# Planification

Tristan Cazenave
Tristan.Cazenave@dauphine.fr

# Références

- Intelligence Artificielle de Stuart Russell et Peter Norvig.
- Cours planification de Pavlos Moraitis et Amal El Fallah.

# STRIPS

- Le langage STRIPS :
- Planification = tâche qui consiste à trouver une séquence d'actions qui réalisera un objectif.
- La représentation des problèmes de planification concerne des états, des actions et des objectifs.
- Le but est de trouver un langage qui est à la fois suffisamment expressif pour décrire une grande varieté de problèmes mais assez restrictif pour permettre à des algorithmes efficaces d'agir.

# STRIPS

- Nous allons considérer des environnements de planification classique (c.a.d. totalement observables, déterministes, finis, statiques -les changements n'étant dûs qu'aux seules actions des agents - et discrets-en temps, action, objets et effets).
- Le langage basique de représentation des planificateurs classiques est le langage STRIPS (Stanford Research Institute Problem Solver)

# STRIPS

- Représentation des états : Les planificateurs décomposent le monde en conditions logiques et représentent un état comme une conjonction de littéraux positifs.
- On peut considérer des littéraux propositionnels : Pauvre ^ Inconnu peut représenter l'état d'un agent infortuné.
- On peut considérer des littéraux du premier-ordre:

  A(Avion1,Paris) ^ A(Avion2, JFK) peuvent représenter un état du problème du frêt
- Les littéraux du premier-ordre doivent être instanciés (ground) et libres de fonctions.
- Des littéraux comme Sur(x,y) ou habite(Pere(Paul), Paris) ne sont pas permis.
- L'hypothèse du monde-clos est utilisé :
- Toute condition non mentionnée dans un état est considérée fausse.

# STRIPS

- Représentation des objectifs:

  Un objectif est un état partiellement spécifié, représenté comme une conjonction de littéraux instanciés positifs comme Riche ^ Reputé ou Dans (Fret1, Avion1)

- Un état propositionnel g satisfait un objectif o si g contient tous les atomes de o :

  L'état Riche ^ Reputé ^ Misérable satisfait l'objectif Riche ^ Reputé

# STRIPS

- Représentation des actions :
- Une action est spécifiée en termes de préconditions qui doivent être valables avant qu'elle puisse être exécutée et en terme d'effets qui s'ensuivent quand elle est exécutée
- Une action pour faire voler un avion d'un endroit à un autre est:

  Action(Voler(avion, départ, arrivée),

  PRECOND: A(avion, départ) ^ Avion(avion) ^ Aéroport(départ) ^ Aéroport(arrivée)

  EFFET: not A(avion, départ) ^ A(avion, arrivée))
- Nous appelons cela un schéma d'action : il représente des actions différentes qui peuvent être dérivées en instanciant avion, départ et arrivée à des constantes différentes.

# STRIPS

- Plus généralement un schéma d'action est constitué de trois parties :
- Le nom de l'action et une liste de paramètres, par exemple Voler(avion, départ, arrivée), sert à identifier l'action
- La précondition est une conjonction de littéraux positifs, libres de fonctions, faisant état de ce qui doit être vrai dans un état avant que l'action puisse être exécutée. Toute variable apparaissant dans la précondition doit aussi apparaître dans la liste de paramètres de l'action
- L'effet est une conjonction de littéraux, libres de fonctions décrivant comment l'état change quand l'action est exécutée. Un littéral positif P dans l'effet est supposé être vrai dans l'état résultant de l'action, tandis qu'un littéral négatif not P est supposé être faux. Les variables dans les effets doivent aussi apparaître dans la liste de paramètres de l'action

# STRIPS

- Ayant défini la syntaxe pour la représentation des problèmes de planification nous pouvons maintenant définir la sémantique:
- La manière la plus directe est de définir comment les actions affectent les états
- Une action est applicable à chaque état qui satisfait la précondition; autrement l'action n'a pas d'effet
- Pour un shéma d'action de premier-ordre, établir l'applicabilité impliquera une substitution theta pour les variables dans la précondition

# STRIPS

- Par exemple considérons que l'état courant est décrit par:

  A(Avion1, JFK) ^ A(Avion2, CDG) ^ Avion(Avion1) ^ Avion(Avion2) ^ Aéroport(JFK) ^ Aéroport(CDG)

- Cet état satisfait la précondition

  A(avion, départ) ^ Avion(avion) ^ Aéroport(départ) ^ Aéroport(arrivée)

  avec la substitution {avion/Avion1, départ/JFK, arrivée/CDG} (entre autres)

- Alors l'action concrète Voler(Avion1, JFK, CDG) est applicable

# STRIPS

- En commençant à l'état s, le résultat de l'exécution une action applicable alpha est un état s' qui est le même que s excepté le fait que chaque littéral positif P dans l'effet de alpha est ajouté dans s' et que chaque littéral négatif not P est retiré de s'
- Après Voler(Avion1, JFK, CDG) l'état courant devient A(Avion1, CDG) ^ A(Avion2, CDG) ^ Avion(Avion1) ^ Avion(Avion2) ^ Aéroport(JFK) ^ Aéroport(CDG)

# STRIPS

- Il faut noter que si un effet positif est déjà dans s celui-ci n'est pas ajouté deux fois et si un effet négatif n'est pas dans s alors cette partie de l'effet est ignorée
- Cette définition incorpore l'hypothèse de STRIPS: chaque littéral non mentionné dans l'effet reste inchangé. De cette façon STRIPS évite le frame problem (c.a.d. représenter toutes les choses qui restent les mêmes)
- La solution d'un problème de planification consiste (dans sa forme la plus simple) en une séquence d'actions qui quand elle est exécutée à l'état initial, a pour résultat un état où l'objectif est satisfait

# STRIPS

- La notation STRIPS est adéquate pour plusieurs domaines réels. Cependant elle ne peut pas résoudre de façon naturelle les ramifications des actions
- S'il existe des personnes, des paquets ou des bagages dans l'avion alors ils doivent aussi changer de position quand l'avion vole.
- Nous pouvons représenter ces changements comme les effets directs du vol, tandis qu'il semble plus naturel de représenter la position du contenu de l'avion comme une conséquence logique de la position de l'avion
- Les systèmes de planification classique ne peuvent aussi résoudre le problème de qualification (c.a.d. le problème de circonstances non représentées qui pourraient causer l'échec de l'action)

# STRIPS

- Exemple : Transport de Frêt Aérien
- Problème de transport de frêt aérien impliquant chargement et déchargement de frêt (cargo), et d'avions et emmenant celui-ci d'une place à l'autre.
- Le problème peut être défini avec trois actions: Charger, Décharger et Voler

# STRIPS

- Les actions affectent deux prédicats: Dans(f,p) signifie que le frêt f est dans l'avion p et A(x,alpha) signifie que l'objet x (soit avion soit frêt) est à l'aéroport alpha
- Le frêt n'est A nulle part quand il est Dans un avion, et par conséquent A signifie "disponible pour utilisation à une position donnée"
- Exercice : Modéliser le problème en STRIPS. On veut transporter du frêt de CDG à JFK par un avion, ainsi que du frêt de JFK à CDG par un autre avion.

# STRIPS

- Init(A(F1,CDG) ^ A(F2, JFK) ^ A(Avion1, CDG) ^ A(Avion2, JFK) ^ Frêt(F1) ^ Frêt(F2) ^ Avion(Avion1) ^ Avion(Avion2) ^ Aéroport(CDG) ^ Aéroport(JFK))
- Objectif(A(F1, JFK) ^ A(F2, CDG))
- Action(Charger(f, avion, alpha),

  PRECOND: A(f, alpha) ^ A(avion, alpha) ^ Frêt(f) ^ Avion(avion) ^ Aéroport(alpha)

  EFFET: not A(f, alpha) ^ Dans(f, avion))

# STRIPS

Action(Décharger(f, avion, alpha),

PRECOND: Dans(f, avion) ^

A(avion, alpha) ^

Frêt(f) ^ Avion(avion) ^

Aéroport(alpha)

EFFET: A(f, alpha) ^ not Dans(f, avion))

# STRIPS

- Action(Voler(avion, départ, arrivée),

  PRECOND: A(avion, départ) ^ Avion(avion) ^

  Aéroport(départ) ^ Aéroport(arrivée)

  EFFET: not A(avion, départ) ^ A(avion, arrivée))
- Solution: [Charger(F1, Avion1, CDG), Voler(Avion1, CDG, JFK), Décharger(F1, Avion1,JFK), Charger(F2, Avion2, JFK), Voler(Avion2, JFK, CDG), Décharger(F2, Avion2, CDG)]

# STRIPS

- Exemple : Changer un pneu
- Problème : on veut remplacer le pneu crevé sur l'essieu par un pneu qui est dans le coffre de la voiture.
- Le problème peut être défini avec quatre actions: Retirer le pneu du coffre, Retirer le pneu crevé de l'essieu, mettre le pneu sur l'essieu, et laisser la voiture pendant la nuit. On considère que la voiture est dans un endroit particulièrement mal famé, et que la laisser pendant la nuit fait disparaître les pneus.
- Exercice : Modéliser le problème en STRIPS.

# STRIPS

- Init(Sur(PneuCreve,Essieu) ^
  Dans(Pneu, Coffre))
- Objectif(Sur(Pneu,Essieu))
- Action(Retirer(Pneu,Coffre),
  PRECOND: Dans(Pneu,Coffre)
  EFFET: not Dans(Pneu,Coffre) ^
  Sur(Pneu,Sol))

# STRIPS

- Action(Retirer(PneuCreve,Essieu),

  PRECOND: Sur(PneuCreve,Essieu)

  EFFET: not Sur(PneuCreve,Essieu) ^

  Sur(PneuCreve,Sol))

# STRIPS

- Action(Mettre(Pneu,Essieu),

  PRECOND: Sur(Pneu,Sol) ^

  not Sur(PneuCreve,Essieu)

  EFFET: not Sur(Pneu,Sol) ^

  Sur(Pneu,Essieu))

# STRIPS

- Action(PartirPendantLaNuit,

  PRECOND:

  EFFET: not Sur(Pneu,Sol) ^

  not Sur(Pneu,Essieu) ^

  not Dans(Pneu,Coffre) ^

  not Sur(PneuCreve,Sol) ^

  not Sur(PneuCreve,Essieu))

# STRIPS

- Exemple : le Monde des Blocs
- C'est le plus fameux exemple des domaines de planification
- Il consiste en un ensemble de blocs, de forme cubique posés sur une table
- Les blocs peuvent s'entasser les uns sur les autres, mais seulement un bloc peut être mis directement sur un autre bloc
- Un bras de robot peut tenir seulement un bloc à la fois, donc il ne peut pas tenir un bloc qui a un autre bloc sur lui

# STRIPS

- L'objectif sera toujours de bâtir une ou plusieurs piles de blocs, spécifiées en termes de quels blocs sont au dessus d'autres blocs
- Par exemple un objectif pourrait être de poser le bloc A sur B et le bloc C sur D
- Nous utiliserons Sur(b, x) pour indiquer que le bloc b est sur x, où x est un autre bloc ou la table
- L'action pour déplacer le bloc b du dessus de x au dessus de y sera Déplacer(b, x, y)
- Une des préconditions pour déplacer b est qu'il n'y ait pas un autre bloc sur lui
- Nous pouvons introduire un prédicat Dégagé(x) qui est vrai quand x n'a rien sur lui

# STRIPS

- L'action Déplacer, déplace un bloc b de x à y si les deux (b et y) sont dégagés. Après que le déplacement soit fait, x est dégagé mais y ne l'est plus :

  Action(Déplacer(b, x, y),

  PRECOND: Sur(b, x) ^ Dégagé(b) ^ Dégagé(y)

  EFFET: Sur(b, y) ^ Dégagé(x) ^ not Sur(b, x) ^ not Dégagé(y))
- Cette action ne maintient pas Dégagé proprement quand x ou y est la table
- Quand x=Table cette action a comme effet Dégagé(Table) mais la table ne doit pas être dégagée et quand y=Table, elle a la précondition Dégagé(Table) mais la table n'a pas à être dégagée pour poser un bloc sur elle.

# STRIPS

- Pour corriger ce problème nous faisons deux choses:
- Nous introduisons une autre action, DéplacerSurTable(b, x), pour déplacer un bloc b de x à la table
- Nous considérons l'interprétation de Dégagé(b) comme "il existe un espace libre sur b pour placer un bloc". Sous cette interprétation Dégagé(Table) sera toujours vrai
- Cependant rien ne peut empêcher le planificateur d'utiliser Déplacer(b, x, Table) au lieu de DéplacerSurTable(b, x)
- Exercice : Au début A, B et C sont sur la table et on veut les empiler. Modéliser le problème en STRIPS.

# STRIPS

- Init(Sur(A, Table) ^ Sur(B, Table) ^ Sur(C, Table))

- Objectif(Sur(A, B) ^ Sur(B, C))

- Action(Déplacer(b, x, y),

  PRECOND: Sur(b, x) ^ Dégagé(b) ^ Dégagé(y) ^ Bloc(b) ^

  Bloc (y) ^ (b <> x) ^ (b <> y) ^ (x <> y)

  EFFET: Sur(b, y) ^ Dégagé(x) ^ not Sur(b, x) ^

  not Dégagé(y))

# STRIPS

- Action(DéplacerSurTable(b, x),

  PRECOND: Sur(b, x) ^ Dégagé(b) ^

  Bloc(b) ^ Bloc (x) ^ (b <> x)

  EFFET: Sur(b, Table) ^ Dégagé(x) ^

  not Sur(b, x))

- Une solution possible est:

  [Déplacer(B,Table, C), Déplacer(A, Table, B)]

# Planification

Recherche dans un espace d'états

# Recherche dans un espace d'états

- L'approche la plus directe est l'utilisation de la recherche dans un espace d'états.
- Puisque les descriptions des actions dans un problème de planification spécifient les préconditions et les effets, il est possible de rechercher dans les deux directions :
- Soit chaînage avant (forward) à partir de l'état initial.
- Soit chaînage arrière (backward) à partir de l'objectif.

# Recherche dans un espace d'états

- Recherche par chaînage avant dans l'espace d'états :
- Cette approche est appelée planification progressive (progressive planning) puisqu'elle se déplace vers l'avant.

# Recherche dans un espace d'états

- Recherche par chaînage avant dans l'espace d'états :
- On commence à l'état initial du problème en considérant des séquences d'actions jusqu'à ce qu'on trouve une séquence qui satisfait un état objectif.
- La formulation des problèmes de planification comme problèmes de recherche dans l'espace d'états, est comme suit:
- L'état initial de la recherche est l'état initial du problème de planification.
- Chaque état sera un ensemble de littéraux positifs instanciés.
- Les littéraux qui n'apparaissent pas sont considérés faux.

# Recherche dans un espace d'états

- Recherche par chaînage avant dans l'espace d'états :
- Les actions qui sont applicables dans un état sont celles dont les préconditions sont satisfaites.
- L'état suivant résultant d'une action est généré en ajoutant les littéraux de l'effet positif et en supprimant les littéraux de l'effet négatif.
- Le test d'objectif (goal test) contrôle si l'état satisfait l'objectif du problème de planification.
- Le coût de l'étape (step cost) de chaque action est typiquement 1 (même s'il est possible de considérer des coûts différents).
- Exercice : Appliquer le chaînage avant au problème du fret.

# Recherche dans un espace d'états

- Recherche par chaînage arrière dans l'espace d'états :
- Cette approche est appelée planification régressive (regression planning) puisqu'elle se déplace vers l'arrière
- L'avantage principal de cette approche est qu'elle permet de considérer seulement les actions pertinentes.
- Une action est pertinente pour un objectif conjonctif si elle accomplit un des conjoints de l'objectif
- Il faut noter que des actions non pertinentes peuvent aussi conduire à l'état objectif.
- Une recherche par chaînage arrière qui permet des actions non pertinentes sera toujours complète mais elle sera moins efficace.

# Recherche dans un espace d'états

- Recherche par chaînage arrière dans l'espace d'états :
- La question principale en planification régressive est: quels sont les états à partir desquels l'application d'une action donnée peut conduire à l'objectif
- Calculer la description de ces états est appelée régresser l'objectif par cette action
- Considérons l'exemple du frêt aérien et soit l'objectif:
- A(F1,B) ^ A(F2, B) ^ A(F3, B) ...A(F20, B) et l'action pertinente Décharger(F1, p, B) qui réalise le premier conjoint
- L'action va être exécutée seulement si ses préconditions sont satisfaites. Par conséquent chaque état prédécesseur doit inclure ces préconditions: Dans(F1, p) ^ A(p, B)
- Le sous but A(F1, B) ne doit pas être vrai à l'état prédécesseur. La description du prédécesseur sera donc: Dans(F1, p) ^ A(p, B) ^ A(F2, B) ^ A(F3, B) ...A(F20, B)

# Recherche dans un espace d'états

- Recherche par chaînage arrière dans l'espace d'états :

  Les actions qui accomplissent les littéraux désirés, ne doivent pas détruire d'autres littéraux désirés.
- Une action qui satisfait cette restriction est appelée cohérente (consistent).
- L'action Charger(F2, P) ne serait pas cohérente puisque elle produirait la négation du littéral A(F2, B)
- Etant données les définitions de pertinence et cohérence nous pouvons décrire le processus général de construction de prédécesseurs pour une recherche en chaînage arrière.

# Recherche dans un espace d'états

- Recherche par chaînage arrière dans l'espace d'états :
- Etant donnée une description d'objectif G, soit A une action pertinente et cohérente. Le prédécesseur correspondant est comme suit:
- Chacun des effets positifs de A apparaissant dans G est détruit.
- Chaque littéral de précondition de A est ajouté à moins qu'il apparaisse déjà.
- Tous les algorithmes classiques de recherche peuvent être utilisés pour effectuer la recherche.
- La fin est atteinte quand est générée une description de prédécesseur qui est satisfaite par l'état initial du problème de planification.

# Recherche dans un espace d'états

- Recherche par chaînage arrière dans l'espace d'états :
- Le chaînage arrière à un facteur de branchement en général plus petit que le chaînage avant.
- Dans l'exemple du frêt aérien, il y a 1000 actions possibles en chaînage avant et seulement 20 en chaînage arrière.
- Exercice : Appliquer le chaînage arrière au problème du frêt avec deux avions.

# Recherche dans un espace d'états

- Recherche en profondeur d'abord :
- On dispose d'un entier (N = 2) et de 2 règles pour modifier cet entier :
- Soustraire 3.
- Ajouter 5.
- On cherche à atteindre la valeur : 0.
- Pour simplifier, on suppose que l'on applique toujours la soustraction avant l'ajout.

# Recherche dans un espace d'états

- Recherche en profondeur d'abord :
- On dispose d'un entier (N = 2) et de 2 règles pour modifier cet entier :
- Soustraire 3.
- Ajouter 5.
- On cherche à atteindre la valeur : 0.
- Pour simplifier, on suppose que l'on applique toujours la soustraction avant l'ajout.
- 2 -> -1 -> -4 -> -7 etc.

# Recherche dans un espace d'états

- Parcours en largeur :
- Se termine toujours (si une solution existe) mais la taille de l'arbre construit est grande par rapport à la difficulté du problème.
- Trouve la solution la plus courte.
- Appliquer la recherche en largeur au problème de soustraire 3 ou ajouter 5, en partant de 2 avec pour objectif 0.

# Recherche dans un espace d'états

- Parcours en largeur :
- Se termine toujours (si une solution existe) mais la taille de l'arbre construit est grande par rapport à la difficulté du problème.
- 2 -> (-1  7) -> (-4  4  4  12) etc.

# Recherche dans un espace d'états

- Stratégie naïve :
- Choisir de développer l’état le plus proche du but.
- Appliquer cette stratégie au problème de soustraire 3 ou ajouter 5.

# Recherche dans un espace d'états

- Stratégie naïve :
- Choisir de développer l’état le plus proche du but.
- Ici, elle conserve la propriété de complétude et diminue fortement la complexité :
- 2 -> (-1  7) ->  (-4  4) ->  (-7  1) -> (-2  6) -> (-5  3) -> 0

# Recherche dans un espace d'états

- Heuristiques pour la Recherche dans l'espace d'états :
- Exercice : Trouver des heuristiques.

# Recherche dans un espace d'états

- Heuristiques pour la Recherche dans l'espace d'états :
- Exercice : Trouver des heuristiques.
- Heuristique simple = nombre de buts non satisfaits.

# Recherche dans un espace d'états

- Heuristiques pour la Recherche dans l'espace d'états :
- Exercice : Trouver des heuristiques.
- Heuristique simple = nombre de buts non satisfaits.
- Heuristique améliorée = nombre  minimal d'actions positives pour atteindre le but.
- Hypothèses = pas d'interactions entre actions, pas d'autres préconditions à vérifier, pas de retrait des but déjà atteints.

# Recherche dans un espace d'états

- Heuristiques admissibles :

  Heuristique optimiste qui ne sous estime le nombre d'actions à effectuer avant d'atteindre le but.
- Permet de trouver efficacement des solutions optimales grâce à l'algorithme A*.

# Recherche dans un espace d'états

- Algorithme A* :
- Pour chaque état visité on a deux valeurs : g et h.
- g = nombre d'actions effectuées pour arriver à l'état.
- h = heuristique admissible = nombre minimal d'actions pour atteindre le but.
- f = g + h

# Recherche dans un espace d'états

- Algorithme A* :
- On a une liste d'ouverts et une liste de fermés.
- On développe l'ouvert qui a le plus petit f.
- On le met dans les fermés et on met ses successeurs dans les ouverts.
- On itère tant que le plus petit f est inférieur au f de l'état objectif.

# Recherche dans un espace d'états

- Exercice :
- Appliquer A* au problème du fret avec deux avions et l'heuristique du nombre minimal d'actions positives pour atteindre le but.
- Commencer par le chaînage avant.
- Puis faire le chaînage arrière.

# Recherche dans un espace d'états

- Exercice :
- Appliquer A* au problème des cubes avec la même heuristique.
- Commencer par le chaînage avant.
- Puis effectuer le chaînage arrière.

# Recherche dans un espace d'états

- Critères d'évaluation :
- Consistance (tout plan produit est correct).
- Cette propriété n'est pas indispensable si pour la plupart des problèmes, le planificateur produit un plan correct.

# Recherche dans un espace d'états

- Critères d'évaluation :
- Complétude (si problème a au moins 1 solution alors le planificateur produit 1 plan)
- Cette propriété n'a de sens que si le planificateur est consistent.

# Recherche dans un espace d'états

- Critères d'évaluation :
- Terminaison
- Soit le problème a au moins une solution et le planificateur produit un plan.
- Soit le problème n'a pas de solution et le planificateur s'arrête au bout d'un temps fini.
- Cette propriété implique la complétude du planificateur.

# Recherche dans un espace d'états

- Critères d'évaluation :
- La complexité (en temps de résolution):
- Cette propriété se mesure difficilement dans toute sa généralité mais prime sur les autres.

# Planification

Graphplan

# Graphplan

- Un graphe de planification donne de meilleures estimations heuristiques que les approches précédentes.
- De plus, une solution peut être extraite directement du graphe de planification en utilisant l'algorithme Graphplan
- Un graphe de planification correspond à une séquence de niveaux qui correspondent aux pas de temps du plan.
- Chaque niveau contient une ensemble de littéraux et un ensemble d'actions. Les littéraux sont ceux qui pourraient être vrais, et les actions celles qui pourraient être appliquées.

# Graphplan

- Exemple :

  Init(Avoir(Gateau))

  Objectif(Avoir(Gateau) ^ Mangé(Gateau))

  Action(Manger(Gateau)

  PRECOND: Avoir(Gateau)

  EFFET: not Avoir(Gateau) ^ Mangé(Gateau))

  Action(Cuisiner(Gateau)

  PRECOND: not Avoir(Gateau)

  EFFET: Avoir(Gateau))

# Graphplan

- Une action persistente est représentée dans le graphe par un carré qui signifie que le littéral n'a pas changé.
- Les liens d'exclusion mutuelle relient les littéraux qui ne peuvent pas être vrais ensemble.
- On continue à développer le graphe tant que le dernier niveau ne se répète pas.
- La complexité de la construction du graphe est polynomiale alors que l'espace de recherche est exponentiel en fonction du nombre de littéraux.

# Graphplan

- Un mutex existe entre deux actions si une de ces trois conditions est vérifiée :
- Effets inconsistants : une action nie l'effet d'une autre. Par exemple Manger(Gateau) et Avoir(Gateau) sont en désaccord sur l'effet Avoir(Gateau).
- Interference : un effet d'une action est la negation de la précondition d'une autre. Par exemple Manger(Gateau) interfere avec la persistence de Avoir(Gateau).
- Compétition de besoins : une préconditions d'une action est mutuellement exclusive avec la précondition d'une autre. Par exemple Cuisiner(Gateau) et Manger(Gateau) sont mutex parce qu'elle sont en compétition pour Avoir(Gateau)

# Graphplan

- Un mutex existe entre deux littéraux au même niveau si l'un est la négation de l'autre ou si chaque paire possible d'actions qui pourrait atteindre les deux littéraux est mutuellement exclusive (cette condition est appelée support inconsistant).
- Par exemple Avoir(Gateau) et Mangé(Gateau) sont en mutex dans S1 car la seule façon d'atteindre Avoir(Gateau) et en mutex avec la seule façon d'atteindre Mangé(Gateau).

# Graphplan

- Exercice :
- Ecrire le graphe de planification pour le problème du gateau.

# Graphplan

- Exercice :
- Ecrire le graphe de planification pour le problème du changement du pneu crevé.

# Graphplan

- Heuristiques utilisant le graphe de planification :
- Le graphe de planification donne beaucoup d'information sur le problème.
- Par exemple un littéral qui n'apparait pas dans le niveau final du graphe ne peut être atteint par aucun plan.
- Cette information peut être utilisée en recherche arrière pour affecter un cout infini aux états contenant un littéral non atteignable.

# Graphplan

- D'une manière plus générale, on peut estimer le cout d'arriver à un but littéral par son niveau de première apparition dans le graphe de planification. C'est son cout de niveau.
- Une heuristique admissible pour évaluer la difficulté d'un état est de prendre le maximum des niveaux des buts.
- Une heuristique non admissible mais qui marche beaucoup mieux est de prendre la somme des niveaux des buts (ce qui correspond à estimer que les buts sont indépendants).

# Graphplan

- Exercice :
- Développer un arbre de recherche en recherche arrière avec A* pour le problème du frêt avec deux avions en utilisant comme heuristique la somme des niveaux des buts.

# Graphplan

- Exercice :
- Développer un arbre de recherche avec A* pour le problème des cubes en utilisant comme heuristique la somme des niveaux des buts.

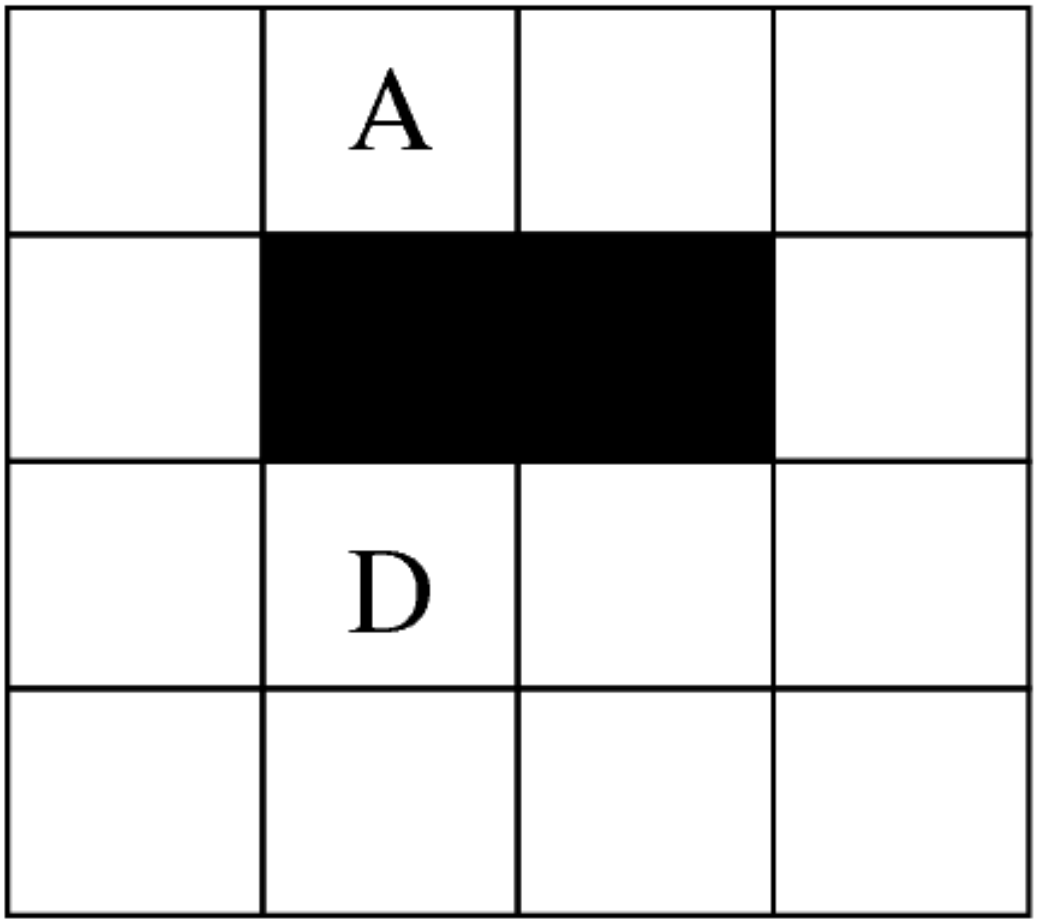
# Graphplan

- Exercice :
- Développer un arbre de recherche avec A* pour le problème du frêt avec deux avions en utilisant comme heuristique la somme des niveaux des buts.

# Recherche heuristique

Recherche de solutions optimales avec A*

# Le plus court chemin



# Le plus court chemin

- Utilisé tel quel dans de nombreux jeux vidéo, par les GPS et en robotique.
- Les algorithmes utilisés pour les plus court chemins ont des applications dans de nombreux domaines :

  Bioinformatique.

  Linguistique.

  Jeux.

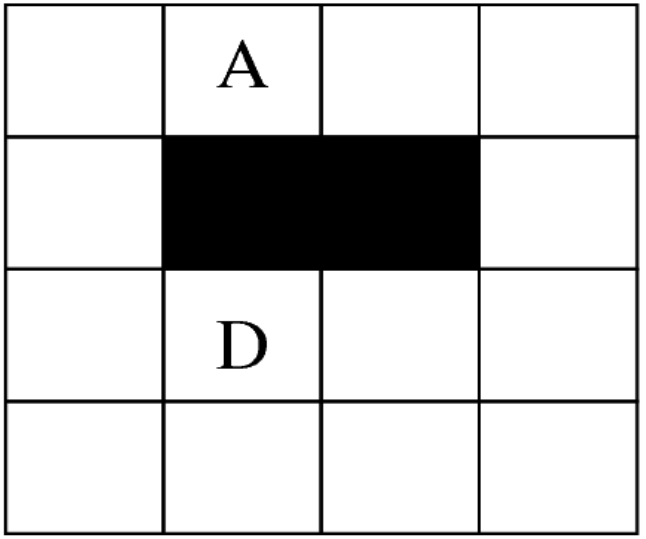
  Planification.

# Dijkstra

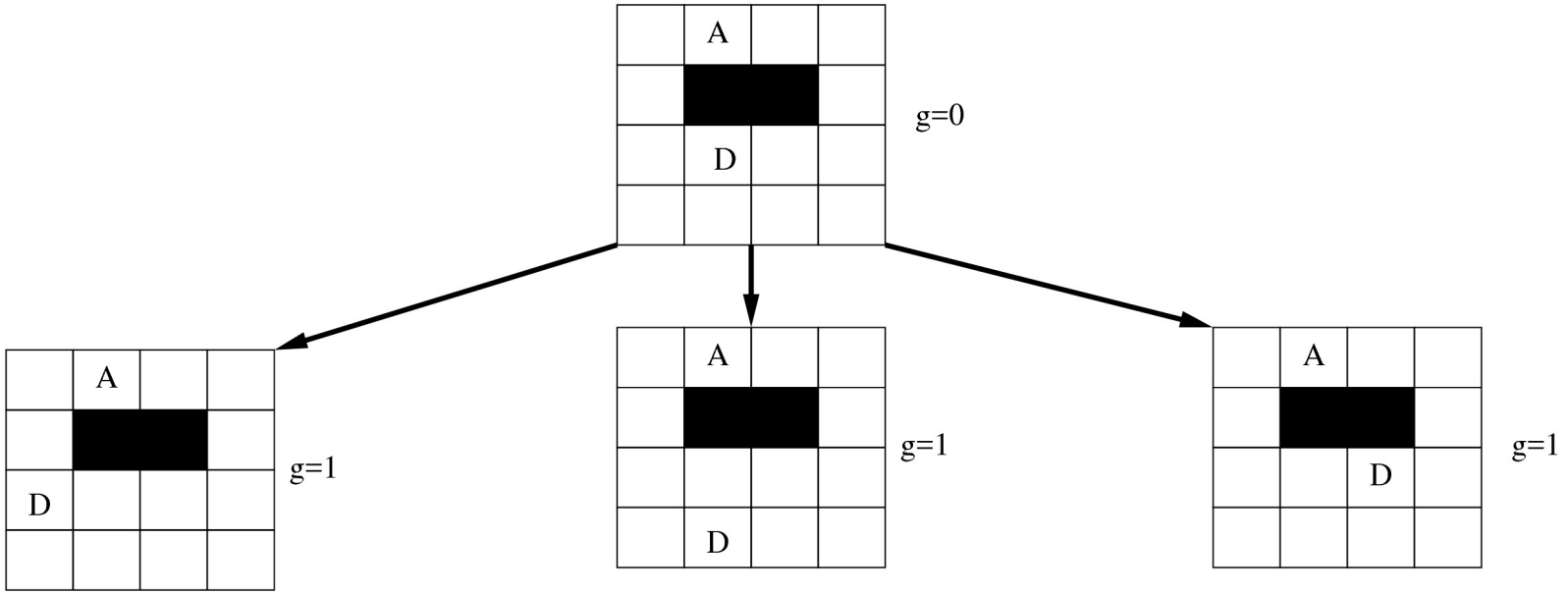
- Développe un arbre avec une position par feuille.
- Principe : développer la feuille qui a le plus petit chemin parcouru.
- Trouve le chemin le plus court en temps linéaire pour les cartes.

# Dijkstra

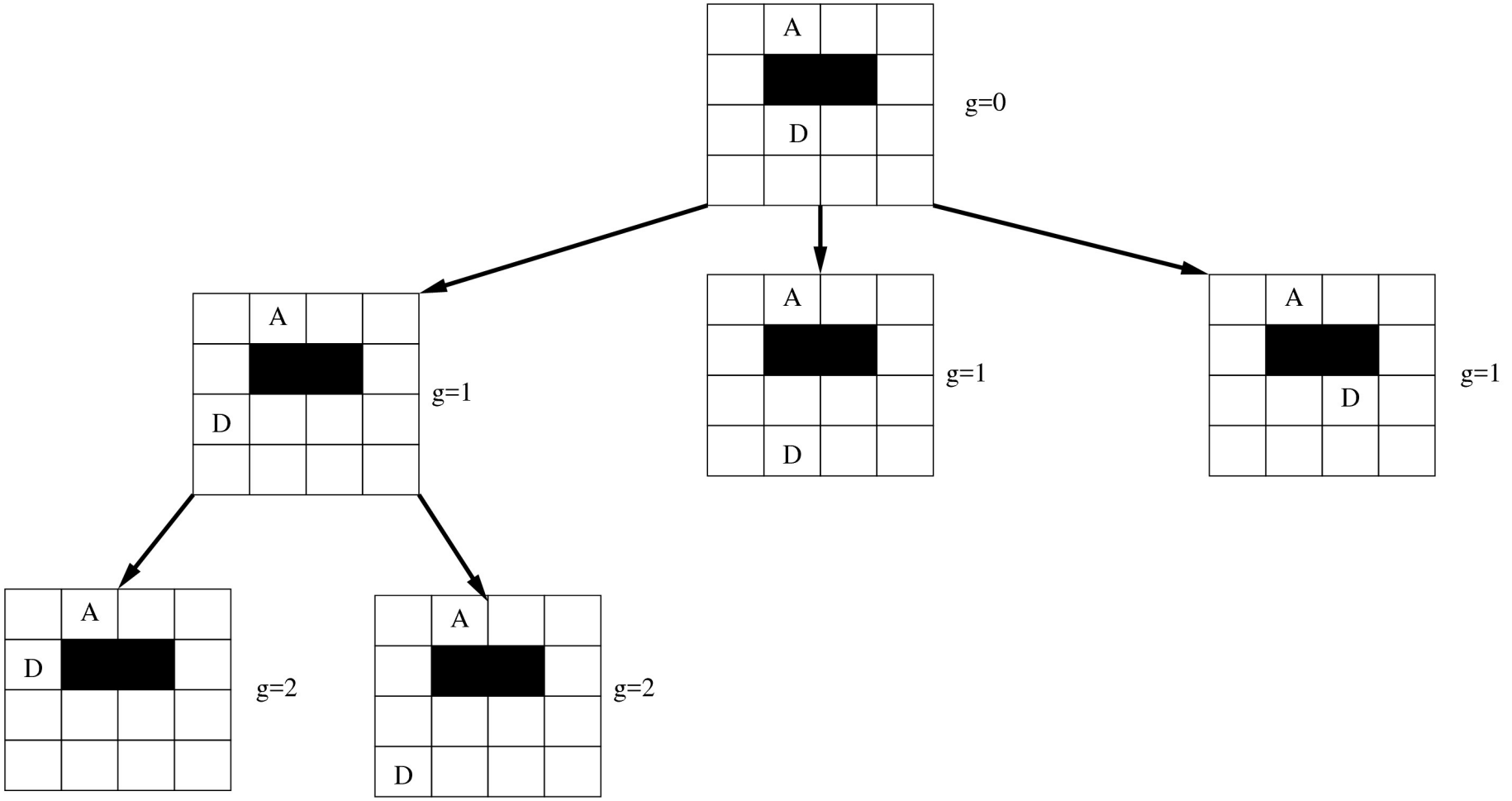
- Exercice
- Appliquer Dijkstra au problème suivant :



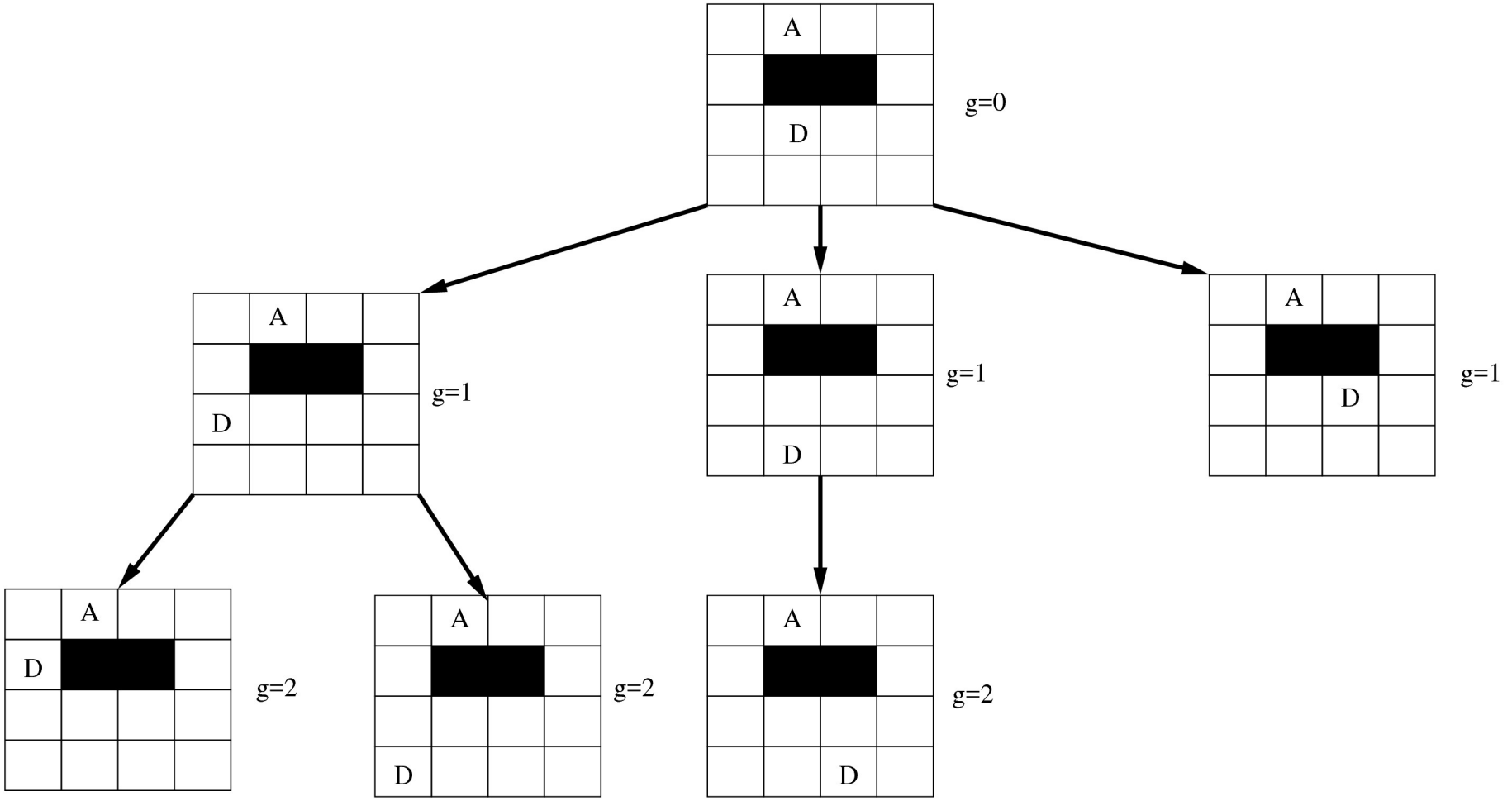
# Dijkstra



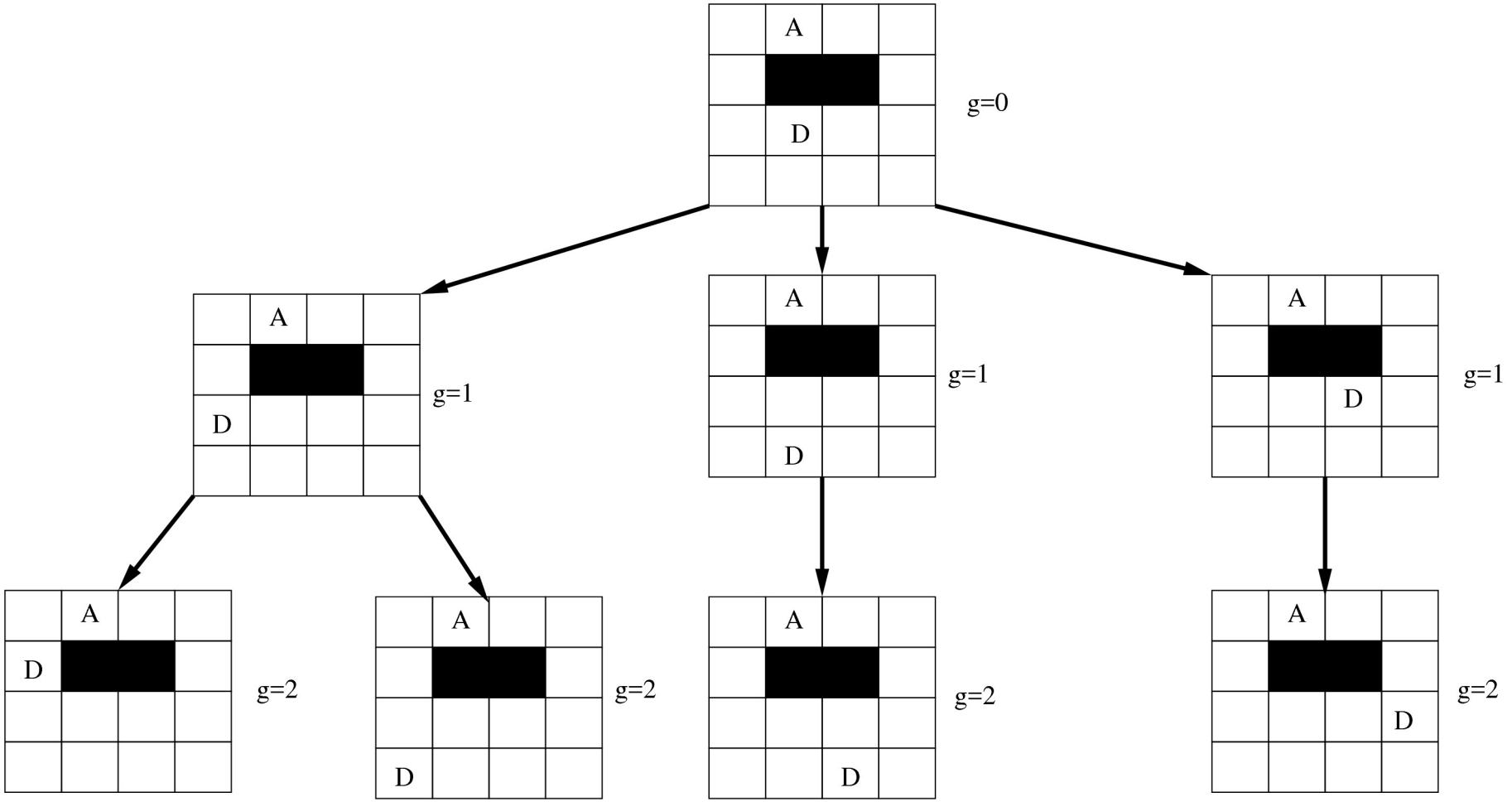
# Dijkstra



# Dijkstra



# Dijkstra



# A*

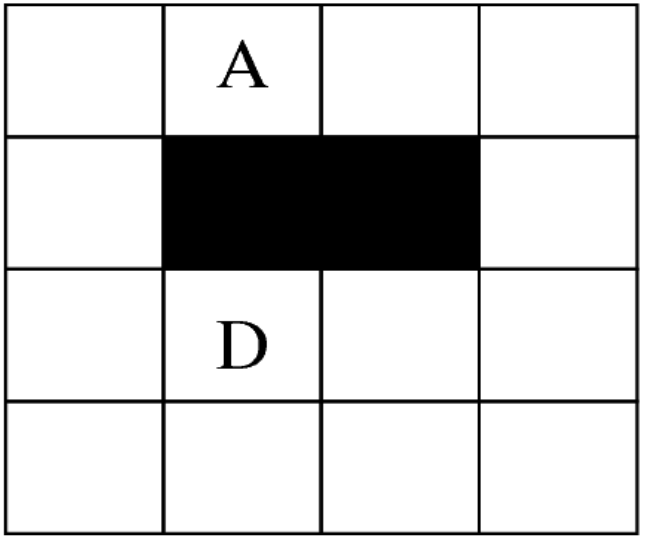
- Développe un arbre avec une position par feuille.
- Principe : développer la feuille qui a le plus petit f = g + h.
- h = heuristique admissible
- Distance de manhattan.
- Trouve le chemin le plus court en temps linéaire mais en moins de feuilles développées que Dijkstra.

# A*

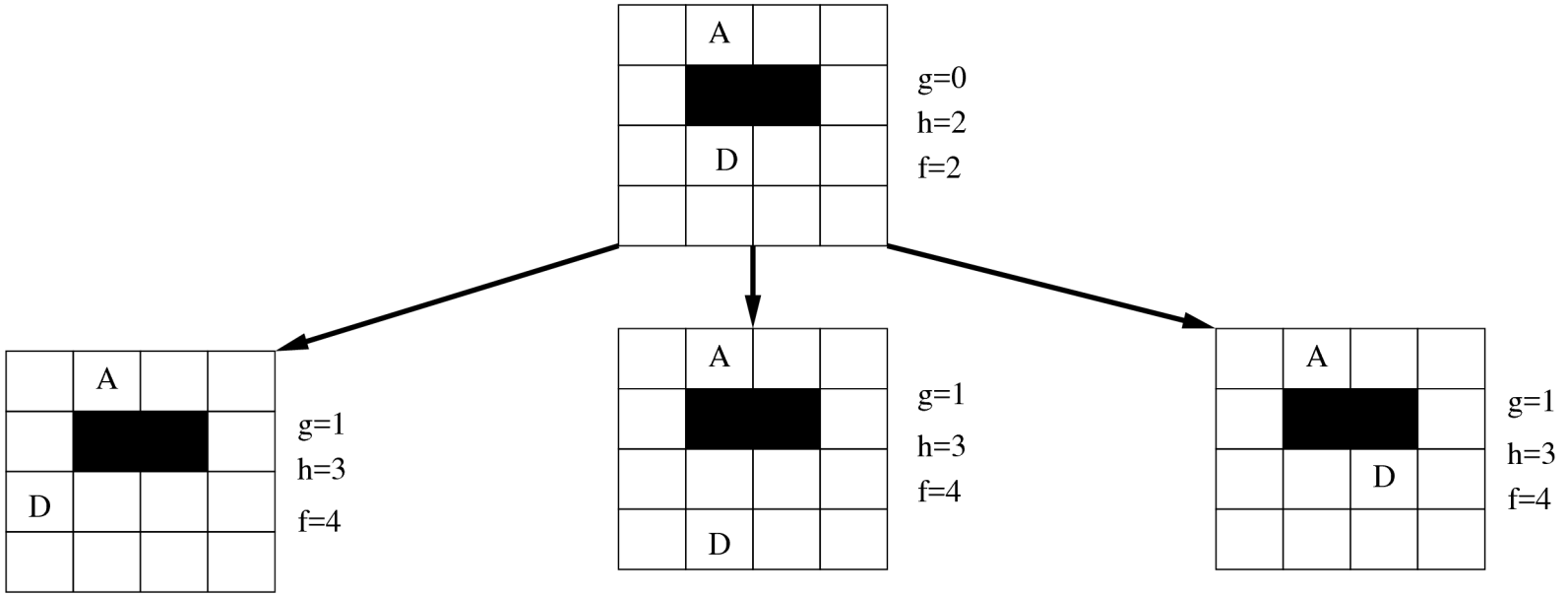
- Quand on veut appliquer A* il faut déterminer une heuristique admissible qui donne toujours une valeur plus petite que le nombre de coups à jouer pour atteindre la solution.
- L'heuristique admissible la plus courante est l'heuristique de Manhattan.
- Elle consiste à considérer que tous les mouvements sont possibles sans prendre en compte les obstacles.

# A*

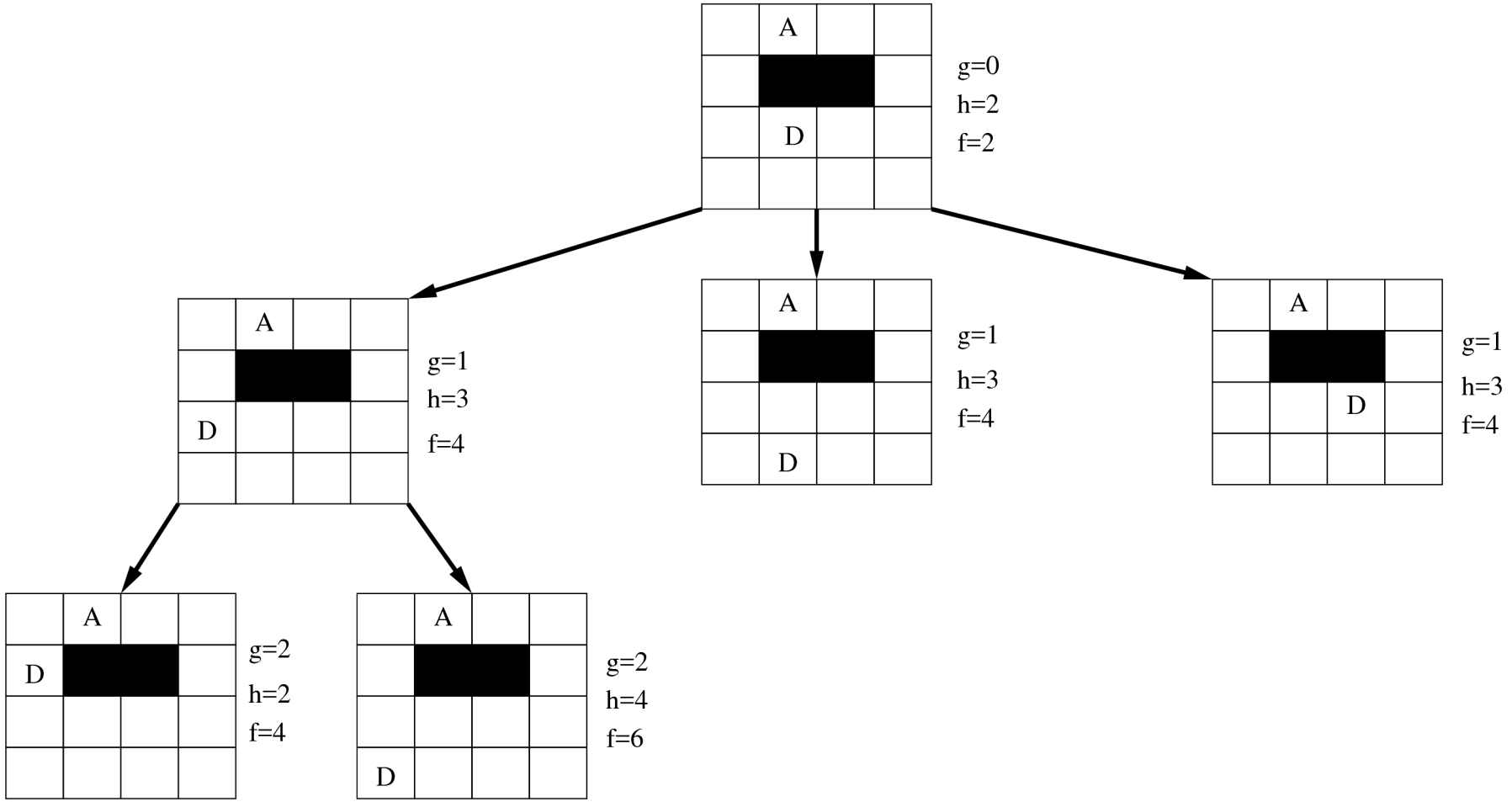
- Exercice
- Appliquer A* au problème suivant :



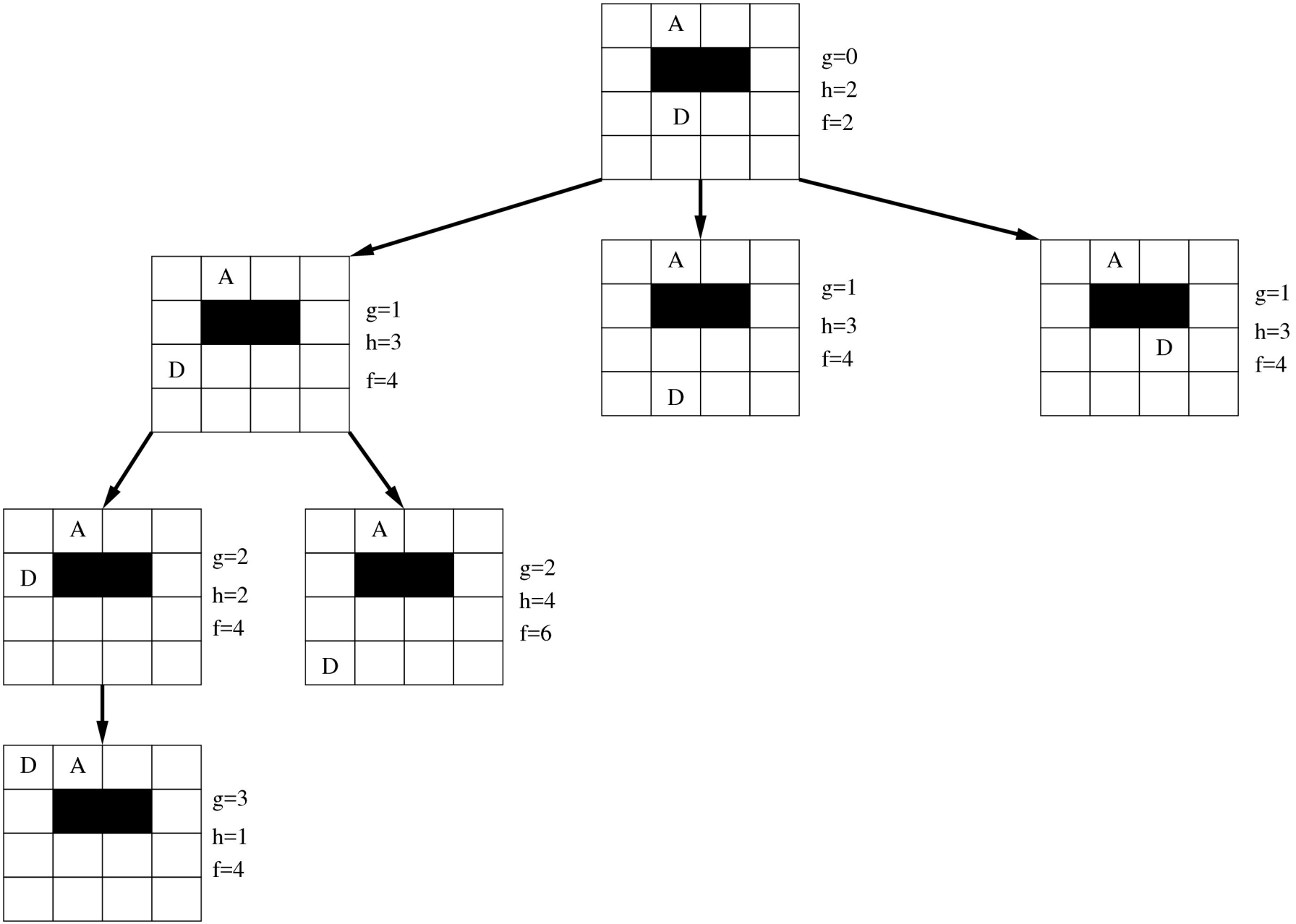
# A*



# A*



# A*


A
D
g=0
h=2
f=2
A
D
g=1
h=3
f=4
A
D
g=1
h=3
f=4
A
D
g=1
h=3
f=4
A
D
g=2
h=2
f=4
A
D
g=2
h=4
f=6
D
A
g=3
h=1
f=4

# A*

- Exercices :
- Trouver une heuristique admissible lorsque les déplacements diagonaux sont autorisés.
- Montrer que A* est une généralisation de Dijkstra.

# A*

- L'heuristique triangulaire
- ALT est une bonne heuristique pour les cartes routières.
- Elle consiste à pré calculer les distances d'un point donné à tous les autres points.
- Ces distances pré calculées peuvent alors être utilisées pour calculer une heuristique admissible.
- Exercice : Trouver une heuristique admissible à partir des distances pré calculées depuis un point P.

# A*

- L'heuristique utilise l'inégalité triangulaire.
- Si par exemple |X,Y| est la longueur du plus court chemin entre X et Y, et si les distances sont pré calculées depuis P, que le noeud courant est en D, et que le but à atteindre est en A, on a les inégalités suivantes :

  $|P,A| \leq |P,D| + |D,A|$

  $|D,P| \leq |D,A| + |A,P|$

# A*

- Avec les deux équations on obtient :

  |D,A| >= | |D,P|  - |A,P| |
- D'où une heuristique admissible qui utilise les distances pré calculées :

  h = | |D,P|  - |A,P| |
- Note : les distances pré calculées peuvent être trouvées avec Dijkstra.

# A*

- Un problème typique de recherche de plus court chemin dans un graphe d'états est le Taquin.
- Le Taquin est un jeu solitaire en forme de damier créé vers 1870 aux États-Unis (Wikipédia) :

# A*

- Un coup au Taquin consiste à déplacer une fiche dans l'emplacement vide :

```
5 6 1          5 6 1
  2 8   =>     2   8
3 4 7          3 4 7
```

# A*

- Dans sa version 3x3, le but est de trouver une séquence minimale de coups qui permet d'atteindre la position suivante :

  1 2 3

  4  5

  6 7 8

# A*

- Dans sa version 4x4, on cherche à atteindre la position suivante :

# A*

- Quelle valeur donne l'heuristique de Manhattan pour le Taquin suivant :

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 13 | 2 | 11 | 10 | | | 1 | 2 | 3 | 4 |
| 8 | 6 | 12 | 14 | | => | 5 | 6 | 7 | 8 |
| 7 | 5 | 3 | 15 | | | 9 | 10 | 11 | 12 |
| 1 | 4 | 9 | | | | 13 | 14 | 15 | |

# A*

- Quelle valeur donne l'heuristique de Manhattan pour le Taquin suivant :

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 13 | 2 | 11 | 10 | | 1 | 2 | 3 | 4 |
| 8 | 6 | 12 | 14 | => | 5 | 6 | 7 | 8 |
| 7 | 5 | 3 | 15 | | 9 | 10 | 11 | 12 |
| 1 | 4 | 9 | | | 13 | 14 | 15 | |

- h = 3 + 0 + 2 + 4 + 3 + 0 + 2 + 4 + 3 + 2 + 2 + 2 + 3 + 5 +3 = 38

# IDA*

- L'inconvénient principal de A* est qu'il nécessite de grandes quantités de mémoire.
- Comme l'information minimale qu'on doit garder en mémoire est la liste des ouverts, A* dépasse les capacités mémoires des machines actuelles pour certains problèmes difficiles.
- L'algorithme IDA* permet de résoudre le problème de mémoire de A* tout en gardant l'optimalité de la solution.
- IDA* est un acronyme pour Iterative Deepening A*.

# IDA*

- Chaque itération de l'algorithme est une recherche en profondeur d'abord qui calcule dynamiquement

  f(Position)=g(Position)+h(Position)

  pour chaque noeud développé.
- Dès que la fonction f d'un noeud excède le seuil propre à l'itération, le noeud est coupé, et la recherche continue sur les autres chemins non coupés.

# IDA*

- IDA* est approprié pour les problèmes qui ont de grands espaces d'états et de grands facteurs de branchement.
- Le seuil est initialisé avec l'évaluation heuristique h de l'état initial.
- A chaque itération, le seuil est incrémenté.
- L'algorithme se termine lorsque un état final est atteint.

# IDA*

- Exercice :
- Ecrire IDA* en pseudo-code.

  (c'est plus simple que A*)

# IDA*

- IDA* est simple à programmer :

```
IDA (int g)
  if (g + h > seuil)
    return false
  if (solved)
    return true
  for (move in possible moves)
    play (move)
    if (IDA (g + 1))
      return true;
    undo (move)
  return false
```