OBJECTIVES:

- Find shortest path in a graph for given a source and a destination using A* Search algorithm.
- Implementing A* search algorithm using your preferred programming language.

BACKGROUND STUDY

- Should have prior knowledge on any programming language to implement the algorithm.
- Need knowledge on queue and priority queue.
- Input a graph.
- A* algorithm that would be covered in theory class and that knowledge will help you here to get the implementation idea.

RECOMMENDED READING

- http://en.wikipedia.org/wiki/A*_search_algorithm
- BOOK

A* SEARCH

A* is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between points, called nodes. Noted for its performance and accuracy, it enjoys widespread use.

A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). As A* traverses the graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way.

It uses a knowledge-plus-heuristic cost function of node x (usually denoted f(x)) to determine the order in which the search visits nodes in the tree. The cost function is a sum of two functions:

- the past path-cost function, which is the known distance from the starting node to the current node x (usually denoted g(x))
- a future path-cost function, which is an admissible "heuristic estimate" of the distance from x to the goal (usually denoted h(x)).

The h(x) part of the f(x) function must be an admissible heuristic; that is, it must not overestimate the distance to the goal.

Starting with the initial node, it maintains a priority queue of nodes to be traversed, known as the open set. The lower f(x) for a given node x, the higher its priority. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). (Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path

to a goal.) The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

Pseudocode that can be used to implement A* algorithm is as following:

```
function A_Star(start,goal)
   closedset := the empty set    // The set of nodes already evaluated.
   openset := {start}    // The set of tentative nodes to be evaluated, initially containing the start
node
   came_from := the empty map    // The map of navigated nodes.

   g_score[start] := 0    // Cost from start along best known path.
   // Estimated total cost from start to goal through y.
   f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

   while openset is not empty
      current := the node in openset having the lowest f_score[] value
      if current = goal
         return reconstruct_path(came_from, goal)

      remove current from openset
      add current to closedset
      for each neighbor in neighbor_nodes(current)
         tentative_g_score := g_score[current] + dist_between(current,neighbor)
         tentative_f_score := tentative_g_score + heuristic_cost_estimate(neighbor, goal)
         if neighbor in closedset and tentative_f_score >= f_score[neighbor]
               continue

         if neighbor not in openset or tentative_f_score < f_score[neighbor]
            came_from[neighbor] := current
            g_score[neighbor] := tentative_g_score
            f_score[neighbor] := tentative_f_score
            if neighbor not in openset
               add neighbor to openset

   return failure
```

openset – is a priority queue where we store nodes in increasing order of their path cost.

closedset – is a set of nodes where we store all evaluated nodes.

came_from – is a map where we store which node is visited from which node that parent-child relations.

g_score – is a list to store g-values of all nodes.

f_score – is a list to store f-values of all nodes.

heuristic_cost_estimate() – is used to find the heuristic value for a specific node. We assume that heuristic value will be given as input for each node and we select heuristic value for a node from those values.

Following pseudocode can be used to find the actual sequence of steps in A* search:

```
function reconstruct_path(came_from, current_node)
   if current_node in came_from
      p := reconstruct_path(came_from, came_from[current_node])
      return (p + current_node)
   else
      return current_node
```

LABROTORY EXERCISES

1. Implement A* algorithm using the pseudocode described above in your preferred language.
   Input: a graph G, start node S and a goal node D.
   Output: Goal found or not. If goal found then print the path.