# Unit 3 – Lesson 8. Introduction to A\* Pathfinding

**Aim:**

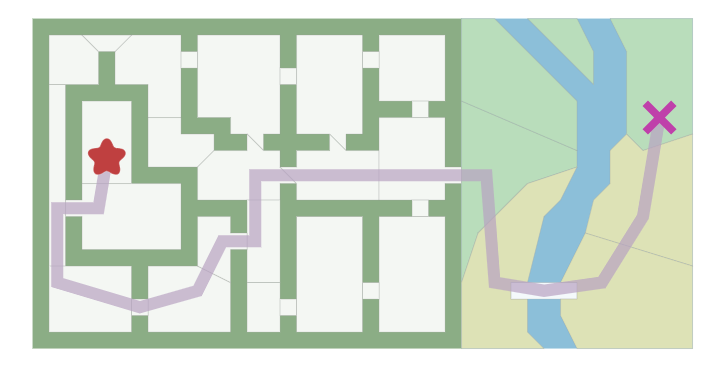
What are some of the pathfinding algorithms? What is the A\* path finding algorithm?

**Objectives:** After the lesson, students should be able to:

* Obtain understanding of the A\* path finding algorithm
* Use the A\* path finding algorithm in game design

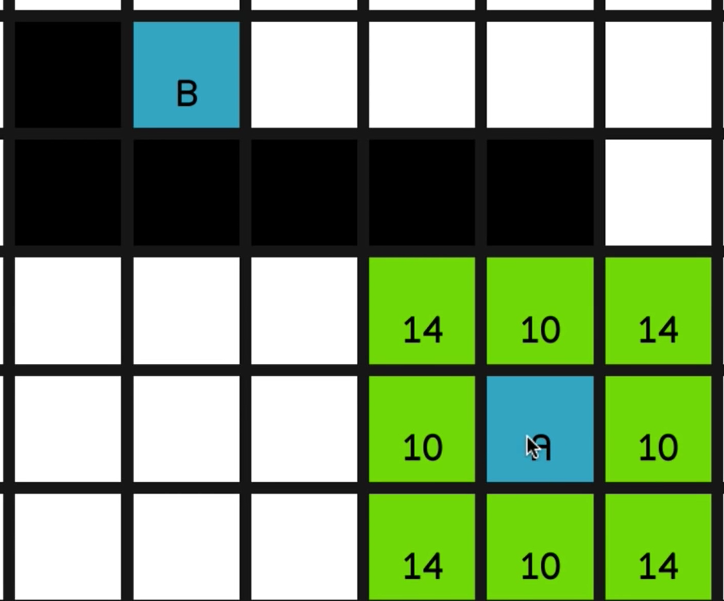
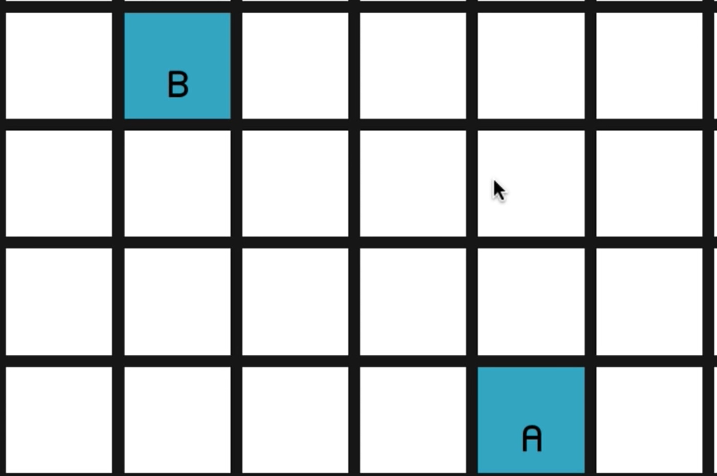
**CLASS PROCEDURE:**

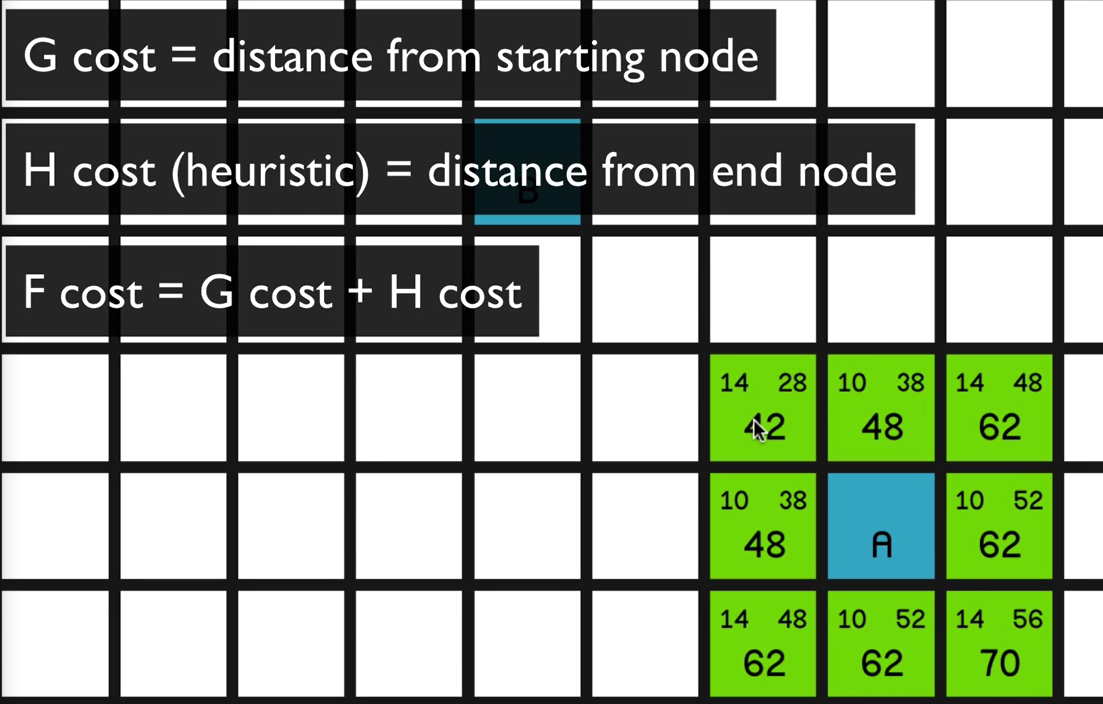
***Do Now:*** What are the different algorithms we can use to find the shortest path?



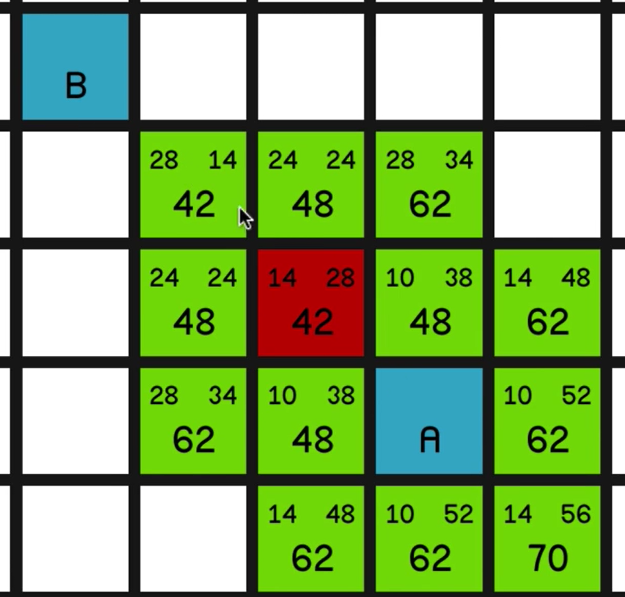
***Class Discussion / Presentation:***

* If we place the tiles in the map. What algorithms can we use to find the shortest path?
* Breadth-First Search
  + Examines all paths of cost 1, then 2, and so on until it finds the goal.
  + Guaranteed to find the shortest path
  + Very slow
* Dijkstra’s Algorithm
  + Selects closest unprocessed node from the start, and updates its value, and that of its neighbors.
  + Nodes are valued by the distance from the start.
* Best-First Search
  + Similar to Dijkstra’s Algorithm, but uses a heuristic to determine the value of node.
  + Main disadvantage is that is does not account well for terrain that is slower.
* What is A\* path finding algorithm?
* Very similar to a Breadth-First Search.
* Orders nodes to search not just by distance from start node, but also by a “heuristic” or a estimate cost remaining to the end point.
* The cost of a node is defined as  
    
  f(n) = g(n) + h(n)   
    
  where  
    
  g(n) is the distance traveled from the start  
    
  h(n) is the heuristic, the estimated distance to the end node
* A\* Examples:





Therefore, we can use A\* to find the shortest path from A to B:



1. Considerations for A\* pathfinding:
   1. For a grid of squares, the heuristic is usually the number of squares in between the current node and the end node.
   2. You must define how diagonal squares work with map grids. If movement is continuous, consider multiplying the cost of diagonal movement by √2
   3. Defining an upper bound cutoff can prevent the search from taking too long.

***Pair – sharing Activity / HW (20 points, due Friday)***

Write a program to solve the 8-puzzle problem (and its natural generalizations) using the A\* search algorithm.

**The problem.** The [8-puzzle problem](http://en.wikipedia.org/wiki/Fifteen_puzzle) is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square tiles labeled 1 through 8 and a blank square. Your goal is to rearrange the tiles so that they are in order, using as few moves as possible. You are permitted to slide tiles horizontally or vertically into the blank square. The following shows a sequence of legal moves from an initial board (left) to the goal board (right).

1 3 1 3 1 2 3 1 2 3 1 2 3

4 2 5 => 4 2 5 => 4 5 => 4 5 => 4 5 6

7 8 6 7 8 6 7 8 6 7 8 6 7 8

initial 1 left 2 up 5 left goal

**Best-first search.** Now, we describe a solution to the problem that illustrates a general artificial intelligence methodology known as the [A\* search algorithm](http://en.wikipedia.org/wiki/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 previous search node. First, insert the initial search node (the initial board, 0 moves, and a null previous 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 tiles 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 tiles 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 tiles to their goal positions, plus the number of moves made so far to get to the search node.

For example, the Hamming and Manhattan priorities of the initial search node below are 5 and 10, respectively.

8 1 3 1 2 3 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8

4 2 4 5 6 ---------------------- ----------------------

7 6 5 7 8 1 1 0 0 1 1 0 1 1 2 0 0 2 2 0 3

initial goal Hamming = 5 + 0 Manhattan = 10 + 0

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 tile that is out of place must move at least once to reach its goal position. For Manhattan priority, this is true because each tile 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.)

**A critical 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 previous search node.

8 1 3 8 1 3 8 1 8 1 3 8 1 3

4 2 4 2 4 2 3 4 2 4 2 5

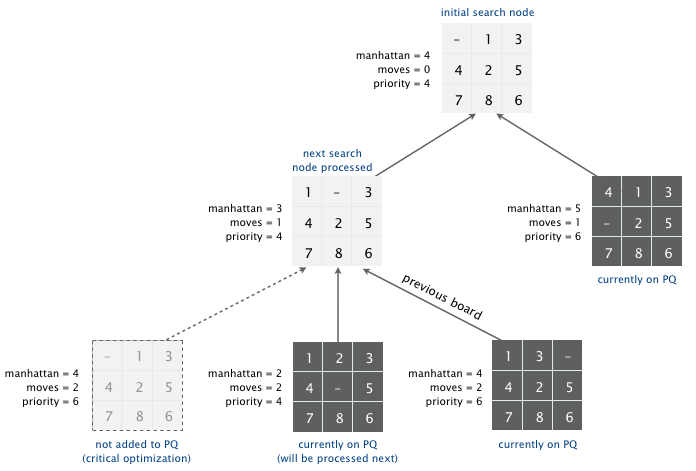
7 6 5 7 6 5 7 6 5 7 6 5 7 6

previous search node neighbor neighbor neighbor

(disallow)

**A second optimization.** To avoid recomputing the Manhattan distance of a board (or, alternatively, the Manhattan priority of a solver node) from scratch each time during various priority queue operations, compute it at most once per object; save its value 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.

**Game tree.** One way to view the computation is as a game tree, where each search node is a node in the game tree and the children of a node correspond to its neighboring search nodes. The root of the game tree is the initial search node; the internal nodes have already been processed; the leaf nodes are maintained in a priority queue; at each step, the A\* algorithm removes the node with the smallest priority from the priority queue and processes it (by adding its children to both the game tree and the priority queue).



**Detecting unsolvable puzzles.** Not all initial boards can lead to the goal board by a sequence of legal moves, including the two below:

1 2 3 1 2 3 4

4 5 6 5 6 7 8

8 7 9 10 11 12

13 15 14

unsolvable

unsolvable

To detect such situations, use the fact that boards are divided into two equivalence classes with respect to reachability: (i) those that lead to the goal board and (ii) those that cannot lead to the goal board. Moreover, we can identify in which equivalence class a board belongs without attempting to solve it.

* Odd board size. Given a board, an inversion is any pair of tiles i and j where i < j but i appears after j when considering the board in row-major order (row 0, followed by row 1, and so forth).
* 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
* 4 5 6 => 4 5 6 => 4 6 => 4 6 => 4 6 7
* 8 7 8 7 8 5 7 8 5 7 8 5
* 1 2 3 4 5 6 8 7 1 2 3 4 5 6 8 7 1 2 3 4 6 8 5 7 1 2 3 4 6 8 5 7 1 2 3 4 6 7 8 5
* inversions = 1 inversions = 1 inversions = 3 inversions = 3 inversions = 3
* (8-7) (8-7) (6-5 8-5 8-7) (6-5 8-5 8-7) (6-5 7-5 8-5)

If the board size N is an odd integer, then each legal move changes the number of inversions by an even number. Thus, if a board has an odd number of inversions, then it cannot lead to the goal board by a sequence of legal moves because the goal board has an even number of inversions (zero).

The converse is also true: if a board has an even number of inversions, then it can lead to the goal board by a sequence of legal moves.

1 3 1 3 1 2 3 1 2 3 1 2 3

4 2 5 => 4 2 5 => 4 5 => 4 5 => 4 5 6

7 8 6 7 8 6 7 8 6 7 8 6 7 8

1 3 4 2 5 7 8 6 1 3 4 2 5 7 8 6 1 2 3 4 5 7 8 6 1 2 3 4 5 7 8 6 1 2 3 4 5 6 7 8

inversions = 4 inversions = 4 inversions = 2 inversions = 2 inversions = 0

(3-2 4-2 7-6 8-6) (3-2 4-2 7-6 8-6) (7-6 8-6) (7-6 8-6)

* Even board size. If the board size N is an even integer, then the parity of the number of inversions is not invariant. However, the parity of the number of inversions plus the row of the blank square is invariant: each legal move changes this sum by an even number. If this sum is even, then it cannot lead to the goal board by a sequence of legal moves; if this sum is odd, then it can lead to the goal board by a sequence of legal moves.
* 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
* 5 6 8 => 5 6 8 => 5 6 7 8 => 5 6 7 8 => 5 6 7 8
* 9 10 7 11 9 10 7 11 9 10 11 9 10 11 9 10 11 12
* 13 14 15 12 13 14 15 12 13 14 15 12 13 14 15 12 13 14 15
* blank row = 1 blank row = 1 blank row = 2 blank row = 2 blank row = 3
* inversions = 6 inversions = 6 inversions = 3 inversions = 3 inversions = 0
* -------------- -------------- -------------- -------------- --------------
* sum = 7 sum = 7 sum = 5 sum = 5 sum = 3

.

**Board and Solver data types.** Organize your program by creating an immutable data type Board with the following API:

**public class Board {**

**public Board(int[][] tiles)** // construct a board from an N-by-N array of tiles

// (where tiles[i][j] = tile at row i, column j)

**public int tileAt(int i, int j)** // return tile at row i, column j (or 0 if blank)

**public int size()** // board size N

**public int hamming()** // number of tiles out of place

**public int manhattan()** // sum of Manhattan distances between tiles and goal

**public boolean isGoal()** // is this board the goal board?

**public boolean isSolvable()** // is this board solvable?

**public boolean equals(Object y)** // does this board equal y?

**public Iterable<Board> neighbors()** // all neighboring boards

**public String toString()** // string representation of this board (in the output format specified below)

**public static void main(String[] args)** // unit testing (required)

}

Corner cases.  You may assume that the constructor receives an N-by-N array containing the N2 integers between 0 and N2 − 1, where 0 represents the blank square. The tileAt() method should throw a java.lang.IndexOutOfBoundsException unless both i or j are between 0 and N − 1.

Performance requirements.  Your implementation should support all Board methods in time proportional to N² (or better) in the worst case, with the exception that isSolvable() may take up to N4 in the worst case.

Also, create an immutable data type Solver with the following API:

**public class Solver {**

**public Solver(Board initial)** // find a solution to the initial board (using the A\* algorithm)

**public int moves()** // min number of moves to solve initial board

**public Iterable<Board> solution()** // sequence of boards in a shortest solution

**public static void main(String[] args)** // solve a slider puzzle (given below)

}

To implement the A\* algorithm, you must use the *MinPQ* data type from *algs4.jar* for the priority queue.

Corner cases.  The constructor should throw a java.lang.IllegalArgumentException if the initial board is not solvable and a java.lang.NullPointerException if the initial board is null.

**Solver test client.** You can use the following test client to read a puzzle from a file (specified as a command-line argument) and print the solution to standard output.

public static void main(String[] args) {

// create initial board from file

In in = new In(args[0]);

int N = in.readInt();

int[][] tiles = new int[N][N];

for (int i = 0; i < N; i++)

for (int j = 0; j < N; j++)

tiles[i][j] = in.readInt();

Board initial = new Board(tiles);

// check if puzzle is solvable; if so, solve it and output solution

if (initial.isSolvable()) {

Solver solver = new Solver(initial);

StdOut.println("Minimum number of moves = " + solver.moves());

for (Board board : solver.solution())

StdOut.println(board);

}

// if not, report unsolvable

else {

StdOut.println("Unsolvable puzzle");

}

}

**Input and output formats.** The input and output format for a board is the board size N followed by the N-by-N initial board, using 0 to represent the blank square.

**% more puzzle04.txt**

3

0 1 3

4 2 5

7 8 6

% **java Solver puzzle04.txt**

Minimum number of moves = 4

3

0 1 3

4 2 5

7 8 6

3

1 0 3

4 2 5

7 8 6

3

1 2 3

4 0 5

7 8 6

3

1 2 3

4 5 0

7 8 6

3

1 2 3

4 5 6

7 8 0

% **more puzzle-unsolvable3x3.txt**

3

1 2 3

4 5 6

8 7 0

% **java Solver puzzle3x3-unsolvable.txt**

Unsolvable puzzle

Your program should work correctly for arbitrary N-by-N boards (for any 1 ≤ N ≤ 32768), even if it is too slow to solve some of them in a reasonable amount of time.

**Challenge for the bored.** Implement a better solution which is capable of solving puzzles that the required solution is incapable of solving.

**Deliverables.** Submit the files Board.java and Solver.java (with the Manhattan priority). We will supply algs4.jar. Your may not call any library functions other than those in java.lang, java.util, and algs4.jar. You must use [MinPQ](http://algs4.cs.princeton.edu/code/javadoc/edu/princeton/cs/algs4/MinPQ.html) for the priority queue. Finally, submit a [readme.txt](ftp://ftp.cs.princeton.edu/pub/cs226/8puzzle/readme.txt) file and answer the questions.