Résolution d'un problème grace à la recherche dans un graphe

M1 Miage 2015–2016 *Intelligence Artificielle*

Stéphane Airiau

LAMSADE

Pourquoi avoir un cours d'IA

- Qu'est-ce que l'IA ?
  question difficile, les contours de l'IA sont parfois assez flous
- Qu'est ce que l'IA fait pour vous ?
  - traduction automatique (par exemple Mandarin → Français)
  - identification d'objet dans des images (ex : chaises, visages, etc)
  - plier votre linge, voiture sans conducteur
  - démontrer ou aider à démontrer de nouveaux théorèmes
  - assistants personnels (ex siri)
  - robots (aides aux personnes âgées, aides musées)

Tracks at IJCAI 2015

International Joint Conference on Artificial Intelligence

Agent and Multiagent Systems
Constraint Optimization
Ontologies
Game Theory
Heuristic Search
Graphical Models
Robotics and Vision
Natural Language Processing
Planning
Relational Learning
Satisfiability
Social Choice Theory
Vision and Perception
Web Mining

Artificial Intelligence and Social Sciences
Auctions and Market-Based Systems
Distributed Search/CSP/Optimization
Knowledge Representation, Reasoning, and Logic
Knowledge Acquisition
Machine Learning
Multidisciplinary Topics and Applications
Constraints, Satisfiability, and Search
Recommender Systems
Model Verification / Model Checking
Sequential Decision Making
Social Networks
Web and Knowledge-Based Information Systems
Special Track on Artificial Intelligence and the Arts

Special Track on Computational Sustainability
Special Track on Knowledge Representation and Reasoning
Special Track on Machine Learning

Tracks at IJCAI 2015

Economie ?

Agent and Multiagent Systems
Constraint Optimization
Ontologies
Game Theory
Heuristic Search
Graphical Models
Robotics and Vision
Natural Language Processing
Planning
Relational Learning
Satisfiability
Social Choice Theory
Vision and Perception
Web Mining

Artificial Intelligence and **Social Sciences**
Auctions and Market-Based Systems
Distributed Search/CSP/Optimization
Knowledge Representation, Reasoning, and Logic
Knowledge Acquisition
Machine Learning
Multidisciplinary Topics and Applications
Constraints, Satisfiability, and Search
Recommender Systems
Model Verification / Model Checking
Sequential Decision Making
Social Networks
Web and Knowledge-Based Information Systems
Special Track on Artificial Intelligence and the Arts

Special Track on Computational Sustainability
Special Track on Knowledge Representation and Reasoning
Special Track on Machine Learning

Tracks at IJCAI 2015

Logique ?

Agent and Multiagent Systems
Constraint Optimization
Ontologies
Game Theory
Heuristic Search
Graphical Models
Robotics and Vision
Natural Language Processing
Planning
Relational Learning
Satisfiability
Social Choice Theory
Vision and Perception
Web Mining

Artificial Intelligence and Social Sciences
Auctions and Market-Based Systems
Distributed Search/CSP/Optimization
Knowledge Representation, Reasoning, and Logic
Knowledge Acquisition
Machine Learning
Multidisciplinary Topics and Applications
Constraints, Satisfiability, and Search
Recommender Systems
Model Verification / Model Checking
Sequential Decision Making
Social Networks
Web and Knowledge-Based Information Systems
Special Track on Artificial Intelligence and the Arts

Special Track on Computational Sustainability
Special Track on Knowledge Representation and Reasoning
Special Track on Machine Learning

Tracks at IJCAI 2015

Optimisation Combinatoire ?

Agent and Multiagent Systems
Constraint Optimization
Ontologies
Game Theory
Heuristic Search
Graphical Models
Robotics and Vision
Natural Language Processing
Planning
Relational Learning
Satisfiability
Social Choice Theory
Vision and Perception
Web Mining

Artificial Intelligence and Social Sciences
Auctions and Market-Based Systems
Distributed Search/**CSP/Optimization**
Knowledge Representation, Reasoning, and Logic
Knowledge Acquisition
Machine Learning
Multidisciplinary Topics and Applications
Constraints, Satisfiability, and Search
Recommender Systems
Model Verification / Model Checking
Sequential Decision Making
Social Networks
Web and Knowledge-Based Information Systems
Special Track on Artificial Intelligence and the Arts

Special Track on Computational Sustainability
Special Track on Knowledge Representation and Reasoning
Special Track on Machine Learning

Tracks at IJCAI 2015

Agent and Multiagent Systems
Constraint Optimization
Ontologies
Game Theory
Heuristic Search
Graphical Models
Robotics and Vision
Natural Language Processing
Planning
Relational Learning
Satisfiability
Social Choice Theory
Vision and Perception
Web Mining

Artificial Intelligence and Social Sciences
Auctions and Market-Based Systems
Distributed Search/CSP/Optimization
Knowledge Representation, Reasoning, and Logic
Knowledge Acquisition
Machine Learning
Multidisciplinary Topics and Applications
Constraints, Satisfiability, and Search
Recommender Systems
Model Verification / Model Checking
Sequential Decision Making
Social Networks
Web and Knowledge-Based Information Systems
Special Track on Artificial Intelligence and the Arts

Special Track on Computational Sustainability
Special Track on Knowledge Representation and Reasoning
Special Track on Machine Learning

Ressources

Ce cours suit en partie le livre suivant :

Artificial Intelligence : A Modern Approach
de *Stuart Russel et Peter Norvig*
Pearson, 2009

Ce livre est le classique pour enseigner l'intelligence artificielle actuellement dans le monde.

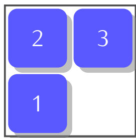
Les chapîtres et sections seront indiquées sur les transparents.

Résoudre des problèmes en cherchant dans un graphe d'état :

l'ntelligence de la force brute

AIMA chapître 2, Sections 1–4

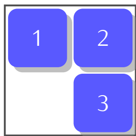
Exemple d'un jeu à résoudre : le taquin



taquin 2×2

Un taquin $n \times n$ est constitué de $n^2 - 1$ jetons carrés à l'intérieur d'un carré pouvant contenir n^2 jetons : on a donc une **case vide**.
Les coups permis sont ceux qui font glisser dans la case vide un des 2, 3 ou 4 jetons contigus.

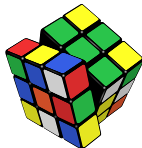
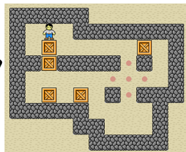
But : partir de l'état du taquin ci-dessus pour arriver à l'état du taquin ci-dessous.



Un programme peut il résoudre ce type de problèmes pour n=2,3,4,5... ?

Jeux dans le même esprit

- poser n reines sur un échiquier de taille $n \times n$ sans qu'aucune reine n'attaque une autre reine.
- résoudre un "rubik's cube"

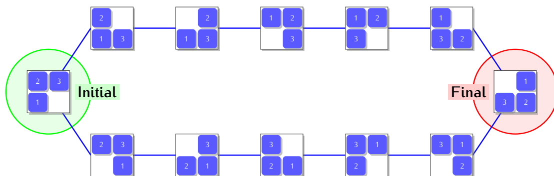
- résoudre un niveau de "Sokoban"
- jouer "aux chiffres" de l'emission des chiffres et des lettres.
- trouver un itinéraire d'un point A à un point B à l'aide d'une carte
- trouver un itinéraire qui passe exactement une fois par chaque ville

On peut aussi se poser des question **d'optimisation** :

- résoudre le taquin en un minimum de coups
- trouver un itinéraire qui minimise le temps de parcours
- ...

Définition du problème : la modélisation

- 1ière étape : décider ce que sont les états $\mathcal{S}$: ensemble d'états et
- 2nde étape : bien définir le problème avec
  - un état de **départ** $s_0 \in \mathcal{S}$
  - les **actions** possibles pour chaque état : $\mathcal{A}_s$ ensemble des actions de l'état $s \in \mathcal{S}$.
  - le modèle de **transition** : c'est une fonction qui donne l'état atteint après avoir pris une action depuis un état : $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$
  - ➩ on peut représenter l'état de départ, les actions et le modèle de transition par un **graphe**.
  - une fonction qui teste si le but est **atteint**. $\mathcal{G} : \mathcal{S} \to \{\top, \bot\}$
  - éventuellement une fonction de coût pour chaque action dans chaque état.

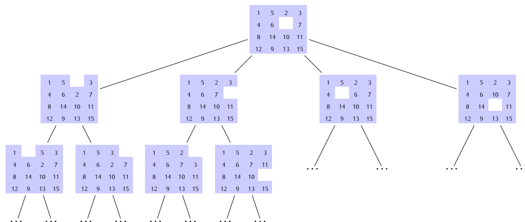


Graphe pour le taquin 2×2

Résolution

La séquence d'actions possibles à partir de l'état initial forme un **graphe de recherche** :

- les noeuds représentent les états
- les branches représentent les différentes actions
- la racine est l'état initial



- Pour résoudre : on choisit un noeud à développer et on teste si les fils sont des états finaux.
- Quelle politique pour choisir le noeud à développer ?
- Attention : on peut répéter des noeuds ⇒ on peut tourner en boucle (sauf si on se rappelle des états visités) !

Taquin $n \times n$

- résoudre un problème de taquin 2×2 a l'air facile !
- Pensons au taquin 4×4
  - Combien y a-t-il de sommets exactement ?
  - ?
  - ➩ la réponse est un entier proche de 2.10^{13}
  - Certains problèmes sont impossibles (il n'existe pas de chemin de l'état initial à l'état final).
  - en fait pour le 4×4, le graphe possède deux composantes connexes
  - ➩ si on tire au sort l'état initial et l'état final
    on a 50% de chance qu'une solution existe.
  - il existe un test simple pour le savoir.
- Peut-on utiliser des algorithmes de la théorie des graphes ?
- ➩ suppose que le graphe puisse être stocké en mémoire
  peut être possible pour 4×4, pas sûr pour 5×5 ou 6×6 !
- on peut utiliser des algorithmes enseignés en cours d'algorithmique.

Résoudre avec des algorithmes de base

- expansion d'un graphe en largeur d'abord
  (**"breadth-first search" (BFS)**)
- expansion d'un graphe en profondeur d'abord
  (**"depth-first search" (DFS)**)
- expansion d'un graphe avec côut uniforme (**"Uniform-cost search"**) : développer le noeud qui a le coût le plus bas.
- recherche en profondeur limitée (**"depth-limited search"**) : utilise DFS jusqu'à une profondeur l donnée
  cela évite le problème du chemin infini, mais pause problème si la solution est plus profonde que l.
- recherche en profondeur itérative (**"iterative deepening DFS"**) : combine DFS et BFS : itérativement utilise DFS jusqu'à une profondeur de 1, 2, ... jusqu'à trouver le but.
  les états sont générés de nombreuses fois
- recherche bidirectionnelle (**"Bidirectional search"**) : effectue deux recherche : une de l'état initial vers l'état final, l'autre dans le sens inverse (donc depuis l'état final).

Répétition des états

On ajoutera à ces stratégies la faculté de se souvenir ou non des noeuds visités

- sans mémoire : **“tree-search”** risque que les états se répètent
  ⇨ risque de boucle
- avec mémoire : **“graph-search”** nécessite des ressources de stockage
  ⇨ évite les boucles !

⇨ on ajoute une liste d’états visités (parfois appelée “explored set” ou “closed list”)
Attention cependant, ceci peut avoir un coût non négligeable en mémoire !

Graph-search

```
1  function Graph Search returns a solution or failure
2    initialise the frontier with initial state
3    initialise the explored set with ∅
4      do
5        if the frontier is empty
6          then return failure
7        else
8          choose a leaf node and remove it from the frontier
9          if the node is a goal state
10           then return the node
11         else
12           add the node to the explored set
13           if the node not in the frontier or explored set
14             expand the node and add the node in the frontier
```

Critères pour comparer ces algorithmes

- **Complétude** : a-t-on une garantie de trouver une solution quand elle existe ?
- **Complexité du temps de calcul** : combien de temps a-t-on besoin (dans le pire des cas)
- **Complexité de l'espace mémoire** : combien d'espace mémoire a-t-on besoin (dans le pire des cas)
- **Optimalité** : est-ce-que la solution trouvée est optimale ?

Comparaison

	BFS	Uniform-Cost	DFS	Depth-limited	Iterative-deepening	Bidirectional
complet ?	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘
complexité temps	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$
complexité mémoire	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$
optimal ?	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘

- b sera le facteur de branchement de l'arbre
- d est la profondeur de la solution la moins profonde
- m est la profondeur maximale de l'arbre de recherche
- l est la limite de profondeur

Guider la recherche grace à des heuristiques

On se place dans les problèmes où il y a une fonction de coût (positive).

- Les techniques précédentes trouvent des solutions, mais ne sont pas efficaces

➡ utilisation de **connaissance** sur la structure du problème

problèmes potentiels :

  - d'où vient cette connaissance ?
  - cette connaissance doit être correct !

Rappel : Uniform cost développe le noeud qui a le coût le plus bas.

➡ le coût le plus bas est une **fonction d'évaluation**, on la notera f.

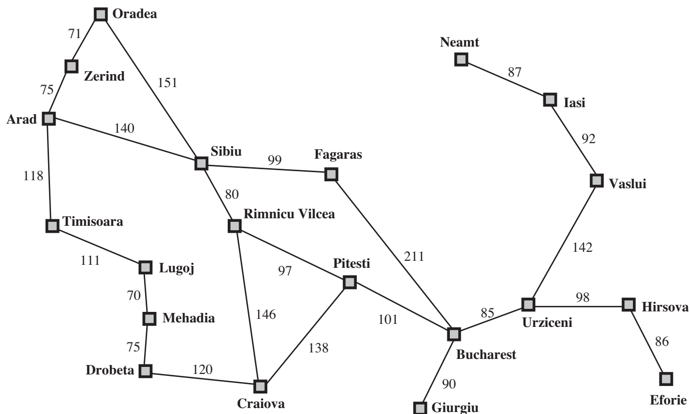
pour le noeud n, $f(n)$ indique si n est prometteur

$f(n)$ **doit indiquer la qualité de la solution en passant par** n

- évidemment, si on était devin et on connaissait f, il n'y aurait pas de problème à résoudre.

➡ on veut développer des heuristiques pour estimer f.

Example : Going from Arad to

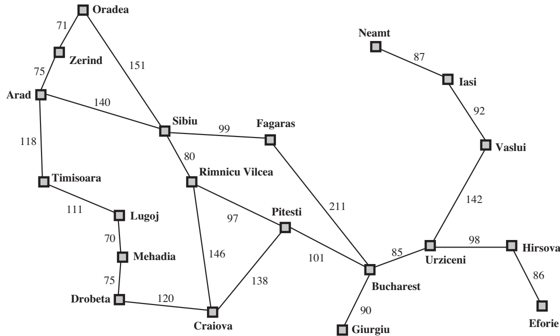


Uniform-cost search

Greedy-search // recherche gloutonne

- Uniform-cost : la fonction d'évaluation était le coût jusqu'à présent
- on peut essayer d'essayer d'estimer la distance jusqu'au but
- ⇨ $h(v)$ va représenter la distance à vol d'oiseau entre une ville v et Bucharest.
- il n'y a rien dans la description du problème qui parle de distance à vol d'oiseau ! On a utilisé notre *"sens commun"*
- il faut trouver cette distance à vol d'oiseau ! (mesure sur une carte par exemple)

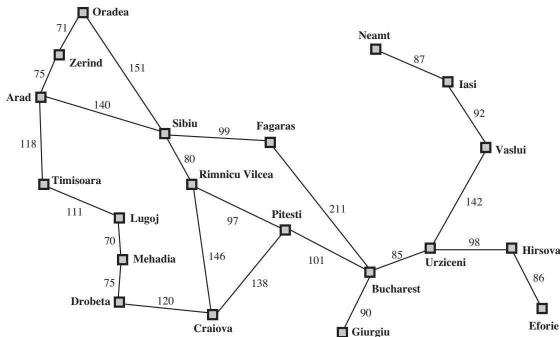
exemple



Distance à vol d'oiseau jusqu'à Bucharest

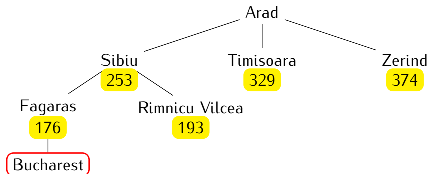
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

exemple



Distance à vol d'oiseau jusqu'à Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



Discussion

- on a trouvé une solution rapidement
- mais la solution **n'**est **pas** optimale ! La solution a un coût de 450, et on peut trouver plus court !
- que se passe-t-il ?
  - parfois, la solution optimale doit "s'éloigner" du but temporairement pour aller plus vite ensuite.

➾ trouver une meilleure fonction d'évaluation

$A^\star$

Soit n un noeud. On note

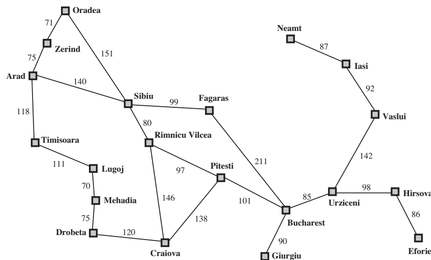
- $g(n)$ le coût pour atteindre le noeud n depuis la racine
- $h(n)$ le coût **estimé** du chemin le moins couteux vers un noeud final.
- $f(n) = g(n) + h(n)$ est donc une estimation du coût minimal passant par le noeud n.

La stratégie d'expansion des noeuds est de développer le noeud avec la plus faible valeur de $f = g + h$.

On a toujours deux versions

- **tree-search** : on ne se rappelle pas des états visités
- **graph-search** : on se rappelle des états visités

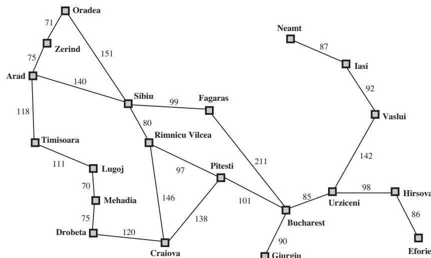
exemple



Distance à vol d'oiseau jusqu'à Bucharest

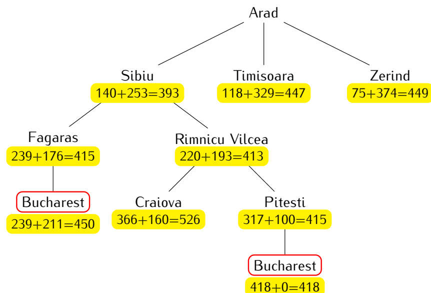
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

exemple



Distance à vol d'oiseau jusqu'à Bucharest

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374



Condition d'Optimalité (tree-search)

Definition (admissibilité)

Une heuristique est **admissible** ssi elle **sous estime** toujours le coût

Dans l'exemple, par définition, la distance à vol d'oiseau sous estime la distance totale, donc l'heuristique est admissible.

Theorem

$A^\star$ dans la version **tree-search** est optimal si la fonction heuristique h est admissible

Condition d'Optimalité (tree-search)

Definition (admissibilité)

Une heuristique est **admissible** ssi elle **sous estime** toujours le coût

Dans l'exemple, par définition, la distance à vol d'oiseau sous estime la distance totale, donc l'heuristique est admissible.

Theorem

$A^\star$ dans la version **tree-search** est optimal si la fonction heuristique h est admissible

Proof

Supposons que l'algorithme atteigne le but n et le chemin retourné est noté l. Le coût du chemin l est donc $g(n)$.
Raisonnons par l'absurde : on fait l'hypothèse que le chemin trouvé l n'est pas optimal. Il existe donc un chemin l' dont le coût est $c < g(n)$. Donc, comme h sous-estime, pour chaque noeud n' de l', on a $g(n') + h(n') \leqslant c < g(n)$. Mais de tels noeuds auraient déjà dûs être développés, on arrive à une contradiction. □

Condition d'Optimalité (graph-search)

Definition (monotonicité)

une heuristique est monotone si pour tout noeud n, tout succésseur n' de n on a $h(n) \leqslant c(n,n') + h(n')$.

Theorem

$A^\star$ dans la version **graph-search** est optimal si la fonction heuristique h est monotone.

Une heuristique monotone est admissible (cf TD). Généralement, les heuristiques admissibles mais non monotones sont assez dures à trouver.

Proof

Supposons que h est monotone. Soit un chemin dans le graphe de recherche et n et n' deux noeuds succéssifs. Alors

$f(n) = g(n) + h(n) \leqslant \underbrace{g(n) + c(n, n')}_{g(n')} + h(n')$ (monotonicité de h)

$f(n) \leqslant g(n') + h(n') = f(n')$. On en conclue donc que f est croissante le long de tout chemin.

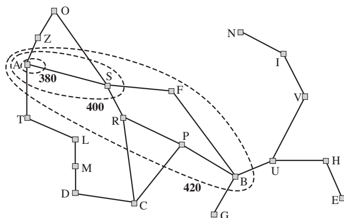
Démontrons maintenenant qu'au moment où $A^\star$ choisit de développer le noeud n, le chemin optimal menant à n a déjà été trouvé. Supposons que ce ne soit pas le cas : il doit exister un noeud n' sur le chemin optimal menant à n. Comme f est croissante, $f(n') \leqslant f(n)$, et donc n' aurait dû être développé, on arrive à une contradiction.

Ainsi, la séquence de noeuds développés par $A^\star$ est en ordre croissant de la fonction f. Donc la première fois qu'on veut développer un noeud but n, on a trouvé une solution optimale, car $f(n) = g(n)$ et chaque noeud développé plus tard aura un coût au moins plus grand.

□

Contours

f étant croissante sur tout chemin, on peut représenter des contours comme sur une carte topographique.



Tous les noeuds à l'intérieur d'un coutour ont une évaluation d'au plus la valeur du contour. $A^\star$ développe tous les noeuds à l'intérieur d'un contour avant d'en développer d'autres. ➾ ex : Timisoara n'est pas développé : **"pruning"** : on peut éliminer cette partie de la recherche sans examen et sans risque de manquer un chemin optimal : **très important pour beaucoup d'algorithmes en IA.**
$A^\star$ ajoute des noeuds dans des "bandes" dont la valeur de f augmente.

- si l'heuristique est n'est pas très intéressante (ex. uniform-cost), les bandes seront plutôt circulaires
- si l'heuristique est proche de la vraie fonction les bandes seront plus étroites et guideront vers une solution optimale

Autre propriétés

Aucun autre algorithme utilisant une même heuristique h ne peut développer moins de noeuds que $A^\star$: ne pas développer un noeud n tel que $f(n) < C^\star$ présente le risque de manquer un chemin optimal.

Pour des problèmes de grandes tailles, c'est généralement la mémoire qui limite l'utilisation de $A^\star$

➡ il existe des variantes pour limiter l'utilisation de la mémoire.

- iterative-deepening $A^\star$: utiliser l'idée de iterative-deepening avec un coût fixé : on cherche une solution au plus d'un coût f_{max}, si on ne trouve pas, on cherche avec un f_{max} plus grand.
- recursive best-first search (RBSF) : combine DFS et $A^\star$
- memory-bound $A^\star$
- simplified memory-bound $A^\star$

Retour au taquin

Heuristique pour le taquin :

- W : nombre de jetons pas à leur place
- P : somme des distances pour placer chaque jeton à leur place

Ces heuristiques sont-elles admissibles ?

Quelle est la meilleure heuristique ?
On a pour chaque noeud n : $W(n) \leqslant P(n)$ (pourquoi ?), donc P est meilleure

Conclusion

- recherche non informée :
  - modelisation est importante
  - différents algorithmes avec différentes propriétés
  - ➩ choisir le plus approprié
- recherche informée :
- algorithme $A^\star$
- qualité de l'heuristique
- n'est pas toujours suffisant