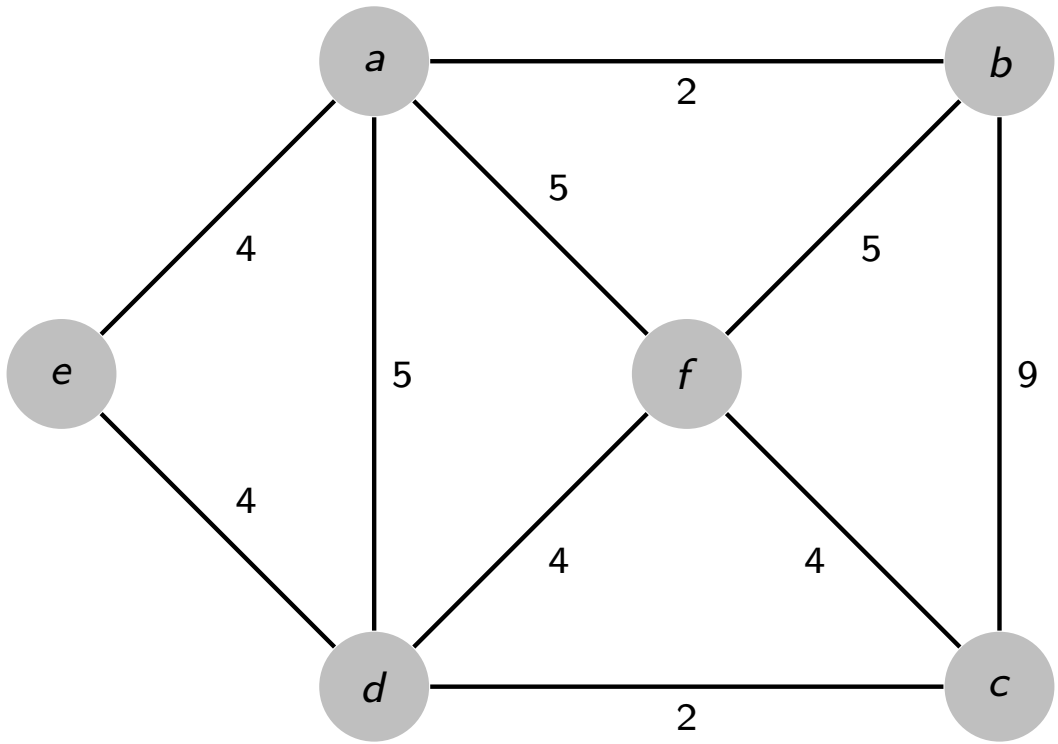
- l'état initial est a et l'état but est g
- les arêtes sont non orientées et le coût des actions y est indiqué



One more time with feeling !

Calculez

Le plan menant de a à c !



Est-ce toujours une bonne idée d'explorer les effets de certaines actions ?

Une action qui nous éloigne du but
Une action qui est très coûteuse

Un algorithme de recherche en « meilleur d'abord »

Objectif

Rechercher la solution dans un graphe d'états à partir d'un état initial et en calculant un meilleur chemin vers cette solution.

Meilleur = Heuristique

$$f(e_i) = g(e_i) + h(e_i)$$

- $g(e_i)$: coût pour aller de e_0 (initial) à e_i (courant)
- $h(e_i)$: estimation du coût pour aller de e_i (courant) à e_n (final)

Équivalence avec l'algorithme de Dijkstra

Si $\forall e_i \in V : h(e_i) = 0$ alors $f(e_i) = g(e_i)$ et A^* fait une recherche en largeur

Équivalence avec la descente de gradient

Si $\forall e_i \in V : f(e_i) = h(e_i)$ alors A^* fait une recherche en profondeur

Propriétés de l'heuristique

Admissibilité

$$\forall e \in V : h(e) \leq h^*(e) \quad \text{où } h^* \text{ est l'heuristique optimale}$$

- l'heuristique h est dite *admissible* si elle minore le coût réel (inconnu) du chemin
- si h est admissible alors l'algorithme retourne le meilleur chemin

Monotonie

$$\forall e_i, e_j \in V, e_j \in \text{successeurs}(e_i) : h(e_i) \leq h(e_j) + k(e_i, e_j)$$

- si l'heuristique h est *monotone* alors elle est *admissible*
- et l'algorithme A^* est de complexité linéaire

Information

$$\forall e \in V : 0 \leq h_1(e) \leq h_2(e)$$

- l'heuristique h_2 est dite *mieux informée* que h_1
- plus une heuristique est informée, moins le nombre d'états explorés sera grand

Principes de l'algorithme A^*

- sélectionner un nœud e_j dont la valeur f est minimale
- ajouter à l'espace de recherche les successeurs de e_j
- conserver pour chaque nœud le meilleur chemin à partir de e_i.
- l'algorithme termine dès que l'on a atteint l'un des nœuds solutions

Utilisation de files d'attente prioritaires

liste ouverte : contient tous les nœuds qui restent à évaluer

liste fermée : contient tous les nœuds qui ont été évalués

Propriété

Si l'heuristique est monotone alors il est inutile d'utiliser des listes.

ASTAR(*problem*)

Require: *plan* $\leftarrow$ MAP(*State*, *Action*), *father* $\leftarrow$ MAP(*State*, *State*)
Require: *distance* $\leftarrow$ MAP(*state*, REAL), *value* $\leftarrow$ MAP(*State*, REAL)
open $\leftarrow$ {*problem.initial_state*}
father[*problem.initial_state*] $\leftarrow$ $\emptyset$
distance[*problem.initial_state*] $\leftarrow$ 0
value[*problem.initial_state*] $\leftarrow$ heuristic(*problem.initial_state*)
while *open* $\neq$ $\emptyset$ **do**
    instantiation $\leftarrow$ $\text{argmin}_{node \in open}$(*value*(*node*))
    if satisfies(*instantiation*, *problem.goal*) **then**
        return get-bfs-plan(*father*, *plan*, *instantiation*)
    else
        open $\leftarrow$ *open* $\setminus$ {*instantiation*}
        for all *action* $\in$ *problem.actions* **do**
            if is_applicable(*action*, *instantiation*) **then**
                next $\leftarrow$ apply(*action*, *instantiation*)
                if not *next* $\in$ *distance* **then**
                    distance[*next*] $\leftarrow$ $+\infty$
                end if
                if *distance*[*next*] $>$ *distance*[*instantiation*] + cost(*action*) **then**
                    distance[*next*] $\leftarrow$ *distance*[*instantiation*] + cost(*action*)
                    value[*next*] $\leftarrow$ *distance*[*next*] + heuristic(*next*)
                    father[*next*] $\leftarrow$ *instantiation*, *plan*[*next*] $\leftarrow$ *action*
                    open $\leftarrow$ *open* $\cup$ {*next*}
                end if
            end if
        end for
    end if
end while
return $\emptyset$ {Can be implemented as a null object}

Exercice : heuristiques admissibles

Jeu du taquin

Le taquin est un jeu solitaire en forme de damier. Il est composé de 15 petits carreaux numérotés de 1 à 15 qui glissent dans un cadre prévu pour 16. Il consiste à remettre dans l'ordre les 15 carreaux à partir d'une configuration initiale quelconque.

Une heuristique possible

Soit l'heuristique h_1 qui associe à chaque état le nombre de carreaux mal placés.

Questions

1. Démontrez que h_1 est admissible,
2. Trouver une heuristique admissible h_2 plus informée que h_1.

Exercice : appliquez A*

Considérez le problème suivant

- soit V 5 variables booléennes a, b, c, d, e
- soit $\{\bar{a}, \bar{b}, \bar{c}, \bar{d}, \bar{e}\}$ l'état initial
- soit $\{a, b, c, d, e\}$ l'état but
- soit l'heuristique h qui associe à chaque état le nombre de variables à faux

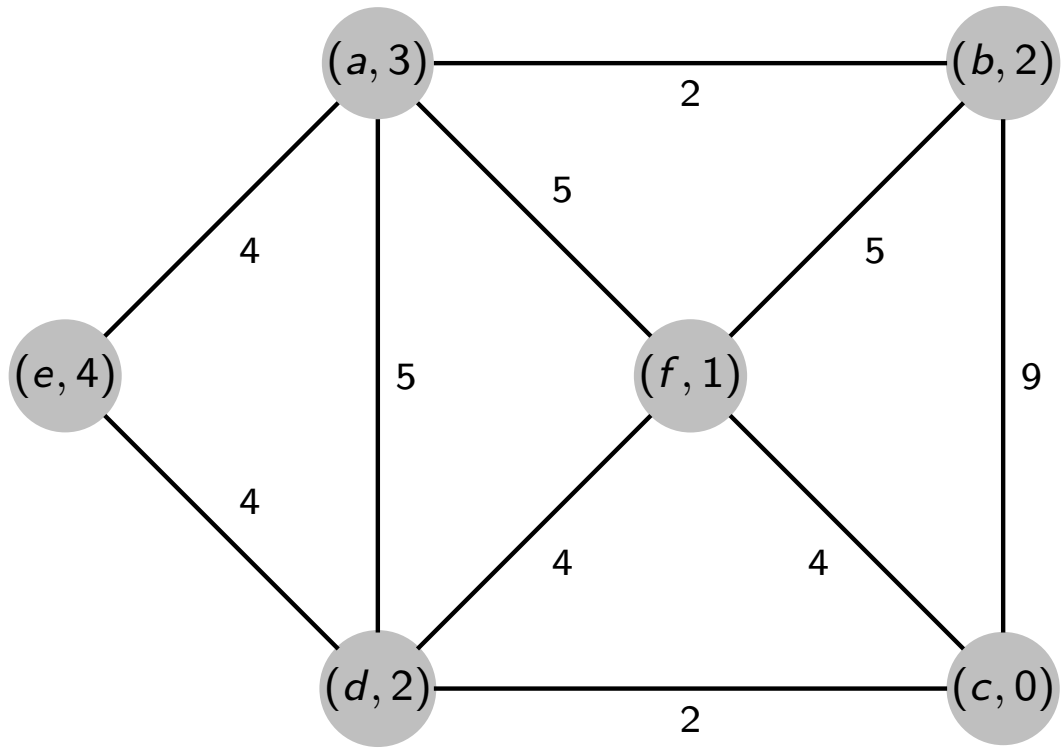
Actions

- $\forall v \in V : \mathtt{S_v} : (\mathtt{PRE} = \top, \mathtt{EFF} = \{v\})$ coûtant 1
- $\mathtt{D_1} : (\mathtt{PRE} = \top, \mathtt{EFF} = \{a, b\})$ coûtant 1.5
- $\mathtt{D_2} : (\mathtt{PRE} = \top, \mathtt{EFF} = \{c, d\})$ coûtant 1.5
- $\mathtt{D_2} : (\mathtt{PRE} = \top, \mathtt{EFF} = \{d, e\})$ coûtant 1.5
- $\mathtt{T_1} : (\mathtt{PRE} = \top, \mathtt{EFF} = \{a, b, c\})$ coûtant 4
- $\mathtt{T_2} : (\mathtt{PRE} = \top, \mathtt{EFF} = \{c, d, e\})$ coûtant 4

One more time with feeling !

Calculez

- Le plan menant de a à c !
- La valeur heuristique est indiquée à coté de l'identifiant du nœud



Recherche heuristique avancée

Algorithme de recherche en faiseau (ou *beam search*)

Problème

Si A^* ne développe pas systématiquement l'arbre de résolution, le nombre de nœuds ouverts peut devenir prohibitif.

Solution

Ne conserver dans les ouverts que les k meilleurs nœuds

Propriétés

- perte de la complétude et de l'optimalité
- si $k \to \infty$ alors la recherche en faiseau se comporte comme A^*

Variante avec approfondissement itératif

- initaliser k avec une petite valeur
- en cas d'échec, recommencer avec $k = k + \Delta$

Algorithme A^* multi-critère

Fonction heuristique

$$f : V \longmapsto \mathbb{R}^k \text{ (idem pour } g \text{ et} h)$$

$$f = g + h$$

Comparer deux vecteurs f

- somme pondérée
- moyenne pondérée
- ordre lexicographique
- etc.

Algorithme B^*

Principe

Les nœuds sont évalués par des intervales plutôt que par des valeurs singletons.

Heuristique

$$f : V \to \mathbb{R}^2 \quad \text{(aussi noté } (f_-, f_+) \text{)}$$

Règles d'évaluation

1. évaluer les successeurs d'un nœud dans un ordre arbitraire
2. $f_-(\text{successeurs}(e_i)) > f_-(e_i) \implies f_-(e_i) \leftarrow f_-(\text{successeurs}(e_i))$
3. $f_+(e_i) > f_+(\text{successeurs}(e_i)) \implies f_+(e_i) \leftarrow f_+(\text{successeurs}(e_i))$

Condition de terminaison

Un successeur de la racine a une valeur pessimiste qui n'est pas inférieure aux valeurs optimistes des autres successeurs

Conclusion

Conclusion

Les problèmes de planification peuvent être modélisés par des graphes

Problème de planification

canonique : graphe entièrement connu (connaissance parfaite)
coûts fixes (statique)

exploration : graphe découvert au fur et à mesure (connaissance imparfaite)
coûts variables (dynamique)

Algorithme A^*

- généralisation de l'algorithme de Dijkstra
- recherche en « meilleur d'abord » selon une fonction heuristique
- de nombreuses variantes (recherche en faiseau, recherche frontalière, etc.)

Bibliographie

Jean-Marc Aliot, Thomas Schiex, Pascal Brisset et Frédérick Garcia
Intelligence artificielle et informatique théorique
Cépaduès, 2007.

Stuart Russell et Peter Norvig
Intelligence artificielle
Pearson, 2010.