Intelligence Artificielle 1 Report

N - Puzzle

Faculty of Science and Technology
University of Limoges

presented by

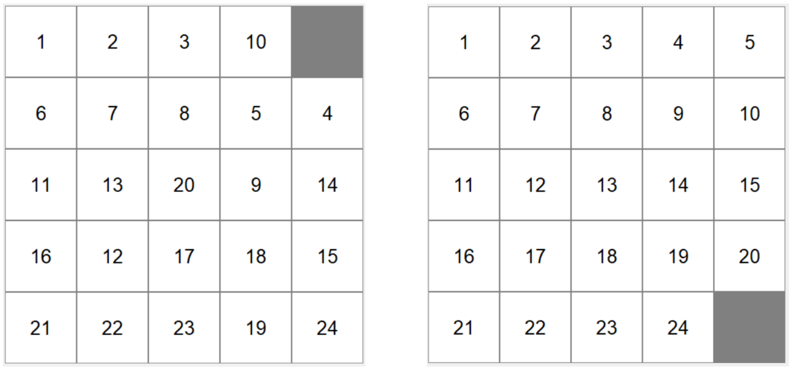
Doan Thi Van Thao
Nguyen Thi Mai Phuong

Master 1 CRYPTIS

January 4, 2021

1 N-Puzzle

N-Puzzle is a sliding-block game on a N * N grid with (N * N) - 1 tiles numbered from 1 to N * N, plus an empty space. The objective is to place the numbers on tiles to their proper orders as given using the empty space. For example,



(a) Initial State (b) Goal State

Figure 1: Example on a 24-Puzzle.

Here the empty space marks the spot to where the elements can be shifted and the final configuration always remains the same the puzzle is solvable.

2 Problem analysis

To solve N-puzzle game, we define three main classes: `State`, `PriorityQueue` and `Operator`.

(a) `State`
We defined this class to present a state of the board as a node in search algorithm. A node contains the following attributes:
- **list_tiles**: tiles of the board represent as a list of integer numbers
- **parent**: the parent of the state.
- **g**: number of moves to reach the state from the initial state.
- **h**: estimated cost from the state to goal state.
- **op**: the move which parent should do to reach this state.

(b) `Operator`
This class acts like an agent, which used to process moving actions to explore new nodes from the current one. Attribute `i` represents moving step.
- **i = 0**: Move the tile down
- **i = 1**: Move the tile up.
- **i = 2**: Move the tile to the right.
- **i = 3**: Move the tile to the left.

Notes: When tracing the result path from the result node, the value of `i` can be reversed.

(c) `PriorityQueue`
This queue allows to manage all nodes explored during the searching process. The queue will be represented as a list of nodes that has been sorted by priority. The priority was calculated based on `State.g` and `State.h`. Main methods of class `PriorityQueue` include:
  - **add**: add a new node the queue.
  - **get**: get the value of the highest priority node (or first element of queue).
  - **remove**: remove a node and re-arrange the queue.
  - **ifcontain**: check if a node exists in the queue or not.

3 Search Algorithms

In this project, we apply A* and Greedy Best-First Search Algorithms to solve N-Puzzle game. In order to compare these two, we apply on them three different heuristics: Number of misplaced tiles, Euclidean distance, and Manhattan distance. The result after all will be evaluated based on two criteria:

1. Number of nodes explored before find out the solution
2. Quality of solution, based on length of path

3.1 A* Algorithm

A* is a generic search algorithm that can be used to find solutions for many problems, especially path-finding and graph traversals. For path-finding, A* repeatedly examines the most promising unexplored location it has seen. When a location is explored, the algorithm is finished if that location is the goal; otherwise, it makes note of all that location's neighbors for further exploration.

The basic idea of A* is to avoid expanding paths that are already expensive according to a valuation function $f(n)$:

- $g(n)$ = cost so far to reach n
- $h(n)$ = estimated cost from n to goal
- $f(n) = g(n) + h(n)$, estimated total cost of path through n to goal

If the space of the states is finite and there is a solvable method to avoid reviewing (repeating) states, then the algorithm A* is complete (can find the solution) - but it is not guaranteed to be optimal.

3.2 Greedy Best-First Search Algorithm

One of the simplest best-first search strategies is to minimize the estimated cost to reach the goal[1]. That is, the node whose state is judged to be closest to the goal state is always expanded first. This is the main idea of Greedy Best-First Search.

The valuation function which calculates such cost estimates is a heuristic function, denoted by the letter f:

1. $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.
2. $f(n) = h(n)$.

Formally speaking, h can be any function but $h(n) = 0$ if n is a goal[1].

The worst-case time complexity for greedy search is $O(b^m)$, where m is the maximum depth of the search space[1]. With a good heuristic function, the space and time complexity can be reduced substantially. The Greedy BFS algorithm selects the path which appears to be the best, it can be known as the combination of depth-first search and breadth-first search. However, it also suffers from the same defects as theirs, that is if it is not optimal, then it is incomplete because it can start down an infinite path and never return to try other possibilities.

3.3 Admissible heuristic

A h function is called an *admissible heuristic*[1] if it never overestimates the cost to reach the goal. This optimism transfers to the f function as well: If h is admissible, $f(n)$ never overestimates the actual cost of the best solution through n. In the scope of the project, we have chosen three admissible heuristics as stated above:

1. **Number of misplaced tiles**: Compute the heuristic function $h1$ by counting the number of different tiles between the puzzle's initial state and target state. $h1$ is admissible, since it is clear that every tile that is out of position must be moved at least once.
2. **Euclidean distance**: Compute the heuristic $h2$ by sum of Euclidean distances of the tiles from their goal. $h2$ is admissible, since in every move, one tile can only move closer to its goal by one step and the euclidean distance is never greater than the number of steps required to move a tile to its goal position.
3. **Manhattan distance**: Compute the function $h3$ by sum of Manhattan distances of the tiles from their goal. $h3$ is an admissible heuristic, since in every move, one tile can only move closer to its goal by one step.

4 Implementation

As presented above, in A* Algorithm, $f(n) = g(n) + h(n)$, while in Greedy Best-First Search $f(n) = h(n)$ only. Therefore, with the aim of making more common resources for both 2 algorithms, we used the same method of calculating $f(n)$ for all two but fixed the value of $g(n)$ in Greedy Best-First Search to 0. Thus, searching steps are as below:

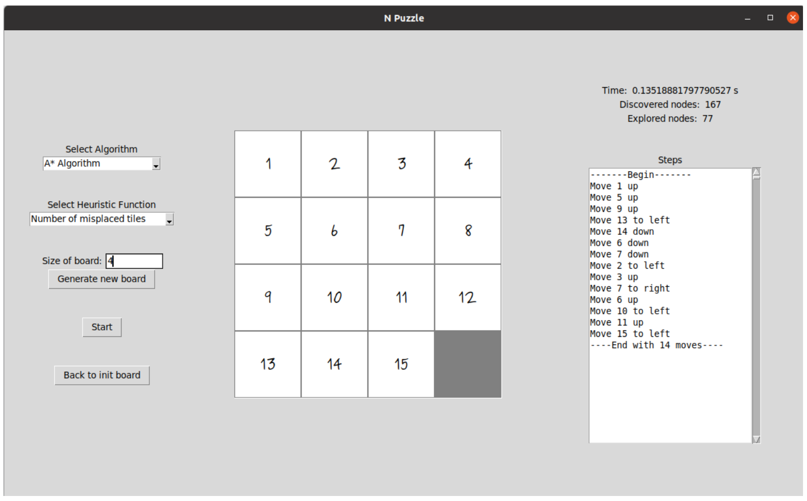
1. **Step 1**: Add initial state of the board to `start` (Start priority queue).

[1]Russell, S. J.,& Norvig, P. (1995). *Artificial intelligence: A modern approach.* Englewood Cliffs, N.J: Prentice Hall.

2. **Step 2**: Pop highest priority state (state `O`) from `start`, add it to `close` (Close priority queue that contains all explored states) and check whether `O` is the goal state or not. Then, we continue with **Step 3a** (if `O` is the goal state) or **Step 3b** (if not).
3. **Step 3a**: Begin to trace the path from `O` by using the attribute `parent` of class `State`. Finish search function.
4. **Step 3b**: Explore new possible states from `O` and add them to `start` if they do not exist in both `start` and `close` queues. Then come back to **Step 2**.

5 Graphical User Interface (GUI)

We use the library `tkinter` to build the GUI. As we can see in Figure 2, this application allows the user to choose a search algorithm (between A* and Greedy Best-First Search Algorithm) to execute on the left side. Moreover, the heuristic function (Number of misplaced tiles, Euclidean distance and Manhattan distance) and size of the game-board also can be changed.



Select Algorithm: A* Algorithm

Select Heuristic Function: Number of misplaced tiles

Size of board: 4

Time: 0.13518881797790527 s
Discovered nodes: 167
Explored nodes: 77

```
-------Begin-------
Move 1 up
Move 5 up
Move 9 up
Move 13 to left
Move 14 down
Move 6 down
Move 7 down
Move 2 to left
Move 3 up
Move 7 to right
Move 6 up
Move 10 to left
Move 11 up
Move 15 to left
----End with 14 moves----
```

Figure 2: Graphic User Interface - Result state

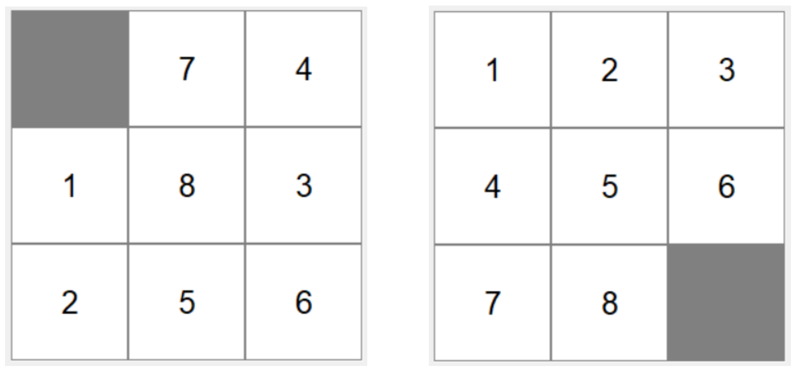
On the right-hand side, steps to solve the N-Puzzle game will be displayed one by one along with simulation in the center board. In addition, the total time to solve, number of discovered nodes, as well as the number of explored nodes needed to find the final result will also be indicated.

6 Experimental results

In this section, we will illustrate the results returned by A* and Greedy Best-First Search Algorithm, implemented with different heuristics in different initial and goal states.

Test Case 1 (TC1)

For general evaluation, we begin with a test case generated by a random function.



(a) Initial State (b) Goal State

Figure 3: Initial and Goal States of TC1

Heuristic Functions	**Results**	
	A* Algorithm	**Greedy B-F-S Algorithm**
Number of misplaced tiles	22 moves Time: 417.5540 Discovered nodes: 12343 Explored nodes: 7938	50 moves Time: 0.8416 Discovered nodes: 493 Explored nodes: 299
Euclidean distance	22 moves Time: 5.4388 Discovered nodes: 1375 Explored nodes: 866	36 moves Time: 0.1868 Discovered nodes: 177 Explored nodes: 108
Manhattan distance	22 moves Time: 5.0458 Discovered nodes: 1375 Explored nodes: 866	36 moves Time: 0.1815 Discovered nodes: 177 Explored nodes: 108

Table 1: Results of TC1

Test Case 2 (TC2)

In this test case, we increase the size of the matrix with random initial state. Initial and Goal States are presented on Figure 1 on section 1.

Heuristic Functions	**Results**	
	A* Algorithm	**Greedy B-F-S Algorithm**
Number of misplaced tiles	18 moves Time: 2.4253 Discovered nodes: 649 Explored nodes: 286	164 moves Time: 314.2072 Discovered nodes: 7117 Explored nodes: 3125
Euclidean distance	18 moves Time: 0.1256 Discovered nodes: 123 Explored nodes: 52	26 moves Time: 0.1527 Discovered nodes: 127 Explored nodes: 53
Manhattan distance	18 moves Time: 0.1290 Discovered nodes: 123 Explored nodes: 52	26 moves Time: 0.1435 Discovered nodes: 127 Explored nodes: 53

Table 2: Results of TC2

Analysis of the Experimental Results

As we can see in Table 1 and Table 2, heuristic functions have high-impact to the number discovered and explored nodes as well as the quality of solutions. With *Number of misplaced tiles* heuristic, the number of discovered and explored nodes is a lot, which leads to very long searching time. Also, the quality of found solutions by using this heuristic is worse than other. On the contrary, the results using *Euclidean distance* and *Manhattan distance* are relatively similar and also are evaluated as better results.

7 Conclusion

Based on three above test-cases and lots of others, it is concluded that the results of A* Algorithm are always better than Greedy Best-First Search's with fewer moves. However, the minimum number of steps to reach the goal had to trade off by a large number of discovered and explored nodes. As another option, Greedy Best-First Search Algorithm does not get the optimal solutions but the time that needed to find out the final result is less than A* Algorithm.

From the above comparison, we conclude that A* Algorithm and Greedy-Best First Search have different strengths and also weaknesses. Depend on particular case we will decide for the compatible algorithm. However, we strongly recommend to use two heuristics Euclidean distances and Manhattan distance to optimize the searching time.

Appendix

The Appendix displays experimental implementation in the program.

```python
class State:
    def __init__(self):
        # parent, op: toan tu
        self.list_tiles = None
        self.parent = None
        self.g = 0
        self.h = 0
        self.op = None

    def getvalues(self):
        if self.list_tiles is None:
            return None
        res = ''
        for x in self.list_tiles:
            res += str(x)
        return res

    def __lt__(self, other):
        if other is None:
            return False
        return self.g + self.h < other.g + other.h

    def docopy(self):
        sn = deepcopy(self)
        return sn

    def printmatrix(self):
        size = matrixsize
        for i in range(size):
            for j in range(size):
                print(self.list_tiles[i * size + j], end=' ')
            print()
        print()

    def getposbyvalue(self, val):
        sz = matrixsize
        for i in range(sz):
            for j in range(sz):
                if self.list_tiles[i * sz + j] == val:
                    return i, j
```

Listing 1: Class State

```python
class PriorityQueue:
    def __init__(self):
        self.queue = []

    def add(self, item):
        heappush(self.queue, item)

    def get(self):
        return self.queue[0]

    def remove(self, item):
        self.queue.remove(item)
        heapify(self.queue)
        h = []
        for value in self.queue:
            heappush(h, value)
        self.queue = h

    def __len__(self):
        return len(self.queue)

    def empty(self):
        return len(self.queue) == 0

    def ifcontain(self, node):
        if node is None:
            return False
        for value in self.queue:
            if equal(node, value):
                return True
        return False
```

Listing 2: Class PriorityQueue

```python
class Operator:
    def __init__(self, i):
        self.i = i

    def checkstate(self, c):
        if c.list_tiles is None:
            return True
        return False

    def position0(self, c):
        sz = matrixsize
        for i in range(sz):  # i= 0 1 2
            for j in range(sz):
                if c.list_tiles[i * sz + j] == 0:
                    return i, j

    def moveup(self, s):
        if self.checkstate(s):
            return None
        x, y = self.position0(s)
        if x == 0:
            return None
        return self.swap(s, x, y, self.i)
    #do the same with movedown, moveright and moveleft

    def swap(self, s, x, y, i):
        ,,,

    def move(self, s):
        if self.i == 0:
            return self.movedown(s)
        if self.i == 1:
            return self.moveup(s)
        if self.i == 2:
            return self.moveright(s)
        if self.i == 3:
            return self.moveleft(s)
        return None
```

Listing 3: Class Operator

```python
# For A*
def RUN_A(S, G):
    start = PriorityQueue()
    close = PriorityQueue()
    S.g = 0
    S.h = hx(S, G)
    start.add(S)
    while True:
        if start.empty():
            print('searching failed')
            return
        O = start.get()
        start.remove(O)
        close.add(O)  # get O from Open and add into Close
        if equal(O, G):
            print('success')
            tracepath(O)
            return
        for i in range(4):  # i = 0 1 2 3
            op = Operator(i)
            child = op.move(O)  # child is the next node
            if child is None:
                continue
            if not start.ifcontain(child) and not close.ifcontain(child):
                child.parent = O  # set parent is O
                child.op = op
                child.g = O.g + 1
                child.h = hx(child, G)
                start.add(child)
```

Listing 4: Implementation of A* Algorithm

```python
# For Greedy
def RUN_B(S, G):
    start1 = PriorityQueue()
    close1 = PriorityQueue()
    S.g = 0
    S.h = hx(S, G)
    start1.add(S)
    enodes = 0
    dnodes = 0
    while True:
        if start1.empty():
            print('searching failed')
            return
        O = start1.get()
        enodes = enodes + 1
        start1.remove(O)
        close1.add(O)  # get O from Open and add into Close
        if equal(O, G):
            print('success')
            tracepath(O)
            return
        for i in range(4):  # i = 0 1 2 3
            op = Operator(i)
            child = op.move(O)  # child is the next node
            if child is None:
                continue
            if not start1.ifcontain(child) and not close1.ifcontain(child):
                child.parent = O  # set parent is O
                child.op = op
                child.g = 0
                child.h = hx(child, G)
                start1.add(child)
```

Listing 5: Implementation of Greedy BFS Algorithm