**Project 1 CS205 Artificial Intelligence**

Achyuth Madhav Diwakar Instructor: Dr. Eamonn Keogh

SID: 862055916

[adiwa001@ucr.edu](mailto:adiwa001@ucr.edu)

01/26/2018

In completing this homework, I have consulted:

<http://www.cs.ucr.edu/~eamonn/205/> : Slides for heuristic search and blind search

<https://stackoverflow.com/questions/5805602/how-to-sort-list-of-objects-by-some-property>

<http://www.java67.com/2012/10/how-to-sort-object-in-java-comparator-comparable-example.html> : For the implementation of the queue structure which sorts based on cost function.

<https://math.stackexchange.com/questions/293527/how-to-check-if-a-8-puzzle-is-solvable>

<https://www.geeksforgeeks.org/check-instance-15-puzzle-solvable/> : used the source to learn about the conditions in which an 8 or 15 puzzle has a solution.

Key code components are written by me and all the solutions, graphs and output trace are my work only. Some code snippets like that of puzzle.hasSolution() have been written by analyzing the sources above.

**Introduction**

The Java application PuzzleSolver was created for the first project in the CS205 Artificial Intelligence Course by Prof. Eamonn Keogh at the University of California Riverside.

The application solves the 8 - puzzle. and provides a choice of three algorithms for the same. The user can select between the Uniform Cost Search, A\* using Manhattan distance heuristic and A\* using misplaced tile heuristic.

The application was developed using Java (version 7) and has an interactive interface that works through the standard output, the input from the user is taken through the standard input.

**Design**

The application consists of three classes described below:

**State.class**: Implements the Comparable interface. Consists of the puzzleState, gVal (g(n)) and hVal (h(n)) as fields. The class consists of getters and setters for the same. It implements the compareTo method of the Comparable interface. It takes a State object as argument and compares the cost (hVal + gVal) of itself to that of the State in the parameter, if the cost of the parameter’s state is equal to its cost it returns **0**, if lesser the method returns **1**, else it returns **-1**.

**NPuzzle.class**: It consists of the fields for goalState, rootNode (initial state), a priority queue of the states (used to store the tree structure), an enum structure for the movements of the tiles and attributes for puzzle size. The class also contains the methods for solving the puzzle by Uniform Cost Search (solveByMethod) which takes the algorithmChoice of the user as parameter and runs the appropriate method (default being the uniform cost). The methods, solveByMisplacedTile() and solveBymanhattan(), compute the h(n) value using the respective methods. The move method takes the currentState and direction to move the tile in that direction. The goalStateReached() method checks if the goal state is reached and prints the trace on the standard output. The notRepeatedState() method checks if the current state is not repeated, only then it is expanded. The class also consists of an “isValid()” method which checks the validity of the puzzle input.

**PuzzleSolver.class**: This class consists of the main() method which handles the interactive interface of the application and has methods to take inputs and run the NPuzzle methods to solve the puzzle.

**Adaptability**

The code was written as generally as possible, with all the loops and other identifiers being created with variable properties. The size of the puzzle (8, 15 – puzzle, etc.) can be changed by altering the *PUZZLE\_SIZE* field in the PuzzleSolver class and the application adapts to the change by taking the appropriate input, creating a goal state and runs the algorithms on the new puzzle size. The default input provided in the PuzzleSolver class is however, hardcoded for the 8 – puzzle only. The application was run once with a 15-puzzle and works accurately.

**Algorithms**

The initial state is first expanded in all its possible subordinate states by checking if the blank space can be moved left, right, up and down. Out of all the valid moves new states are formed. Of these new states, the best state to expand is chosen using the cost calculated by the algorithm chosen by the user. In each iteration, the state is checked for a match with the goal state and also for repeated states.

**Uniform Cost Search**

This method searches the branches with the same cost. It is basically the A\* algorithm with h(n) set to zero. The cost of expanded node is g(n) which is one.

**Misplaced Tile Heuristic**

In this method, each non-blank element in the current state is compared to the goal state and if, it is different. the ‘h’ value is incremented. After traversing through the entire grid, the final ‘h’ value is set to that node. For example,

**Goal State: 1 2 3 Puzzle State: 1 2 4**

**4 5 6 3 0 6**

**7 8 0 7 8 5**

The underlined elements are the non-matching ones between puzzle state and goal state. The blank element in the puzzle state is ignored. Therefore, the misplaced tile heuristic is **3**. The h(n) value for this puzzle state is set as **3**.

**Manhattan Distance Heuristic**

This heuristic is calculated by finding the number of places the initial state of the element is away from its place in the goal state. This value is calculated for each element and then the sum of those values is set as the h(n) value for that state.

**Goal State: 1 2 3 Puzzle State: 1 2 4**

**4 5 6 3 0 6**

**7 8 0 7 8 5**

h4= 1 + 2 = 3, h3 = 1 + 2 = 3, h5 = 1 + 1 = 2

Therefore h(n) for this state = 3 + 3 + 2 = 8

**Results**

The PuzzleSolver was tested with the following 5 puzzles:

**P1: 1 2 3 P2: 4 8 0 P3: 3 5 8**

**4 5 6 6 5 7 4 2 6**

**7 8 0 3 2 1 0 1 7**

**P4: 5 1 8 P5: 5 1 3**

**2 4 6 8 6 0**

**7 3 0 2 7 4**

**Number of States Expanded**

Arranged in increasing order of difficulty.

|  |  |  |  |
| --- | --- | --- | --- |
| **Puzzle** | **Uniform Cost** | **A\* Misplaced Tile** | **A\* Manhattan Dist.** |
| **P1** | **39** | **7** | **5** |
| **P3** | **38872** | **2195** | **226** |
| **P5** | **65990** | **4249** | **368** |
| **P4** | **136149** | **14352** | **2179** |
| **P2** | **177645** | **50680** | **1573** |

**Fig 1.1 Number of expanded Nodes for each puzzle**

From the fig. 1.1 it can be seen that, for the relatively simpler puzzle P1 the difference in the number of nodes expanded is too small to be observed on the graph, whereas with the more challenging variants (P2, P4) it can be seen just how few the expanded nodes are in Manhattan heuristic is comparison to Uniform Search and Misplaced tile.

**Maximum Number of Nodes in the Queue**

|  |  |  |  |
| --- | --- | --- | --- |
| **Puzzle** | **Uniform Cost** | **A\* Misplaced Tile** | **A\* Manhattan Dist.** |
| **P1** | **50** | **16** | **11** |
| **P3** | **33472** | **2358** | **253** |
| **P5** | **48817** | **4193** | **369** |
| **P4** | **61936** | **13816** | **2105** |
| **P2** | **62101** | **41121** | **1914** |

**Fig. 1.2 Maximum size of the queue**

From the fig. 1.2 it can be observed that, the size of the queue is much smaller in the case of the heuristic search algorithms in comparison to the uniform cost search.

**Conclusion**

It can be seen from the result that the performance of Uniform Cost Search is the worst among the three, this is because the A\* algorithm with h(n) = 0 has a time complexity of O(b^d). Among the other heuristic search algorithms, the Manhattan distance method shows the best performance. Both the Manhattan distance and the Misplaced Tile heuristic methods greatly improve the performance from the Uniform Cost Search as is evident from the results.

**Output Trace**

Welcome to the 8-Puzzle Solver

Type '1' to use a default puzzle, or '2' to enter your own puzzle.

1

Using default puzzle

1 2 3

4 8 0

7 6 5

Enter your choice of algorithm

1. Uniform Cost Search

2. A\* with the Misplaced Tile heuristic.

3. A\* with the Manhattan distance heuristic

3

The best state to expand with a g(n)=0 and a h(n)=0 is:

1 2 3

4 8 0

7 6 5

The best state to expand with a g(n)=1 and a h(n)=4 is:

1 2 3

4 8 5

7 6 0

The best state to expand with a g(n)=2 and a h(n)=3 is:

1 2 3

4 8 5

7 0 6

The best state to expand with a g(n)=3 and a h(n)=2 is:

1 2 3

4 0 5

7 8 6

The best state to expand with a g(n)=4 and a h(n)=1 is:

1 2 3

4 5 0

7 8 6

Goal!!

To solve this problem search algorithm expanded a total of 5 states.

The maximum number of nodes in the queue at any one time was 11

Depth of the goal:5

Enter 1 to use different algo OR 0 to exit.