COMPUTER SCIENCE AND ENGINEERING

Artificial Intelligence
UE17CS325

Vishwas N.S	PES1201701321
Pavan Mitra	PES1201700239
Abhishek M.	PES1201701563

Comparative Implementation of state-space search based solutions of the M x N-Puzzle problem

Contents

1 History Bits **2**

2 Relevance in Today's context **2**

3 Keywords and Concepts **2**

3.1 State Space 3

3.1.1 Node 3

3.1.2 D_from_Goal 3

3.2 Traversal 3

3.2.1 Cost 3

3.2.2 Path 3

4 Possible Solutions to the problem **3**

4.1 DFS 4

4.2 BFS 4

5 Introducing Heuristics **5**

5.1 A* search Algorithm 5

6 Improving on A* **6**

7 A* with various distance measures **6**

7.1 Manhattan distance 6

7.2 Misplaced Tile Count 6

8 VAP distance **7**

9 OBSERVATIONS **8**

10 Possible Improvements **8**

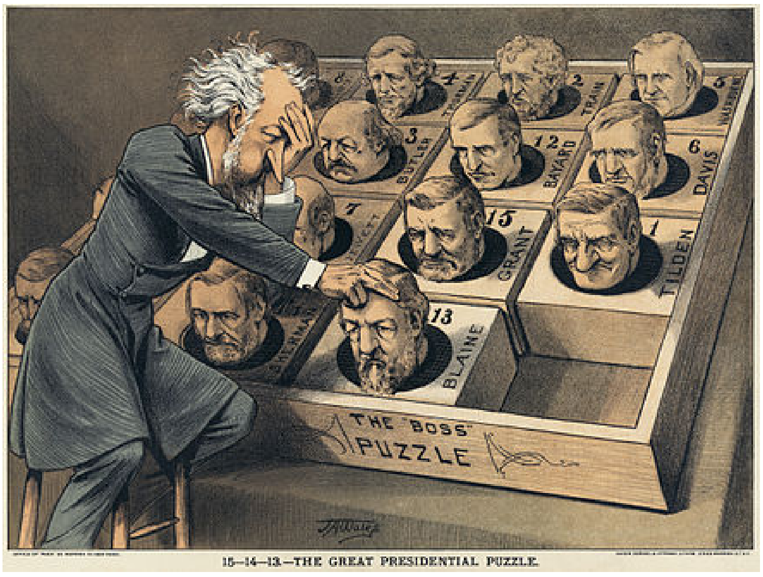
11 Implementation **9**

12 Conclusion **9**

1 History Bits

The N-Puzzle [originally 15 puzzle] is said to have been "invented" by Noyes Palmer Chapman, way back in 1874. It is said to have waded off of the 16-puzzle, a predecessor consisting of 16 numbered blocks that were to be put together in rows of four, each summing to 34.

Interestingly, the first strands of mass production were carried out by students of "The American School for the deaf" in Boston, Massachusetts. The game quickly became a craze in early 1880, and shipped to Japan in 1889.



2 Relevance in Today's context

In the 21st century however, the n-puzzle is a powerful learning concept that is used as a concept base in the fields of Logic, Learning and Artificial Intelligence.

3 Keywords and Concepts

We use the following terms to logically formulate the problem.

3.1 State Space

3.1.1 Node

Every element in the State Space tree, that represents a snapshot/**state** of the puzzle at time *t*.

3.1.2 D_from_Goal

An optional metric, that evaluates how quantitatively distant,a given state is, from the goal state.

3.2 Traversal

The act of changing the current state from one node to another, by means of committing a **move**.

3.2.1 Cost

The overhead sustained in committing a traversal. It is the property of an edge in the state space tree.

3.2.2 Path

A set of abut edges, that could connect 2 nodes anywhere in the tree.

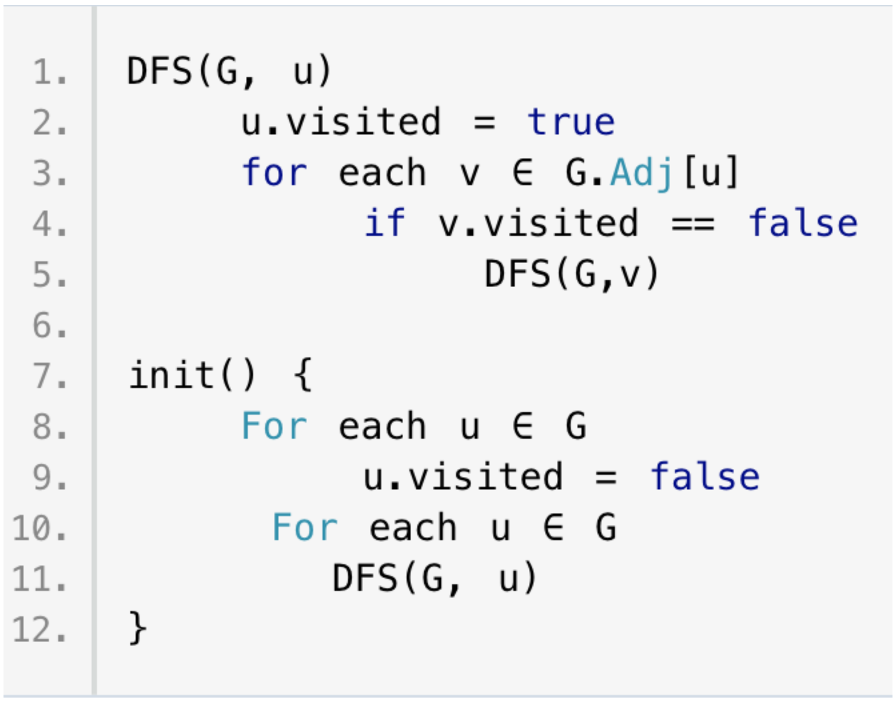
4 Possible Solutions to the problem

Johnson & Story[1879] showed that provided m and n are at least 2, all even permutations are *solvable*. A brute force approach would be to enumerate all possible moves, given a certain state, pick one at random and execute it. Suffice to say, this is inefficient. It ends up with a worst case time $\mathcal{O}(h * n^n)$ where h is the number of moves allowed at each state, which is typically 4.

Not to mention, not only are some states not solvable, but also some movesets might end up in an infinite loop [ex. LRLRLR... or UDUD...] Going one step further would be to organise when and how we make these state traversals.

Heuristically, we could say the best move forward would be to ”stick to” one method(say m1) of traversal and ”switch to” another(say m2) once all possible m1 are done and solution has not yet been found, which we formally name Depth First Search (DFS).

4.1 DFS



```
DFS(G, u)
        u.visited = true
        for each v ∈ G.Adj[u]
                if v.visited == false
                        DFS(G,v)

init() {
        For each u ∈ G
                u.visited = false
          For each u ∈ G
              DFS(G, u)
}
```

Another way we could advance from brute force following a similar thought process, would be to ”finish all” possible traversals, and if none of them are a solution, do the same to each state recursively. Formally, we would call this Breadth First Search (BFS).

4.2 BFS

Noticing that although these methods cut down on traversal time by ordering the move-sets rather than randomising them, they miss out on an opportunity. The opportunity is that each state encodes information about itself, and we could exploit this information to take move decisions.

```
procedure BFS(G,start_v):
    let Q be a queue
    label start_v as discovered
    Q.enqueue(start_v)
    while Q is not empty
        v = Q.dequeue()
        if v is the goal:
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered:
                label w as discovered
                w.parent = v
                Q.enqueue(w)
```

5 Introducing Heuristics

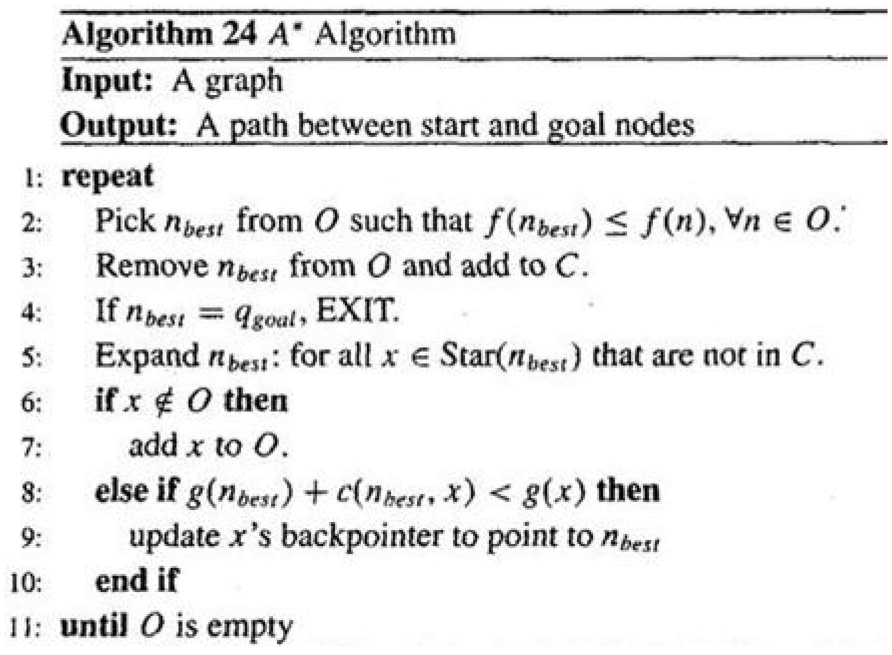
A heuristic is a semi-formed idea or formulation, which means it is meaningless by itself, but can be used to quantify a state, which gives us a mathematical base for choosing moves, and hence enhance pathfinding.

5.1 A* search Algorithm

A* is the most popular choice for pathfinding, because it's fairly flexible and can be used in a wide range of contexts.

A* is like Dijkstra's Algorithm in that it can be used to find a shortest path. A* is like Greedy Best-First-Search in that it can use a heuristic to guide itself. In the simple case, it is as fast as Greedy Best-First-Search.

It works by computing distance, through a function that takes in current state and goal state, and outputs a numeric value that represents how far away we are from the goal. It next computes the same for state after every possible move, and picks the most favourable one.



Algorithm 24 A^* Algorithm

Input: A graph

Output: A path between start and goal nodes

```
1: repeat
2:   Pick $n_{best}$ from $O$ such that $f(n_{best}) \leq f(n), \forall n \in O$.
3:   Remove $n_{best}$ from $O$ and add to $C$.
4:   If $n_{best} = q_{goal}$, EXIT.
5:   Expand $n_{best}$: for all $x \in \text{Star}(n_{best})$ that are not in $C$.
6:   if $x \notin O$ then
7:     add $x$ to $O$.
8:   else if $g(n_{best}) + c(n_{best}, x) < g(x)$ then
9:     update $x$'s backpointer to point to $n_{best}$
10:  end if
11: until $O$ is empty
```

6 Improving on A*

As is obvious, the effectiveness of our A* results depends, to a considerable extent on the quality of metric used to quantify a state(Here,D_from_goal which we shall further refer to as the distance function). The better, a representation of the state our function is, the better the results.

7 A* with various distance measures

7.1 Manhattan distance

$$H = \Sigma \quad [abs(current\_cell.x - goal.x) + abs(current\_cell.y - goal.y)] \tag{1}$$

Uses the above equation as metric

7.2 Misplaced Tile Count

```
""" Calculates the different between the given puzzles """
temp = 0
```

```
for i in range(0,self.n):
    for j in range(0,self.n):
        if start[i][j] != goal[i][j] and start[i][j] != '_':
            temp += 1
return temp
```

So we've observed how a few standard mappings fare against the state space of the sliding puzzle. But can we do better?? What if we manually affix a distance function specific to the problem?

8 VAP distance

We combine both Manhattan distance an the intuition that our agent start solving from a corner.

We do this by assigning weights to the Manhattan distances calculated, rather than taking an absolute sum.

$$VAP = \Sigma \quad [(m * n - k)[(current\_cell.x – goal.x) + (current\_cell.y – goal.y)]] \tag{2}$$

9 OBSERVATIONS

So we have a few search algorithms, we have a few results, what do they conclusively say about the n*m-puzzle

- While the VAP-enabled A* shows least time, it also accrues the overhead of having to quantify VAP-distance, and hence is beat down by blind search algorithms for small values of m and n.
- The distance metric used is specific to the problem, since different mappings cover different sub-spaces of the entire state space.
- We assume that the time taken to compute aforementioned metric is constant (which is pretty much true), which allows for massive scalability.

  t $\in \mathcal{O}(1)$

- It is interesting to note that while BFS internally uses a queue, A* internally uses a priority queue. Some operations(ex. Dequeue) are less expensive on a queue. $[\mathcal{O}(n) vs$[O$(\log(n))$

- While A* differs from Best First Search in that it also takes into account, the cost associated with moves, we pretty much are doing a BestFS here in the name of A*,since we rightly assume that each move has the same cost i.e.

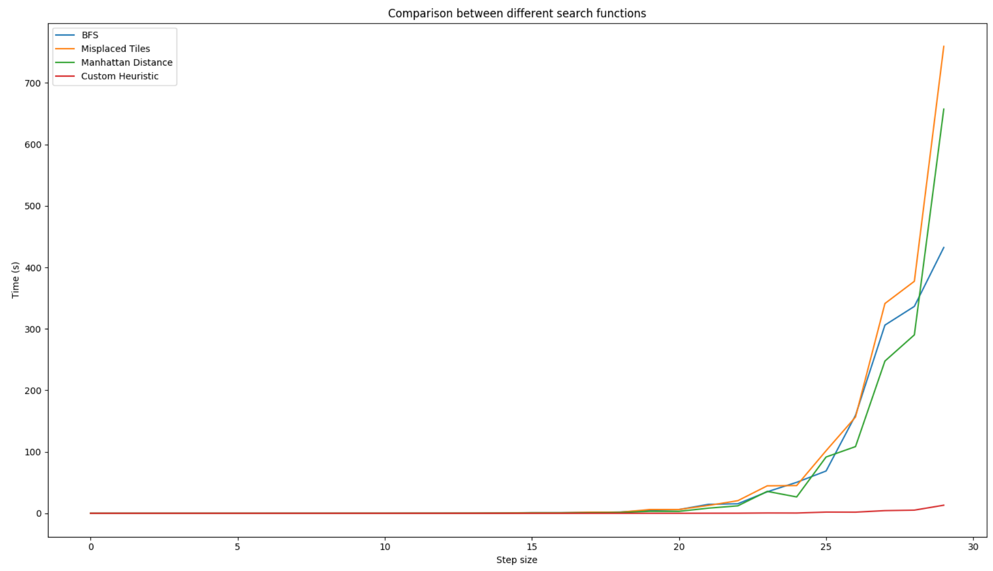
  c[L]=c[R]=c[D]=c[U]

10 Possible Improvements

- A possible improvement is a hybrid tool that picks the most suitable algorithm by itself, by virtue of 'm' and 'n'.
- Further, we could use Calculus (or numerical methods) to regress the best fit distance function, based on current set of state spaces, and re-iterate the function as we see better states
- This would add the additional requirement that the seed state be solvable. Else, function weights may not converge.
- We could allow for fine-tuning the "greediness" of A* manually, based on space-solvability tradeoff. A greedier algorithm saves space but lowers solvability (and possibly increases time) and vice-versa.

11 Implementation

A range of readings were taken for 30 plot-points (SS = number of steps we are, from the goal state).
We generate state spaces that are a given (SS) number of steps away from the solution, so we can get a quantitative measure of efficacy. This is plotted against time on Y.



12 Conclusion

- It is evident from the graph that the exponential time decay is abhorrently apparent, starting from (SS $\simeq$ 20).

- Our custom distance metric pulls through traditional algorithms by a long stretch, and is apparent by the linear slope even at SS $\simeq$ 30