Une image contenant texte

Description générée automatiquement

***- PROJECT REPORT –***

**ARTIFICIAL INTELLIGENCE**

**PROJECT: N-SLIDING PUZZLE**

**2022-2023**

***Project realized by***

***Lina AMRANI-HANCHI***

***Dina BENMOUSSA***

***M1 – I2D***

***Project supervised by***

***Paolo VIAPPIANI***

**INTRODUCTION**

The problem we are solving is the n-sliding puzzle problem where we have n tiles to order and the empty tile having to be on the bottom right corner. We used two approaches, one with informed search and the second with uninformed search.

For the informed search we used the A\* algorithm with two different heuristics (Manhattan distance and displaced tiles heuristic). For the uninformed search we used the Uniform cost search algorithm.

**FORMALIZATION OF THE PROBLEM**

Set of states: determined by the position of the empty tile

Initial State: √(n+1) x √( (n+1) matrix (list of lists) given by the user (randomly or in a file)

Goal State: √( (n+1) x √( (n+1) matrix (list of lists) with ordered tiles

Une image contenant texte

Description générée automatiquement

example of a goal state for a 15-sliding puzzle

Actions:

moving empty tile down / moving a tile up

moving empty tile up / moving a tile down

moving empty tile left / moving a tile right

moving empty tile right / moving a tile left

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementTransition function:

RESULT( , right) =

Regarding the heuristics we chose two heuristics: the Manhattan distance and the displaced tiles heuristic. We implemented the function ‘*heuristic\_fun’* that chooses the heuristic to use depending on the choice of the user.

**Manhattan distance:**

-is admissible because each tile will have to be moved at least the number of cases between itself and its goal place.

-is consistent: n=current state, n’=state after action , c: cost is always 1

h(n) <= h(n’) + c(n,n’)

h(n) - h(n’) <= c(n,n’)

case 1: the action puts a tile in its goal place or moves it closer to its goal state:

h(n’)=h(n)-1

h(n) - h(n’) <= c(n,n’) ó 1<=1 OK

case 2: the action moves a tile to a place further than its goal state:

h(n’)=h(n)+1

h(n) - h(n’) <= c(n,n’) ó -1<=1 OK

h(n) <= h(n’) + c(n,n’)

this inequality always holds so the heuristic is consistent

**Misplaced tiles heuristic:**

-is admissible: because each tile which is misplaced will have to be moved of at least one tile to get to its goal place. So if there are n misplaced tiles we need at least n moves to put them all at their place.

-is consistent:

case 1: the action puts a tile in its goal place:

h(n’)=h(n)-1

h(n) - h(n’) <= c(n,n’) ó 1<=1 OK

case 2: the action moves a tile that is not in its place to a place that is not its goal place :

h(n’)=h(n)

h(n) - h(n’) <= c(n,n’) ó 0<=1 OK

case 3: the action moves a tile that is in its goal place to its goal place:

h(n’)=h(n)+1

h(n) - h(n’) <= c(n,n’) ó -1<=1 OK

h(n) <= h(n’) + c(n,n’)

this inequality always holds so the heuristic is consistent

Both heuristics are admissible and consistent so A\* is optimal.

**IMPLEMENTATION CHOICES**

For the implementation of the project, we began by creating different classes to represent the problem.

1. **Puzzle class**

This class is a representation of the configuration of a n-sliding puzzle.

It is initialized by two attributes: *self.size* and *self.tiles*. *self.size* is the number of columns and rows of the puzzle (given a size n, it will be a (n2-1)-puzzle), and *self.tiles* is the configuration of the tiles of the puzzle that we want to solve.

For this class, we defined a function that is called *find\_empty\_tile*  which is used to find the position of an empty tile of a puzzle and give its indexes. We also defined the *is\_solvable* function which return True if the given puzzle is solvable and False otherwise. The *possible\_actions* function gives all the possible actions that can be done at the current state of the puzzle.

1. **Node class**

The Node class represent the configuration of the nodes of the state space graph where the nodes are the states and the arcs are the actions. A *Node* is initialized by several attributes:

*Self.tiles, self.puzzle, self.g, self.h, self.f, self.parent, self.heuristic, self.goal and self.n.*

*self.tiles* is the configuration of the tiles of the puzzle at the current Node represent as a matrix (a list of lists), *self.puzzle* is the puzzle corresponding to the state as defined above, *self.f* is the value of the evaluation function*, self.h the value of the* heuristic and *self.g the value of the path cost of the current state. self.parent* is the parent node of the current node, *self.goal* is the goal state corresponding to the puzzle we are trying to solve. *Self.n* is the number of columns and rows of the puzzle (given a size n, it will be a (n2-1)-puzzle). And finally, *self.heuristic* is the heuristic function used during the algorithm (‘manhattan’ for the Manhattan distance, ‘displaced’ for the displaced tiles heuristic, both for A\* and fixed to ‘ucs’ when using the Uniform Cost Search algorithm).

We also implemented different functions for this class that are used after. *Def\_goal* allows to define the goal state according to the puzzle the algorithm is going to solve and *isGoal* returns True if the tiles on which the function is applied corresponds to a goal state and False otherwise. The function *heuristic\_fun* defines the computation of the heuristic function for each case, Manhattan distance, displaced tiles or it equal to zero when the algorithm used is Uniform cost search and uploads the values of *f, g and h* of the current node.

1. **Other function implemented**

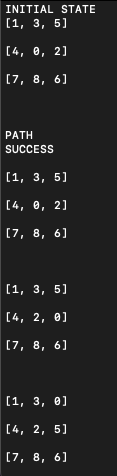
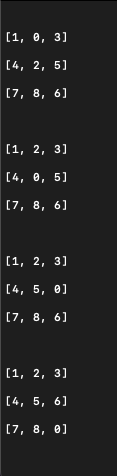
And to proceed with the execution of the algorithm, we also defined function that were useful for that algorithm. The *expand()* function expands the current node, which means that it returns a list of the children of the current node given the possible actions. Then *is\_in\_expanded()* allows to say if a given node has already been expanded, ie if the node is in the list of the expanded nodes. The *find\_node()* function is used to retrieve the indexes of the correct position (position in the goal state) of an element (value of a tile).

Then *print\_path()* allows to print the path of the solution using *print\_state()* to display the tiles in the form of a puzzle.

1. **Implementation of the different algorithms**

**Examples of solving 8-puzzles with the different algorithms:**

file: 8puzzle\_1.txt



A\* with Manhattan distance heuristic:



A\* with Displaced tiles heuristic:



Uniform Cost Search:

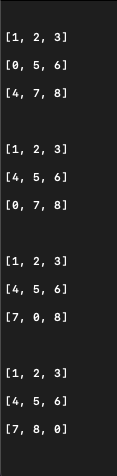


Breadth-First Search:



Une image contenant texte

Description générée automatiquementfile: 8puzzle\_2.txt

Une image contenant texte, écran, noir, fermer

Description générée automatiquement

A\* with Manhattan distance heuristic:

A\* with Displaced tiles heuristic:

Uniform Cost Search:



Breadth-First Search:



file: 8puzzle\_5.txt

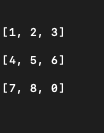
Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

file: 15puzzle\_2.txt

Une image contenant texte

Description générée automatiquementUne image contenant texte, noir, écran, fermer

Description générée automatiquement

**Comparison of the heuristics and algorithms**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Puzzle** |  | Une image contenant texte  Description générée automatiquement | Une image contenant texte  Description générée automatiquement | Une image contenant texte  Description générée automatiquement |
| **A\* with h1**  **(displaced)** | **Time** | 0.004604 | 0.019448 | 30.57770 | 0.00165 |
| **Space Complexity\*** | Path length 7  Frontier 12  Exp nodes 11 | path length 11  frontier 32  exp nodes 38 | path length 20  frontier 1525  exp nodes 2466 | path length 5  frontier 5  exp nodes 4 |
| **A\* with h2**  **(manhattan**) | **Time** | 0.005553 | 0.016477 | 0.496920 | 0.003181 |
| **Space Complexity\*** | path length 7  frontier 9  exp nodes 7 | path length 11  frontier 14  exp nodes 15 | path length 20  frontier 223  exp nodes 325 | path length 5  frontier 5  exp nodes 4 |
| **Uniform cost Search** | **Time** | 0.039179 | 2.337301 | 2627.34362 | 0.01465 |
| **Space Complexity\*** | Path length 7  Frontier 72  exp nodes 120 | path length 11  frontier 523  exp nodes 887 | path length 20  frontier 18351  exp nodes 40276 | path length 5  frontier 50  exp nodes 43 |
| **Breadth first search** | **Time** | 0.015965 | 0.10912 | 293.89876 | 0.004359 |
| **Space Complexity\*** | Path length 7  Frontier 58  Exp nodes 64 | Path length 11  Frontier 305  Exp nodes 437 | path length 20  frontier 15026  exp nodes 28440 | path length 5  frontier 22  exp nodes 16 |

Regarding the examples above, it shows us that the A\* algorithm is more efficient than Unifor Cost Search, for both heuristics. A\* allows to avoid visiting nodes of the tree that can’t take to a shortest path to a goal state.

But A\*’s complexity still is exponential. In the worst case, all the tree needs to be explored.

For A\* itself, both heuristics don’t have the same performance:

The number misplaced tiles heuristic is the easiest for this problem, but it doesn’t give information about how the tiles are misplaced such as how far a misplaced tile is from being correct, so it doesn’t give a really good performance.

The Manhattan distance performs better given that it provides information on the distance of the tile from its goal place than the misplaced tiles one.