Modélisation et Résolution de Problèmes

Tristan Cazenave
LAMSADE
Université Paris Dauphine - PSL

Tristan.Cazenave@dauphine.fr

Recherche en meilleur d’abord

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.

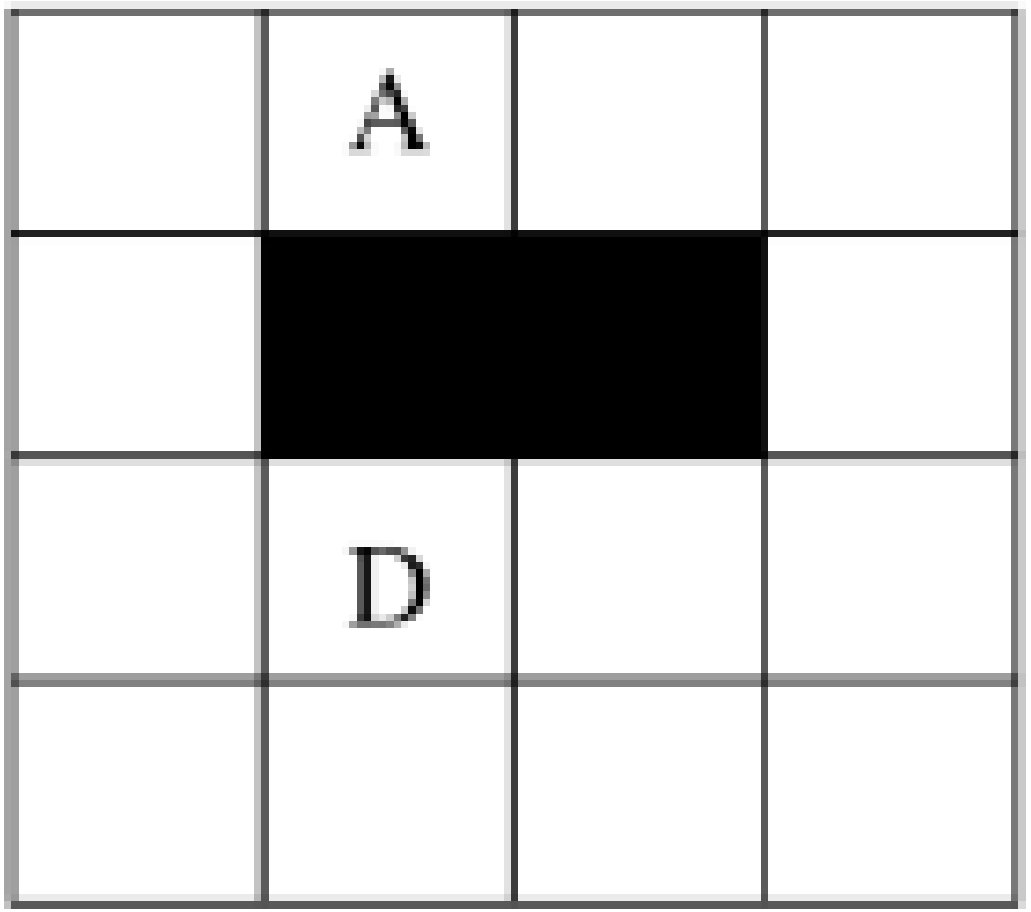
Recherche dans un espace d'états

- Greedy Best First Search :
- 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

- Greedy Best First Search :
- 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

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.

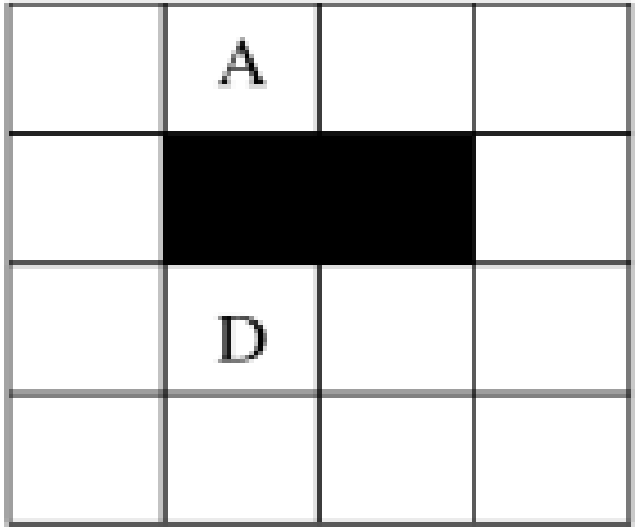
  Planification.

Dijkstra

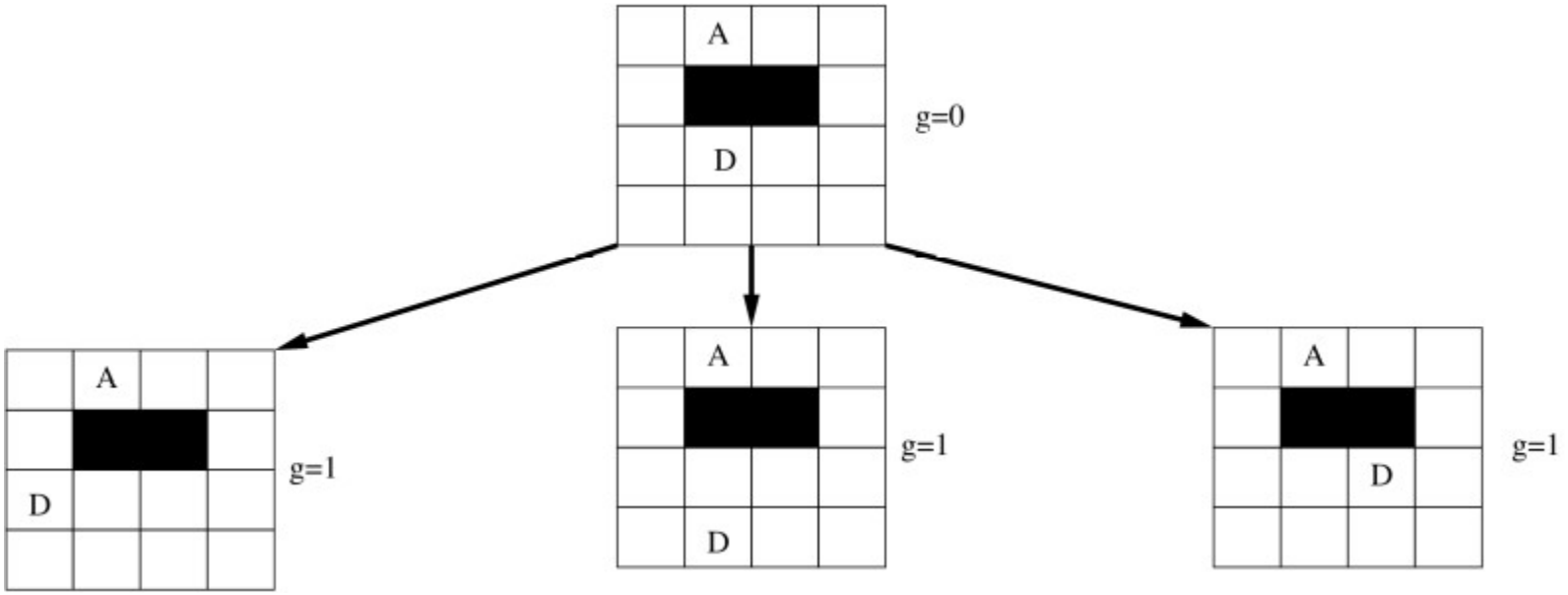
- Développe un arbre avec une position par feuille.
- Principe : développer la feuille qui a le plus petit chemin parcouru.
- Trouve le chemin le plus court en temps linéaire pour les cartes.

Dijkstra

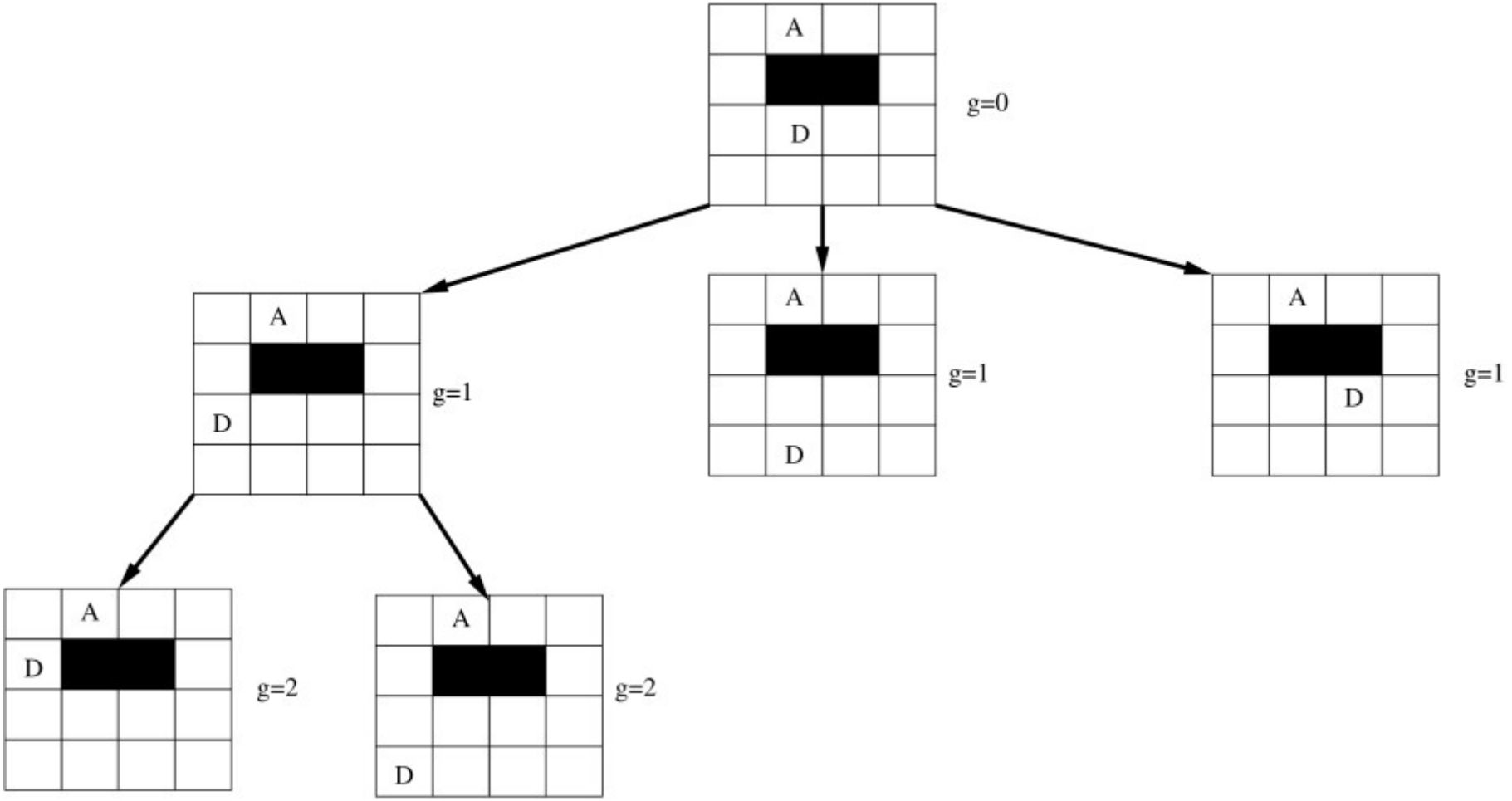
- Exercice
- Appliquer Dijkstra au problème suivant :



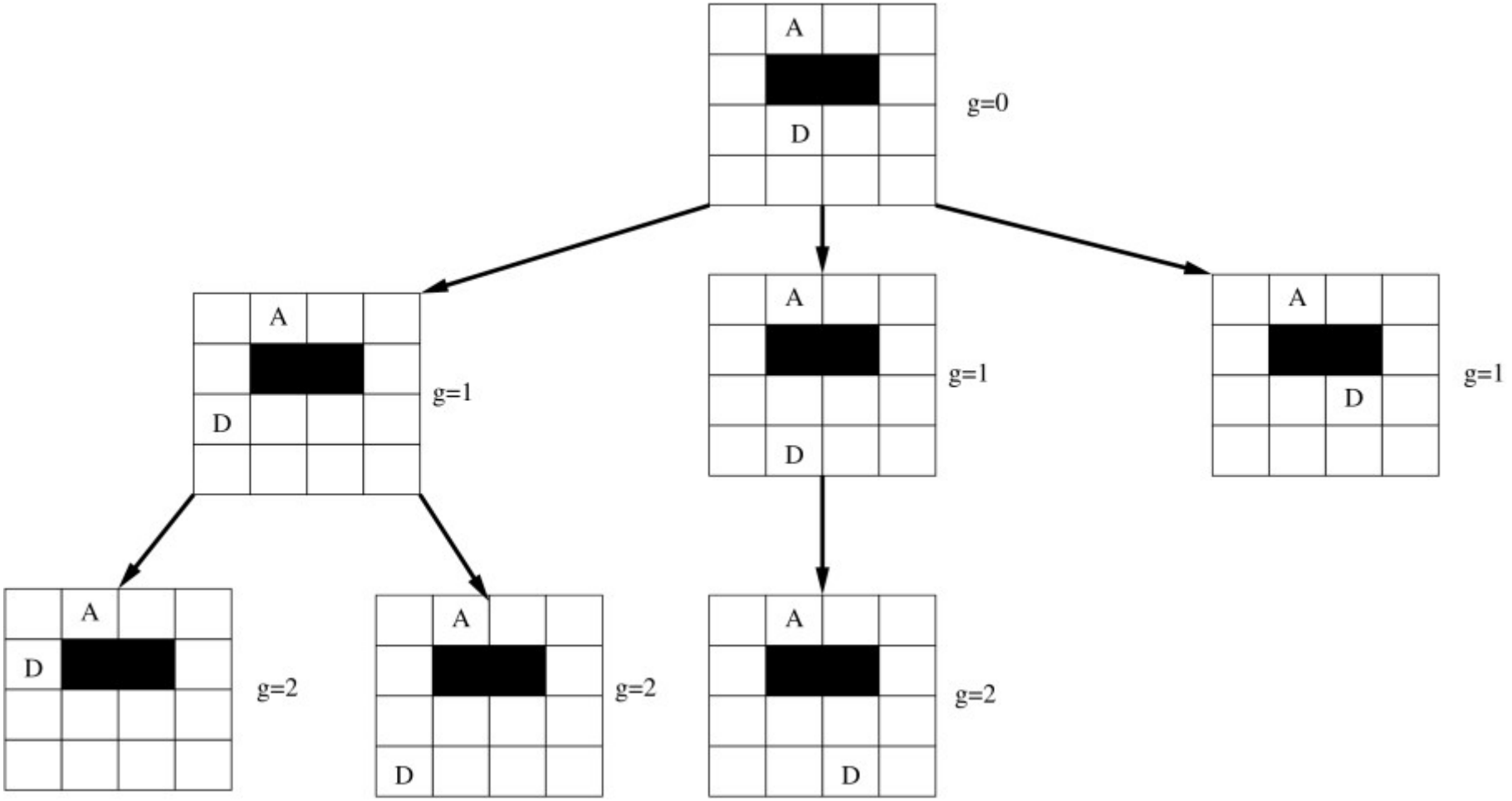
Dijkstra



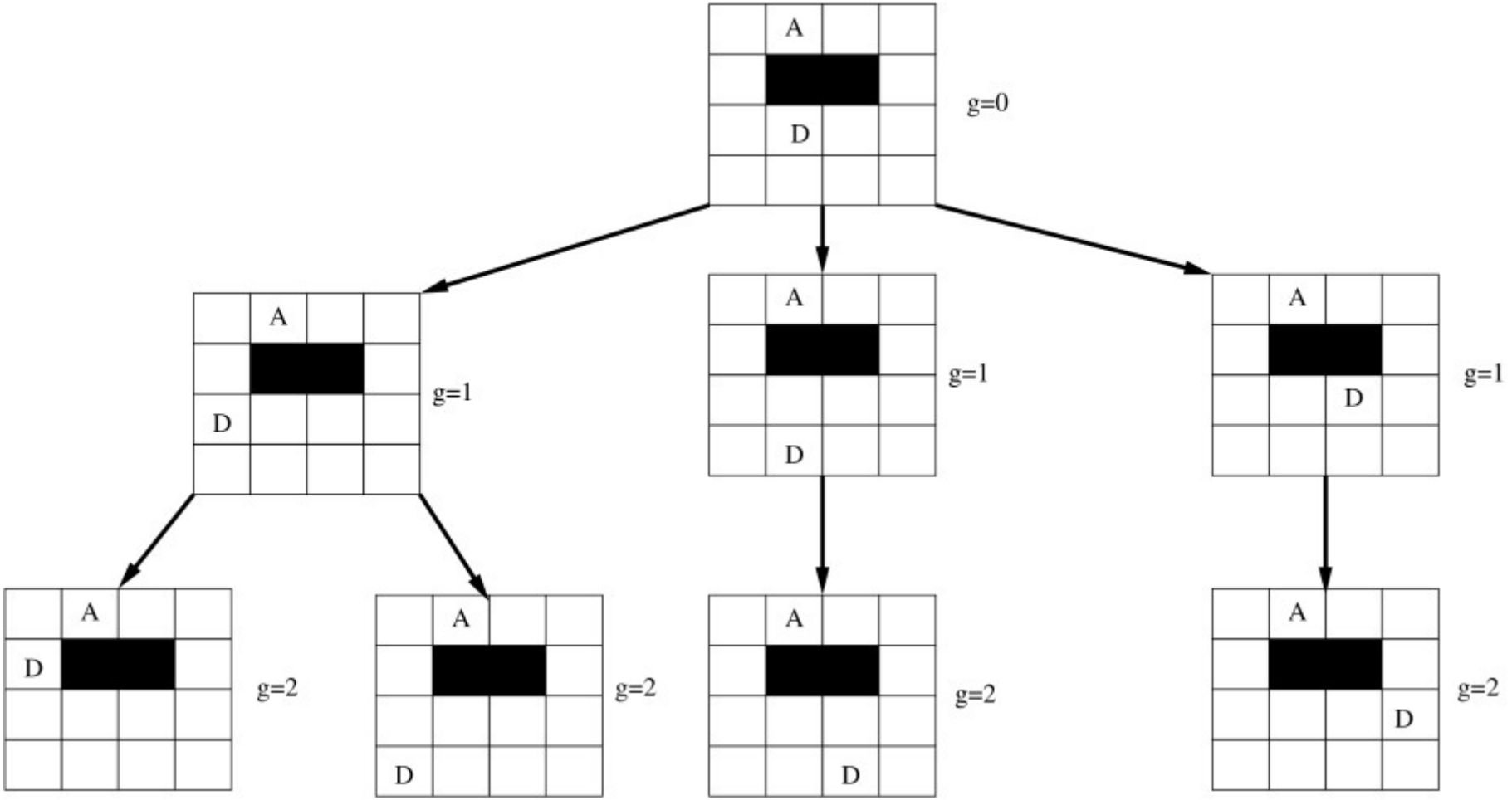
Dijkstra



Dijkstra



Dijkstra



A*

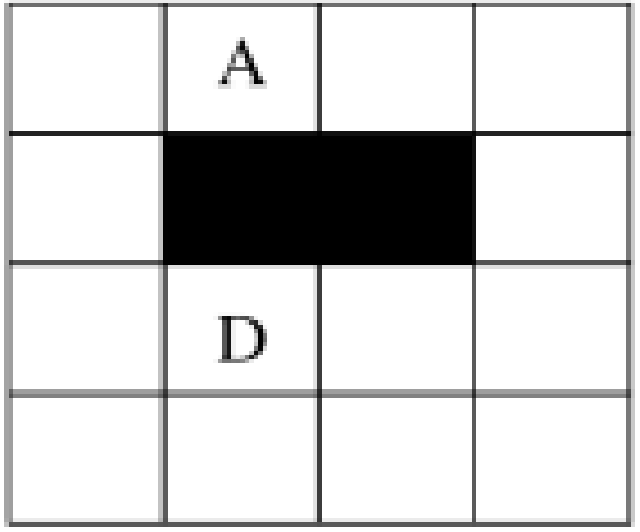
- Développe un arbre avec une position par feuille.
- Principe : développer la feuille qui a le plus petit f = g + h.
- h = heuristique admissible
- Distance de manhattan.
- Trouve le chemin le plus court en temps linéaire mais en moins de feuilles développées que Dijkstra.

A*

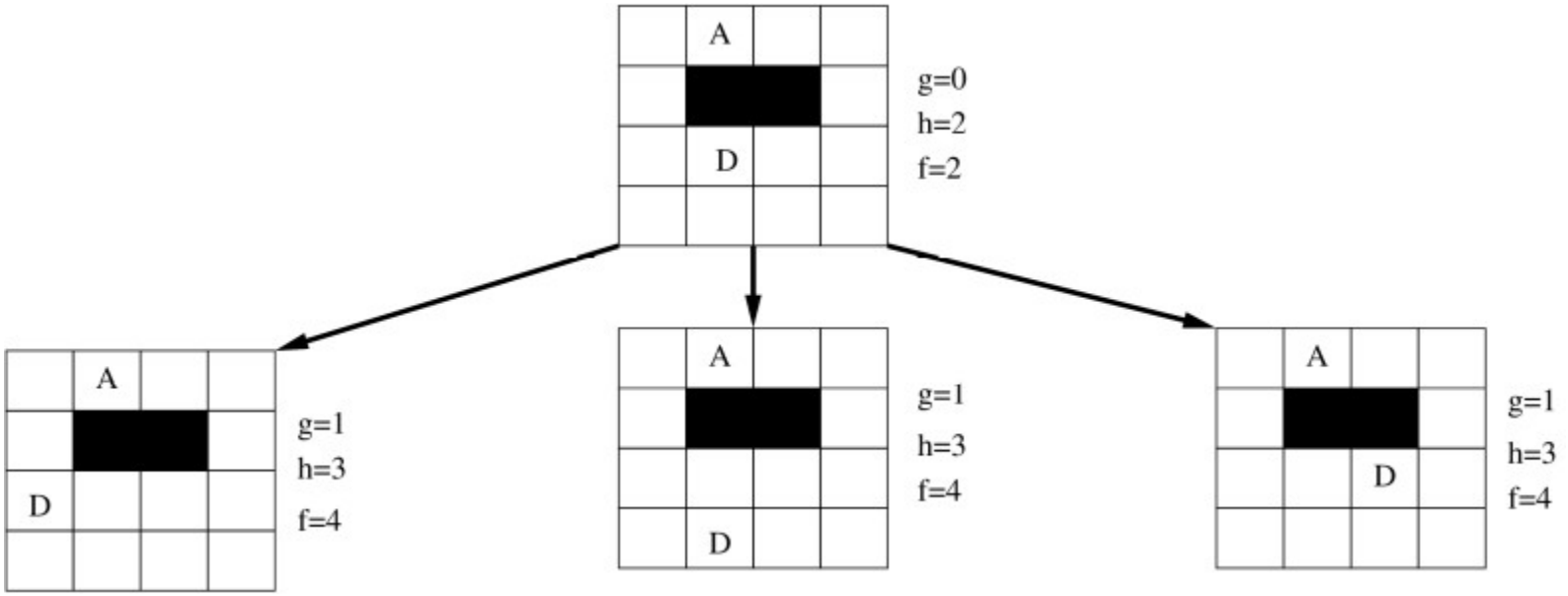
- Quand on veut appliquer A* il faut déterminer une heuristique admissible qui donne toujours une valeur plus petite que le nombre de coups à jouer pour atteindre la solution.
- L'heuristique admissible la plus courante est l'heuristique de Manhattan.
- Elle consiste à considérer que tous les mouvements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

A*

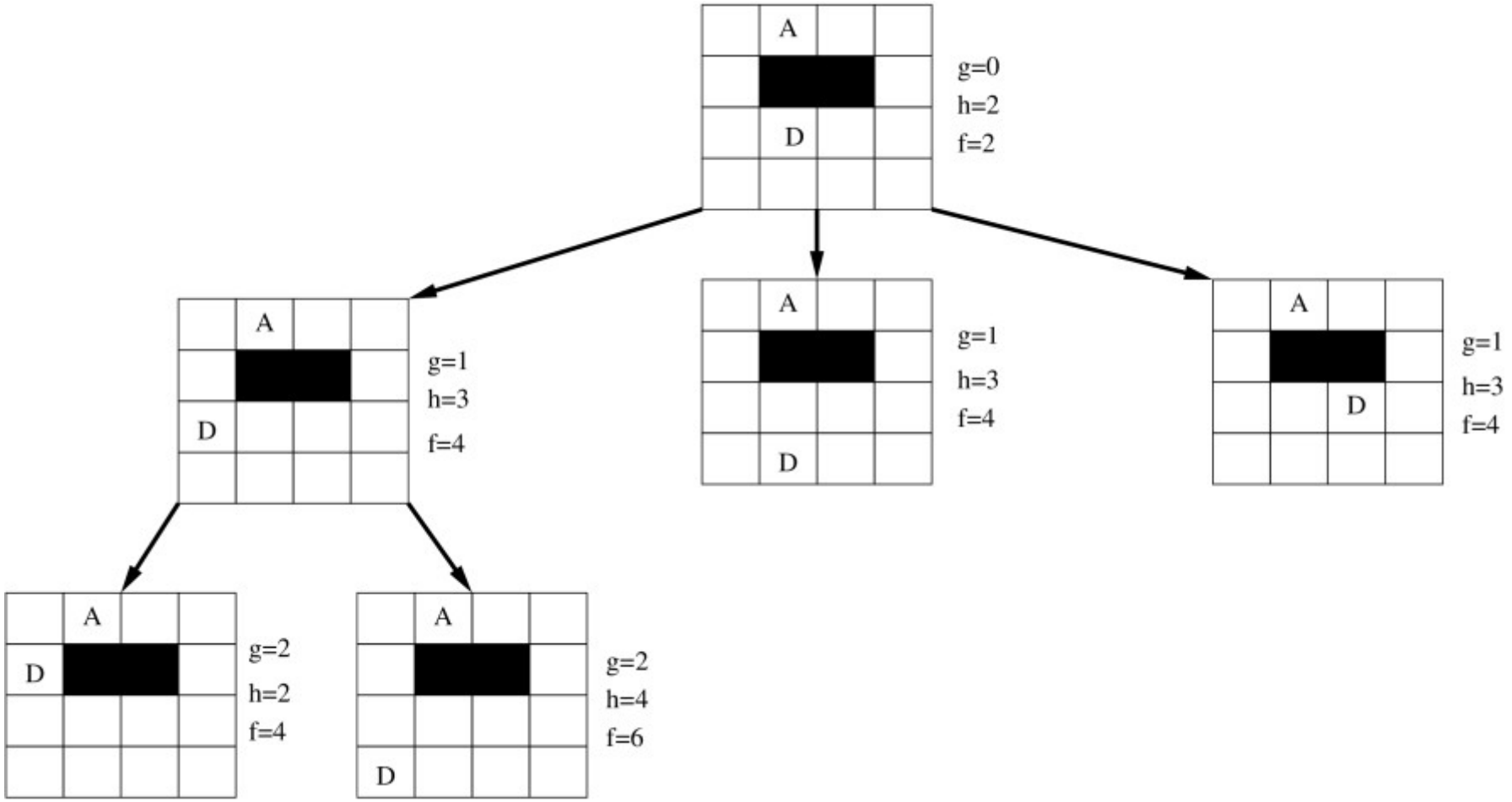
- Exercice
- Appliquer A* au problème suivant :



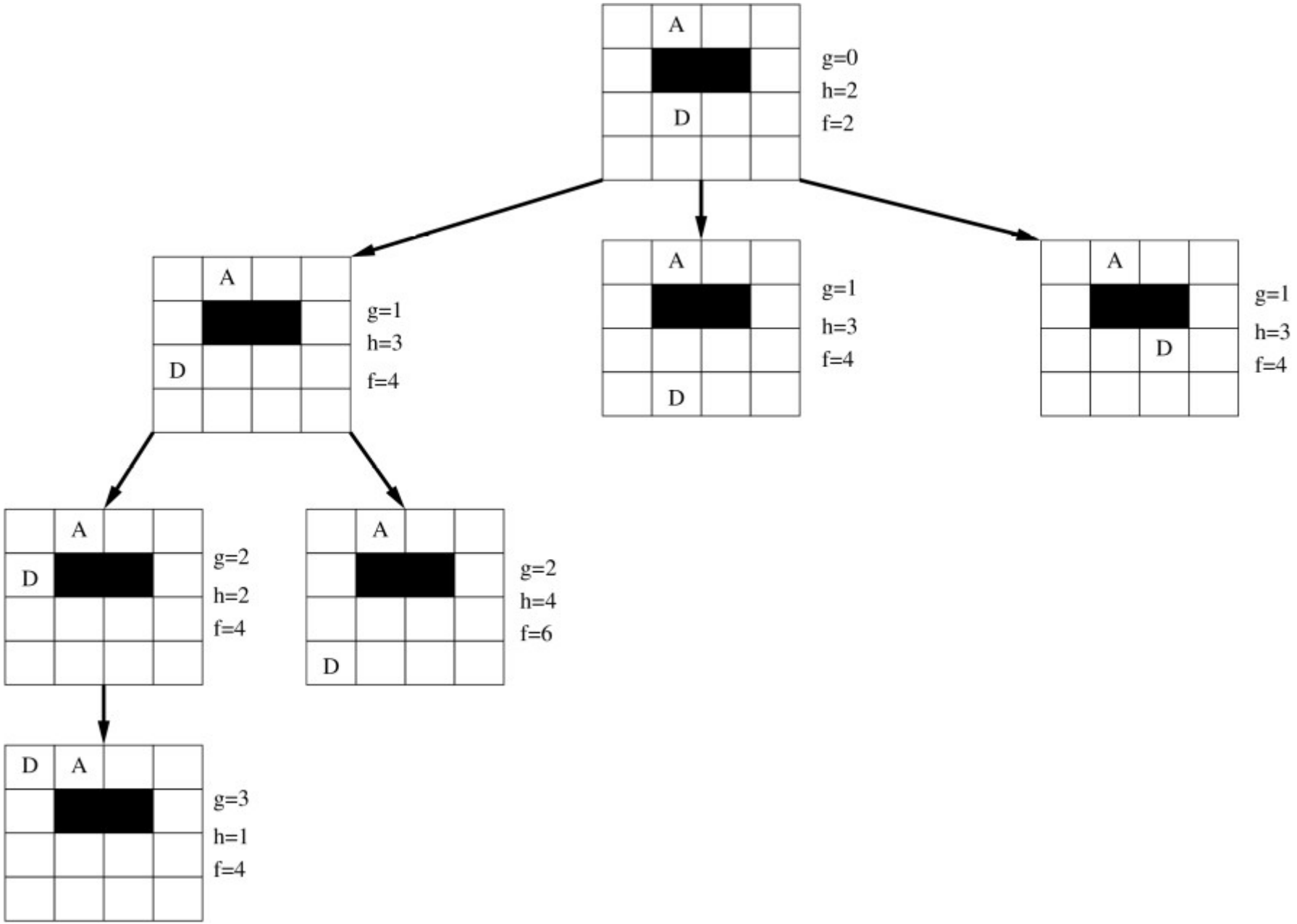
A*



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*

- Exercice :
- Trouver une heuristique admissible lorsque les déplacements diagonaux sont autorisés.
- Pour les déplacements horizontaux et verticaux l'heuristique de Manhattan est :

$$h = |x_D - x_A| + |y_D - y_A|$$

A*

- Exercice :
- 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 [Cazenave 2006].

Optimizations of data structures, heuristics and algorithms for path-finding on maps

Tristan Cazenave
Labo IA
Dept. Informatique
Université Paris 8, Saint-Denis, France
cazenave@ai.univ-paris8.fr

***Abstract*— This paper presents some optimizations of A* and IDA* for pathfinding on maps. The best optimal pathfinder we present can be up to seven times faster than the commonly used pathfinders as shown by experimental results. We also present algorithms based on IDA* that can be even faster at the cost of optimality. The optimizations concern the data structures used for the open nodes, the admissible heuristic and the re-expansion of points. We uncover a problem related to the non re-expansion of dead-ends for sub-optimal IDA*, and we provide a way to repair it.**

Keywords: path-finding, game maps, mazes, A-star, IDA*

I. Introduction

Path-finding is an important part of many applications, including commercial games and robot navigation. In games it is important to use an optimized path-finding algorithm because the CPU resources are also needed by other algorithms, and because many games are real-time. The particular problem addressed in this paper is grid-based path-finding. It is often used in real-time strategy games for example, in order to find the shortest path for an agent to its *goal* location on the map.

A* [1] is the standard algorithm for finding shortest paths. The usual heuristic associated to A* is the Manhattan heuristic. We present data structures and heuristics that enable A* to be up to seven times faster than the usual implementation.

The contributions of this paper are :

- It is better to use an array of stacks than a priority queue for maintaining the open nodes of an A* search.
- The $ALTBest_P$ heuristic is introduced, and is shown to perform better than the Manhattan heuristic and than the ALT heuristic [2].
- It is useful for IDA* to maintain a two-step lazy cache of the length of the shortest paths found so far.
- Adding a constant to the next threshold of IDA* enables large speed-ups at the cost of optimality. However the lengths of the paths are within 2% of the optimal. Sub-optimal IDA* can be competitive with A*.
- Recording the minimum f for each searched location is useful to find dead-ends, but it can cut valid paths when used with a sufficiently large added constant to the threshold of IDA*. Our program can detect and repair the problem.

Section two describes related work. Section three presents optimizations related to the choice of the best open node. Section four deals with the re-expansion of points. Section five presents the different admissible heuristics we have tested. Section six details experimental results. Section seven concludes and outlines future work.

II. Related work

A* [1] is a search algorithm that finds shortest paths. It uses a fast heuristic function that never over-estimates the length of the path to the goal state, i.e. an admissible heuristic often named h. For each node, it knows g the cost of the path from the root of the search to the node, and it computes f = g + h. The f function is used to develop the tree in a best first manner: A* expands the node with the smallest possible f. IDA* [5] is the iterative deepening version of A*. It can be enhanced with a transposition table that detects positions that have already been searched [6].

Fringe Search [4] is a hybrid of A* and IDA* that reduces slightly the computation time compared to A*. The Fringe Search paper also presents useful optimizations of IDA* for game maps.

The use of map abstractions can be combined with A* to make it faster. For example, Path-Refinement A* [7] builds high level plans and progressively refines them into low-level actions.

The admissible heuristic used for path-finding on road maps, which are much more constrained than game maps, can be improved (i.e. give greater admissible values) using the ALT heuristic [2]. The heuristics used on road maps can be reused for game maps, especially when the game maps are complex and are close to mazes.

III. Choosing the best open node

Each time it expands a node, A* chooses the node with the smallest f function. Therefore the cost of finding the node with the smallest f is very important for A*. In this section we present three methods and the associated data structures used to find the best open node. The first method is using a list of open nodes. The second method is widely used and implements the open list as a priority queue. The third method is faster than the two others and uses an array of stacks.

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	

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 Python.

  (c'est plus simple que A*)

IDA*

- IDA* est simple à programmer :

```
IDA (s, g, seuil)
  if (g + h (s) > seuil)
    return false
  if (s == goal state)
    return true
  for (move in possible moves)
    s1 = play (s, move)
    if (IDA (s1, g + 1, seuil))
      return true;
  return false
```

IDA*

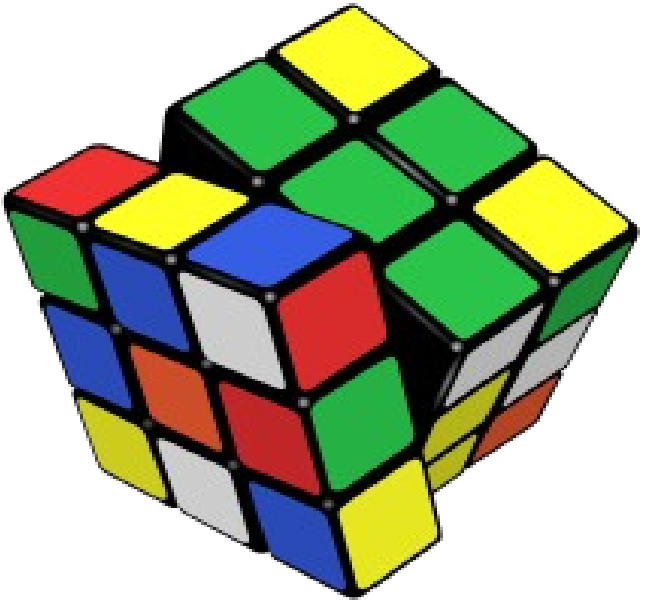
```
IDA (s)
  seuil = h (s)
  while (IDA (s, 0, seuil) == False)
      seuil = seuil + 1
```

Recherche heuristique

Heuristiques admissibles pour A*

Le Rubik's cube

- Le Rubik’s Cube est un casse-tête inventé en 1974 par le Hongrois Ernő Rubik, et qui s’est rapidement répandu sur toute la planète au cours des années 1980 (Wikipédia).



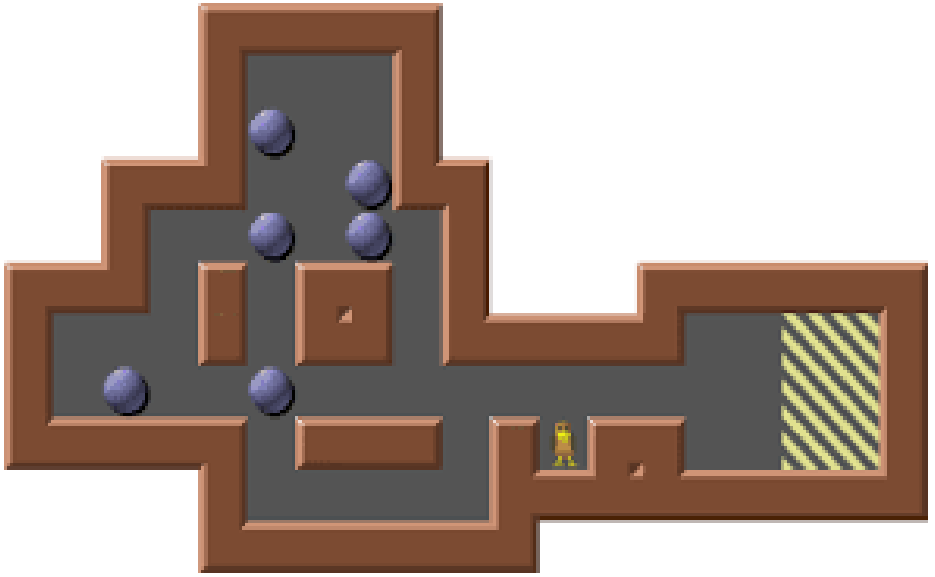
Le Rubik's cube

- Le Rubik's cube peut être résolu pratiquement instantanément par un programme qui utilise des macro coups.
- Ce sont des suites de coups qui échangent deux cubes sans modifier la place des autres.
- Nous nous intéressons à la résolution optimale du Rubik's cube qui est plus difficile.
- La résolution optimale consiste à trouver une solution contenant le plus petit nombre de coups possibles.
- Le diamètre du Rubik's cube est de 20.

Le Rubik's cube

- Le Rubik's Cube se prête bien à une résolution par IDA*.
- Le coût des recherches exhaustives à une profondeur inférieure à celle du seuil n'est que de 8% du coût de la recherche à la profondeur du seuil.
- L'utilisation de IDA* dans ce cas est donc justifiée.
- Elle est même nécessaire étant donné le très grand nombre de noeuds à développer pour les profondeurs utiles.
- Exercice : Trouver une heuristique admissible simple basée sur la distance de Manhattan pour le Rubik's cube.
- Est il possible de l'améliorer simplement ?

Sokoban



- Le jeu original a été écrit par Hiroyuki Imabayashi et comportait 50 niveaux.
- Il remporte en 1980 un concours de jeu vidéo pour ordinateur (Wikipedia).

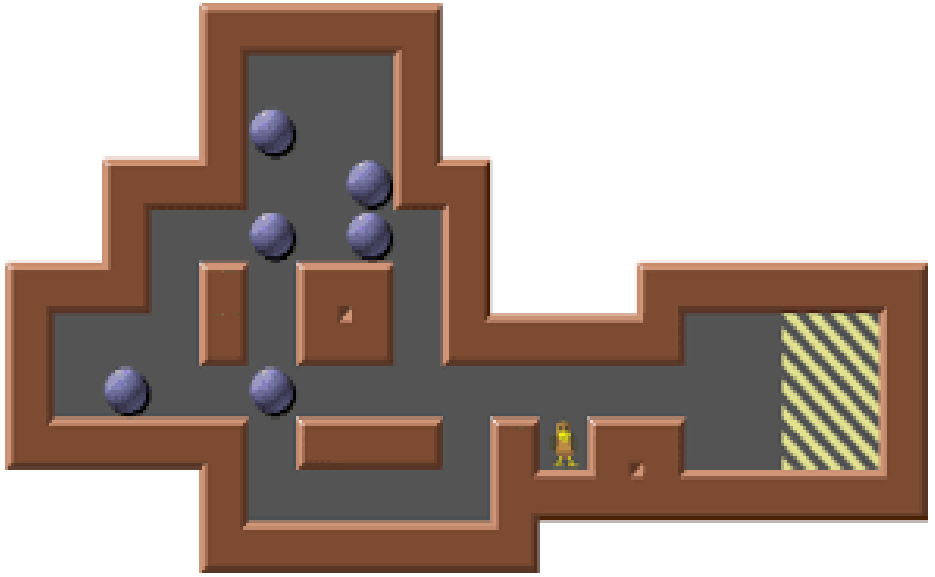
Sokoban

- Le joueur doit ranger des caisses sur des cases cibles.
- Il peut se déplacer dans les quatre directions, et pousser (mais pas tirer) une seule caisse à la fois.
- Une fois toutes les caisses rangées, le niveau est réussi et le joueur passe au niveau suivant.
- L'idéal est de réussir avec le moins de coups possibles.

Sokoban

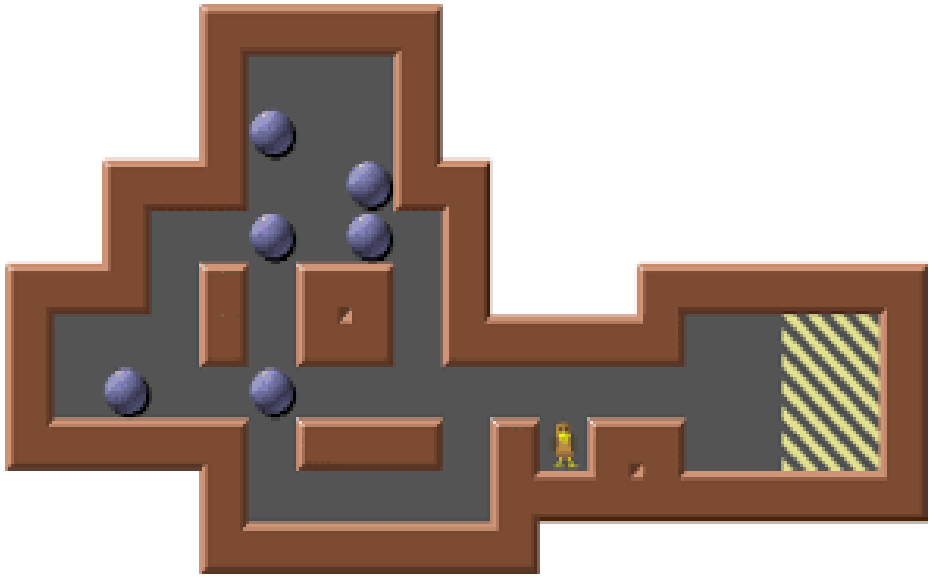
- Sokoban est le Go des puzzles.
- Une heuristique admissible simple est de calculer l'affectation optimale des caisses aux cibles en fonction de leurs distances.
- Algorithme hongrois en n^3, optimisations possibles.
- Thèse de Andreas Junghanns : Pushing the Limits: New Developments in Single-Agent search, 1999, University of Alberta.

Sokoban



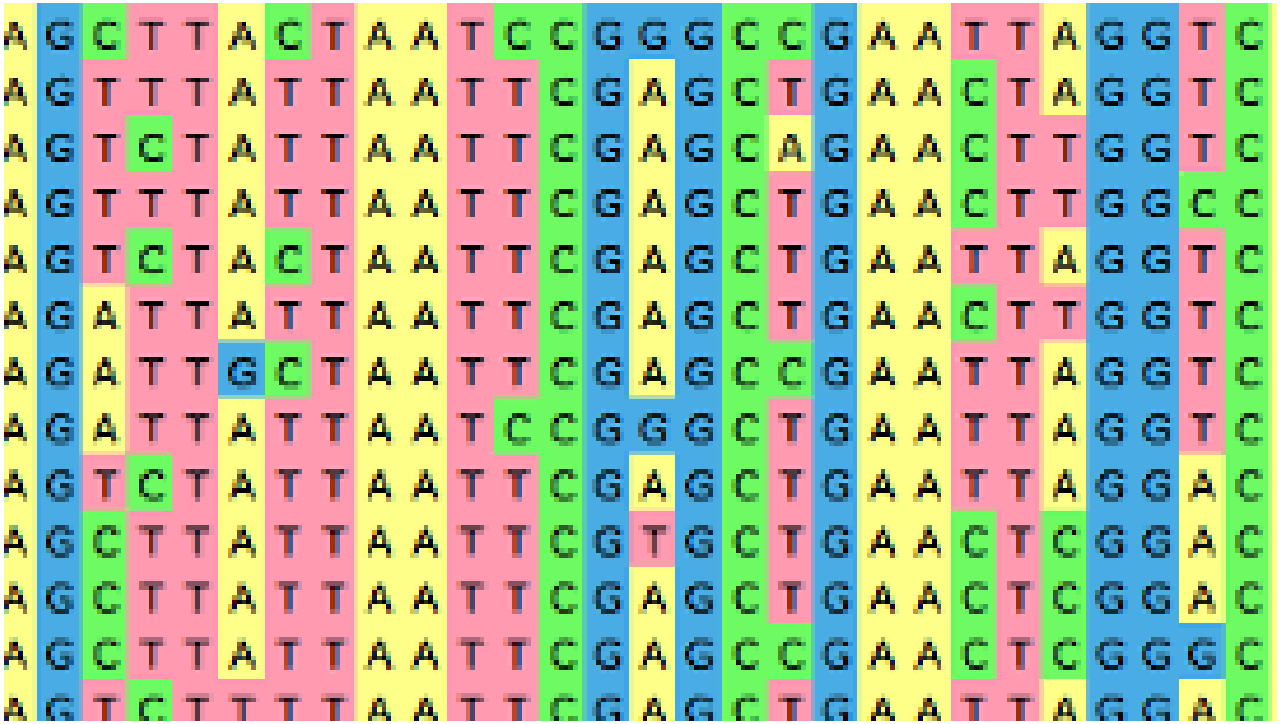
- Que vaut h pour cette position ?
- On ne compte que les poussées.

Sokoban



- h = 11 +14 + 14 + 16 + 12 + 13 = 80

Multiple Sequence Alignment



Multiple Sequence Alignment

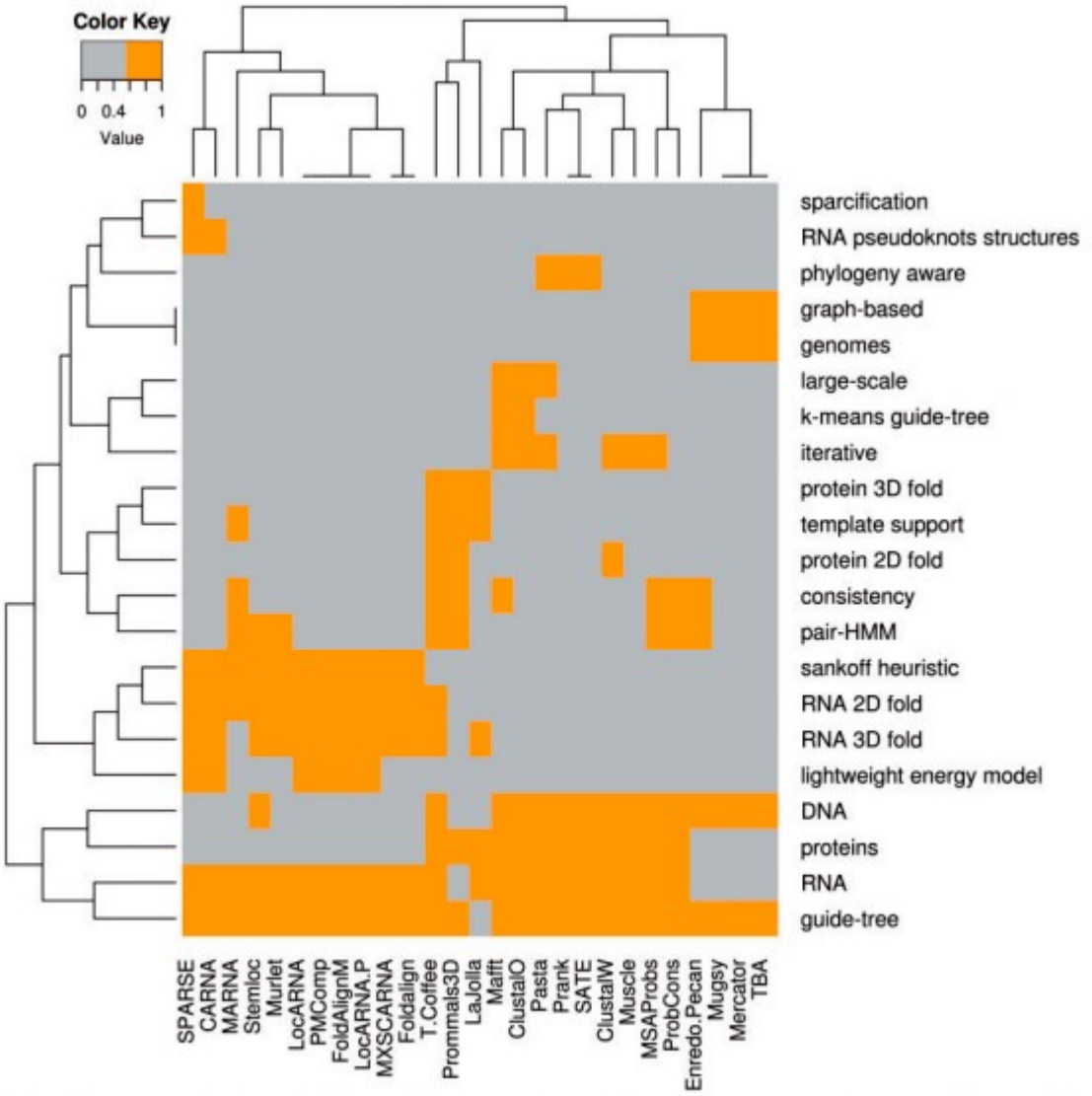


Figure 1. Main algorithmic components of the most widely used multiple aligners. On the heatmap, orange entries indicate a feature implemented in the considered method. Both the aligners and the components were clustered by similarity using the R-package. A colour version of this figure is available online at BIB online: https://academic.oup.com/bib.

Partial Move A*

Tristan Cazenave
LAMSADE
Université Paris-Dauphine
Paris, France

***Abstract*—Some shortest path problems that can be solved using the A* algorithm have a large branching factor due to the combination of multiple choices at each move. Multiple sequence alignment and multi-agent pathfinding are examples of such problems. If the search can be stopped after each choice instead of being stopped at each combination of choices, it takes much less memory and much less time. The goal of the paper is to show that Partial Move A* is much better than A* for these problems when the branching factor is large due to a large combination of choices at each move. When there is such a large combination of choices at each move, Partial Move A* can yield large memory gains and speedups over regular A*.**

I. Introduction

When the moves of a problem contain multiple choices, the evaluation of each choice separately can be interesting instead of evaluating all the possible combinations of choices. When each choice is evaluated separately the search can be stopped at a partial move instead of being stopped at each combination that contains this partial move. Partial Move A* (PMA*) is an iterative deepening A* algorithm [1] that stops search inside a move. It can yield large memory gains and speedups on difficult problems. The algorithm can be applied to multiple sequence alignment and to multi-agent pathfinding. We show in this paper that it enables large memory gains and large speedup on these two problems when the branching factor is large.

Multiple sequence alignment is an NP-hard problem that can be addressed with the A* algorithm [2]. It is commonly believed that iterative deepening A* is not appropriate for multiple sequence alignment since there are many paths passing through the same nodes, with the exception of the work on iterative deepening dynamic programming [3]. We show that PMA* is appropriate for multiple sequence alignment even if it is an iterative deepening A* algorithm.

The multi-agent pathfinding problem is PSPACE-hard [4]. Existing literature on the multi-agent pathfinding problem mainly deals with inexact algorithm as the problem is considered intractable. An example of an algorithm combining individual paths is cooperative pathfinding [5]. We are interested in exact algorithms for this problem. The standard exact algorithm is A*. PMA* improves much on A* search for this problem. PMA* is complete and optimal for the multi-agent pathfinding problem. There are good approximation algorithms for this problem such as [6], however these algorithm are not complete nor optimal.

An algorithm related to PMA* is Partial Expansion A* [7]. Partial Expansion A* uses less memory than A* but uses more time, whereas PMA* uses less memory than A* and less time on difficult problems. PMA* also uses less memory and less time than Partial Expansion A*.

The idea underlying PMA* is that it is very beneficial both for memory consumption and for speed to be able stop search at each move component. For example in multiple sequence alignment it is important to be able to stop search after each choice of either aligning a letter in a sequence or inserting a gap in the same sequence. Concerning multi-agent pathfinding it is important to be able to stop search after the move of each agent and not only after all the agents have moved.

The outline of this paper is as follows: section 2 compares A*, Partial Expansion A* and Partial Move A*. Section 3 presents the application of PMA* to multiple sequence alignment and section 4 presents its application to multi-agent pathfinding.

II. Comparison of algorithms for A*

In this section the differences between A*, Partial Expansion A* and PMA* are outlined.

A. A*

In A*, all the children of a node are added to the Open structure, and the node is placed in the closed structure after expansion.

B. Partial Expansion A*

Partial expansion A* does not expand all the children of a node. It selects the children that have a F value smaller than the F value of the node plus a constant C.

It defines two sets :

$SUCC_{\leq C} \leftarrow \{n_j | n_j \in succ(n), f(n_j) \leq F(n) + C\}$

$SUCC_{>C} \leftarrow \{n_k | n_k \in succ(n), f(n_k) > F(n) + C\}$

It only expands the children in $SUCC_{\leq C}$ and puts the node n in the closed nodes only if $SUCC_{>C} = \emptyset$.

Each time the algorithm comes again on an already partially expanded node, it has to develop again the node in order to build the SUCC sets of nodes. When there are many sequences, the computing time of the algorithm is dominated by the computation of the SUCC sets.

C. Partial Move A*

When running Partial Expansion A* on problems with a large number of children for each node, most of the time of the algorithm is consumed in finding all the possible children of a node and computing the sets $SUCC_{\leq C}$ and $SUCC_{>C}$. For

Multiple Sequence Alignment

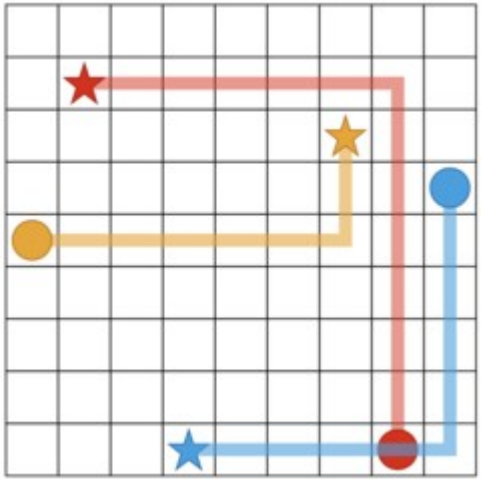
- Exercise:
- Code the state and the possible moves for aligning 3 sentences [Cazenave 2010].
- Code the admissible heuristic.
- Apply IDA*.

Multi-Agent Pathfinding

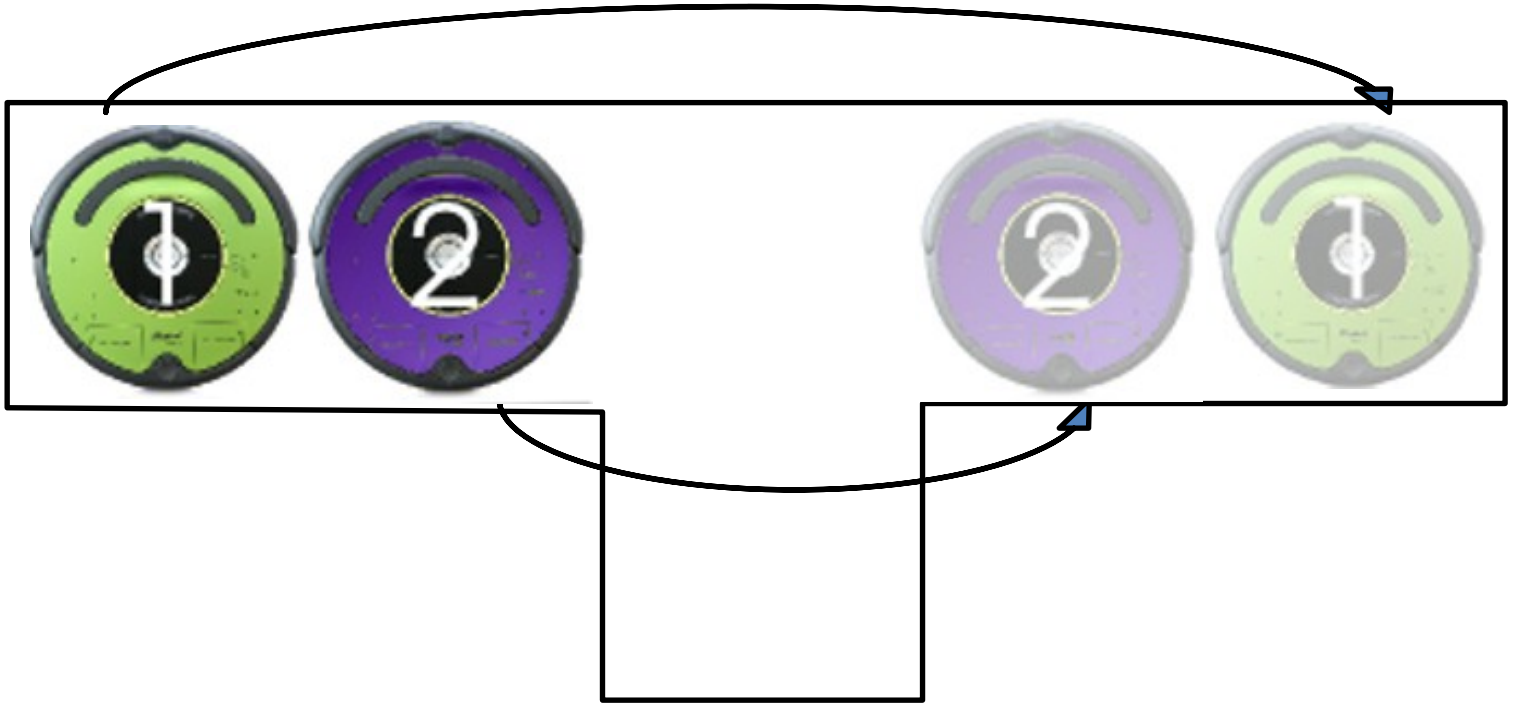
- The Multi-Agent Pathfinding (MAPF) problem is the fundamental problem of planning paths for multiple agents.
- The key constraint is that the agents will be able to follow these paths concurrently without colliding with each other.
- Applications of MAPF:
  - Automated warehouses (e.g. Amazon).
  - Autonomous vehicles.
  - Path planning in games (e.g. Real Time Strategy: StarCraft).

Multi-Agent Pathfinding

- Cooperative Pathfinding (Silver 2005)
  - A* search for the first agent.
  - A* search for the second agent avoiding the first.
  - …
  - A* search for the last agent avoiding the previous.

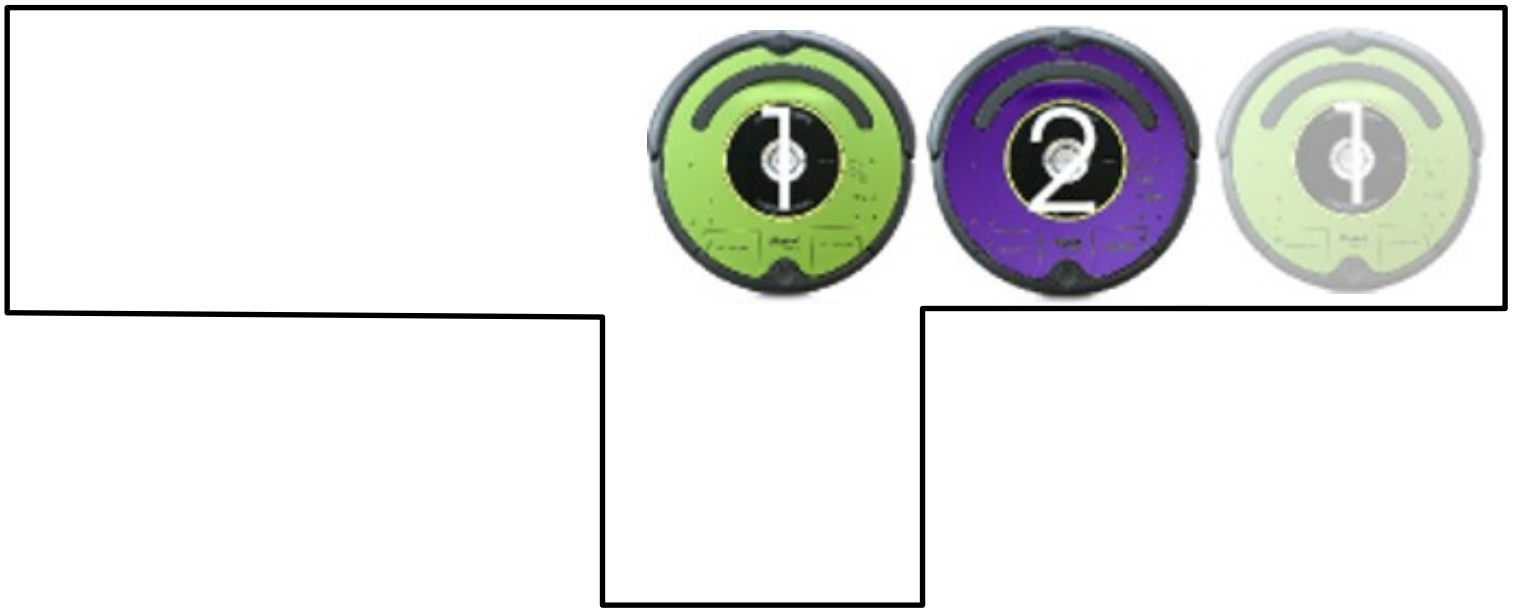


Multi-Agent Path Finding (MAPF)

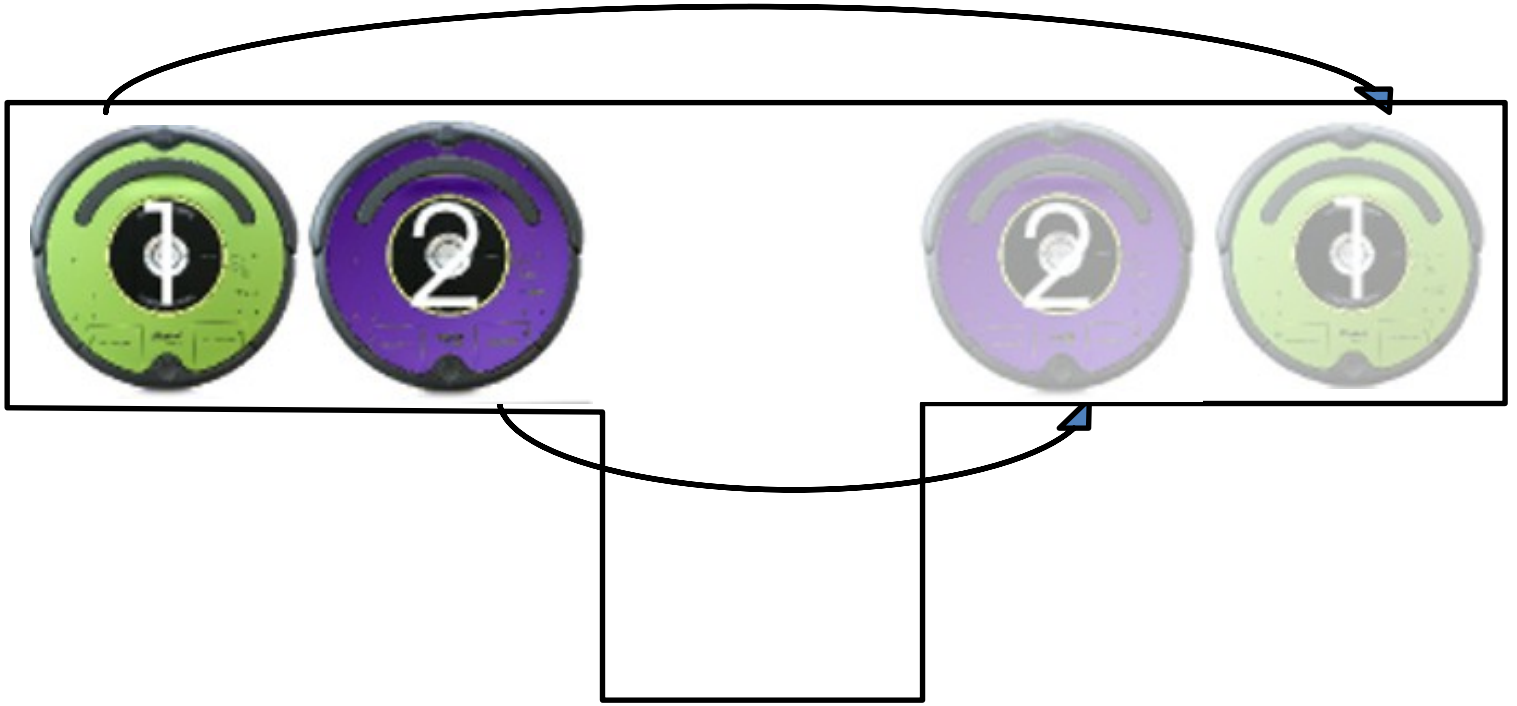


Multi-Agent Path Finding (MAPF)

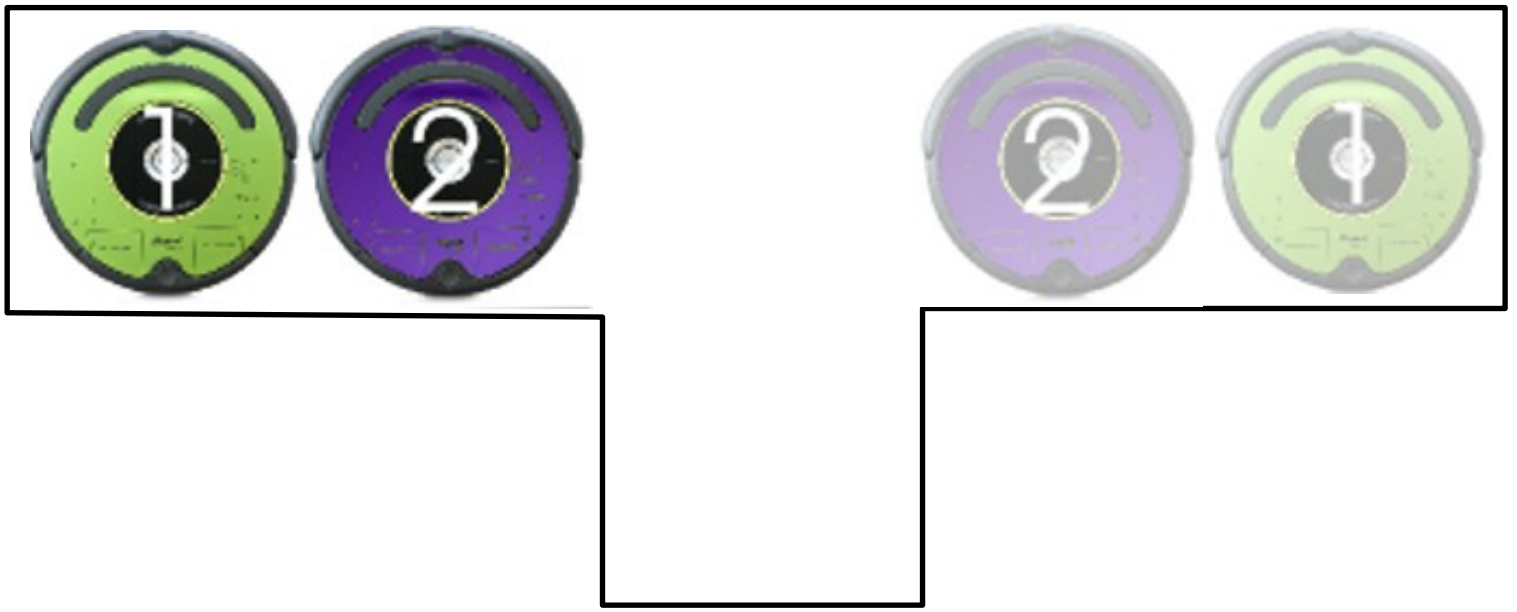
Multi-Agent Path Finding (MAPF)



Multi-Agent Path Finding (MAPF)



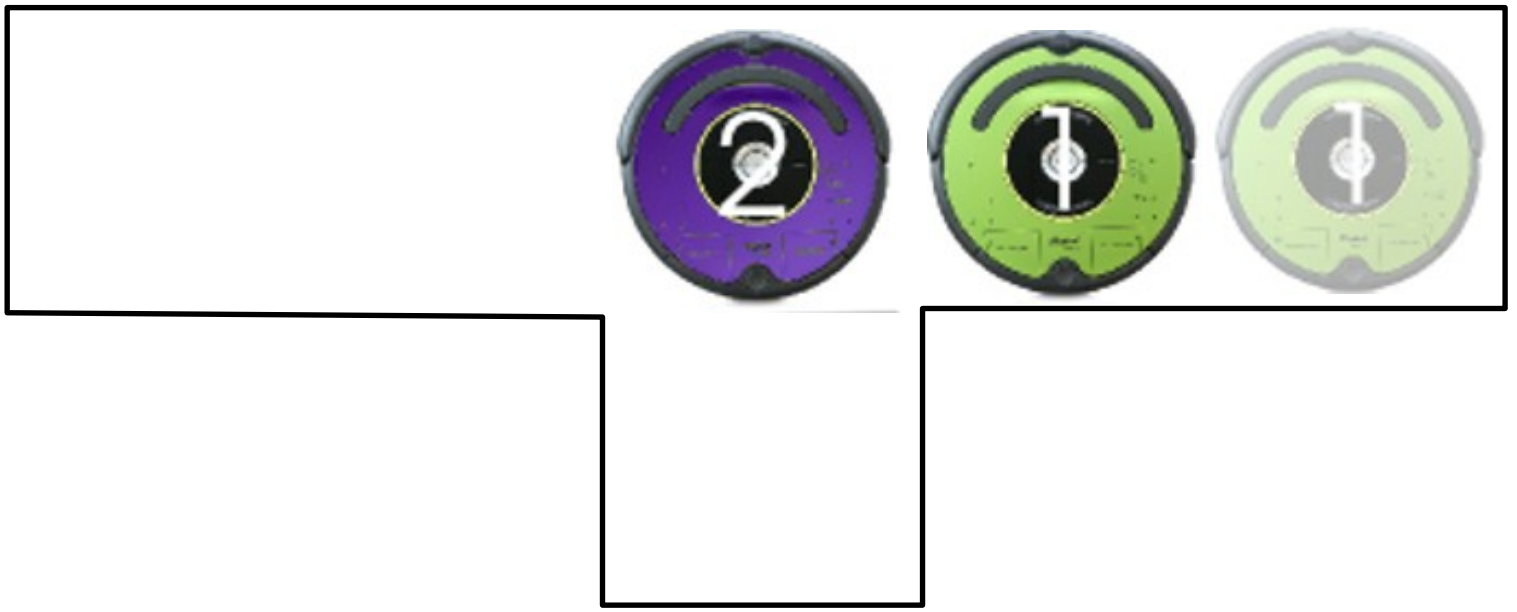
Multi-Agent Path Finding (MAPF)



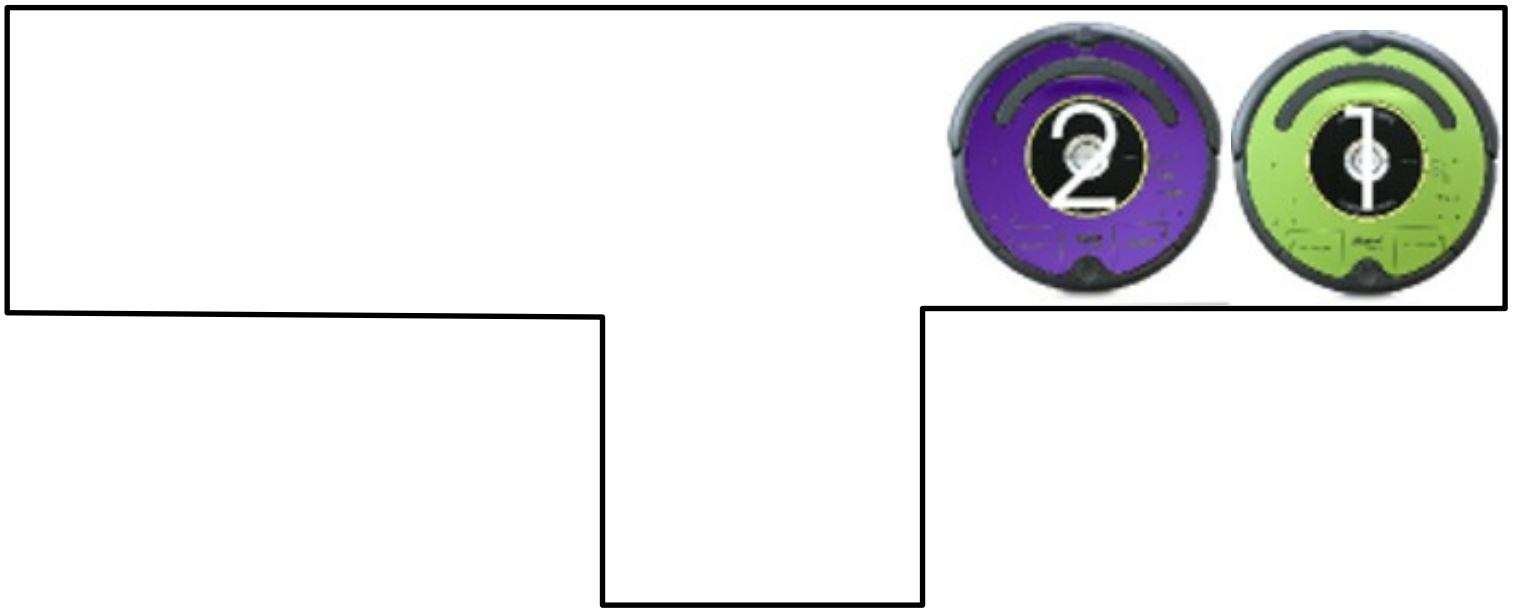
Multi-Agent Path Finding (MAPF)

Multi-Agent Path Finding (MAPF)

Multi-Agent Path Finding (MAPF)



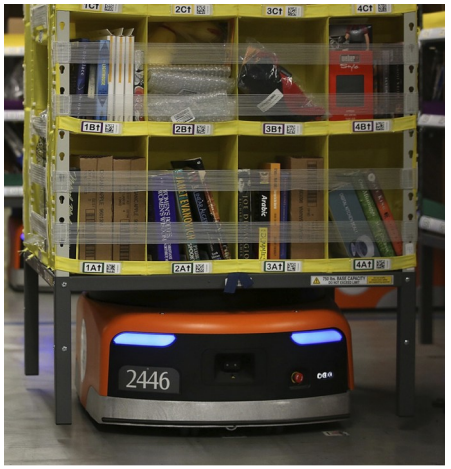
Multi-Agent Path Finding (MAPF)



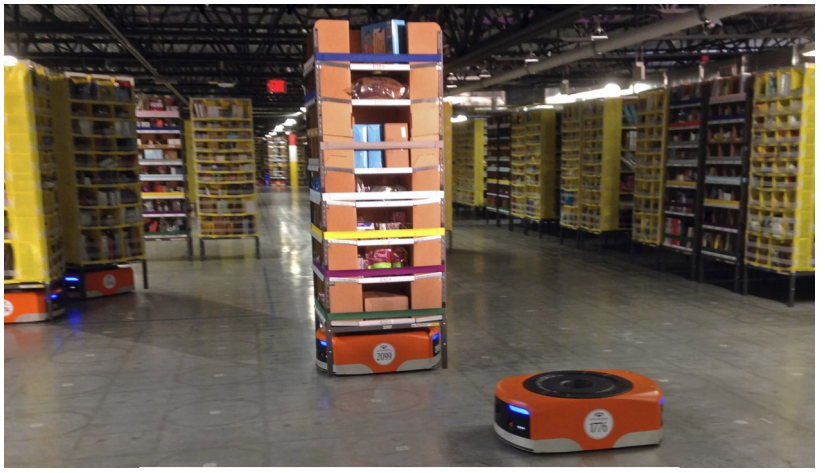
- Optimization problem with the objective
  to minimize task-completion time (called makespan) or
  the sum of travel times (called flowtime)

Multi-Agent Path Finding (MAPF)

- Application: Amazon fulfillment centers
- 2003 Kiva Systems founded
- 2012 Amazon acquires Kiva Systems for $775 million
- 2015 Kiva Systems becomes Amazon Robotics



[www.npr.org – Getty Images]



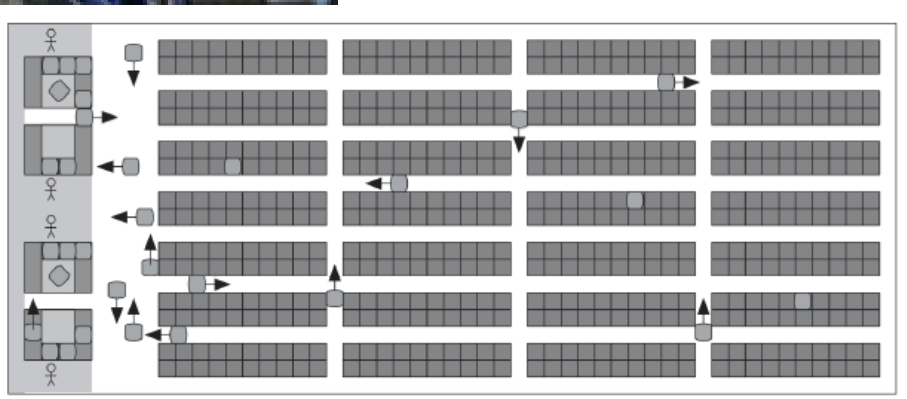
[www.theguardian.com - AP]

- > 3,000 robots on > 110,000 square meters in Tracy, California

Multi-Agent Path Finding (MAPF)

- Application: Amazon fulfillment centers

[from: YouTube]



[Wurman, D'Andrea and Mountz]

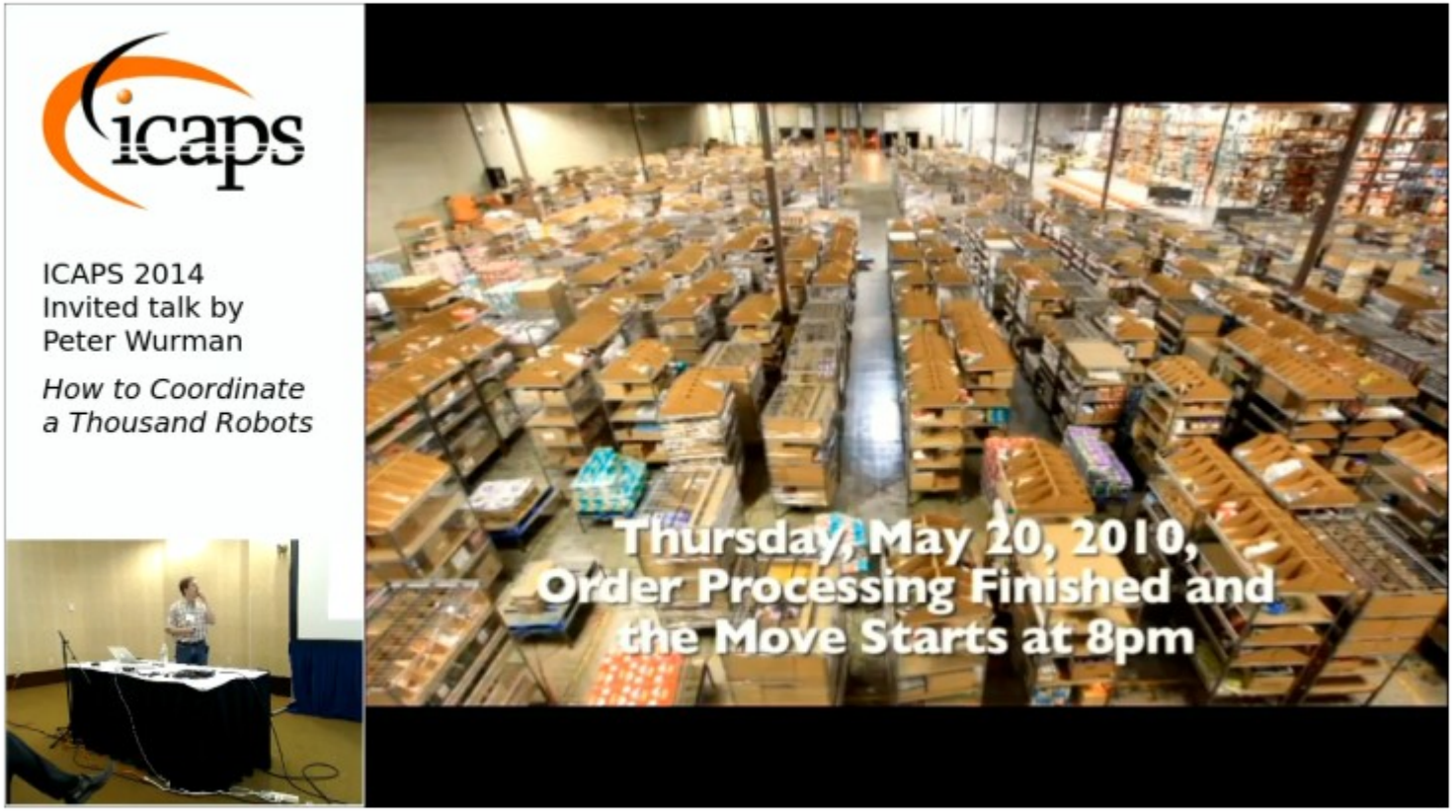
Multi-Agent Path Finding (MAPF)

- Application: Amazon fulfillment centers

[from: YouTube]

Multi-Agent Path Finding (MAPF)

- Application: Amazon fulfillment centers

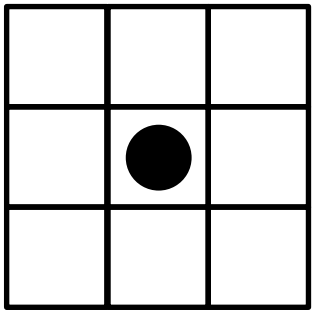


[from: YouTube]

Multi-Agent Path Finding (MAPF)

Robot

Agent



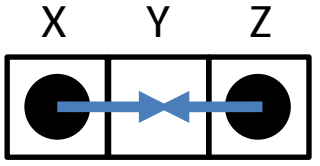
[from: YouTube]

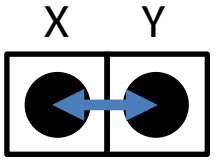
Stickers on the ground establish a grid!

- Simplifying assumptions
  - Point agents
  - No kinematic constraints
  - Discretized environment
    - we use grids here but most techniques work on planar graphs in general

Multi-Agent Path Finding (MAPF)

- Each agent can move N, E, S or W into any adjacent unblocked cell (provided an agent already in that cell leaves it while the agent moves into it or earlier) or wait in its current cell
- Not allowed ("vertex collision")
  - Agent 1 moves from X to Y
  - Agent 2 moves from Z to Y
- Not allowed ("edge collision")
  - Agent 1 moves from X to Y
  - Agent 2 moves from Y to X





Multi-Agent Path Finding (MAPF)

- Suboptimal MAPF algorithms
  - Theorem [Yu and Rus]: MAPF can be solved in polynomial time on undirected grids without makespan or flowtime optimality
  - Unfortunately, good throughput is important in practice!

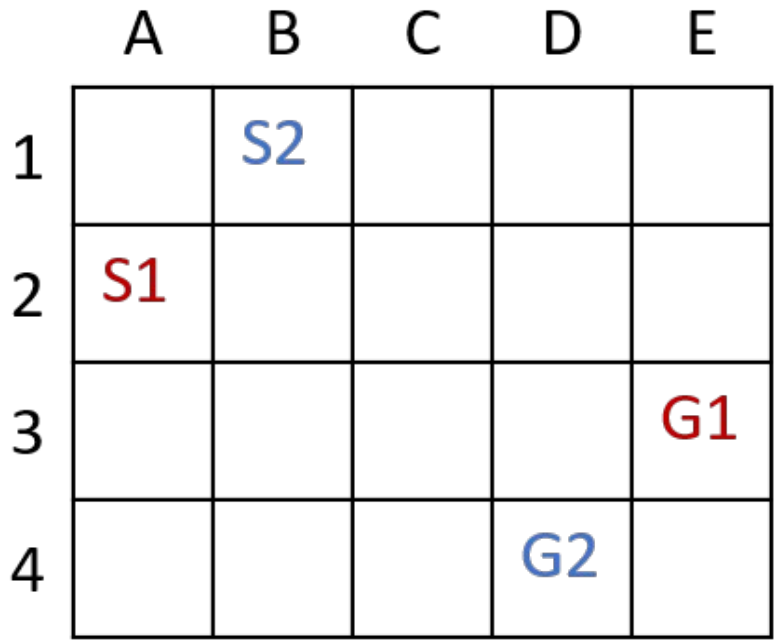
Multi-Agent Path Finding (MAPF)

- Optimal MAPF algorithms
  - Theorem [Yu and LaValle]: MAPF is NP-hard to solve optimally for makespan or flowtime minimization

[www.random-ideas.net]

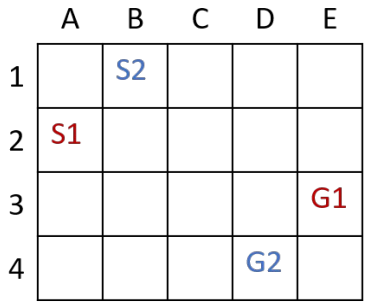
- Bounded-suboptimal MAPF algorithms
  - Theorem [Ma, Tovey, Sharon, Kumar and Koenig]: MAPF is NP-hard to approximate within any factor less than 4/3 for makespan minimization on graphs in general

Multi-Agent Path Finding (MAPF)

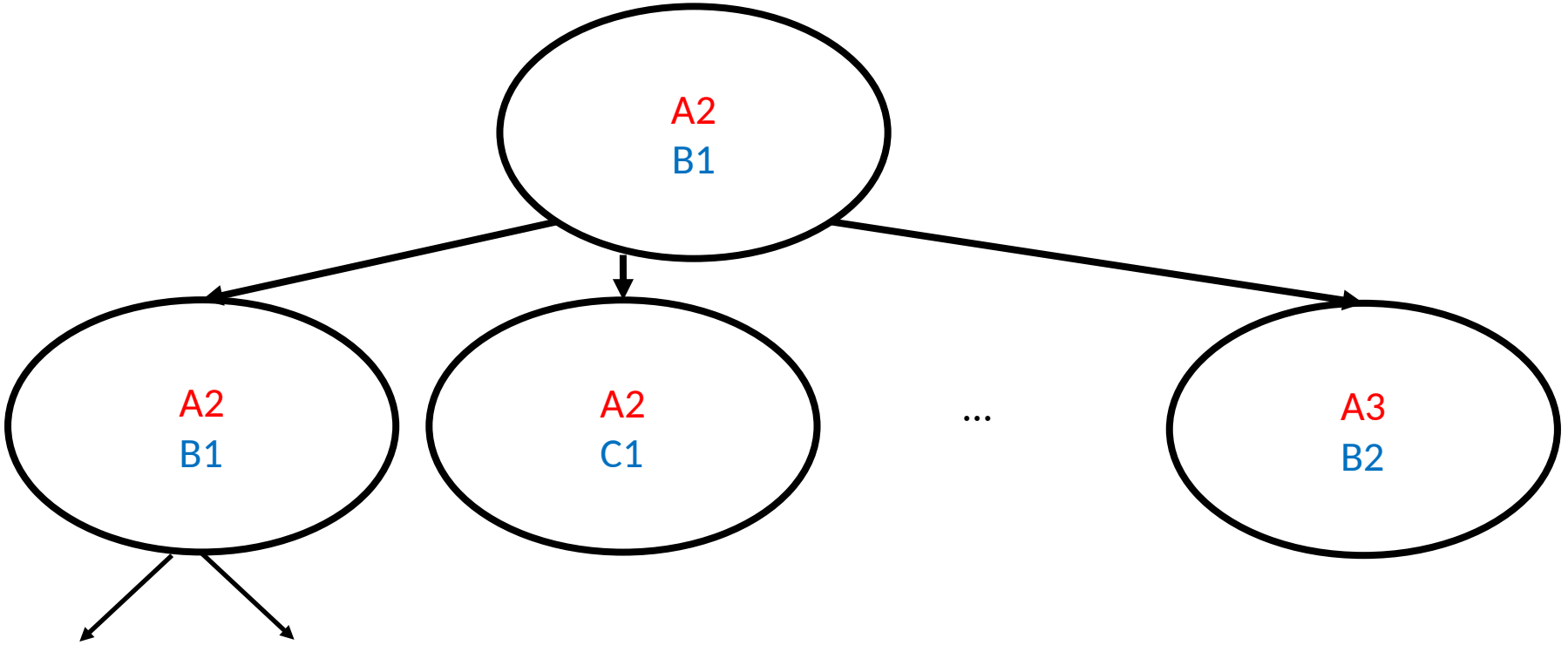


S1 (S2) = start cell of the red (blue) agent
G1 (G2) = goal cell of the red (blue) agent

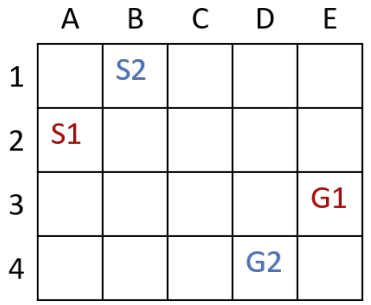
A*-Based Search



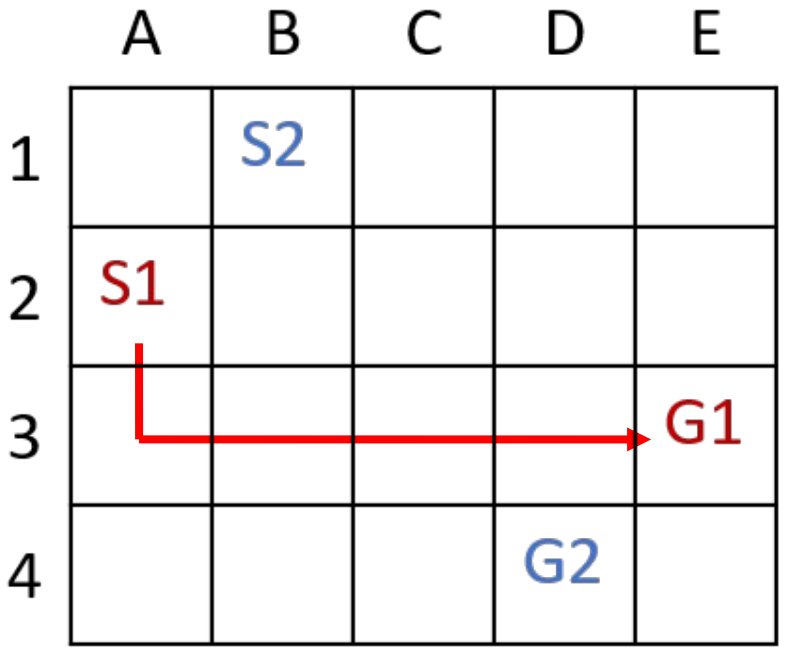
- A*-based search in the joint cell space: Optimal (or bounded-suboptimal) but extremely inefficient MAPF solver



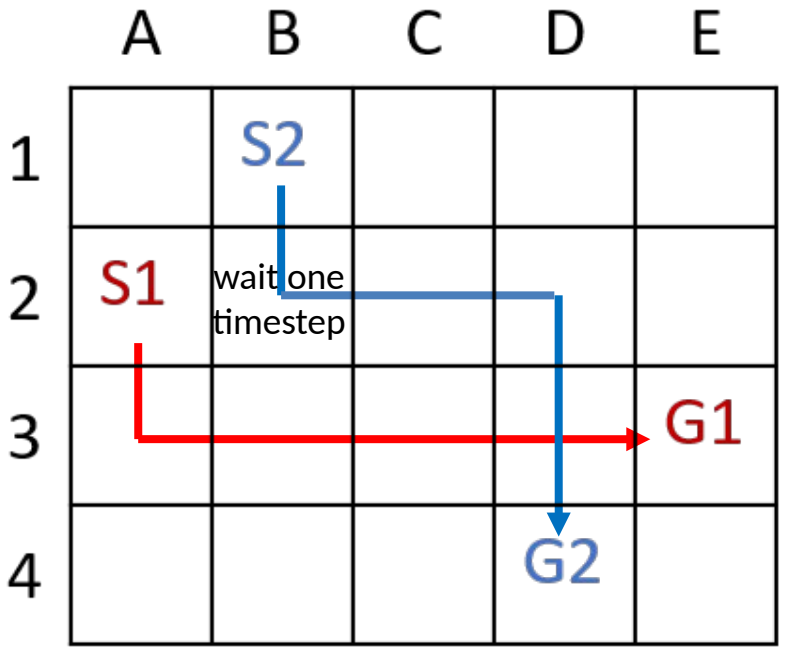
Priority-Based Search



- Priority-based (= sequential) search (plan for one agent after another in space (= cell)-time space in a given order): efficient but suboptimal (and even incomplete) MAPF solver

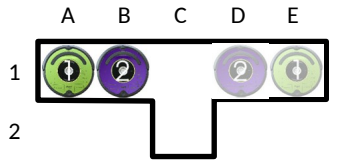


First, find a time-minimal path for the agent with priority 1.

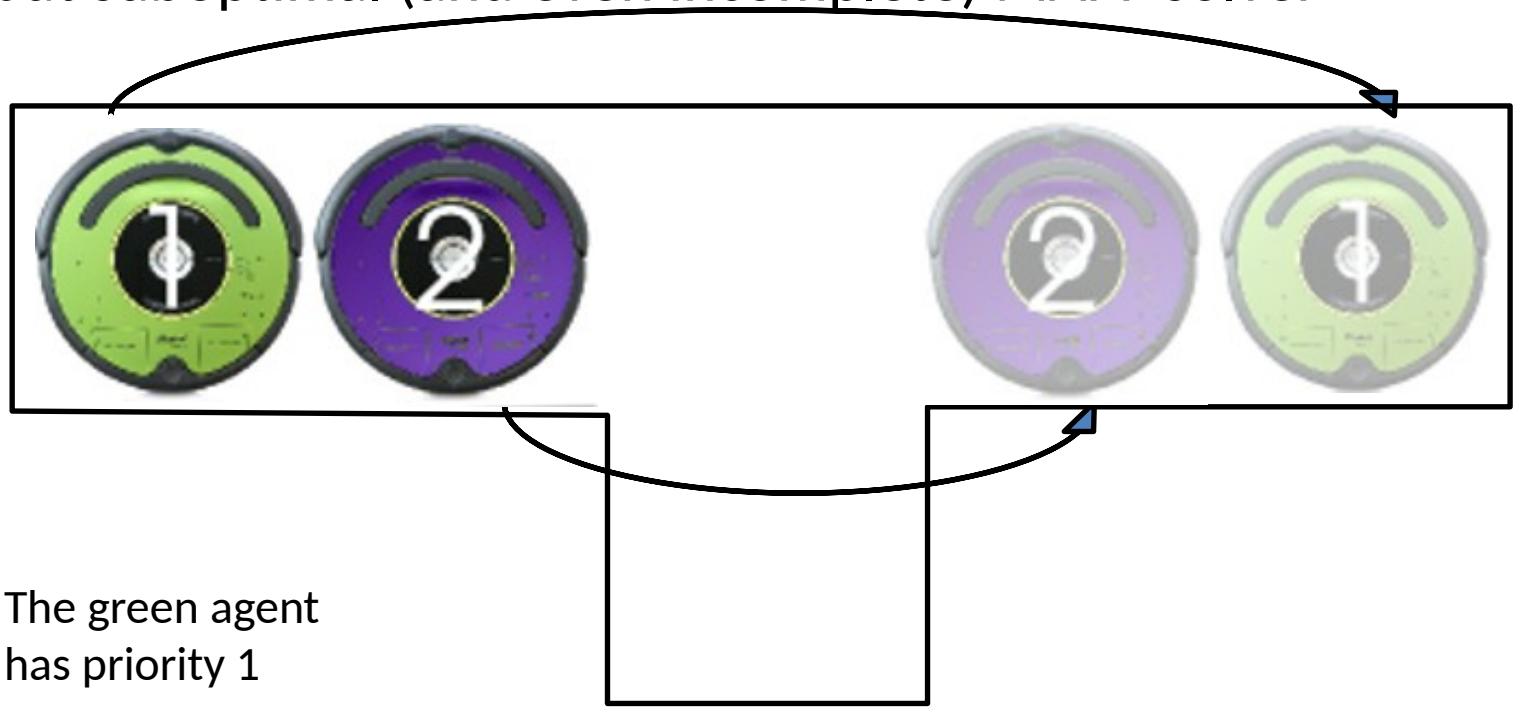


Then, find a time-minimal path for the agent with priority 2 that does not collide with the paths of higher-priority agents.

Priority-Based Search



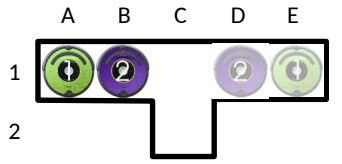
- Priority-based (= sequential) search (plan for one agent after another in space (= cell)-time space in a given order): efficient but suboptimal (and even incomplete) MAPF solver



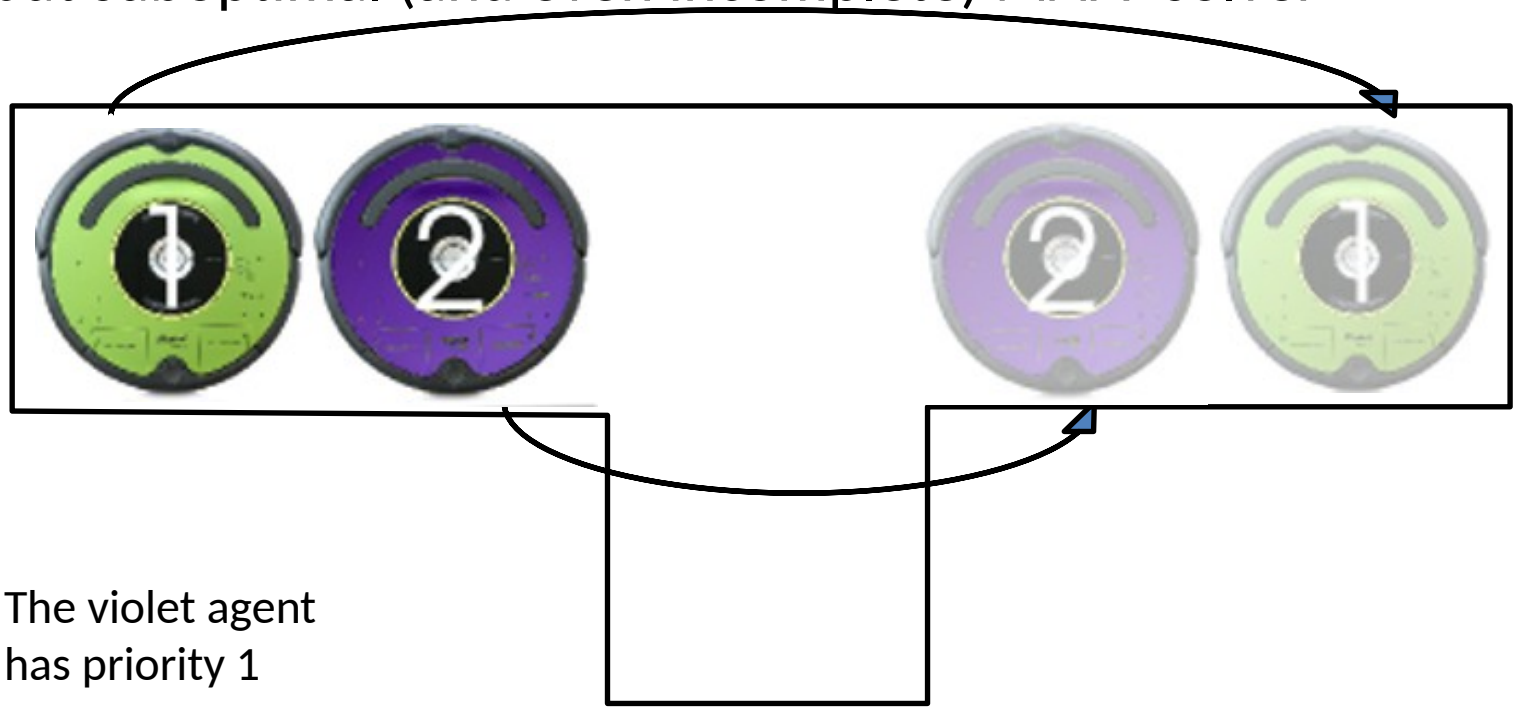
The green agent has priority 1

- Priority-based search finds first path A1, B1, C1, D1, E1 for the green agent and then path B1, C1, C2, C1, D1 for the violet agent. Thus, priority-based search finds a solution.

Priority-Based Search



- Priority-based (= sequential) search (plan for one agent after another in space (= cell)-time space in a given order): efficient but suboptimal (and even incomplete) MAPF solver



The violet agent has priority 1

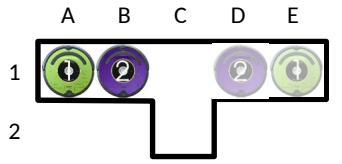
- Priority-based search finds first path B1, C1, D1 for the violet agent and then no path for the green agent. Thus, priority-based search does not find a solution.

Priority-Based Search

- You could implement space (= cell)-time A* with a reservation table (specific for a particular agent) as follows

- The states are pairs (cell, t) for all cells and times
- If the agent can move from cell X to cell Y (in the absence of other agents), create direct edges
  - from state (X,0) to state (Y,1)
  - from state (X,1) to state (Y,2)
  - …
- If the agent is not allowed to be in cell X at time t (because a collision with a higher-priority agent would result), delete state (X,t)
- If the agent is not allowed to move from cell X to cell Y at time t (because a collision with a higher-priority agent would result), delete the directed edge from state (X,t) to state (Y,t+1)
- Search the resulting state space for a time-minimal path from state (start cell, 0) to any state (goal cell, t) for all times t

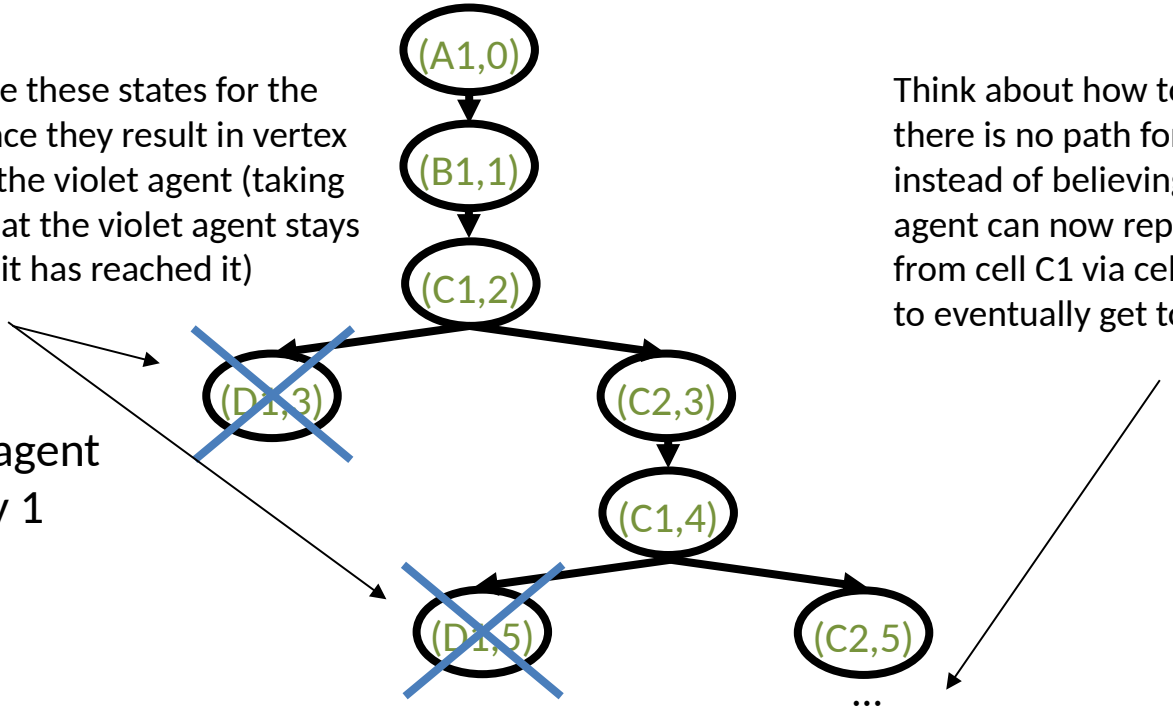
Priority-Based Search



- You could implement space (= cell)-time A* with a reservation table (specific for a particular agent) but you might not want to build it explicitly since it is often large. Rather, you never want to generate the states or edges that you would have deleted in the reservation table in the A* search tree

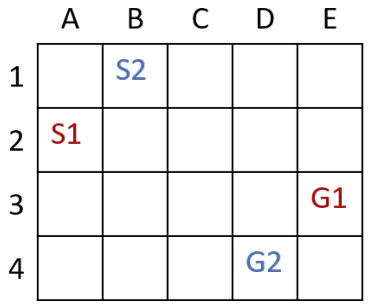
Do not generate these states for the green agent since they result in vertex collisions with the violet agent (taking into account that the violet agent stays in cell D3 once it has reached it)

Think about how to detect that there is no path for the green agent instead of believing that the green agent can now repeatedly move from cell C1 via cell C2 back to cell C1 to eventually get to its goal cell E1



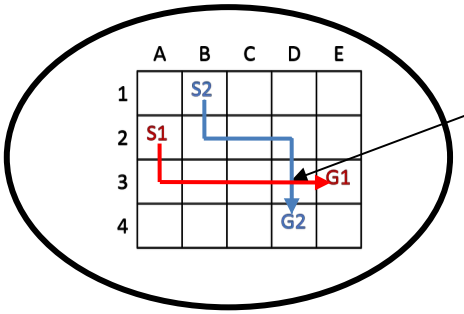
The violet agent has priority 1

Conflict-Based Search



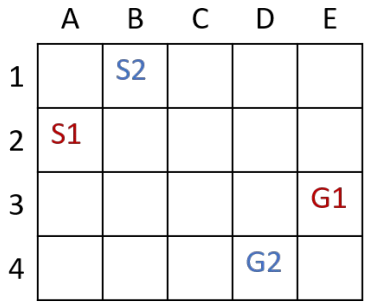
- Conflict-based search [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible

Find time-minimal paths for all agents independently



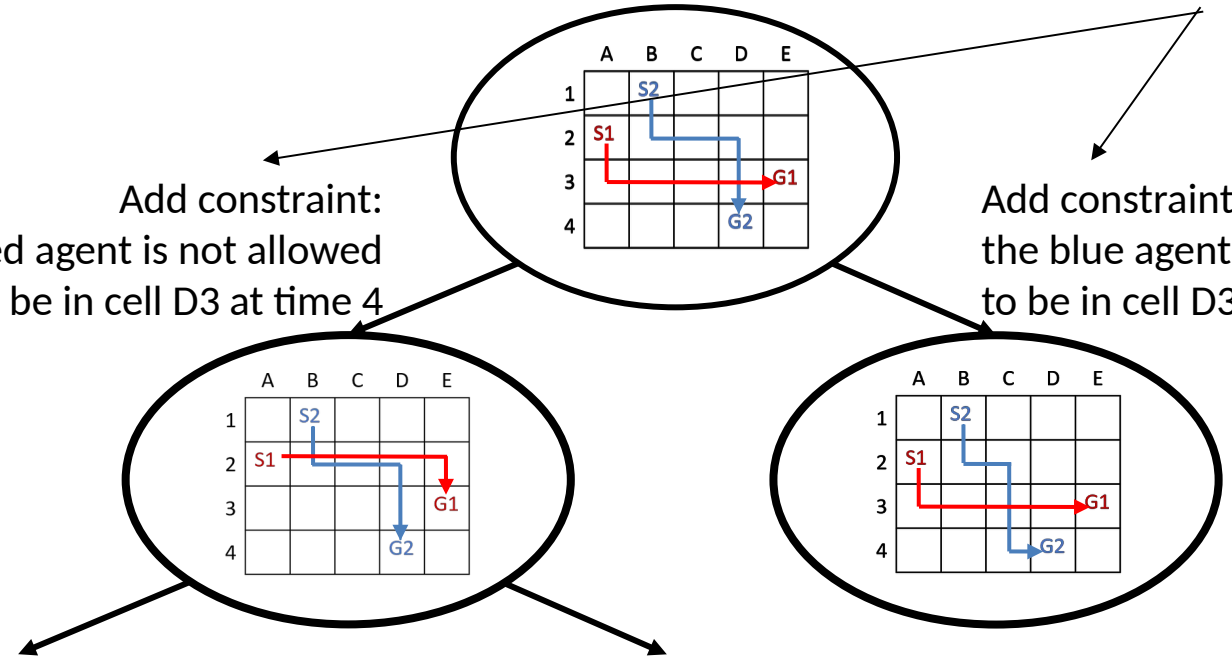
Conflict (here: vertex collision)

Conflict-Based Search



- Conflict-based search [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible

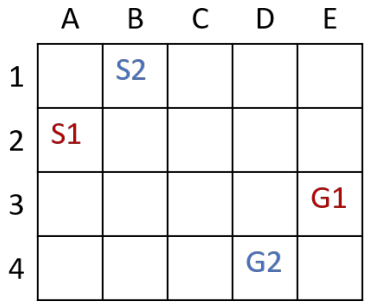
Such vertex constraints simply correspond to one blocked cell each in the reservation table



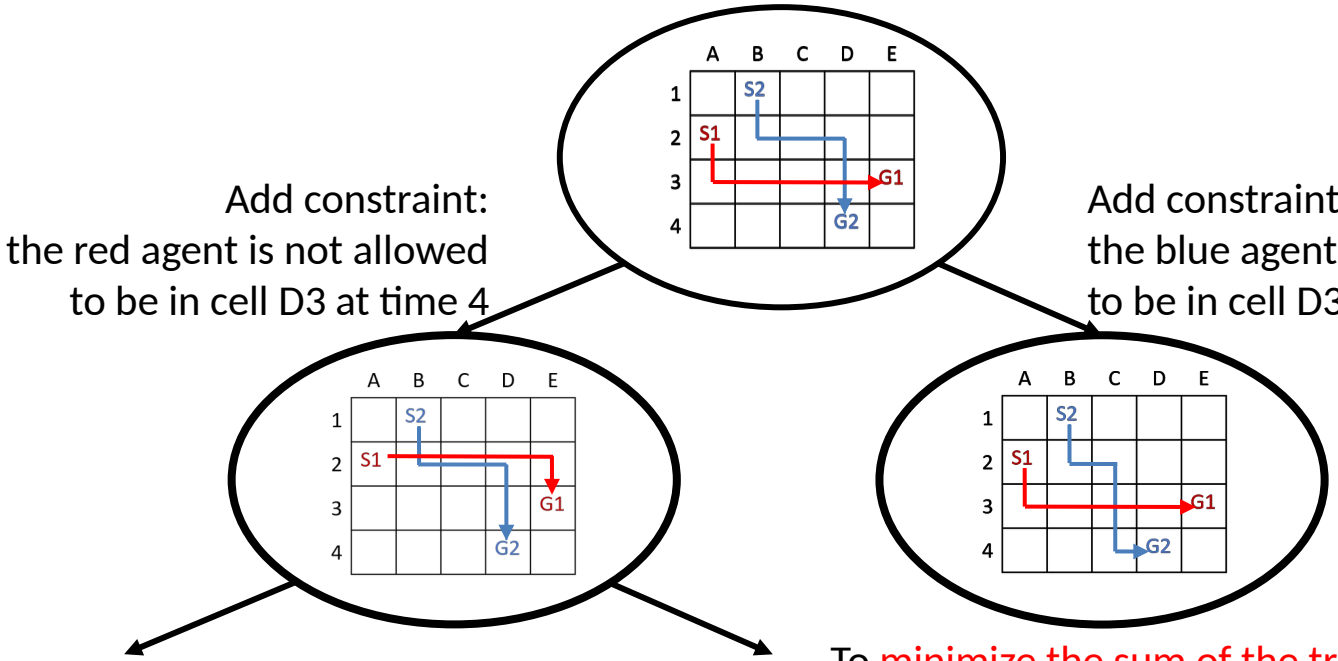
Add constraint:
the red agent is not allowed to be in cell D3 at time 4

Add constraint:
the blue agent is not allowed to be in cell D3 at time 4

Conflict-Based Search



- Conflict-based search [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible



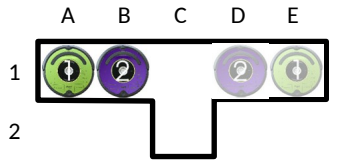
Add constraint:
the red agent is not allowed
to be in cell D3 at time 4

Add constraint:
the blue agent is not allowed
to be in cell D3 at time 4

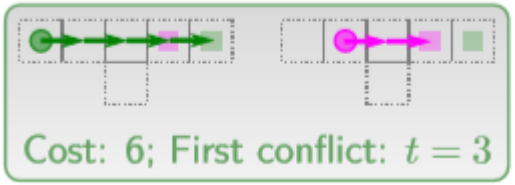
To minimize the sum of the travel times of all agents perform a best-first search on this tree with

- g = cost = sum of travel times of all agents (here: 10)
- h = 0

Conflict-Based Search



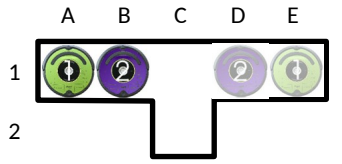
- Conflict-based search [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible



A1, B1, C1, D1, E1 B1, C1, D1

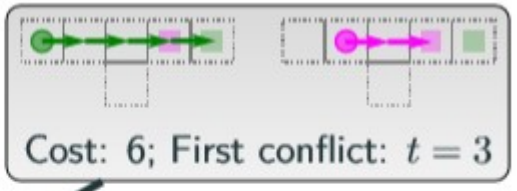
- Find time-minimal paths for both agents independently, which results in a vertex collision in cell D1 at time 3; clearly, the green agent cannot be in cell D1 at time 3 or the violet agent cannot be in cell D1 at time 3

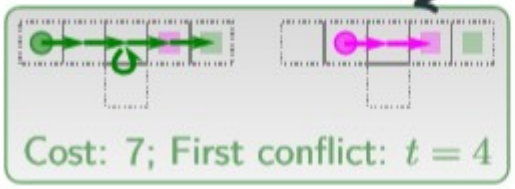
Conflict-Based Search



- Conflict-based search [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible

the green agent is not allowed to be in cell D1 at time 3

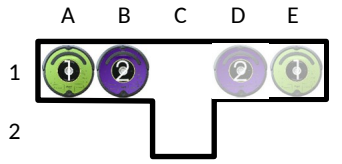




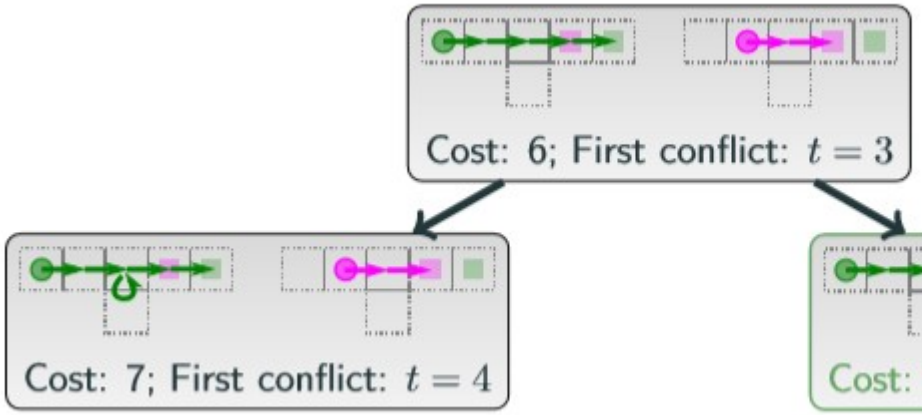
A1, B1, C1, C1 (= wait), D1, E1 B1, C1, D1

- Work on the leaf node with the smallest cost; impose the vertex constraint: the green agent is not allowed to be in cell D1 at time 3; create a new child node, and replan the path of the green agent, which results in a vertex collision in cell D1 at time 4

Conflict-Based Search



- Conflict-based search [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible

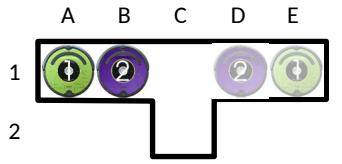


the violet agent is not allowed to be in cell D1 at time 3

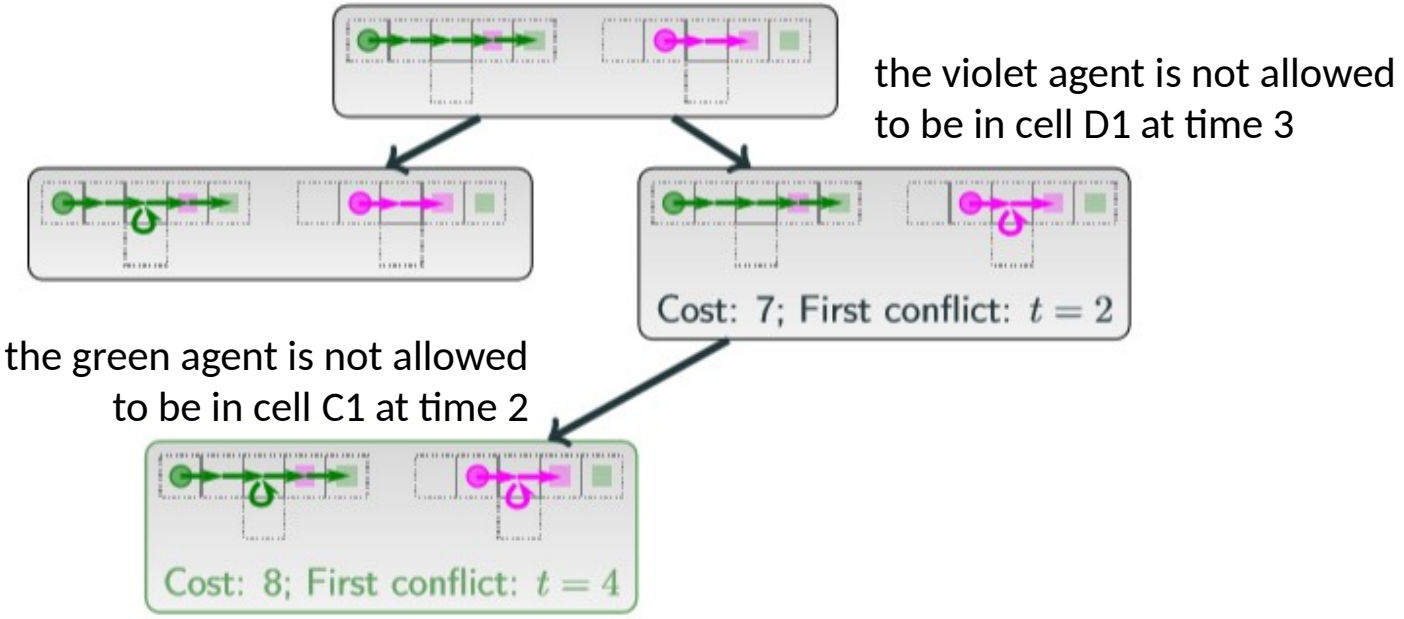
A1, B1, C1, D1, E1 B1, C1, C1 (= wait), D1

- Impose also the vertex constraint: the violet agent is not allowed to be in cell D1 at time 3, create a new child node, and replan the path of the violet agent, which results in a vertex collision in cell C1 at time 2

Conflict-Based Search

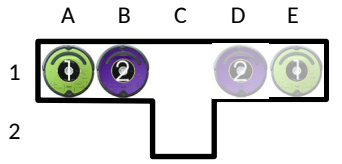


- Conflict-based search [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible

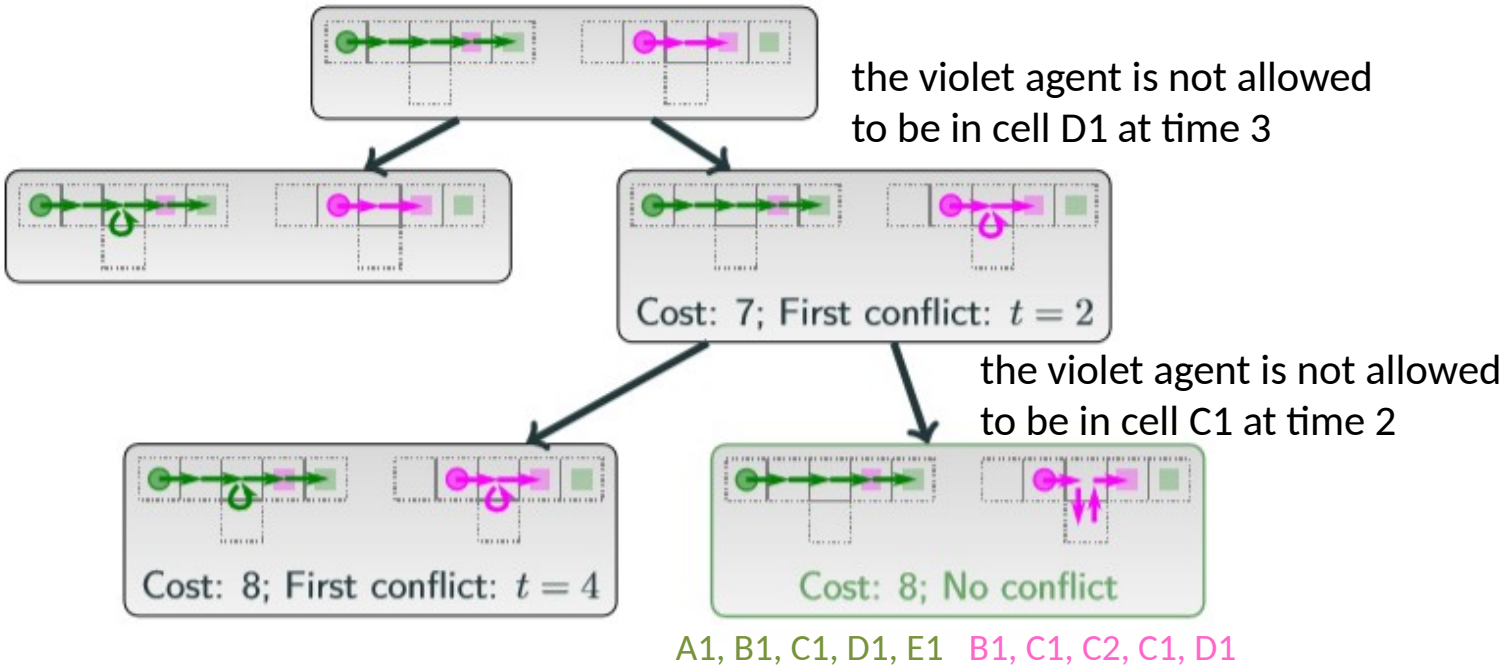


- Work on the leaf node with the smallest cost; impose the vertex constraint: the green agent is not allowed to be in cell C1 at time 2 (in addition to the previous vertex constraint), create a new child new, and replan the path of the green agent, which results in a vertex collision in cell D1 at time 4

Conflict-Based Search

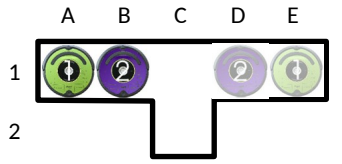


- Conflict-based search [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible

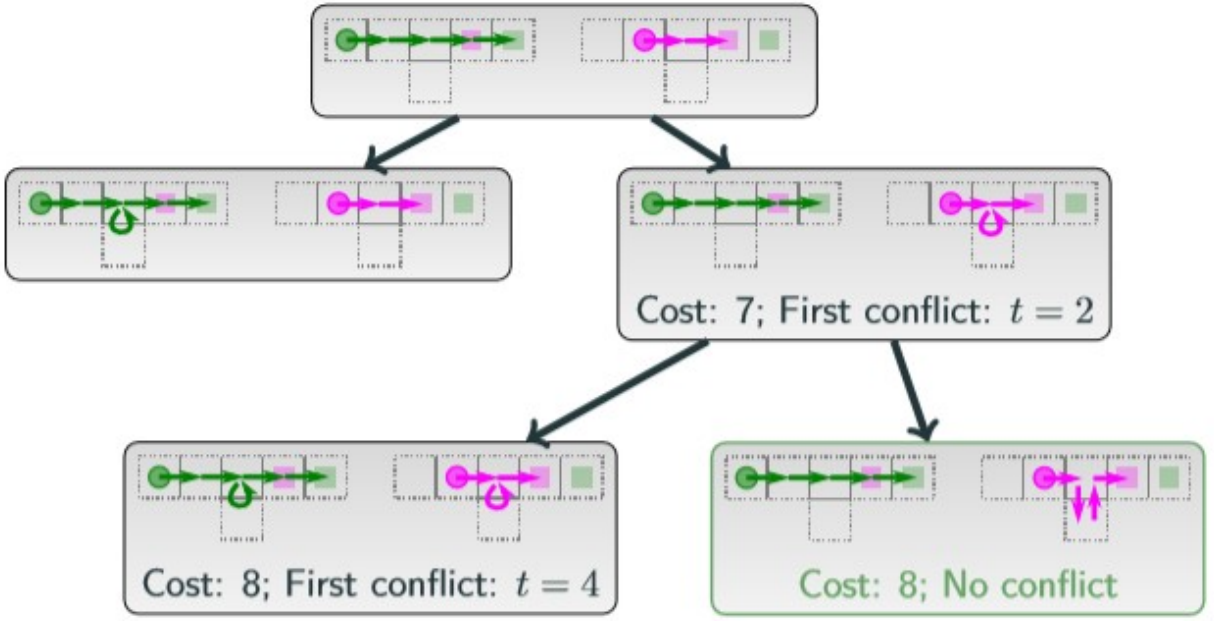


- Impose also the vertex constraint: the violet agent is not allowed to be in cell C1 at time 2 (in additional to the previous vertex constraint), work on the child node with the smallest cost, and replan the path of the violet agent, which results in no vertex or edge collisions

Conflict-Based Search



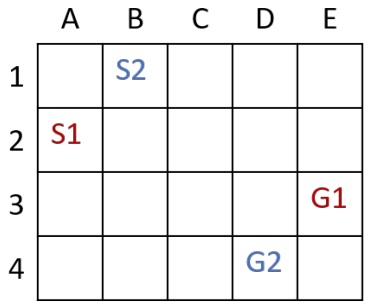
- Conflict-based search [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible



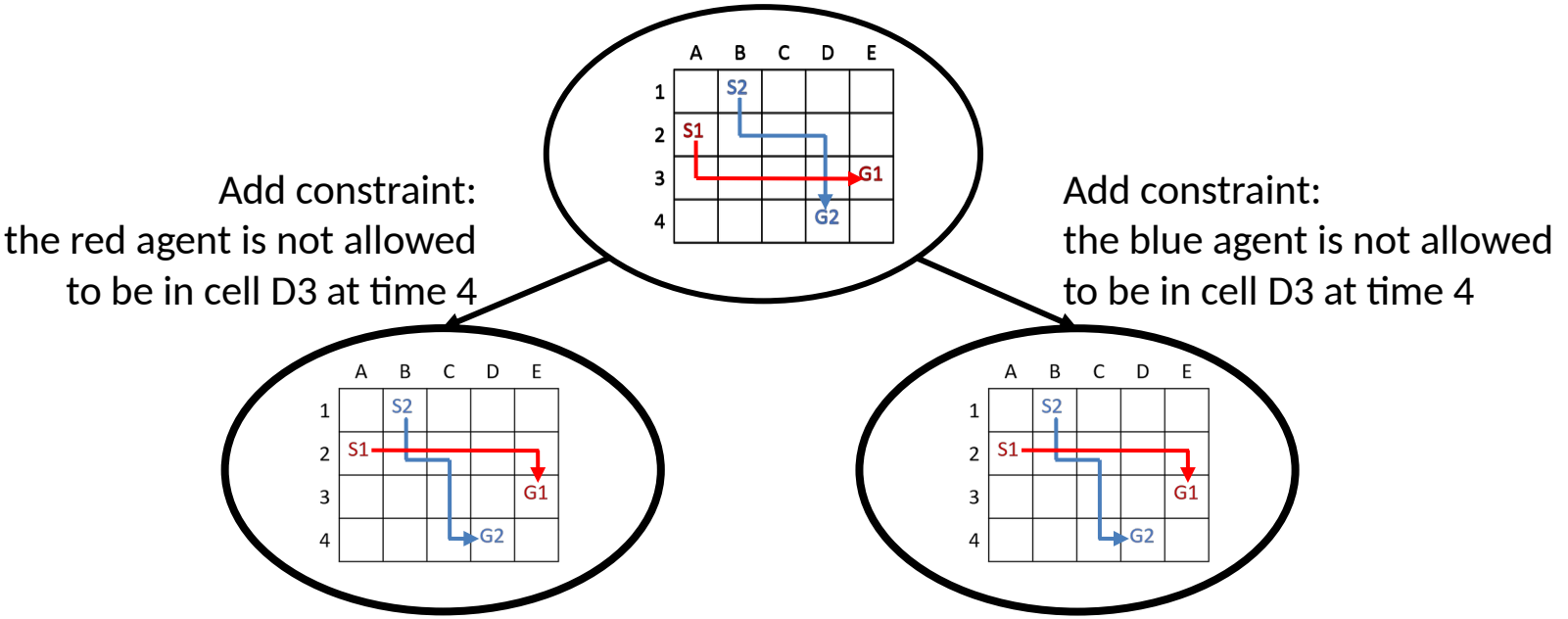
A1, B1, C1, D1, E1 B1, C1, C2, C1, D1

- Work on the leaf node with the smallest cost and terminate since this node has no vertex or edge collisions

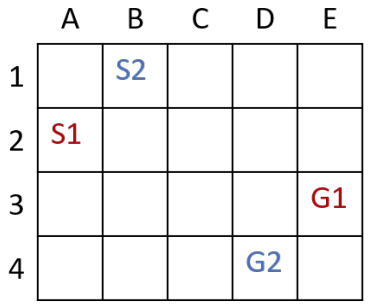
Conflict-Based Search with Disjoint Splitting



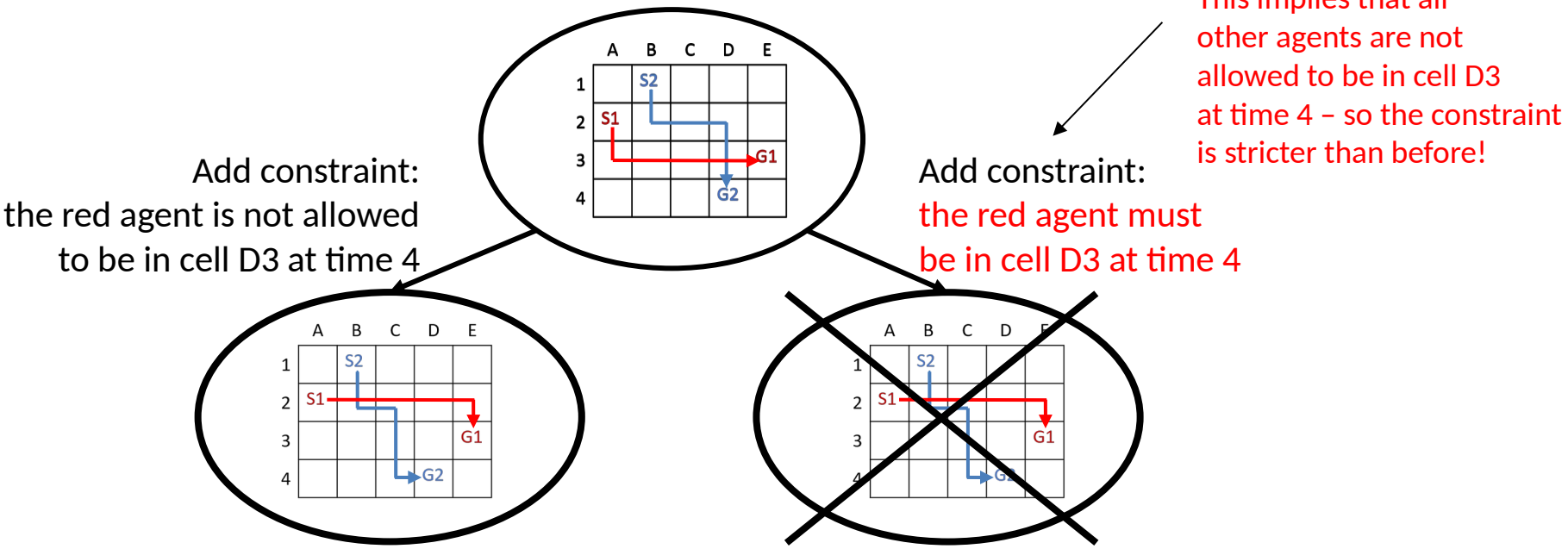
- Conflict-based search (without disjoint splitting) [Sharon, Stern, Felner and Sturtevant]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible



Conflict-Based Search with Disjoint Splitting

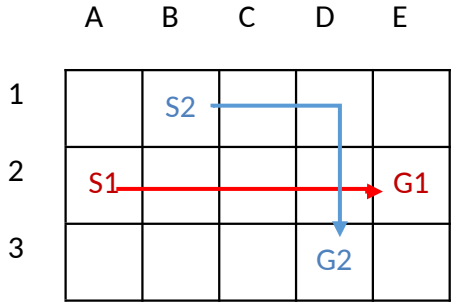


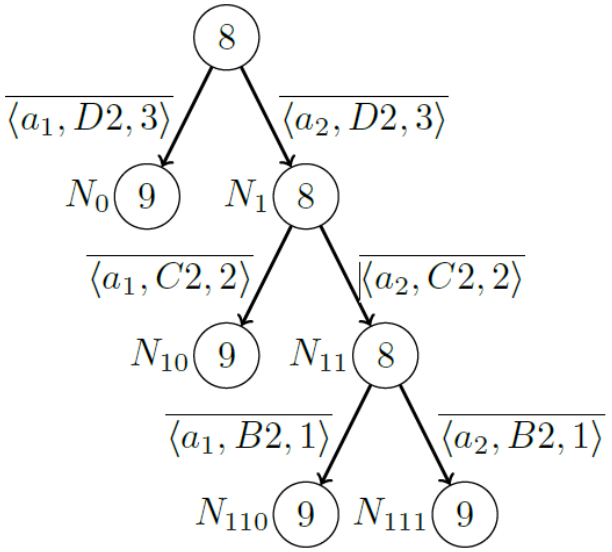
- Conflict-based search with disjoint splitting [Li, Harabor, Stuckey, Felner, Ma and Koenig]: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible



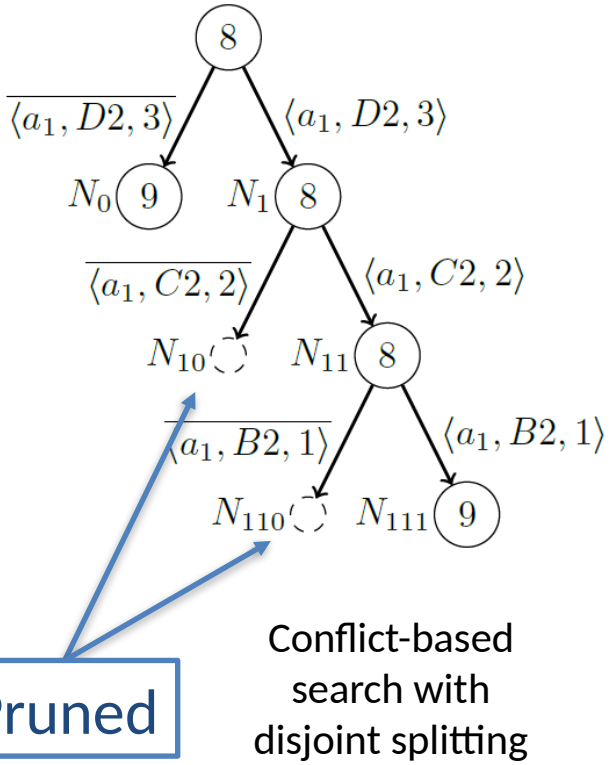
Conflict-Based Search with Disjoint Splitting

- Conflict-based search with disjoint splitting: Optimal (or bounded-suboptimal) MAPF solver that plans for each agent independently, if possible



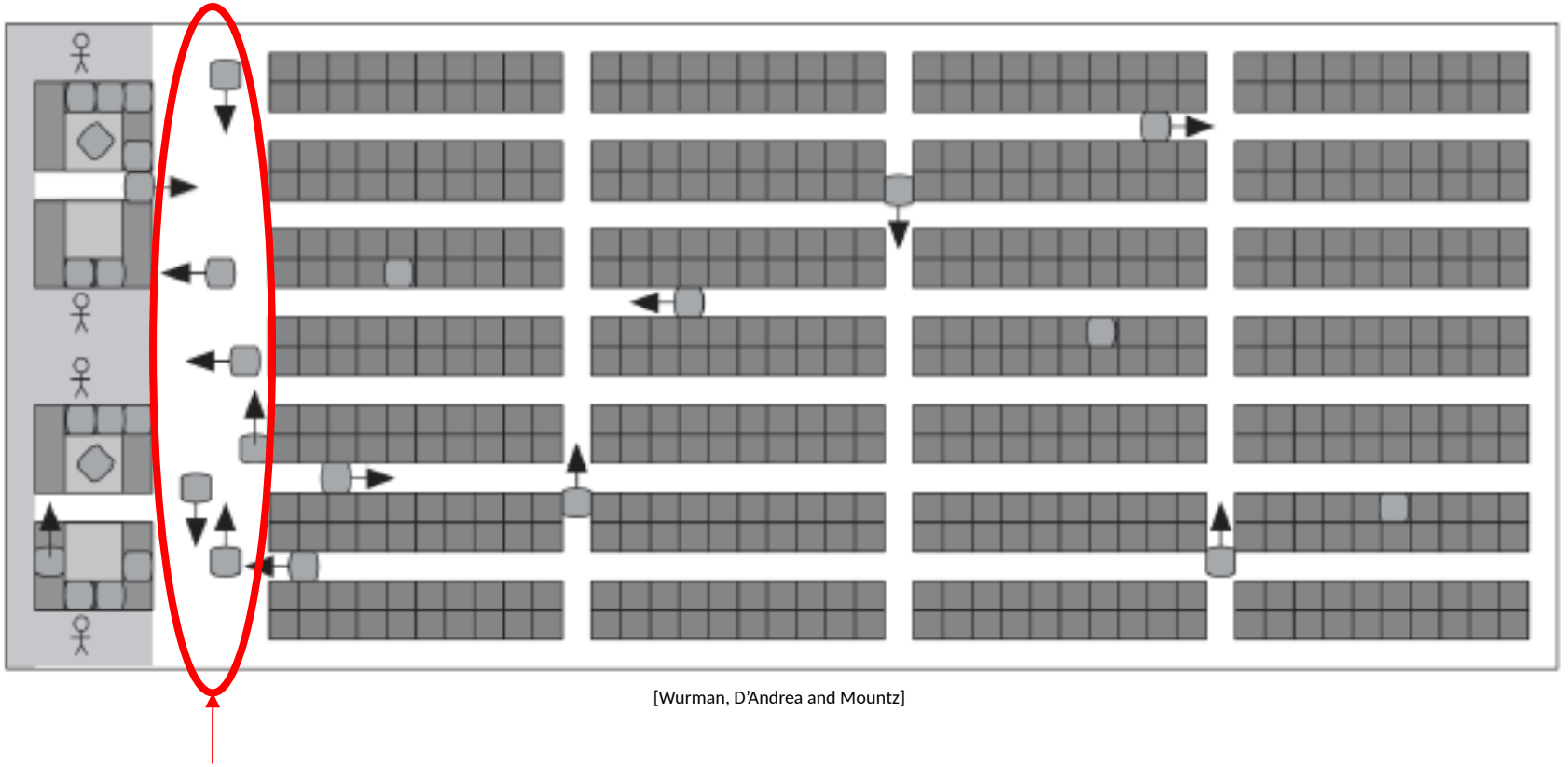


Conflict-based search without disjoint splitting



Conflict-based search with disjoint splitting

Execution of MAPF Plans



[Wurman, D'Andrea and Mountz]

Use the MAPF methods here (in a small area of high congestion but with few agents) rather than over the whole fulfillment center

Execution of MAPF Plans

- Want to learn more about multi-agent path finding?
- Visit: http://mapf.info/

mapf.info webmaster: Sven Koenig

Search Go

Learn all about Multi-Agent Path Finding (MAPF)

Home Page

Benchmarks

Mailing List

Meetings

Publications

Researchers

Software

Tutorials

Welcome!

Multi-Agent Path Finding (MAPF) is the problem of computing collision-free paths for a team of agents from their current locations to given destinations. Application examples include autonomous aircraft towing vehicles, automated warehouse systems, office robots, and game characters in video games. Practical systems must find high-quality collision-free paths for such agents quickly.

Consider, for example, automated warehouses. Path planning for robots in such warehouses is tricky since most warehouse space is used for storage, resulting in narrow corridors where robots cannot pass each other. Warehouse robots operate all day long but a simplified one-shot version of the path-planning problem is the simplest form of a MAPF problem, which can be described as follows: On math paper, some cells are blocked. The blocked cells and the current cells of *n* robots are known. A different unblocked cell is assigned to each of the *n*

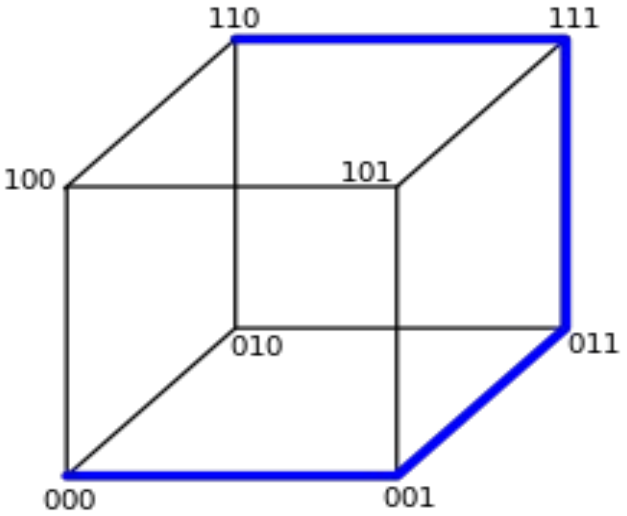
Project

- MAPF project:

  http://idm-lab.org/project-p/project.html

- Read Project text
- Run the Python code
- Add code for the different search algorithms

Snake in the box



- A path such that for every node only two neighbors are in the path.
- Applications: Electrical engineering, coding theory, computer network topologies.
- World records with NRPA [Dang et al. 2023].

Solving the Snake in the Box Problem with Heuristic Search: First Results

Alon Palombo, Roni Stern, Rami Puzis and **Ariel Felner**
Department of Information Systems Engineering
Ben-Gurion University of the Negev
Beer Sheva, Israel
palomboalon@gmail.com
sternron, puzis, felner at *post.bgu.ac.il*

Scott Kiesel and **Wheeler Ruml**
Department of Computer Science
University of New Hampshire
Durham, NH 03824 USA
palomboalon@gmail.com
skiesel and ruml at *cs.unh.edu*

Abstract

Snake in the Box (SIB) is the problem of finding the longest simple path along the edges of an n-dimensional cube, subject to certain constraints. SIB has important applications in coding theory and communications. State of the art algorithms for solving SIB apply uninformed search with symmetry breaking techniques. We formalize this problem as a search problem and propose several admissible heuristics to solve it. Using the proposed heuristics is shown to have a huge impact on the number of nodes expanded and, in some configurations, on runtime. These results encourage further research in using heuristic search to solve SIB, and to solve maximization problems more generally.

Introduction

The longest simple path (LPP) problem is to find the longest path in a graph such that any vertex is visited at most once. This problem is known to be NP-Complete (Garey and Johnson 1990). The *Snake in the Box* (SIB) problem is a special case of LPP in which: (1) the searched graph, denoted Q_n, is an n-dimensional cube, and (2) the only paths allowed are *induced paths* in Q_n. An induced path in a graph G is a sequence of vertices such that every two vertices adjacent in the sequence are connected by an edge in G and every pair of vertices from the sequence that are not adjacent in it are not connected by an edge in G. An optimal solution to SIB is the longest induced path in Q_n. For example, the optimal solution to Q_3 is shown in Figure 1 by the green line (for now, ignore the difference between solid and dashed lines). The *Coil in the Box* (CIB) is a related problem where the task is to find the longest induced cycle in Q_n.

SIB and CIB have important applications in error correction codes and communications. In particular, a snake of length d in an n-dimensional cube represents a special type of *Gray code* that encodes each of the numbers $0, \ldots, d$ to n bits (Kautz 1958).

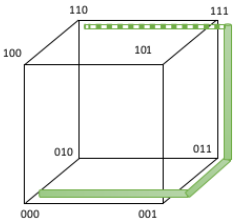


Figure 1: An example of a snake in a 3 dimensional cube.

The mapping between a snake of length d and a code is as follows. Every vertex in the n-dimensional cube is labeled by a binary bit vector of length n (see Figure 1 for an example of this labeling). A snake $s = (v_0, \ldots, v_d)$ represents the code that encodes each number $i \in (0, \ldots, d)$ by the label of v_i. Gray codes generated from snakes are particularly useful for error correction and detection because they are relatively robust to one-bit errors: flipping one bit in the label of v_i is either the label of v_{i-1}, v_{i+1}, or a label not in s. Thus, a one-bit error would result in the worst case in an error of one in the decoded number (mistaking i for $i-1$ or $i+1$).

Due to its importance, SIB and CIB have been studied for several decades (Kautz 1958; Douglas 1969; Abbott and Katchalski 1988; Potter et al. 1994; Kochut 1996; Hood et al. 2011; Kinny 2012). Existing suboptimal solvers use genetic algorithms (Potter et al. 1994) and Monte Carlo tree search algorithms (Kinny 2012). In this work we focus on finding optimal solutions to SIB and CIB. Existing approaches to optimally solving SIB and CIB employ exhaustive search, symmetry breaking, and mathematical bounds to prune the search space (Kochut 1996; Hood et al. 2011).

In this paper we formulate SIB and CIB as heuristic search problems and report initial results on using heuristic search techniques to solve them. SIB and CIB are both maximization problems (MAX problems), and thus search algo-

Snake in the Box

- Admissible heuristic:

  https://ojs.aaai.org/index.php/SOCS/article/download/18357/18148

Snake in the Box

- Exercise:
- Code the state and the possible moves for the Snake in the Box of dimension N.
- Code the admissible heuristic.
- Apply IDA* to dimension 5.

Bases de patterns

- L'analyse rétrograde peut être utilisée pour calculer des bases de patterns qui permettent d'accélérer la résolution de problèmes.
- Un pattern est un sous-ensemble d'une position.
- La fonction h a une grande influence sur l'efficacité de A* : plus les valeurs renvoyées par h sont élevées (tout en restant admissibles), plus A* trouvera la solution rapidement.
- La fonction utilisée traditionnellement pour calculer une heuristique admissible est la distance de Manhattan.
- On peut améliorer h en utilisant les bases de données de patterns.

Bases de patterns

Le Taquin

- Pour construire une base de patterns pour le Taquin, on choisit un sous-ensemble des pièces, et on calcule pour chaque arrangement possible de ce sous ensemble le nombre minimal de déplacements qu'il est nécessaire de faire pour remettre toutes les pièces du sous-ensemble à leurs places.

Bases de patterns

Le Taquin

- Les patterns engendrés donnent un chemin minimal jusqu'à la position finale qui est plus grand que la distance de Manhattan, mais plus petit que le nombre réel.
- Pour évaluer le chemin qui reste à parcourir jusqu'à la position finale depuis une position donnée, le programme reconnaît tous les sous-buts qu'il a calculé sur la position.
- L'heuristique admissible consiste à prendre le maximum de toutes les valeurs calculées pour ces sous-buts.
- L'utilisation de bases de données de patterns permet de résoudre les problèmes de Taquin 4x4 avec mille fois moins de nœuds.

Bases de patterns

Le Taquin 4x4

- On décide de calculer toutes les configurations des 7 premières pièces.
- Quelle est la taille de la base de patterns qu'on va engendrer ?

Bases de patterns

Le Rubik's cube :

- La recherche de solutions optimales au Rubik's Cube est un problème résoluble par IDA*.
- On peut pour cela utiliser des bases de données de patterns afin de mieux évaluer la fonction h

Bases de patterns

Le Rubik's cube :

- On peut créer des patterns qui ne contiennent que les 8 cubes de coins.
- La position et l'orientation du dernier cube est déterminée par la position et l'orientation des cubes précédents.
- Exercice :

  Combien y a-t-il de combinaisons possibles des 8 cubes de coin ?

Bases de patterns

Le Rubik's cube :

- On peut énumérer les patterns dans une base de données et leur associer le nombre de coups nécessaires pour atteindre la position finale.
- Le nombre de coups varie de 0 à 11, on utilise donc 4 bits pour stocker un nombre, la table utilise alors 42 Mo de mémoire.
- La valeur moyenne de h pour cette table est de 8.764 comparée à 5.5 pour la distance de Manhattan.
- Exercice :

  Combien y a-t-il de combinaisons possibles pour des patterns composés de 6 des 12 cubes de bord ?

Bases de patterns

Le Rubik's cube :

- Pour utiliser ces deux heuristiques à la fois, la seule façon de faire est de prendre le maximum des deux pour h.
- La combinaison d'IDA* et des bases de patterns permet de résoudre optimalement pratiquement tous les problèmes de Rubik's cube.
- Sans les bases de patterns la résolution est beaucoup plus lente.

Bases de patterns

Les bases de patterns additives :

- On peut faire mieux que prendre le maximum des distances précalculées.
- Si deux bases de patterns ne contiennent que des pièces différentes, on peut additionner les distances.
- Exercice :

  Trouver une heuristique additive simple pour le Taquin 4x4.

Bases de patterns

Les bases de patterns additives :

- Une heuristique admissible au Taquin 4x4 consiste à additionner la distance précalculée des 7 premières pièces à celle des 7 dernières pièces et à la distance de Manhattan de la dernière pièce.

Bases de patterns

La compression de bases de patterns :

- Lorsqu'une base de patterns a une taille trop grande pour être contenue en mémoire vive, on peut la compresser.
- On choisit un sous ensemble des pièces de la base la plus grande, et on calcule pour chaque configuration du sous ensemble la distance minimale sur toutes les configurations du sur ensemble contenant la configuration du sous ensemble.
- Par exemple, si on a calculé toutes les distances des 9 premières pièces du Taquin, on calcule pour chaque configuration de 7 pièces la distance minimum sur toutes les configurations à 9 pièces qui contiennent la configuration à 7 pièces.
- Cette distance compressée est plus grande que la distance de la base de 7 pièces et donne donc une meilleure heuristique admissible.

Bases de patterns

Sokoban :

- Il n'est possible de pousser qu'une seule caisse à la fois.
- Si deux caisses sont voisines au bord cela forme un interblocage, aucune des deux caisses ne peut plus être déplacée et le problème devient insoluble.
- Il existe une multitude d'autres interblocages.
- Exercice : trouver un pattern d’interblocage.

Bases de patterns

Sokoban :

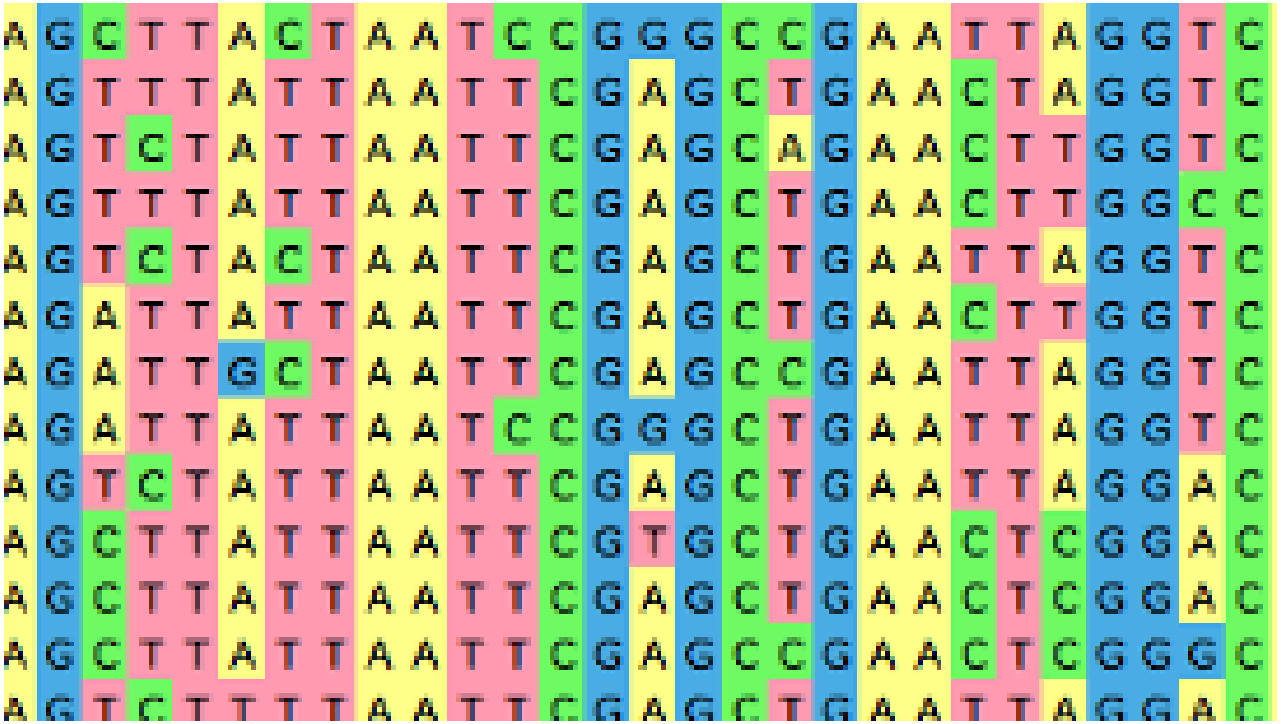
- Une façon de se prémunir contre les interblocages d'un labyrinthe et de les énumérer avec de l'analyse rétrograde et de construire une base de patterns des interblocages spécifiques à un labyrinthe.
- Détecter ainsi les interblocages permet à IDA* d'être plus efficace en arrêtant la recherche dès qu'une position est prouvée insoluble.

Bases de patterns

Le Taquin 4x4

- Exercice : Écrire un algorithme qui permet d'engendrer la base de patterns des 7 premières pièces du Taquin 4x4.
- Écrire tout d'abord les variables nécessaires.
- On veut aussi une fonction qui code une position, c'est à dire qui fasse une bijection entre entiers et positions.
- Écrire ensuite un algorithme qui engendre toutes les configurations possibles et qui teste tous les coups possibles de chaque configuration pour trouver si la configuration est à une distance donnée.
- Écrire ensuite la fonction principale d'analyse rétrograde.

Multiple Sequence Alignment



Multiple Sequence Alignment

- Dynamic programming enables to compute in advance the cost of the optimal alignment of two sequences.
- The result is in a matrix of costs.
- Each cell of the matrix is associated to indices in the two sequences.
- The value in the matrix are computed from the bottom right to the upper left.

Multiple Sequence Alignment

- A move in diagonal in the matrix is the alignment of two letters.
- A move down and a move right are the alignement of a gap and of a letter.
- Compute by hand the matrix for two given sequences.

Multiple Sequence Alignment

- Dynamic programming :

	A	C	G	T	_
A					
A					
G					
T					
_					

Multiple Sequence Alignment

- Dynamic programming :

	A	C	G	T	_
A					4
A					3
G					2
T					1
_	4	3	2	1	0

Multiple Sequence Alignment

- Dynamic programming :

	A	C	G	T	_
A	1	2	2	3	4
A	1	1	1	2	3
G	2	1	0	1	2
T	3	2	1	0	1
_	4	3	2	1	0

Partial Move A*

Tristan Cazenave
LAMSADE
Université Paris-Dauphine
Paris, France

***Abstract*—Some shortest path problems that can be solved using the A* algorithm have a large branching factor due to the combination of multiple choices at each move. Multiple sequence alignment and multi-agent pathfinding are examples of such problems. If the search can be stopped after each choice instead of being stopped at each combination of choices, it takes much less memory and much less time. The goal of the paper is to show that Partial Move A* is much better than A* for these problems when the branching factor is large due to a large combination of choices at each move. When there is such a large combination of choices at each move, Partial Move A* can yield large memory gains and speedups over regular A*.**

I. Introduction

When the moves of a problem contain multiple choices, the evaluation of each choice separately can be interesting instead of evaluating all the possible combinations of choices. When each choice is evaluated separately the search can be stopped at a partial move instead of being stopped at each combination that contains this partial move. Partial Move A* (PMA*) is an iterative deepening A* algorithm [1] that stops search inside a move. It can yield large memory gains and speedups on difficult problems. The algorithm can be applied to multiple sequence alignment and to multi-agent pathfinding. We show in this paper that it enables large memory gains and large speedup on these two problems when the branching factor is large.

Multiple sequence alignment is an NP-hard problem that can be addressed with the A* algorithm [2]. It is commonly believed that iterative deepening A* is not appropriate for multiple sequence alignment since there are many paths passing through the same nodes, with the exception of the work on iterative deepening dynamic programming [3]. We show that PMA* is appropriate for multiple sequence alignment even if it is an iterative deepening A* algorithm.

The multi-agent pathfinding problem is PSPACE-hard [4]. Existing literature on the multi-agent pathfinding problem mainly deals with inexact algorithm as the problem is considered intractable. An example of an algorithm combining individual paths is cooperative pathfinding [5]. We are interested in exact algorithms for this problem. The standard exact algorithm is A*. PMA* improves much on A* search for this problem. PMA* is complete and optimal for the multi-agent pathfinding problem. There are good approximation algorithms for this problem such as [6], however these algorithm are not complete nor optimal.

An algorithm related to PMA* is Partial Expansion A* [7]. Partial Expansion A* uses less memory than A* but uses more time, whereas PMA* uses less memory than A* and less time on difficult problems. PMA* also uses less memory and less time than Partial Expansion A*.

The idea underlying PMA* is that it is very beneficial both for memory consumption and for speed to be able stop search at each move component. For example in multiple sequence alignment it is important to be able to stop search after each choice of either aligning a letter in a sequence or inserting a gap in the same sequence. Concerning multi-agent pathfinding it is important to be able to stop search after the move of each agent and not only after all the agents have moved.

The outline of this paper is as follows: section 2 compares A*, Partial Expansion A* and Partial Move A*. Section 3 presents the application of PMA* to multiple sequence alignment and section 4 presents its application to multi-agent pathfinding.

II. Comparison of algorithms for A*

In this section the differences between A*, Partial Expansion A* and PMA* are outlined.

A. A*

In A*, all the children of a node are added to the Open structure, and the node is placed in the closed structure after expansion.

B. Partial Expansion A*

Partial expansion A* does not expand all the children of a node. It selects the children that have a F value smaller than the F value of the node plus a constant C.

It defines two sets :

$SUCC_{\leq C} \leftarrow \{n_j | n_j \in succ(n), f(n_j) \leq F(n) + C\}$

$SUCC_{>C} \leftarrow \{n_k | n_k \in succ(n), f(n_k) > F(n) + C\}$

It only expands the children in $SUCC_{\leq C}$ and puts the node n in the closed nodes only if $SUCC_{>C} = \emptyset$.

Each time the algorithm comes again on an already partially expanded node, it has to develop again the node in order to build the SUCC sets of nodes. When there are many sequences, the computing time of the algorithm is dominated by the computation of the SUCC sets.

C. Partial Move A*

When running Partial Expansion A* on problems with a large number of children for each node, most of the time of the algorithm is consumed in finding all the possible children of a node and computing the sets $SUCC_{\leq C}$ and $SUCC_{>C}$. For

Partial Move A*

- Combination of actions in a state.
- Examples :
  - Multiple Sequence Alignment
  - Multi-Agent Pathfinding
- Partial Move A*:

  Stop search after each action instead of stopping the search after each combination of actions.

Partial Move A*

- A* with a combination of 3 actions :

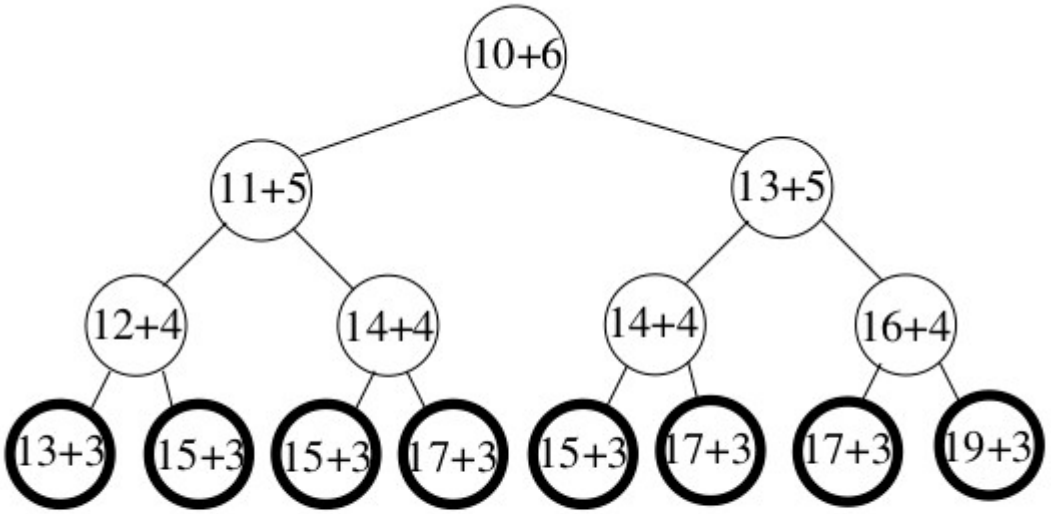


Fig. 1. The expansion behavior of A*.

Partial Move A*

- Partial Expansion A* with a combination of 3 actions:

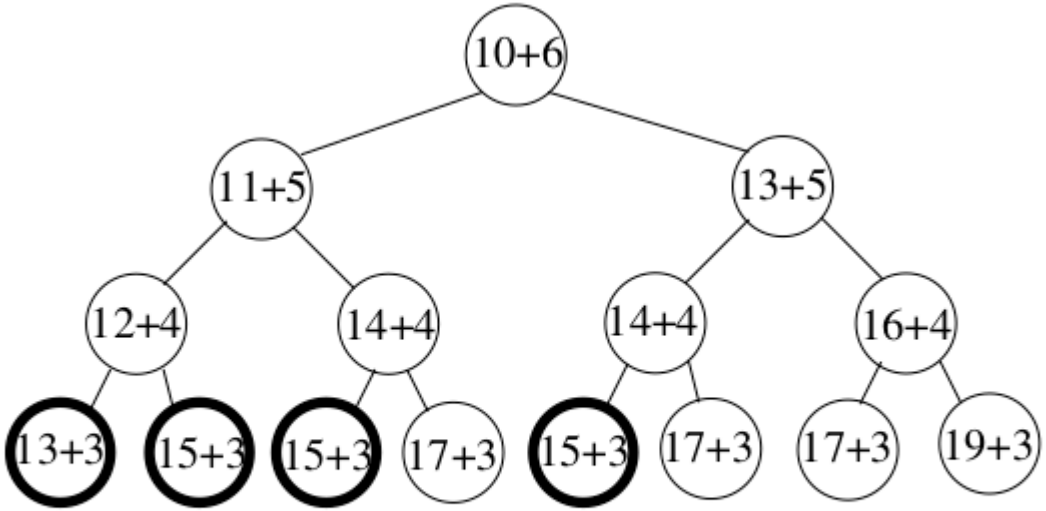


Fig. 2. The expansion behavior of Partial Expansion A*.

Partial Move A*

- Partial Move A* with a combination of 3 actions:

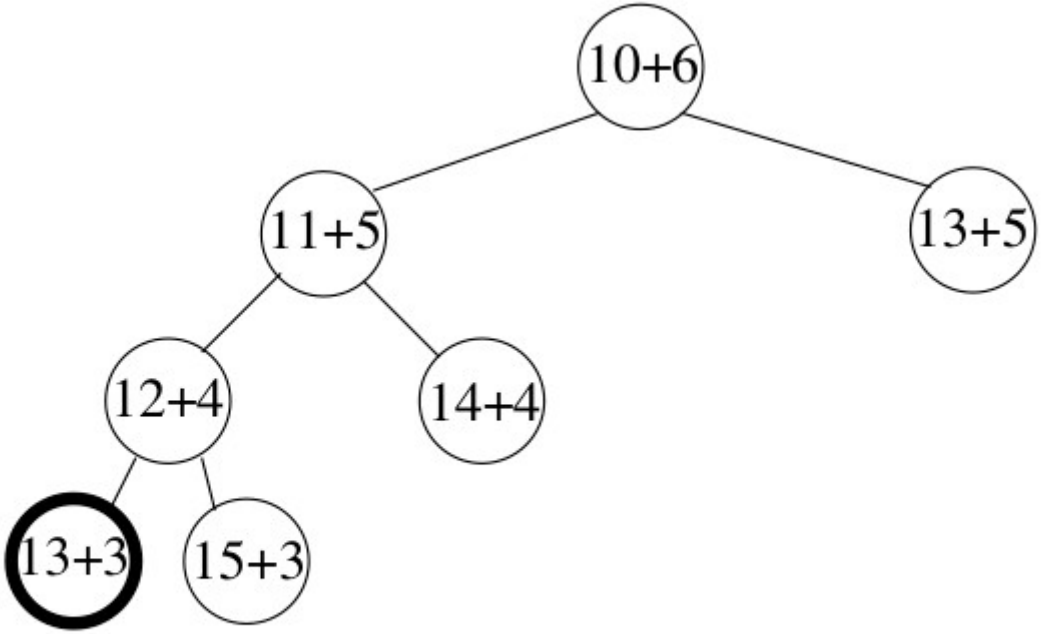


Fig. 3. The expansion behavior of Partial Move A*.

Partial Move A*

Algorithm 1 Pseudo-code for Partial Move A*.

```
PMA (int g, int index, position previous, position current)
f = g + h (index, previous, current)
if f > f of current iteration then
   return false;
end if
if index = number of steps in a move then
   if h (current) = 0 then
      return true;
   end if
   if currentPosition already visited with a smaller g then
      return false;
   end if
   add current to already visited
   index ← 0
   previous ← current
end if
for all possible partial moves do
   compute δg the increase in g due to the partial move
   update current with the partial move
   if PMA(g + δg, index + 1, previous, current) then
      return true;
   end if
end for
return false
```

Partial Move A*

Algorithm 2 Pseudo-code for iterative deepening Partial Move A*.

```
iterativePMA ()
f of current iteration = h (root)
while true do
    init hash table
    add root to already visited
    if PMA (0, 0, root, root) then
        return f of current iteration;
    end if
    f of current iteration = f of current iteration + 1
end while
```

Partial Move A*

		a_1	a_2	a_3	a_4		
		a_5	a_6	a_7	a_8		

TABLE IV
EXCHANGING PLACES FOR 8 AGENTS.

Agents	A* nodes	PMA* nodes	gain
2	216	11	19.64
4	30,713	181	169.69
6	5,233,301	3,671	1425.58
8	>40,000,000	79,631	>502.32
	A* time	PMA* time	speedup
2	0.003281 s.	0.000793 s.	4.14
4	0.011146 s.	0.001565 s.	7.12
6	236.91 s.	0.038093 s.	6,219.25
8	>8,599 s.	2.46 s.	>3,495.53

TABLE V
OPTIMAL EXCHANGING PLACES FOR DIFFERENT NUMBERS OF AGENTS.

Partial Move A*

		a_1	a_2	a_3	a_4		
		a_5	a_6	a_7	a_8		

TABLE IV
EXCHANGING PLACES FOR 8 AGENTS.

Agents	A* nodes	PMA* nodes	gain
2	216	11	19.64
4	30,713	181	169.69
6	5,233,301	3,671	1425.58
8	>40,000,000	79,631	>502.32
	A* time	PMA* time	speedup
2	0.003281 s.	0.000793 s.	4.14
4	0.011146 s.	0.001565 s.	7.12
6	236.91 s.	0.038093 s.	6,219.25
8	>8,599 s.	2.46 s.	>3,495.53

TABLE V
OPTIMAL EXCHANGING PLACES FOR DIFFERENT NUMBERS OF AGENTS.

Partial Move A*

g_5							
		g_4					
a_2							a_0
	a_3	a_6	g_7	a_5		g_2	
					a_1	a_4	a_7
		g_3				g_1	g_6

TABLE X
THE PROBLEM WITH MAXIMAL SPEEDUP AND GAIN.

Partial Move A*

Agents	mean gain	median gain	max gain
2	18.53	19.29	46.00
3	86.39	94.50	193.00
4	364.93	415.62	1,248.50
5	1,586.63	1,827.88	8,339.25
6	3,325.39	7,608.50	89,977.16
7	14,438.44	23,743.57	602,400.31
8	>26,698.17	122,229.20	4,125,671.00

TABLE VIII
MEMORY GAINS FOR UP TO 8 AGENTS.

Agents	mean speedup	median sp.	max sp.
2	14.73	18.84	33.74
3	11.20	16.84	24.66
4	17.10	19.22	24.12
5	28.07	26.28	132.87
6	1,672.62	59.72	20,109.39
7	94,771.70	697.16	288,140.06
8	>184,804.06	18,519.66	11,438,055.00

TABLE IX
SPEEDUPS FOR UP TO 8 AGENTS.

Partial Move A*

- Goal : Move 5 agents.
- Random start and goal points on a 8x8 map.
- Heuristic : Manhattan distance.
- Compare Partial Move A* to A*.

Partial Move A*

Algorithm 3 Pseudo-code for computing the admissible heuristic at a partial move.

```
h (int index, position previous, position current)
h = 0
for i from 0 to nbSequences - 1 do
   for j from i + 1 to nbSequences - 1 do
      if ((i < index) and (j ≥ index)) then
         increase h by the minimum cost between inserting a
         gap in sequence j and moving forward in sequence
         j plus the pairwise heuristic
      else
         increase h by the pairwise heuristic
      end if
   end for
end for
return h
```

Partial Move A*

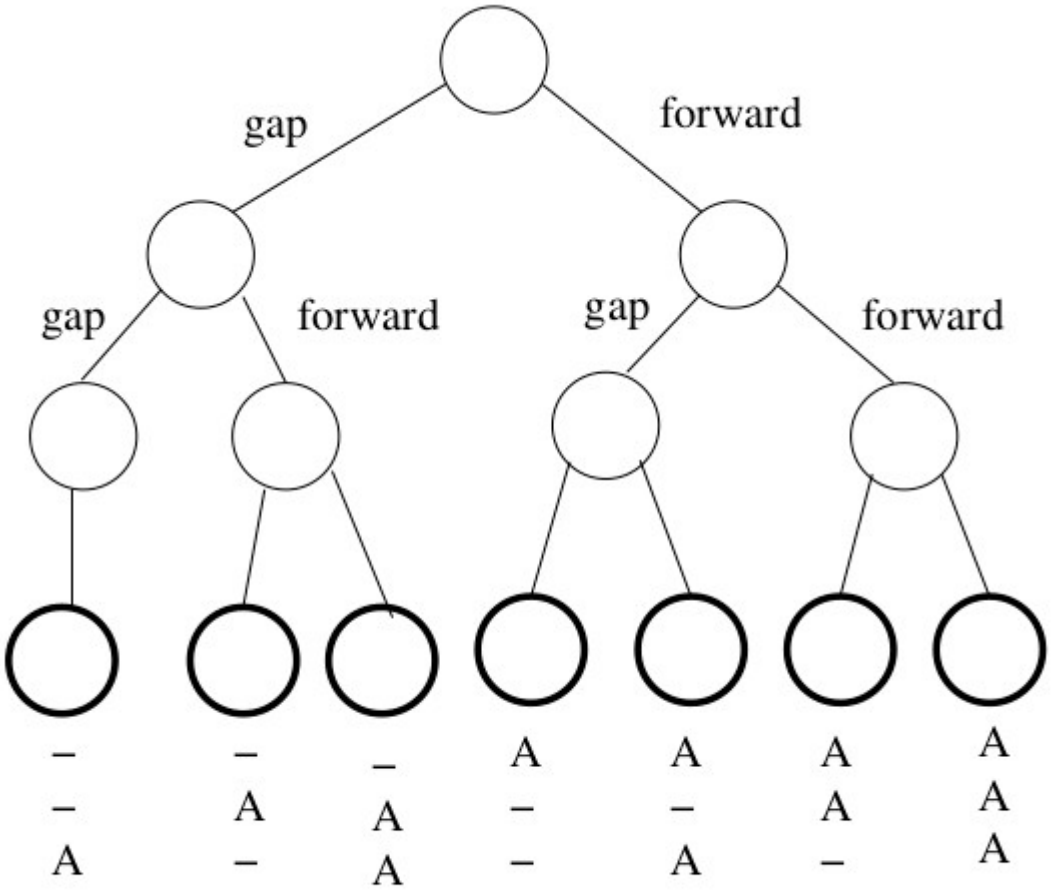


Fig. 4. Behavior of A* for the first move of the sequence.

Partial Move A*

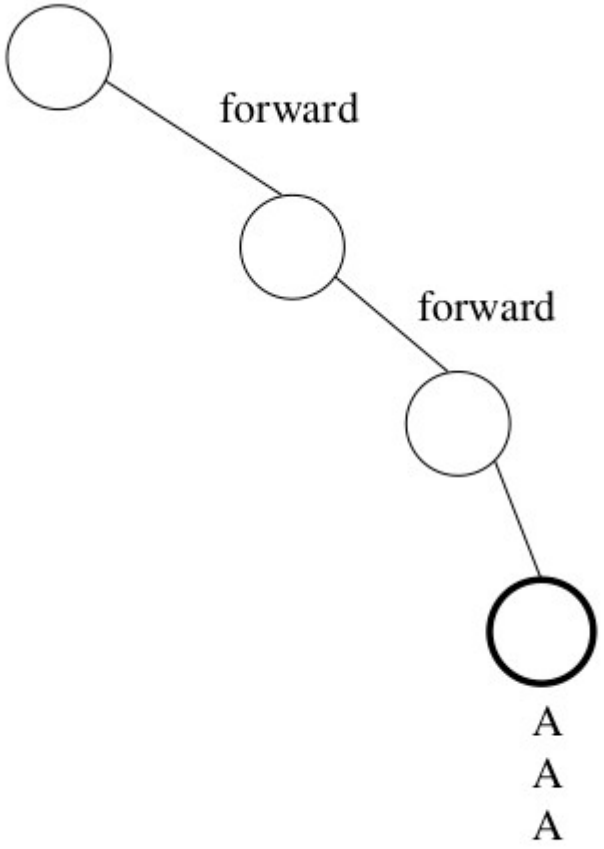


Fig. 5. Behavior of PMA* for a f threshlod of 0.

Partial Move A*

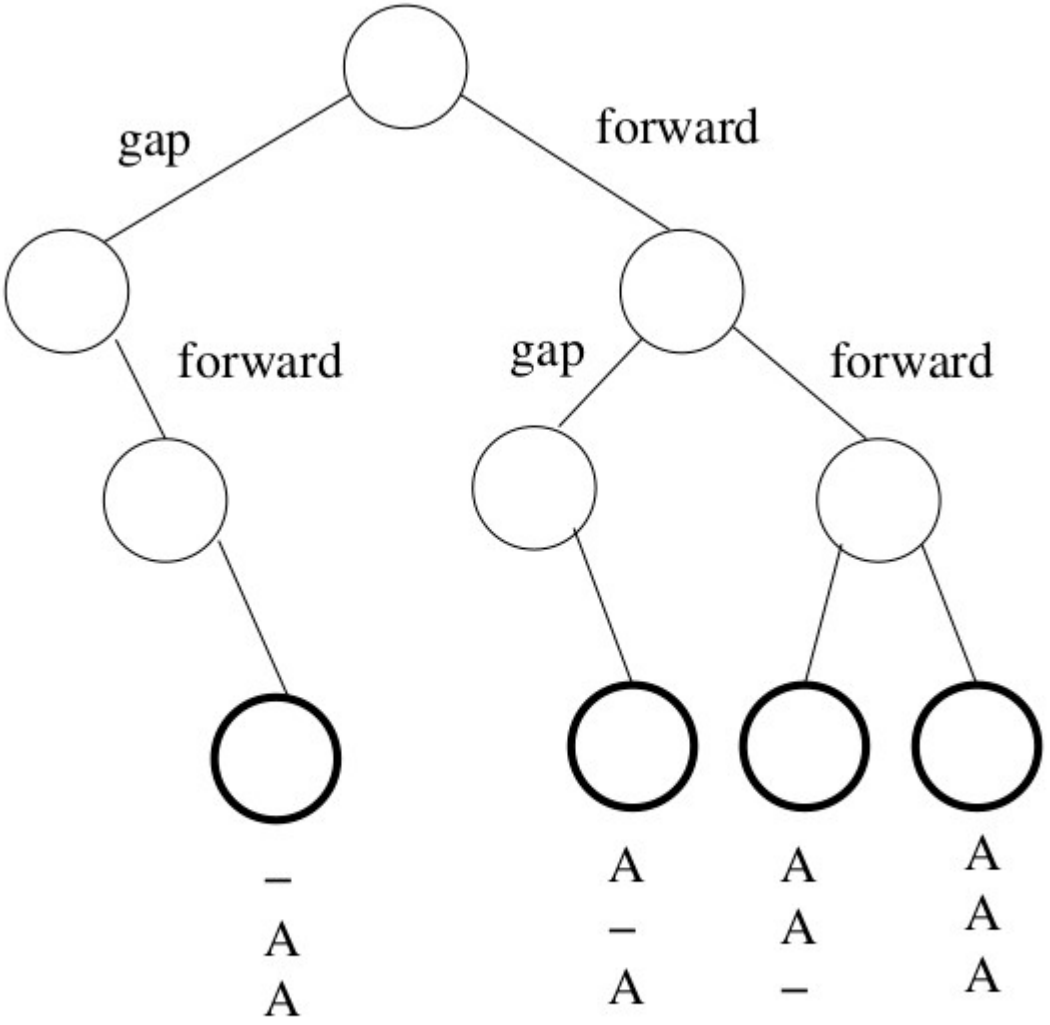


Fig. 6. Behavior of PMA* for a f threshold of 2.

Partial Move A*

name	*s*	*algo.*	*time*	*nodes*
2trx-ref2	20	PMA*	19,043.88	2,021,479
2trx-ref2	19	PMA*	500.66	95,732
2trx-ref2	18	PMA*	9.95	3,718
2trx-ref2	17	PMA*	1.18	769
2trx-ref2	16	PMA*	0.44	452
2trx-ref2	16	A*	138.03	9,244,517
1tvxA-ref2	15	PMA*	15.08	15,779
1tvxA-ref2	15	A*	>242.42	>10,000,000
1tvxA-ref2	14	PMA*	0.99	1,587
1tvxA-ref2	14	A*	147.73	7,490,488
1tvxA-ref2	13	PMA*	0.44	908
1tvxA-ref2	13	A*	13.28	2,127,596
1tvxA-ref2	12	PMA*	0.09	309
1tvxA-ref2	12	A*	0.93	509,547
1tvxA-ref2	11	PMA*	0.03	178
1tvxA-ref2	11	A*	0.22	202,299
1aboA-ref1	5	PMA*	8.21	414,817
1aboA-ref1	5	A*	2.23	475,654

TABLE II
RESULTS FOR SEQUENCES FROM BALIBASE.

name	*s*	*speedup*	*gain*
2trx-ref2	16	313.70	20,452.47
1tvxA-ref2	14	149.22	4,719.90
1tvxA-ref2	13	30.18	2,343.17
1tvxA-ref2	12	10.33	1,649.02
1tvxA-ref2	11	7.33	1,136.51
1aboA-ref1	5	0.27	1.15

TABLE III
SPEEDUP AND GAIN FOR SEQUENCES FROM BALIBASE.

Multiple Sequence Alignment

- Goal : align 3 DNA sequences.
- Heuristic : h = 0
- Cost of gap = 2
- Cost of misalignment = 1
- Cost of alignement = 0
- Compare Partial Move A* to A*.

Multiple Sequence Alignment

- Goal : align 3 DNA sequences.
- Compute the dynamic programming matrices for all pair of sequences.
- Use precomputed matrices to compute an admissible heuristic.
- Compare Partial Move A* to A*.

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))

STRIPS

- 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))

STRIPS

- 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) ^ Bloc (A) ^ Bloc (B) ^
  Bloc (C))

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

STRIPS

- 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

- 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.
- Effectuer la recherche en 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 :
- Ecrire le graphe de planification pour le problème du fret avec deux avions.

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.