National University of San Agustin
System Engineer School

ADA - Lab 06 - A*

Christian E. Portugal-Zambrano

November 05, 2018

1 Introduction

In this practice we will be introducing an algorithm to solve the 8-puzzle problem using the A^* search algorithm. This practices is based on Princeton - 8-puzzle

2 Pre-requisites

You need your heap implemented from the last practice, here we will be using it as a priority queue, some good OOP practices will be necessary.

3 Programming environment

You can use all the tools described on practices before, but here you will need some graphics tools, I recommend to use Netbeans for Java or Qt for C++, both tool brings up with a graphics library enough for our practices objectives.

4 Algorithms presentation

So, as we are working with LaTeX algorithms presentation will be easy, you just need to add this at the header of your document:

```
\usepackage{algorithm}
\usepackage{algorithmicx}
\usepackage[noend]{algpseudocode}
```

4.1 The 8-puzzle problem

The 8-puzzleis a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order, using as few moves as possible. You are permitted to slide blocks horizontally or vertically into the blank square. The following Figure 4.1shows a sequence of legal moves from an initial board.

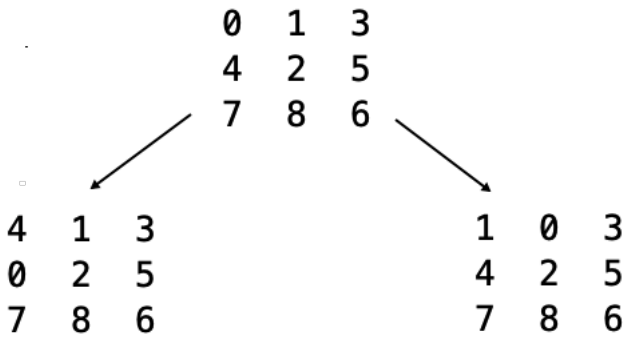

Figure 4.1: Possible moves from a parent board

4.2 Best First Search

Now, we describe a solution to the problem that illustrates a general artificial intelligence methodology known as the $A*$ search algorithm. We define a search node of the game to be a board, the number of moves made to reach the board, and the predecessor search node. First, insert the initial search node (the initial board, 0 moves, and a null predecessor search node) into a priority queue. Then, delete from the priority queue the search node with the minimum priority, and insert onto the priority queue all neighboring search nodes (those that can be reached in one move from the dequeued search node). Repeat this procedure until the search node dequeued corresponds to a goal board. The success of this approach hinges on the choice of priority function for a search node. We consider two priority functions:

- Hamming priority function: The number of blocks in the wrong position, plus the number of moves made so far to get to the search node. Intuitively, a search node with a small number of blocks in the wrong position is close to the goal, and we prefer a search node that have been reached using a small number of moves.

- Manhattan priority function. The sum of the Manhattan distances (sum of the vertical and horizontal distance) from the blocks to their goal positions, plus the number of moves made so far to get to the search node.

In Figure 4.2 we can appreciate the results of both metrics:

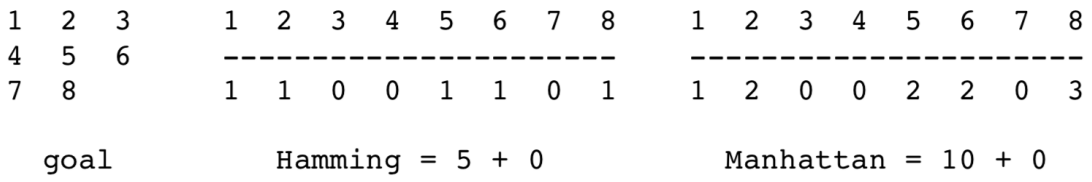

Figure 4.2: Heuristics of moves for each board using Hamming or Manhatan

We make a key observation: To solve the puzzle from a given search node on the priority queue, the total number of moves we need to make (including those already made) is at least its priority, using either the Hamming or Manhattan priority function. (For Hamming priority, this is true because each block that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each block must move its Manhattan distance from its goal position. Note that we do not count the blank square when computing the Hamming or Manhattan priorities.) Consequently, when the goal board is dequeued, we have discovered not only a sequence of moves from the initial board to the goal board, but one that makes the fewest number of moves. (Challenge for the mathematically inclined: prove this fact.)

One Optimization Best-first search has one annoying feature: search nodes corresponding to the same board are enqueued on the priority queue many times. To reduce unnecessary exploration of useless search nodes, when considering the neighbors of a search node, don't enqueue a neighbor if its board is the same as the board of the predecessor search node.

Second Optimization To avoid recomputing the Manhattan priority of a search node from scratch each time during various priority queue operations, pre-compute its value when you construct the search node; save it in an instance variable; and return the saved value as needed. This caching technique is broadly applicable: consider using it in any situation where you are recomputing the same quantity many times and for which computing that quantity is a bottleneck operation.
This optimizations can be observed on Figure 4.3, here we need to verify that a generated board isn't equal to a predecessor board, then we must also consider the step performed (move) into the heuristic.

4.3 Warm-up

For this practice you will have to implement the heap to support Boards as insertion elements. You can use the next code to perform this:

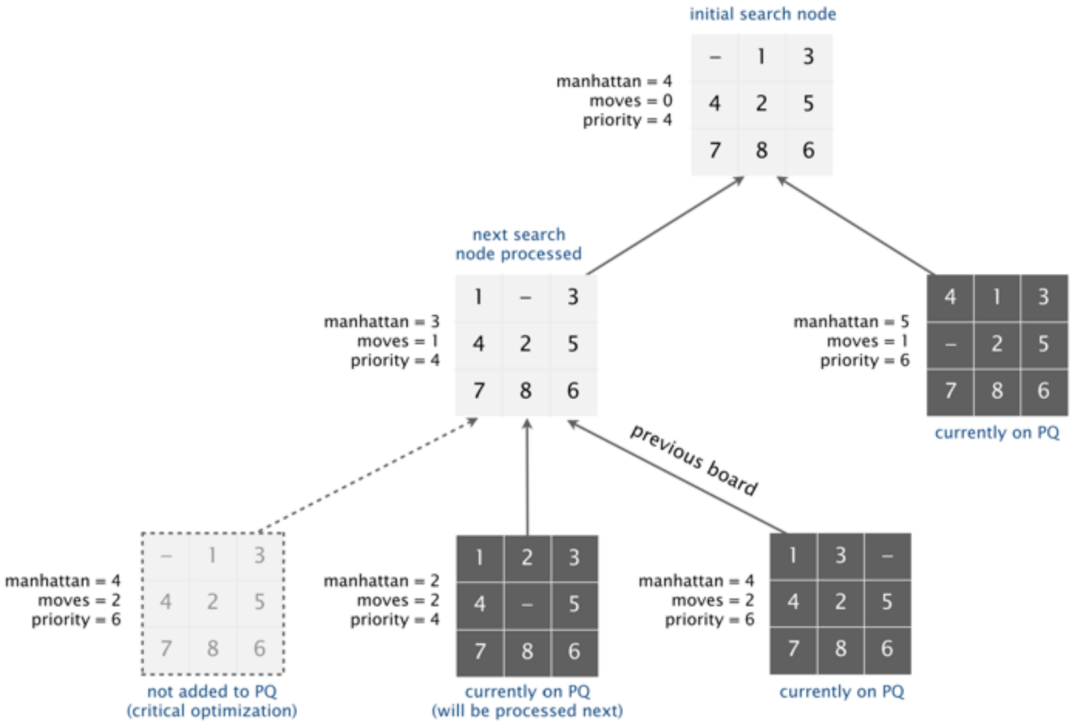

Figure 4.3: A* Algorithm using a priority queue, consider the two optimization to perform the selection of minimum board to follow. Picture from Princeton - 8-puzzle

```
import java.util.ArrayList;

public class Board {
    //construct a board from an nxn array of blocks
    public Board(int[][] toBlocks) {
    }
    //board dimension n
    public int dimension() {
    }
    //number of blocks out of place
    public int hamming() {
    }
    //sum of manhattan distances between blocks and goal
    public int manhattan() {
    }
    //is this board the goal board
    public boolean isGoal() {
    }
    //a board that is obtained by exchanging any pair of blocks
    public Board twin() {
    }
    //does this board equal y
```

```
    public boolean equals(Object y) {
    }
    //all possible board generated from parent
    public Iterable<Board> neighbors() {
    }

    public String toString() {
    }
}
```

Then we need to validate our board methods with some examples, see Figure 4.4:

```
Hamming distance:5
Manhatan distance:5
3
 4  1  3
 0  2  6
 7  5  8

Twin:
3
 4  3  1
 0  2  6
 7  5  8
```

```
Hamming distance:7
Manhatan distance:16
3
 1  6  4
 7  0  8
 2  3  5

Twin:
3
 1  4  6
 7  0  8
 2  3  5
```

```
Hamming distance:10
Manhatan distance:10
9
 1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 35 36
37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63
64  0 65 67 68 78 69 70 72
73 74 66 75 76 77 79 71 80

Twin:
9
 1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18
19 20 21 22 23 24 25 26 27
28 29 30 31 32 33 34 36 35
37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54
55 56 57 58 59 60 61 62 63
64  0 65 67 68 78 69 70 72
73 74 66 75 76 77 79 71 80
```

```
Hamming distance:13
Manhatan distance:33
4
 9  2  8 11
 0  5 13  7
15  1  4 10
 3 14  6 12

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

```
Hamming distance:14
Manhatan distance:38
4
 2  9  3  5
 8 11 12  7
15  4  0 13
 6  1 10 14

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

Figure 4.4: Some examples to demonstrate the use of the board class.

4.4 To Do:

For this section you will have to implement the A* with the priority queue you did before and the Board class given, you can use the next code as a guideline to perform this:

```java
/**
 *
 * @author christian
 */
public class Solver {
    private boolean solvable;
    private int moves;
    private Board[] steps;

    private class SearchNode implements Comparable<SearchNode> {
        private final Board board;
        private final int moves;
        private final SearchNode snode;
        private final int priority;

        public SearchNode(Board board, SearchNode predecessor) {
            this.board = board;
            if (predecessor != null) moves = predecessor.getMoves() + 1;
            else moves = 0;
            snode = predecessor;
            priority = moves + board.manhattan();
        }

        public int getMoves() {
            return moves;
        }

        public Board getBoard() {
            return board;
        }

        public SearchNode getPredecessor() {
            return snode;
        }

        public int getPriority() {
            return priority;
        }

        @Override
        public int compareTo(SearchNode sn) {
            if (priority > sn.getPriority()) {
                return 1;
            }
            if (priority < sn.getPriority()) {
                return -1;
            }
            if (priority == sn.getPriority()) {
                if ((this.priority - this.moves) > (sn.priority - sn.moves)) {
                    return 1;
                } else {
                    return -1;
                }
```

```
            }
            return 0;
        }
    }

    public Solver(Board initial) {
        solvable = false;

        MinPQ<SearchNode> minpq = new MinPQ<>();
        MinPQ<SearchNode> twinminpq = new MinPQ<>();
        if (initial == null)
            throw new java.lang.IllegalArgumentException("Null argument");
        // A* algorithm
        if (initial.isGoal()) {
            solvable = true;
            steps = new Board[1];
            steps[0] = initial;
        } else {
            minpq.insert(new SearchNode(initial, null));
            twinminpq.insert(new SearchNode(initial.twin(), null));
            SearchNode predecessor;
            SearchNode twinpredecessor;

            while (true) {
                // doing the A* stuff, you must implement this



            }
        }
    }

    public boolean isSolvable() {
        return solvable;
    }

    public int moves() {
        return moves;
    }

    public Iterable<Board> solution() {
        if (steps == null) return null;
        return Arrays.asList(steps);
    }


    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // code to test our algorithm
        // you must implement this
    }
```

```
}
```

You have to implement only the code where it indicates you.

5 Deep inside

This section is just for anyone who wants to make a deep inside into the theory and like challenges[1], you will have to prepare a presentation of just 5 minutes to explain and run the algorithm, then you will have to defend yourself another five minutes of questions. I want that all students benefit from your presentation, remember that we are here to learn. If you want to do this please email to cportugalz@unsa.edu.pe

6 Deadline

For this practice one score will be taken at class time at nearly November 20. Remember that plagiarism must be avoided and if it is detected the grade will be zero and repetition informed to superior authorities. All question and doubts must be done to the same email.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[2] S. J. Russell and P. Norvig, *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[3] A. Levitin, *Introduction To Design And Analysis Of Algorithms, 2/E*. Pearson Education India, 2008.

[1] This must not be included into the report