Intelligence Artificielle

basé sur "Artificial Intelligence : A modern approach" de Russel et Nowig

ENSISA 2A

Jonathan Weber

Automne 2024

Recherche heuristiques

Recherche heuristiques

Définition

Best-first search

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds

Best-first search

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre

Best-first search

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre
  - **Objectif** : Estimer la proximité d'un état par rapport à l'objectif

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre
    - **Objectif** : Estimer la proximité d'un état par rapport à l'objectif
    - fonction $f(n)$ mesurant l'utilité d'un nœud qui peut-être composée :

Best-first search

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre
  - **Objectif** : Estimer la proximité d'un état par rapport à l'objectif
  - fonction $f(n)$ mesurant l'utilité d'un nœud qui peut-être composée :
    - d'une ou plusieurs fonctions heuristiques $h(n)$ qui estiment le coût du chemin le plus court pour se rendre au but

Best-first search

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre
  - **Objectif** : Estimer la proximité d'un état par rapport à l'objectif
  - fonction $f(n)$ mesurant l'utilité d'un nœud qui peut-être composée :
    - d'une ou plusieurs fonctions heuristiques $h(n)$ qui estiment le coût du chemin le plus court pour se rendre au but
    - d'une fonction $g(n)$ mesure le coût du chemin de l'état initial au nœud n

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre
  - **Objectif** : Estimer la proximité d'un état par rapport à l'objectif
  - fonction $f(n)$ mesurant l'utilité d'un nœud qui peut-être composée :
    - d'une ou plusieurs fonctions heuristiques $h(n)$ qui estiment le coût du chemin le plus court pour se rendre au but
    - d'une fonction $g(n)$ mesure le coût du chemin de l'état initial au nœud n
- Une heuristique n'a pas besoin d'être exacte

Best-first search

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre
  - **Objectif** : Estimer la proximité d'un état par rapport à l'objectif
  - fonction $f(n)$ mesurant l'utilité d'un nœud qui peut-être composée :
    - d'une ou plusieurs fonctions heuristiques $h(n)$ qui estiment le coût du chemin le plus court pour se rendre au but
    - d'une fonction $g(n)$ mesure le coût du chemin de l'état initial au nœud n
- Une heuristique n'a pas besoin d'être exacte
- Exemple : Distance à vol d'oiseau pour de l'itinéraire routier

Best-first search

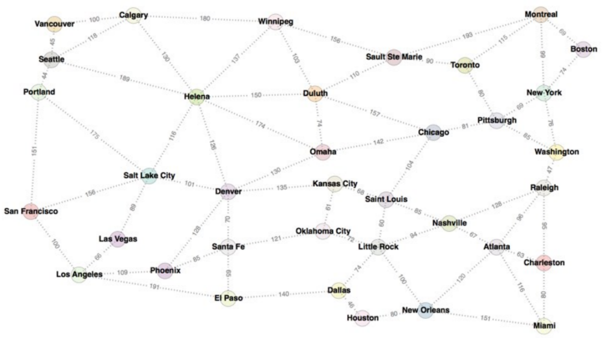
- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre
  - **Objectif** : Estimer la proximité d'un état par rapport à l'objectif
  - fonction $f(n)$ mesurant l'utilité d'un nœud qui peut-être composée :
    - d'une ou plusieurs fonctions heuristiques $h(n)$ qui estiment le coût du chemin le plus court pour se rendre au but
    - d'une fonction $g(n)$ mesure le coût du chemin de l'état initial au nœud n
- Une heuristique n'a pas besoin d'être exacte
- Exemple : Distance à vol d'oiseau pour de l'itinéraire routier
- Algorithmes :

Best-first search

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre
  - **Objectif** : Estimer la proximité d'un état par rapport à l'objectif
  - fonction $f(n)$ mesurant l'utilité d'un nœud qui peut-être composée :
    - d'une ou plusieurs fonctions heuristiques $h(n)$ qui estiment le coût du chemin le plus court pour se rendre au but
    - d'une fonction $g(n)$ mesure le coût du chemin de l'état initial au nœud n
- Une heuristique n'a pas besoin d'être exacte
- Exemple : Distance à vol d'oiseau pour de l'itinéraire routier
- Algorithmes :
  - Recherche gloutonne (greedy search)

Best-first search

- **Rappel** : Stratégie de recherche définit l'ordre de développement des nœuds
- **Idée** : utiliser les connaissances du domaine pour améliorer cet ordre
  - **Objectif** : Estimer la proximité d'un état par rapport à l'objectif
  - fonction $f(n)$ mesurant l'utilité d'un nœud qui peut-être composée :
    - d'une ou plusieurs fonctions heuristiques $h(n)$ qui estiment le coût du chemin le plus court pour se rendre au but
    - d'une fonction $g(n)$ mesure le coût du chemin de l'état initial au nœud n
- Une heuristique n'a pas besoin d'être exacte
- Exemple : Distance à vol d'oiseau pour de l'itinéraire routier
- Algorithmes :
  - Recherche gloutonne (greedy search)
  - $A^\star$



Atlanta	272
Boston	240
Calgary	334
Charleston	322
Chicago	107
Dallas	303
Denver	270
Duluth	110
El Paso	370
Helena	254
Houston	332
Kansas City	176
Las Vegas	418
Little Rock	240
Los Angeles	484
Miami	389
Montreal	193
Nashville	221
New Orleans	322
New York	195
Oklahoma City	237
Omaha	150
Phoenix	396
Pittsburgh	152
Portland	452
Raleigh	251
Saint Louis	180
Salt Lake City	344
San Francisco	499
Santa Fe	318
Sault Ste Marie	0
Seattle	434
Toronto	90
Vancouver	432
Washington	238
Winnipeg	156

L'heuristique représente la distance à vol d'oiseau depuis Sault Ste Marie.

Recherche heuristiques

Recherche gloutonne

- Fonction d'évaluation $f(n) =$

- Fonction d'évaluation $f(n) = h(n)$

- Fonction d'évaluation $f(n) = h(n)$
- Recherche gloutonne ignore le coût du parcours déjà effectué

- Fonction d'évaluation $f(n) = h(n)$
- Recherche gloutonne ignore le coût du parcours déjà effectué
  - développe le nœud le plus proche de l'objectif (selon l'heuristique)

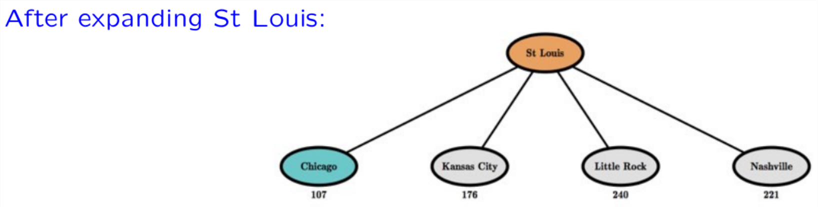
- Fonction d'évaluation $f(n) = h(n)$
- Recherche gloutonne ignore le coût du parcours déjà effectué
  - développe le nœud le plus proche de l'objectif (selon l'heuristique)
- $h(n)$: estimation du coût du parcours du nœud n vers l'état final

- Fonction d'évaluation $f(n) = h(n)$
- Recherche gloutonne ignore le coût du parcours déjà effectué
  - développe le nœud le plus proche de l'objectif (selon l'heuristique)
- $h(n)$: estimation du coût du parcours du nœud n vers l'état final
- Exemple $h_{\text{DVO}}(n)$ = distance à vol d'oiseau entre la ville n et l'objectif

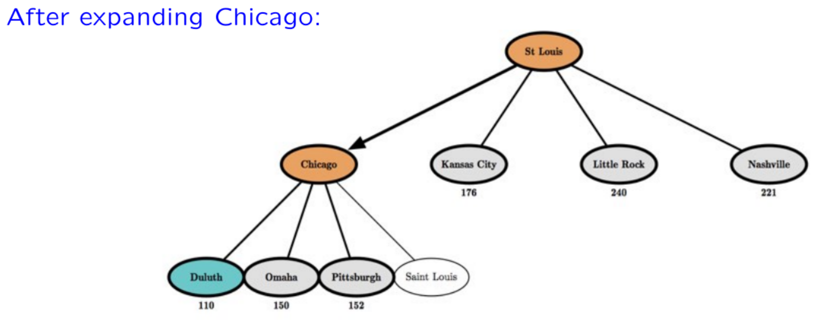
```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

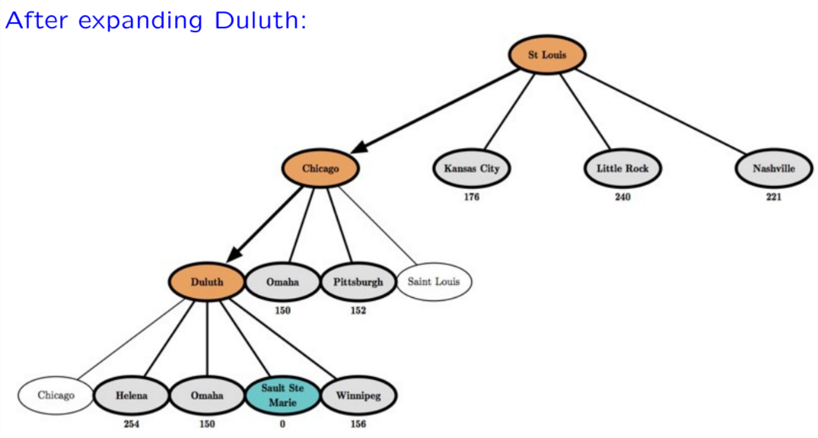
Recherche gloutonne

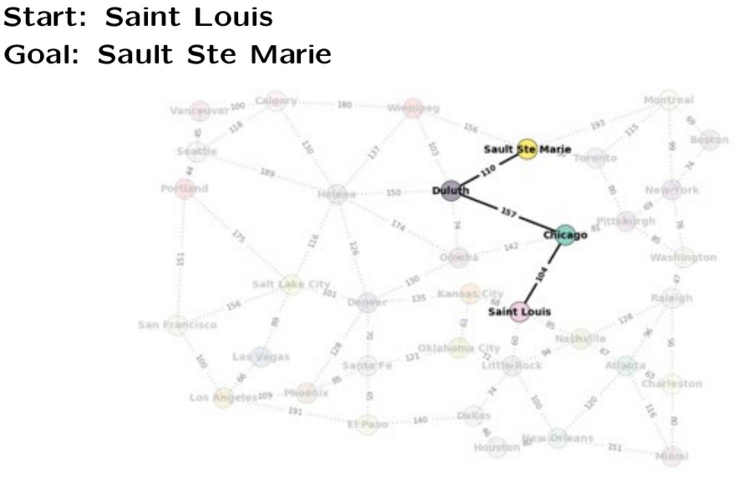
©Ansaf Salleb-Aouissi, Columbia



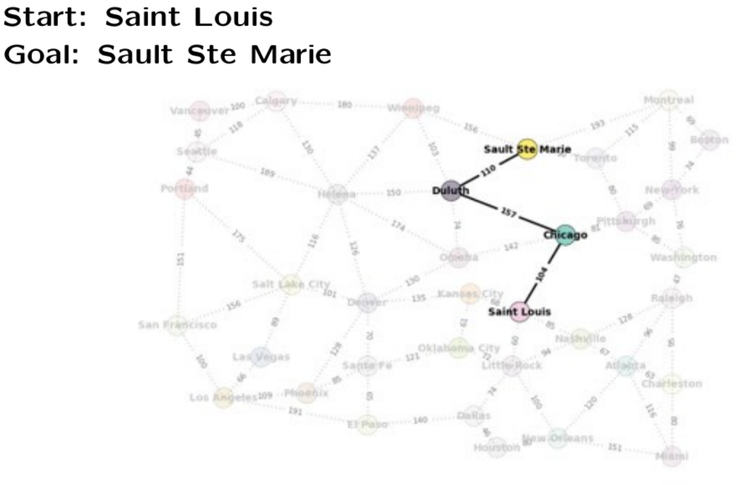
©Ansaf Salleb-Aouissi, Columbia


After expanding Chicago:
St Louis
Chicago
Kansas City
176
Little Rock
240
Nashville
221
Duluth
110
Omaha
150
Pittsburgh
152
Saint Louis


After expanding Duluth:
St Louis
Chicago
Kansas City
176
Little Rock
240
Nashville
221
Duluth
Omaha
150
Pittsburgh
152
Saint Louis
Chicago
Helena
254
Omaha
150
Sault Ste Marie
0
Winnipeg
156



©Ansaf Salleb-Aouissi, Columbia



Recherche gloutonne donne un trajet de 371 km

Recherche gloutonne

- complétude : Non

- complétude : Non
  - Risque de boucle

- complétude : Non
  - Risque de boucle
  - Complet si ajout d'un test pour éviter les boucles

- complétude : Non
  - Risque de boucle
  - Complet si ajout d'un test pour éviter les boucles
- complexité en temps : $O(b^m)$

- complétude : Non
  - Risque de boucle
  - Complet si ajout d'un test pour éviter les boucles
- complexité en temps : $O(b^m)$
  - Performances réelles dépendantes de l'heuristique

- complétude : Non
  - Risque de boucle
  - Complet si ajout d'un test pour éviter les boucles
- complexité en temps : $O(b^m)$
  - Performances réelles dépendantes de l'heuristique
- complexité en mémoire : $O(b^m)$

- complétude : Non
  - Risque de boucle
  - Complet si ajout d'un test pour éviter les boucles
- complexité en temps : $O(b^m)$
  - Performances réelles dépendantes de l'heuristique
- complexité en mémoire : $O(b^m)$
- optimalité : Non

Recherche heuristiques

$A^\star$

- Fonction d'évaluation $f(n) =$

- Fonction d'évaluation $f(n) = g(n) +$

$A^{\star}$

- Fonction d'évaluation $f(n) = g(n) + h(n)$

A*

- Fonction d'évaluation $f(n) = g(n) + h(n)$
- A* cherche à minimiser la totalité du trajet (trajet parcouru + estimation du trajet restant)

- Fonction d'évaluation $f(n) = g(n) + h(n)$
- A* cherche à minimiser la totalité du trajet (trajet parcouru + estimation du trajet restant)
  - $g(n)$: chemin parcouru jusqu'au nœud n

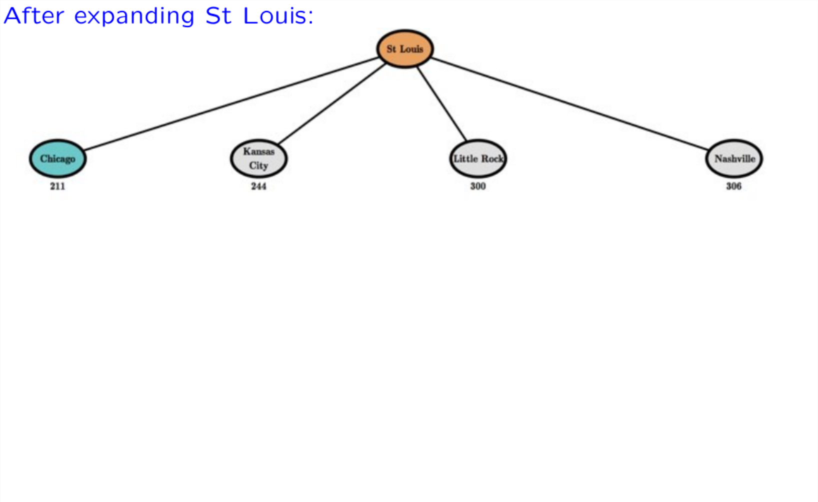
- Fonction d'évaluation $f(n) = g(n) + h(n)$
- $A^\star$ cherche à minimiser la totalité du trajet (trajet parcouru + estimation du trajet restant)
  - $g(n)$: chemin parcouru jusqu'au nœud n
  - $h(n)$: estimation du coût du parcours du nœud n vers l'état final

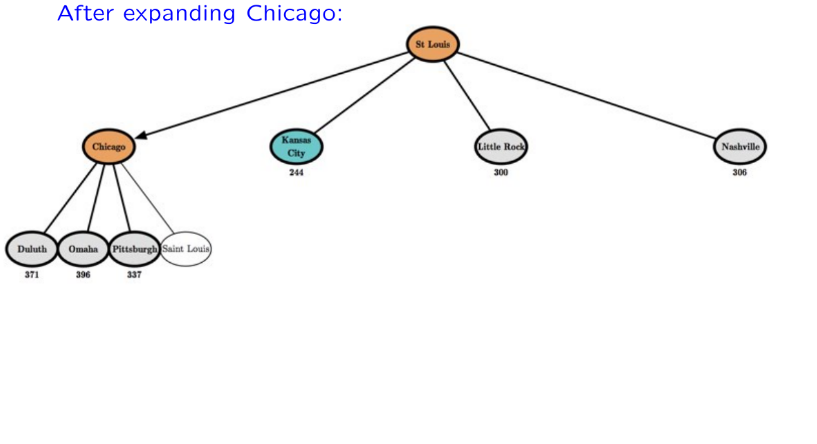
- Fonction d'évaluation $f(n) = g(n) + h(n)$
- A* cherche à minimiser la totalité du trajet (trajet parcouru + estimation du trajet restant)
  - $g(n)$: chemin parcouru jusqu'au nœud n
  - $h(n)$: estimation du coût du parcours du nœud n vers l'état final
- Si $h(n) = 0$ pour tout n, alors A* est équivalent au parcours à coût uniforme

A*

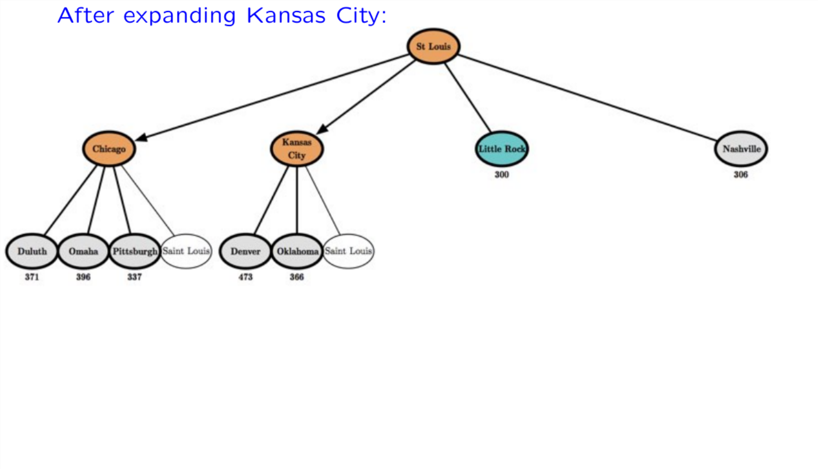
```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

The initial state:

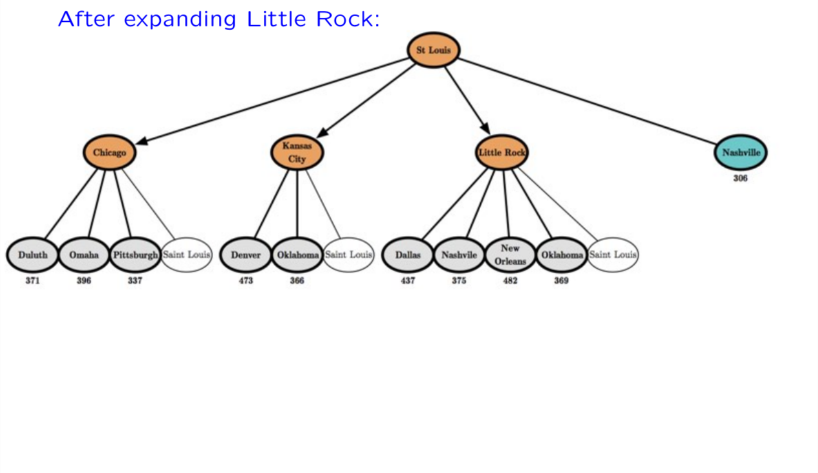

After expanding St Louis:
St Louis
Chicago
211
Kansas City
244
Little Rock
300
Nashville
306



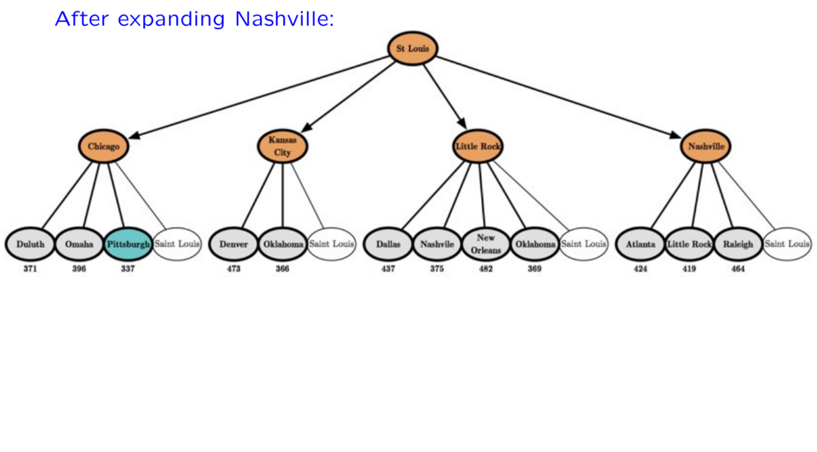
©Ansaf Salleb-Aouissi, Columbia



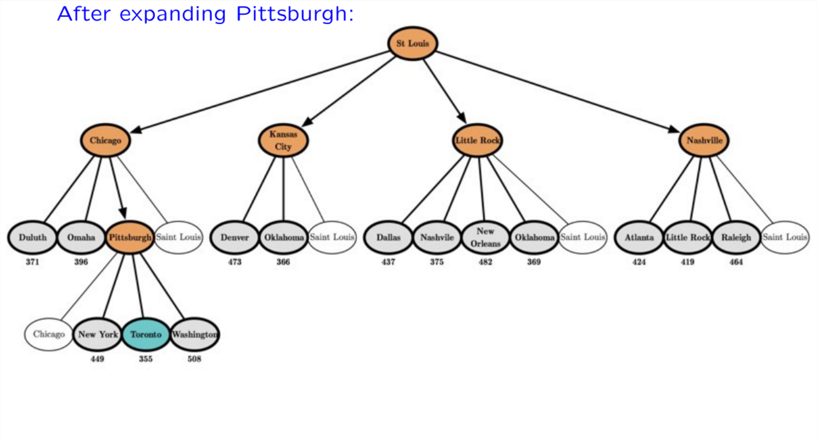
©Ansaf Salleb-Aouissi, Columbia

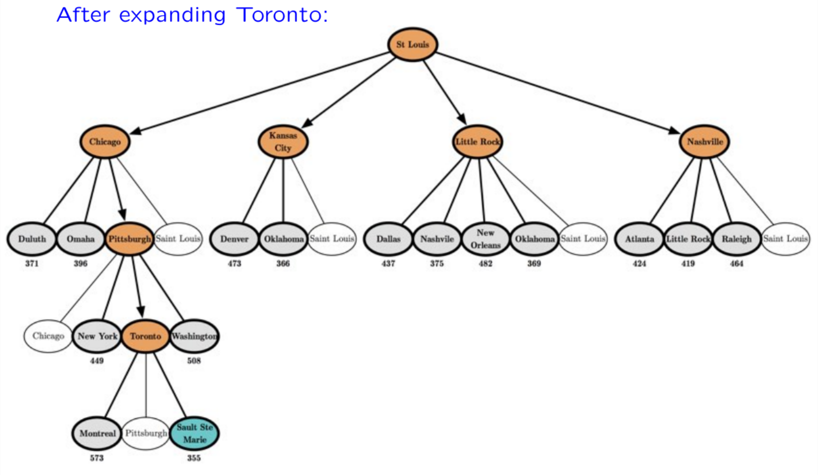


©Ansaf Salleb-Aouissi, Columbia



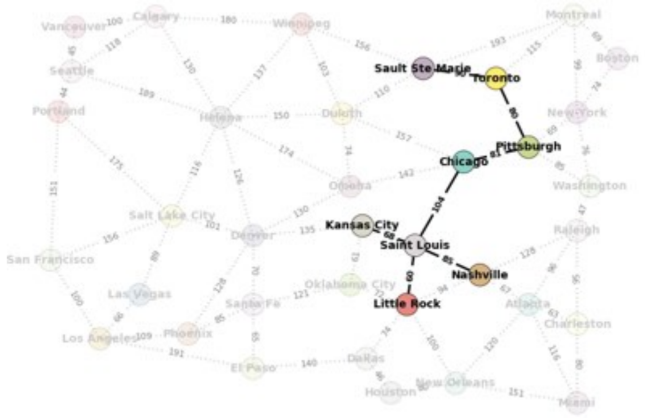
©Ansaf Salleb-Aouissi, Columbia


After expanding Pittsburgh:
St Louis
Chicago
Kansas City
Little Rock
Nashville
Duluth
371
Omaha
396
Pittsburgh
Saint Louis
Denver
473
Oklahoma
366
Saint Louis
Dallas
437
Nashvile
375
New Orleans
482
Oklahoma
369
Saint Louis
Atlanta
424
Little Rock
419
Raleigh
464
Saint Louis
Chicago
New York
449
Toronto
355
Washington
508



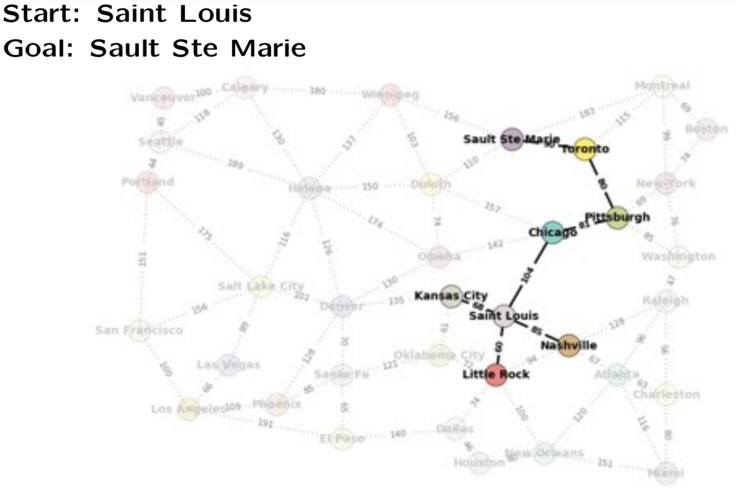
©Ansaf Salleb-Aouissi, Columbia

Start: Saint Louis
Goal: Sault Ste Marie



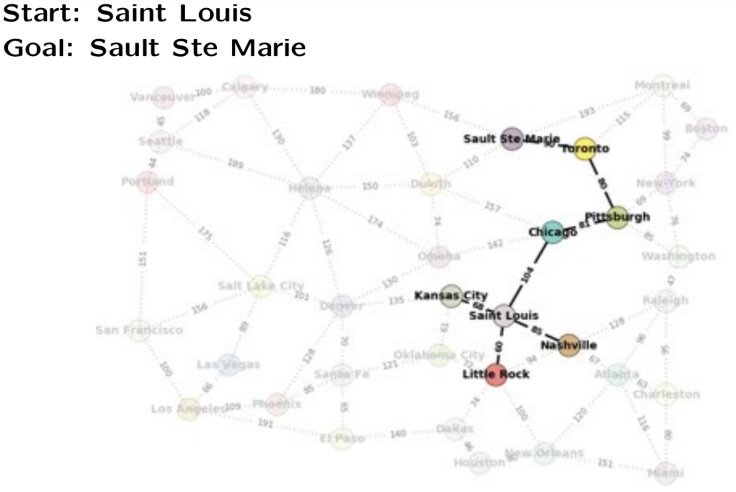
©Ansaf Salleb-Aouissi, Columbia

A*



©Ansaf Salleb-Aouissi, Columbia

A* donne un trajet de 355 km



©Ansaf Salleb-Aouissi, Columbia

A★ donne un trajet de 355 km

Rappel : Recherche gloutonne donne un trajet de 371 km

- complétude : Oui (sauf s'il y a une infinité de nœuds)

A⋆

- complétude : Oui (sauf s'il y a une infinité de nœuds)
- complexité en temps : $O(b^d)$

A*

- complétude : Oui (sauf s'il y a une infinité de nœuds)
- complexité en temps : $O(b^d)$
- complexité en mémoire : $O(b^d)$

A*

- complétude : Oui (sauf s'il y a une infinité de nœuds)
- complexité en temps : $O(b^d)$
- complexité en mémoire : $O(b^d)$
- optimalité : Oui

- ▷ complétude : Oui (sauf s'il y a une infinité de nœuds)
- ▷ complexité en temps : $O(b^d)$
- ▷ complexité en mémoire : $O(b^d)$
- ▷ optimalité : Oui

⚠ En réalité la complexité de A⋆ dépend de l'heuristique et notamment de son facteur de branchement effectif (b^*), les deux complexités devenant $O((b^*)^d)$

Recherche heuristiques

Heuristique

Admissibilité d'une heuristique

- A* nécessite une heuristique admissible

Admissibilité d'une heuristique

- $A^\star$ nécessite une heuristique admissible
  - Une heuristique admissible ne surestime jamais le coût réel pour atteindre le but. Elle est optimiste

- $A^\star$ nécessite une heuristique admissible
  - Une heuristique admissible ne surestime jamais le coût réel pour atteindre le but. Elle est optimiste
  - $h(n) \leq h^\star(n)$, avec $h^\star(n)$ le coût réel pour aller depuis n jusqu'à l'objectif

- A* nécessite une heuristique admissible
  - Une heuristique admissible ne surestime jamais le coût réel pour atteindre le but. Elle est optimiste
  - $h(n) \leq h^\star(n)$, avec $h^\star(n)$ le coût réel pour aller depuis n jusqu'à l'objectif
  - L'heuristique h_{DVO} ne surestime jamais la distance par exemple

- A* nécessite une heuristique admissible
  - Une heuristique admissible ne surestime jamais le coût réel pour atteindre le but. Elle est optimiste
  - $h(n) \leq h^\star(n)$, avec $h^\star(n)$ le coût réel pour aller depuis n jusqu'à l'objectif
  - L'heuristique h_{DVO} ne surestime jamais la distance par exemple
- Optimalité de A*

Admissibilité d'une heuristique

- A* nécessite une heuristique admissible
  - Une heuristique admissible ne surestime jamais le coût réel pour atteindre le but. Elle est optimiste
  - $h(n) \leq h^\star(n)$, avec $h^\star(n)$ le coût réel pour aller depuis n jusqu'à l'objectif
  - L'heuristique h_{DVO} ne surestime jamais la distance par exemple
- Optimalité de A*
  - Si $h(n)$ est admissible alors A* est optimale

- Que faire si f décroît ?

- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin

- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin
  - Par exemple, si p est un successeur de n, il est possible d'avoir :

- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin
  - Par exemple, si p est un successeur de n, il est possible d'avoir :
    - n : $g = 4$, $h = 8$, $f = 12$

- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin
  - Par exemple, si p est un successeur de n, il est possible d'avoir :
    - n : $g = 4$, $h = 8$, $f = 12$
    - p : $g = 5$, $h = 6$, $f = 11$

- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin
  - Par exemple, si p est un successeur de n, il est possible d'avoir :
    - n : $g = 4$, $h = 8$, $f = 12$
    - p : $g = 5$, $h = 6$, $f = 11$
  - On perd de l'information :

- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin
  - Par exemple, si p est un successeur de n, il est possible d'avoir :
    - n : $g = 4$, $h = 8$, $f = 12$
    - p : $g = 5$, $h = 6$, $f = 11$
  - On perd de l'information :
    - $f(n)$ = 12, donc le vrai coût d'un chemin à travers n est ≥ 12

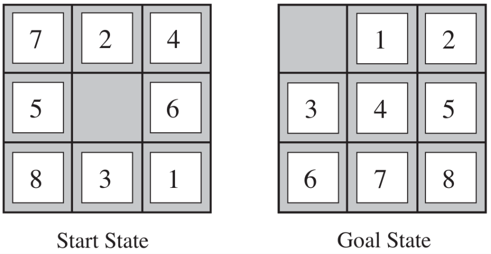
- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin
  - Par exemple, si p est un successeur de n, il est possible d'avoir :
    - n : $g = 4$, $h = 8$, $f = 12$
    - p : $g = 5$, $h = 6$, $f = 11$
  - On perd de l'information :
    - $f(n)$ = 12, donc le vrai coût d'un chemin à travers n est ≥ 12
    - Donc le vrai coût d'un chemin à travers p est aussi ≥ 12

- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin
  - Par exemple, si p est un successeur de n, il est possible d'avoir :
    - n : $g = 4$, $h = 8$, $f = 12$
    - p : $g = 5$, $h = 6$, $f = 11$
  - On perd de l'information :
    - $f(n)$ = 12, donc le vrai coût d'un chemin à travers n est ≥ 12
    - Donc le vrai coût d'un chemin à travers p est aussi ≥ 12
  - Au lieu de $f(p) = g(p) + h(p)$, on utilise :

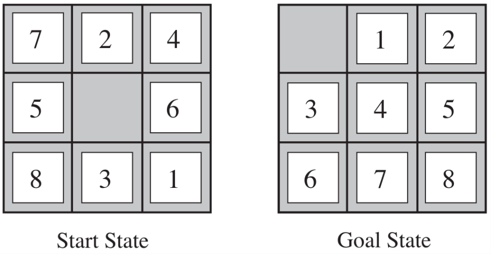
- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin
  - Par exemple, si p est un successeur de n, il est possible d'avoir :
    - n : $g = 4$, $h = 8$, $f = 12$
    - p : $g = 5$, $h = 6$, $f = 11$
  - On perd de l'information :
    - $f(n)$ = 12, donc le vrai coût d'un chemin à travers n est ≥ 12
    - Donc le vrai coût d'un chemin à travers p est aussi ≥ 12
  - Au lieu de $f(p) = g(p) + h(p)$, on utilise :
    - $f(p) = max(g(p) + h(p), f(n))$

- Que faire si f décroît ?
  - Avec une heuristique admissible, f peut décroître au cours du chemin
  - Par exemple, si p est un successeur de n, il est possible d'avoir :
    - n : $g = 4$, $h = 8$, $f = 12$
    - p : $g = 5$, $h = 6$, $f = 11$
  - On perd de l'information :
    - $f(n)$ = 12, donc le vrai coût d'un chemin à travers n est ≥ 12
    - Donc le vrai coût d'un chemin à travers p est aussi ≥ 12
  - Au lieu de $f(p) = g(p) + h(p)$, on utilise :
    - $f(p) = max(g(p) + h(p), f(n))$

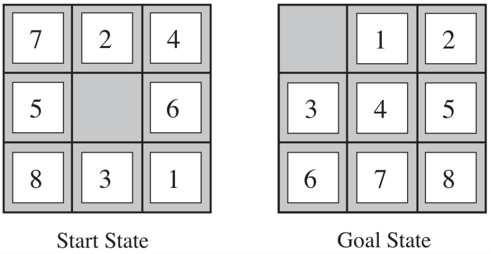
  ⇒ f ne décroît jamais le long du chemin



- La solution est en 26 étapes

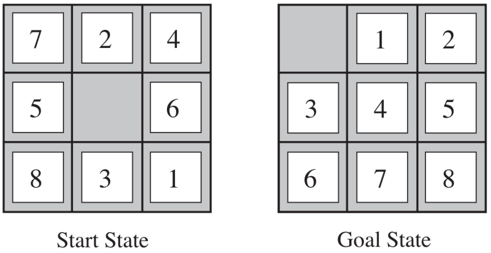


- La solution est en 26 étapes
- $h_1(n)$ = nombre de tuiles mal placées



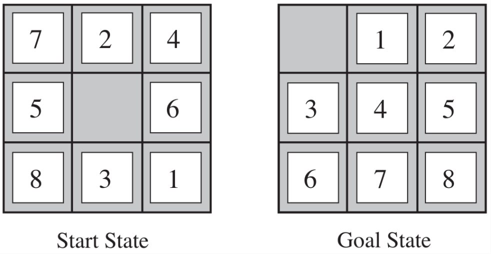
- La solution est en 26 étapes
- $h_1(n)$ = nombre de tuiles mal placées
  - $h_1(start) = 8$

Exemple d'heuristique : le taquin



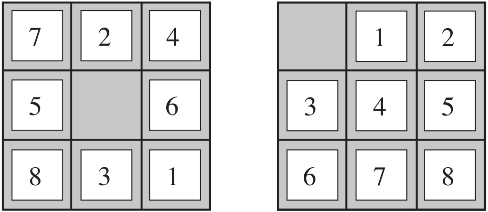
- La solution est en 26 étapes
- $h_1(n)$ = nombre de tuiles mal placées
  - $h_1(start) = 8$
- $h_2(n)$ = distance de Manhattan entre la position de départ et la position finale de chaque pièce

Exemple d'heuristique : le taquin



- La solution est en 26 étapes
- $h_1(n)$ = nombre de tuiles mal placées
  - $h_1(start) = 8$
- $h_2(n)$ = distance de Manhattan entre la position de départ et la position finale de chaque pièce
  - $h_2(start) = 3+1+2+2+2+3+3+2 = 18$

Exemple d'heuristique : le taquin



Start State Goal State

- La solution est en 26 étapes
- $h_1(n)$ = nombre de tuiles mal placées
  - $h_1(start) = 8$
- $h_2(n)$ = distance de Manhattan entre la position de départ et la position finale de chaque pièce
  - $h_2(start) = 3+1+2+2+2+3+3+2 = 18$

⇒ Ces deux heuristiques sont admissibles car elles ne surestiment pas le nombre d'étapes vers la solution

- Comment choisir entre deux heuristiques admissibles ?

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Domination

- Comment choisir entre deux heuristiques admissibles ?
  - Si on regarde, les deux heuristiques précédentes, on peut facilement montrer que $h_1(n) \leq h_2(n)$

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- Comment choisir entre deux heuristiques admissibles ?
  - Si on regarde, les deux heuristiques précédentes, on peut facilement montrer que $h_1(n) \leq h_2(n)$
  - On dit alors que h_2 domine h_1

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- Comment choisir entre deux heuristiques admissibles ?
  - Si on regarde, les deux heuristiques précédentes, on peut facilement montrer que $h_1(n) \leq h_2(n)$
  - On dit alors que h_2 domine h_1
  - Cette domination se traduit par une efficacité accrue

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- Comment choisir entre deux heuristiques admissibles ?
  - Si on regarde, les deux heuristiques précédentes, on peut facilement montrer que $h_1(n) \leq h_2(n)$
  - On dit alors que h_2 domine h_1
  - Cette domination se traduit par une efficacité accrue
- Mesure de performance d'une heuristique = facteur de branchement effectif

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Domination

- Comment choisir entre deux heuristiques admissibles ?
  - Si on regarde, les deux heuristiques précédentes, on peut facilement montrer que $h_1(n) \leq h_2(n)$
  - On dit alors que h_2 domine h_1
  - Cette domination se traduit par une efficacité accrue
- Mesure de performance d'une heuristique = facteur de branchement effectif
  - Facteur de branchement effectif = $\dfrac{\text{nombre de nœuds développés}}{\text{profondeur de la solution}}$

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Conception d'heuristique

- Comment trouver une heuristique admissible ?

- Comment trouver une heuristique admissible ?
    - Considérer une version simplifiée du problème

- Comment trouver une heuristique admissible ?
  - Considérer une version simplifiée du problème
  - Le coût exact d'une solution optimale du problème simplifié est une heuristique admissible pour le problème original

- Comment trouver une heuristique admissible ?
  - Considérer une version simplifiée du problème
  - Le coût exact d'une solution optimale du problème simplifié est une heuristique admissible pour le problème original
  - Exemple : simplification des règles du taquin :

- Comment trouver une heuristique admissible ?
  - Considérer une version simplifiée du problème
  - Le coût exact d'une solution optimale du problème simplifié est une heuristique admissible pour le problème original
  - Exemple : simplification des règles du taquin :
    - une pièce peut être déplacée partout

Conception d'heuristique

- Comment trouver une heuristique admissible ?
  - Considérer une version simplifiée du problème
  - Le coût exact d'une solution optimale du problème simplifié est une heuristique admissible pour le problème original
  - Exemple : simplification des règles du taquin :
    - une pièce peut être déplacée partout
    - ⇒ $h_1(n)$ donne le coût de la solution optimale

- Comment trouver une heuristique admissible ?
  - Considérer une version simplifiée du problème
  - Le coût exact d'une solution optimale du problème simplifié est une heuristique admissible pour le problème original
  - Exemple : simplification des règles du taquin :
    - une pièce peut être déplacée partout
    - ⇒ $h_1(n)$ donne le coût de la solution optimale
    - une pièce peut être déplacée vers les places adjacentes

- Comment trouver une heuristique admissible ?
    - Considérer une version simplifiée du problème
    - Le coût exact d'une solution optimale du problème simplifié est une heuristique admissible pour le problème original
    - Exemple : simplification des règles du taquin :
        - une pièce peut être déplacée partout
        - $\Rightarrow$ $h_1(n)$ donne le coût de la solution optimale
        - une pièce peut être déplacée vers les places adjacentes
        - $\Rightarrow$ $h_2(n)$ donne le coût de la solution optimale

Recherche heuristiques

Réduire le coût mémoire de A*

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*
- **Iterative-deepening A*** (IDA*)

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*
- **Iterative-deepening A*** (IDA*)
  - Même principe que le parcours itératif en profondeur

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*
- **Iterative-deepening A*** (IDA*)
  - Même principe que le parcours itératif en profondeur
  - Remplace la limite de profondeur par une limite du coût f

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*
- **Iterative-deepening A*** (IDA*)
  - Même principe que le parcours itératif en profondeur
  - Remplace la limite de profondeur par une limite du coût f
  - À chaque itération, on remplace la limite f précédente par la valeur de f la plus petite qui excédait la limite f précédente

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*
- **Iterative-deepening A*** (IDA*)
    - Même principe que le parcours itératif en profondeur
    - Remplace la limite de profondeur par une limite du coût f
    - À chaque itération, on remplace la limite f précédente par la valeur de f la plus petite qui excédait la limite f précédente
    - Propriétés

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*
- **Iterative-deepening A*** (IDA*)
  - Même principe que le parcours itératif en profondeur
  - Remplace la limite de profondeur par une limite du coût f
  - À chaque itération, on remplace la limite f précédente par la valeur de f la plus petite qui excédait la limite f précédente
  - Propriétés
    - complétude : Oui (sauf s'il y a une infinité de nœuds)

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*
- **Iterative-deepening A*** (IDA*)
  - Même principe que le parcours itératif en profondeur
  - Remplace la limite de profondeur par une limite du coût f
  - À chaque itération, on remplace la limite f précédente par la valeur de f la plus petite qui excédait la limite f précédente
  - Propriétés
    - complétude : Oui (sauf s'il y a une infinité de nœuds)
    - complexité en temps : $O(b^d)$

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*
- **Iterative-deepening A*** (IDA*)
  - Même principe que le parcours itératif en profondeur
  - Remplace la limite de profondeur par une limite du coût f
  - À chaque itération, on remplace la limite f précédente par la valeur de f la plus petite qui excédait la limite f précédente
  - Propriétés
    - complétude : Oui (sauf s'il y a une infinité de nœuds)
    - complexité en temps : $O(b^d)$
    - complexité en mémoire : $O(bd)$

IDA*

- **Idée** : Adapter le concept du parcours itératif en profondeur à A*
- **Iterative-deepening A*** (IDA*)
  - Même principe que le parcours itératif en profondeur
  - Remplace la limite de profondeur par une limite du coût f
  - À chaque itération, on remplace la limite f précédente par la valeur de f la plus petite qui excédait la limite f précédente
  - Propriétés
    - complétude : Oui (sauf s'il y a une infinité de nœuds)
    - complexité en temps : $O(b^d)$
    - complexité en mémoire : $O(bd)$
    - optimalité : Oui

- **Idée** : utiliser une mémoire limitée pour stocker les noeuds

SMA*

- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- **Simplified Memory-bounded A*** (SMA*)

- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- **Simplified Memory-bounded A*** (SMA*)
  - SMA* fonctionne comme A* jusqu'à ce que la mémoire soit pleine

SMA*

- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- **Simplified Memory-bounded A*** (SMA*)
  - SMA* fonctionne comme A* jusqu'à ce que la mémoire soit pleine
  - Ensuite, l'exploration d'un nouveau nœud nécessitera la suppression du nœud ayant la plus grande valeur f

SMA*

- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- **Simplified Memory-bounded A*** (SMA*)
  - SMA* fonctionne comme A* jusqu'à ce que la mémoire soit pleine
  - Ensuite, l'exploration d'un nouveau nœud nécessitera la suppression du nœud ayant la plus grande valeur f
  - Propriétés

SMA*

- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- **Simplified Memory-bounded A*** (SMA*)
  - SMA* fonctionne comme A* jusqu'à ce que la mémoire soit pleine
  - Ensuite, l'exploration d'un nouveau nœud nécessitera la suppression du nœud ayant la plus grande valeur f
  - Propriétés
    - complétude : Oui (sauf si la solution est à une profondeur d supérieure à la taille mémoire)

SMA*

- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- **Simplified Memory-bounded A*** (SMA*)
  - SMA* fonctionne comme A* jusqu'à ce que la mémoire soit pleine
  - Ensuite, l'exploration d'un nouveau nœud nécessitera la suppression du nœud ayant la plus grande valeur f
  - Propriétés
    - complétude : Oui (sauf si la solution est à une profondeur d supérieure à la taille mémoire)
    - complexité en temps : $O(b^d)$

SMA*

- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- **Simplified Memory-bounded A*** (SMA*)
  - SMA* fonctionne comme A* jusqu'à ce que la mémoire soit pleine
  - Ensuite, l'exploration d'un nouveau nœud nécessitera la suppression du nœud ayant la plus grande valeur f
  - Propriétés
    - complétude : Oui (sauf si la solution est à une profondeur d supérieure à la taille mémoire)
    - complexité en temps : $O(b^d)$
    - complexité en mémoire : minimum de $O(b^d)$ et taille de la mémoire

SMA*

- **Idée** : utiliser une mémoire limitée pour stocker les noeuds
- **Simplified Memory-bounded A*** (SMA*)
  - SMA* fonctionne comme A* jusqu'à ce que la mémoire soit pleine
  - Ensuite, l'exploration d'un nouveau nœud nécessitera la suppression du nœud ayant la plus grande valeur f
  - Propriétés
    - complétude : Oui (sauf si la solution est à une profondeur d supérieure à la taille mémoire)
    - complexité en temps : $O(b^d)$
    - complexité en mémoire : minimum de $O(b^d)$ et taille de la mémoire
    - optimalité : Oui (même conditions que la complétude)

Recherche heuristiques

Recherche locale

- Vu en optimisation avec M. Idoumghar

- Vu en optimisation avec M. Idoumghar
- Algorithmes génétiques

- Vu en optimisation avec M. Idoumghar
- Algorithmes génétiques
- Essaim particulaire

- Vu en optimisation avec M. Idoumghar
- Algorithmes génétiques
- Essaim particulaire
- Meta-heuristiques