Artificial Intelligence project report

👥 Group : Maxime Bourdon, Loïc Etienne

---

1. Introduction (project goal)
2. Formalizing the project as a search problem
3. Definition of different algorithms
    - A*
    - BFS
4. Comparison of executions and results on sample puzzles

Introduction (project goal)

The goal of this project is to solve the N-puzzle game using the A* search algorithm or one of its variants.

We start with a 2-dimensional array consisting of n*n cells. One of these cells will be "0", representing an empty cell, the others will contain numbers starting from 1, which will be unique in this instance of the puzzle.

The search algorithm will have to find a valid sequence of moves in order to reach the final state, which depends on the size of the puzzle. This final state is reached once every tile is set in ascending order, empty one at the end.

Formalizing the project as a search problem

- State : list of integer (location of each tile)
- Actions : move the tiles to left, right, up or down
- Result(state, action, i):

```
if not solvable then return ‘not solvable’

let move;

let dim = sqrt(len(state))
if action == Up then move = state[i-dim];

if action == Down then  move = state[i+dim];
if action == Left then move = state[i-1];
if action == Right then move = state[i+1];

state[i-dim] = i;

i = move;
```

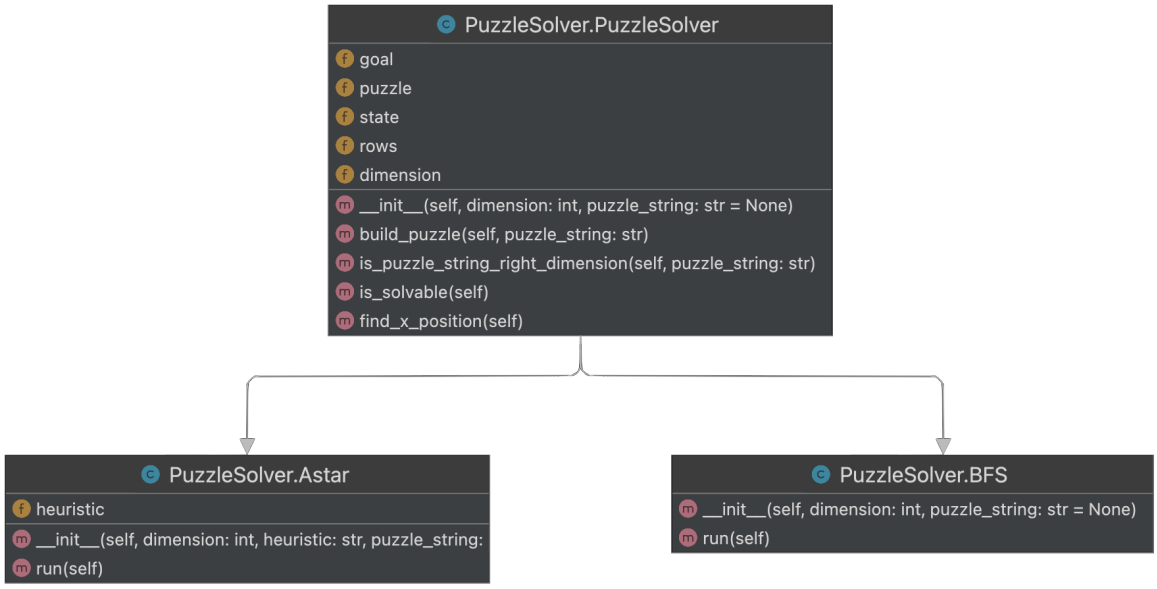
- Goal : list of integers (location of each tile)

Definition of different algorithms

In order to solve this n-puzzle problem, we have realized two algorithms:

- BFS (Breath-First-Search)
- A* (including two heuristics, h1 : Manhattan Distance and h2 : Misplaced Tiles)

These algorithms are implemented as two classes `BFS` & `Astar` inheriting both from a common class `PuzzleSolver`. The latter presents a set of attributes allowing to build the puzzle and to materialize it, as well as a method for this purpose. The latter takes as input a string with integers separated by commas, and builds the matrix associated with the puzzle. This class also presents other methods to check the solvability of the puzzle or the integrity of the input data. Finally, the two classes inheriting from `PuzzleSolver` (`BFS` & `Astar`) implement a `run()` method. The latter returns the result of the execution of the corresponding algorithm, using the attributes of the class.



Before looking at how the algorithms work, it is important to talk about puzzle solvency:

```
First we check the size of the matrix, two cases:

Either it is of odd size (example: 8-puzzle → matrix of size 3*3):

  in this case, if the number of inversion is even, then the puzzle is solvable

Or it is of even size (example : 15-puzzle → matrix of size 4*4)

  in this case:

    if "0" is on an even row and its inversion number is odd

      if "0" is on an odd row and its inversion number is even

        then the puzzle is solvable
```

We have been reusing the source code from GeeksForGeeks for this function.

BFS

Our BFS algorithm is running thanks to the following code :

```
#let nodes equal empty list of nodes
nodes = Queue()
#let explored equal empty list of nodes
explored = set()
# put the start Node into nodes
```

```
initial_node = Node(self.state, None, None, 0, 0, self.goal)
nodes.put(initial_node)
counter = 0

# while nodes is not empty
while not nodes.empty():

    current_node:Node = nodes.get()
    current_state:list = current_node.state

    explored.add(str(current_state))
    # if current_Node is the goal
    if current_node.check_if_its_goal_state():
        #return the solution
        return current_node.solution(self.dimension), len(explored)
    #else generate possible children of the state
    children:List[Node] = current_node.expand(self.dimension)
    for child in children:
        if str(child.state) not in explored:
            counter += 1
            #put the child into nodes
            nodes.put(child)
return
```

A*

Our A* algorithm uses the following code to resolve the search problem :

```
#let nodes equal empty list of nodes
nodes = PriorityQueue()
#let explored_nodes equal empty list of nodes
explored_nodes = []
counter = 0
# let initial_node, the initial Start Node
initial_node = Node(self.state, None, None, 0, 0, self.goal)
if self.heuristic == 'manhattan':
    score = initial_node.manhattan_distance(self.dimension)
else:
    score = initial_node.misplaced_tiles(self.dimension)
# put the start Node into nodes
nodes.put((score, counter, initial_node))
while not nodes.empty():
    #Get the current node
    current_node:tuple = nodes.get()
    current_node:Node = current_node[2]
    explored_nodes.append(current_node.state)
    # if current_Node is the goal
    if current_node.check_if_its_goal_state():
        #return the solution
        return current_node.solution(self.dimension), len(explored_nodes)
    #else generate possible children of the state
    children = current_node.expand(self.dimension)
    for child in children:
        if child.state not in explored_nodes:
```

```
            counter += 1
            #call the heuristic function to generate the score
            if self.heuristic == 'manhattan':
                score = child.manhattan_distance(self.dimension)
            else:
                score = child.misplaced_tiles(self.dimension)
            #put the child into nodes
            nodes.put((score, counter, child))
return
```

This code can use two heuristics specified as an input of the program, when prompted to the user. Those heuristics are coded in by the following two methods of the `Node` class :

Misplaced Tiles heuristic

```
def misplaced_tiles(self,dimension:int):
      counter = 0
      self.heuristic = 0
      for i in range(dimension**2):
          #check if the tile at index i is the same as the tile
          #at index i of the goal state
          if (self.state[i] != self.goal[i]):
              counter += 1
      self.heuristic = self.heuristic + counter
      #once the score is generated, Â
      #we return this score + the path cost (path cost is incremented at each node)
      self.AStar_evaluation = self.heuristic + self.cost

      return self.AStar_evaluation
```

Manhattan distance heuristic

```
def manhattan_distance(self ,dimension:int):
    self.heuristic = 0
    for i in range(1 , dimension**2):
        dist = abs(self.state.index(i) - self.goal.index(i))

    #we generate a score as the distance between the current state and goal state
        self.heuristic = self.heuristic + dist/dimension + dist%dimension
    #once the score is generated,
    #we return this score + the path cost (path cost is incremented at each node)
    self.AStar_evaluation = self.heuristic + self.cost

    return self.AStar_evaluation
```

Comparison of executions and results on sample puzzles

We have set a time limit of 10 minutes for the solving of a puzzle. We obtained results below this limit for puzzle size n = 8 and n = 15. We will compare our results :

for n = 8, we have used the matrix $\begin{bmatrix} 1 & 8 & 7 \\ 3 & 0 & 5 \\ 4 & 6 & 2 \end{bmatrix}$.

for n = 15, we have used the matrix $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 14 & 15 & 0 \\ 13 & 10 & 11 & 12 \end{bmatrix}$.

(you can reuse those programs with the file "puzzles_examples.txt" in the program)

Here are the results for the different algorithms :

For the matrix of size 3*3 :

	Duration	Nodes explored	Moves
BFS	**2057.2 MS**	102.869	22
A* Manhattan	**16.1 MS**	614	22
A* Misplaced Tiles	**4597.0 MS**	11.523	22

For the matrix of size 4*4 :

	Duration	Nodes explored	Moves
BFS	**22.114 S**	1.352.057	19
A* Manhattan	**0.0657 S**	1.014	19
A* Misplaced Tiles	**3.38 S**	9.386	19

We notice that the A* algorithm is the fastest with the heuristic manhattan.

However, the BFS algorithm is faster than the A* with misplaced tiles despite of exploring 10 times more nodes.

It is important to note that the result of the bfs is bad, compared to the A*, 19 seconds of gaps while it was faster in the case of a 3*3 matrix, in addition to an exploration much heavier + 1 million nodes.

Step-by-step resolution of a puzzle

Here is a step by step python-printed resolution of the 4*4 algorithm mentioned above, using both BFS and A* (using both heuristics). The pathway is the same for every algorithm and heuristic.

| 1 2 3 4 |
| 5 6 7 8 |
| 9 14 15 0 |
| 13 10 11 12 |

| 1 2 3 4 |
| 5 6 7 0 |
| 9 14 15 8 |
| 13 10 11 12 |
| || Move -> Up ||

| 1 2 3 4 |
| 5 6 0 7 |
| 9 14 15 8 |
| 13 10 11 12 |
| || Move -> Left ||

| 1 2 3 4 |
| 5 0 6 7 |
| 9 14 15 8 |
| 13 10 11 12 |
| || Move -> Left ||

| 1 2 3 4 |
| 5 14 6 7 |
| 9 0 15 8 |
| 13 10 11 12 |
| || Move -> Down ||

| 1 2 3 4 |
| 5 14 6 7 |
| 9 10 15 8 |
| 13 0 11 12 |
| || Move -> Down ||

| 1 2 3 4 |
| 5 14 6 7 |
| 9 10 15 8 |
| 13 11 0 12 |
| || Move -> Right ||

| 1 2 3 4 |
| 5 14 6 7 |
| 9 10 0 8 |
| 13 11 15 12 |
| || Move -> Up ||

| 1 2 3 4 |
| 5 14 6 7 |
| 9 0 10 8 |
| 13 11 15 12 |
| || Move -> Left ||

| 1 2 3 4 |
| 5 0 6 7 |
| 9 14 10 8 |
| 13 11 15 12 |
| || Move -> Up ||

| 1 2 3 4 |
| 5 6 0 7 |
| 9 14 10 8 |
| 13 11 15 12 |
| || Move -> Right ||

| 1 2 3 4 |
| 5 6 7 0 |
| 9 14 10 8 |
| 13 11 15 12 |
| || Move -> Right ||

| 1 2 3 4 |
| 5 6 7 8 |
| 9 14 10 0 |
| 13 11 15 12 |
| || Move -> Down ||

| 1 2 3 4 |
| 5 6 7 8 |
| 9 14 10 12 |
| 13 11 15 0 |
| || Move -> Down ||

| 1 2 3 4 |
| 5 6 7 8 |
| 9 14 10 12 |
| 13 11 0 15 |
| || Move -> Left ||

| 1 2 3 4 |
| 5 6 7 8 |
| 9 14 10 12 |
| 13 0 11 15 |
| || Move -> Left ||

| 1 2 3 4 |
| 5 6 7 8 |
| 9 0 10 12 |
| 13 14 11 15 |
| || Move -> Up ||

| 1 2 3 4 |
| 5 6 7 8 |
| 9 10 0 12 |
| 13 14 11 15 |
| || Move -> Right ||

| 1 2 3 4 |
| 5 6 7 8 |
| 9 10 11 12 |
| 13 14 0 15 |
| || Move -> Down || | 1 2 3 4 |
| 5 6 7 8 |
| 9 10 11 12 |
| 13 14 15 0 |
| || Move -> Right ||

We can conclude by saying that in solving a n-puzzle, BFS is quite powerless compared to search algorithms like A*.