Artificial Intelligence

Implementation and Comparison of the A* and Jump Point Search (JPS) algorithms

Table of Contents

Abstract

The comparison of the *A** and *Jump Point Search (JPS)* algorithms in a grid where an ant needs to move from its nest to the food supply, showed that these two algorithms are complete and optimal. However, in a 16x16 grid, an ant using the *JSP* algorithm can reach the food supply using 25% of the A* tree search expansion nodes, with approximately 5% of the A* neighbours to consider. This is a considerable memory foot print gain for the *JSP*. The time to reach the destination also seems to play in favour of the *JSP* algorithm.

Introduction

The purpose of this project is to implement and compare two informed search algorithms in Java within the AntWorld project provided by [0]. The main body of this report is organised in four parts. The first part (the introduction) provides a high level functional description of each selected search algorithm; namely the *A** and Jump Point Search *(JPS)*. It then details the inner working of each algorithm using pseudo code (the details of the Java implementation is referenced in the Appendix). The following paragraph concentrates on the *A** algorithm inherent limitations and establishes why the *JPS* algorithm could provide a better alternative. At this point, a list of performance measurements is presented to evaluate the performance of each algorithm in different environments. The second part outlines the assumptions, methodology and scenarios employed to investigate the algorithms behaviour. The third part provides the performance results of each algorithm in each scenario. The final part evaluates the results and draws a conclusion for this experiment.

A* Search Algorithm Description & Pseudo Code

High Level Description

The *A** search belongs to the family of informed search strategies, and more precisely to the best-first search algorithm types [4]. The search moves one step at a time, in all the quadrant directions, including the diagonal. From a given location (a.k.a. node), it expands all neighbour locations in its vicinity. In other words, the algorithm expands each reachable node in the tree search. The *A** search algorithm formally aims at minimising the estimated cost to reach destination, namely f(n), when the destination is reachable. Therefore, it cannot miss an optimal solution. For this, it combines

- g(n): the cost to reach each node, and
- h(n) the cost to get from the node to the goal

Formula1 below formalises the approach.

$$f(n) = g(n) + h(n)$$

where:

- g(n) corresponds to the path cost from the start node (n) to the potential next node
- h(n) corresponds to the path cost from the start node (n) next node to the goal node

Formula 1 – The estimated cost to reach destination

When the goal is reachable, the *A** search is both complete and optimal. In the proposed implementation, the algorithm is optimal because h(n) is an admissible heuristic [4]. It never overestimates the cost of reaching the goal. Furthermore, this implementation also ensures h(n) is consistent. In other words, the cost of moving from the current location to the goal is inferior or equal to:

- i) the cost of moving to the next location, following an action, and
  ii) the cost of moving from the next location to the goal.

This concept is detailed in Formula 2 below.

$$f(n) <= c(n,a,n1) + h(n1)$$

where:

- c(n,a,n1) corresponds to the cost of moving from the current location (n) to the next location (n1), given the action (a)
- h(n1) corresponds to the cost of moving from the next location (n1) to the goal.

Formula 2 – The estimated cost to reach destination

In this report, the heuristic h(n) implements a Manhattan distance . This is the distance between a given location and the goal location as shown in Formula 3. The picture displayed in Figure 1 was borrowed from [3].

md = abs(Xcurrent_location - Xgoal_location) + abs(Ycurrent_location - Ygoal_location)
where:
- Xcurrent_location: the x value of the current location coordinate
- Xgoal_location: the x value of the goal location coordinate
- Ycurrent_location: the y value of the current location coordinate
- Ygoal_location: the y value of the goal location coordinate

Formula 3 – The Manhattan distance formula

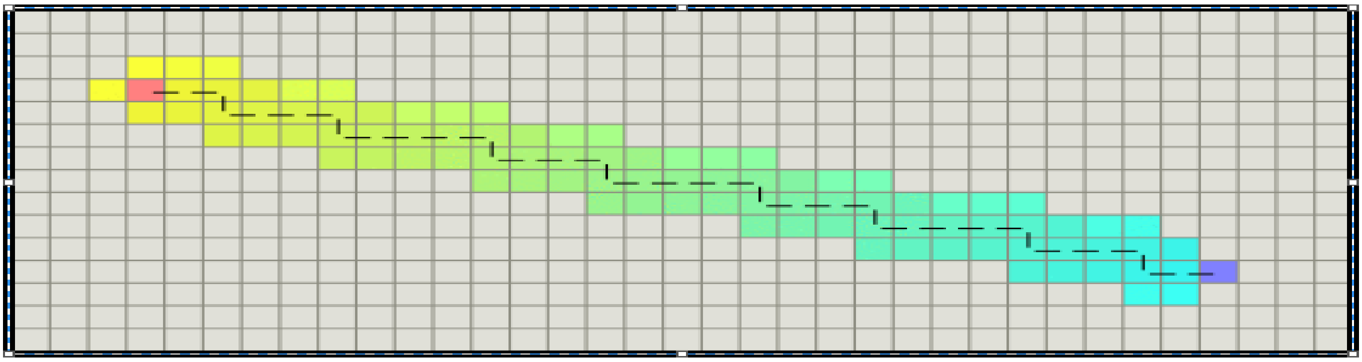


Figure 1 – Graph representation of a Manhattan distance calculation

Visually, the *A**can be presented as on Figure 2, borrowed from [5]. It displays an example of an *A** algorithm where nodes are cities connected by roads, and h(x) is the straight-line distance to target point. As shown in Figure [5], the distance between the green and the blue cities is the shortest when the search algorithm navigates via the city '*e*' as it accumulates a total cost of 7 (2+3+2) compared to a total cost of 10.5 (1.5+2+3+4), when it goes through city 'c'.

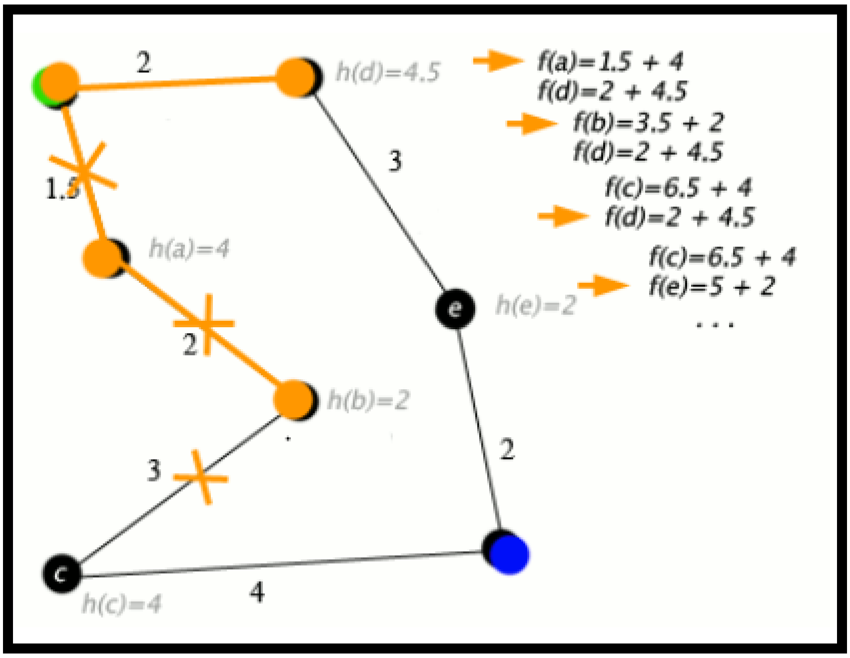


Figure 2 - An example of an *A** algorithm.

Pseudo Code

The following section describes the pseudo code for the *A** algorithm. Each section contains the core of the algorithm logic. Each pseudo function is tagged with the following label *implemented in [Code Block X].* The logic implementation has been implemented in Java 1.8 and is available in the Appendix under the same heading. The pseudo code only contains the most important variable and methods that describe the algorithm inner workings. The java code implementation could contain extra utility functions to support the code running in the *AntWorld* framework. When they were not considered vital for the understanding for the search algorithm, they have been removed from the pseudo code to avoid unnecessary cluttering.

As a general note; the *locationWrapper is called by GET_SHORTEST_PATH(startLocation, goal)* function described *below. The locationWrapper* object inherits from the *location* object, provided in [0]. The description of the *locationWrapper* logic is detailed after the A* pseudo code.

```
Function GET_SHORTEST_PATH(startLocation, goal ) returns shortestPath /*i.e. When the goal is reachable: it returns the
solution which is the list of positions that define the shortest path to get the starting  point to the destination point.
When the goal is not reachable: it returns the list of all locations that were discovered.   */

/*The set of currently discovered nodes still to be evaluated.*/
openSet  ←empty priority queue. The priority queue order the neighbours based on their fValue score.
/*The set of nodes already evaluated.*/
closedSet  ←empty set
/*A list only used in case of failure of the search to discover the goal. It contains all discovered node (evaluated or not)*/
retainedLocations ←[ ]
/*Count the number of times, the tree has been expanded*/
expansionCount  ← 0
/*Establishes whether the goal has been reached or not*/
isGoalReached ← false
/*The map key corresponds to a given location and
the value relates to the most efficient location the key (location) can be reached from*/
locationOrigins <- empty map

/*Start with the known node*/
openSet .ADD(startLocation)

current <- null

while loop( openSet is not empty)
     expansionCount = increment the expansionCount by one
     current = get the location in the openSet having the lowest fScore value
    /*add the current location to the retainedLocations array*/
    retainedLocations .ADD(current)

    if (current = goal) then
        isGoalReached = true
       /*returns the shortest path to reach the destination*/
      return GET_PATH()
   end if

    /*add the current location to the closedSet  */
    closedSet.ADD(current)

    /* Get the list of neighbours one step away from the current location, that are an obstacle*/
    neighbours ←GET_ALL_NEIGHBOURGS(current)
```

```
  for each neighbour in neighbours
     /*the new gscore is the old one incremented by 1 as it corresponds to one step move*/
     gScore = current.GET_G_VALUE() +  1
    if the closedSet contains the neighbour and the gScore >=  neighbour.GET_G_VALUE ()
   then this is not the best path, so continue

   /*This is the best path so far, so store it*/
   if the openSet does not contain the neighbour or the gScore <  neighbour. GET_G_VALUE () then
      /*Update the locationOrigins key with the latest neighbour vs current map*/
      locationOrigins.REMOVE(neighbour)
     locationOrigins.REMOVE(neighbour, current)

     /*Set the neighbour gScore and fSCore*/
     neighbour.Set_G_VALUE(gScore)
     neighbour.Set_F_VALUE(neighbour.GET_G_VALUE () + GET_HEURISTIC(neighbour))
    /*Update the openSet with the latest neighbour information*/
     openSet.REMOVE(neighbour);
     openSet.ADD(neighbour);
  end if
 end for
end loop

  /*This is the case where the goal cannot be reached*/
  isGoalReached ←false;
 return retainedLocations;
```

/*Implemented in [Code Block 1]*/

Function *GET_PATH()* ***returns*** *the list of location that represents the shortest path from the start to the destination node.*
*/*It uses the locationOrigins map to retrace the path taken between two point*/*

/*Implemented in [Code Block 3]*/

Function *GET_ALL_NEIGHBOURGS(currentLocation)* ***returns*** *list of allowed location to move to in the quadrant, i.e. any location that is not an obstacle .*

/*Implemented in [Code Block 2]*/

Function *SET_G_VALUE () /*set the gValue to a locationWrapper object */*

/*Implemented in [Code Block 13]*/

Function *GET_G_VALUE ()* ***returns*** *the gValue from a locationWrapper object*

/*Implemented in [Code Block 13]*/

Function *GET_HEURISTIC(currentLocation, destinationLocation)* ***returns*** *the Manhattan distance /*the Manhattan distance as explained in [3]. The sum of the absolute value of difference of the number of rows and the absolute value of difference of the number of columns between a given location and the destination location */*

/*Implemented in [Code Block 9]*/

The *locationWapper* object implements the *getter* and *setter* functions for the *gValue* and *fValue*, but more importantly, it embeds the logic called by the priority queue to rank location objects in the order based on their *fValue* values.

function *compareTo (LocationWrapper other)* ***returns*** *return -1 if the current location has a smaller fValue than the other location, else 1* */*Implemented in [Code Block 13]*/*
function *equals (LocationWrapper other)* ***returns*** *return true when two locations have the same position* */*Implemented in [Code Block 13]*/*
function *getGValue ()* ***returns*** *the gValue to the location object* */*Implemented in [Code Block 13]*/*
function *setGValue (double value)* */*set the gValue to the location object*/* */*Implemented in [Code Block 13]*/*
function *getFValue ()* ***returns*** *get the fValue to the location object* */*Implemented in [Code Block 13]*/*
function *setFValue (double value)* */*set the fValue to the location object*/* */*Implemented in [Code Block 13]*/*
function *hashCode* ***returns*** *a unique identifier for the object.* */*Implemented in [Code Block 13]*/*

Jump Point Search Algorithm Description & Pseudo Code

High Level Description

The *Jump Point Search* (JPS) belongs to the same family as the *A** algorithm. It is both complete and optimal and it uses a heuristic to move from the source to the destination. In order to perform a like for like comparison, the heuristic chosen for the *JPS* implementation is the same as the one chosen for the *A** algorithm. The main improvement of the *JPS* is that it reduces symmetries in the search procedure via graph pruning. Consequently, the JPS algorithm allows for long 'jumps' along the horizontal, vertical and diagonal cells rather than performing a step by step move. Figure 3, borrowed from [2], illustrates the two main cases. Case a) represents an optional jump from node x to node y (plain black arrow). It involves applying a recursive pruning rule. In this case, node z cannot be reached optimally unless it visits both the node x and y. Consequently, the other suboptimal paths are not evaluated. This reduces the number of tree expansions and therefore reduces the memory load. Case b) corresponds to an optimal diagonal jump from x to y. There are two steps in this case; first there is a horizontal and vertical recursion that attempt to discover a jump point (y). If not jump point is found, then the diagonal step is chosen.

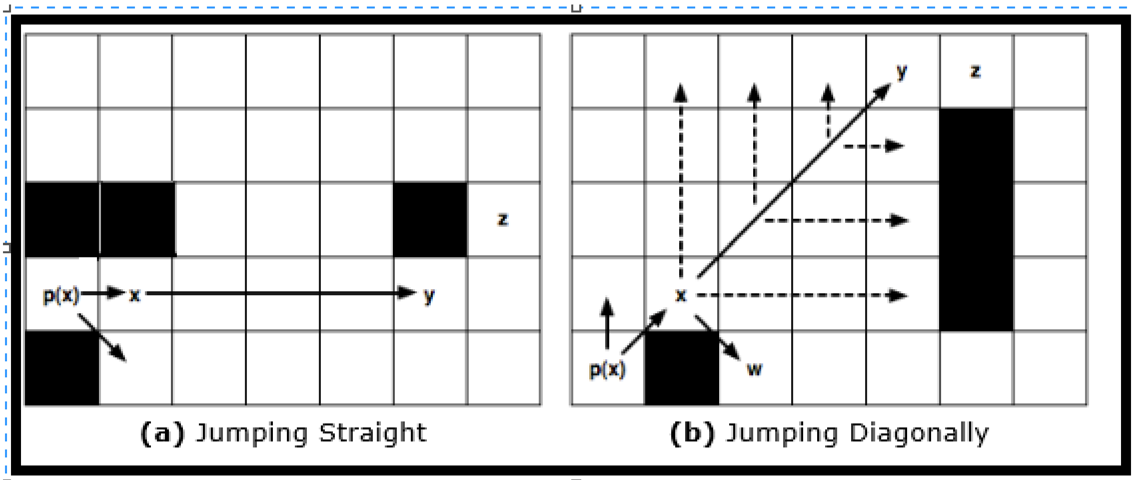


Figure 3 – Jumping in action. Node x is currently being expanded; p(x) is its parent.

The pruning algorithm contains two rules; one relating to horizontal and vertical moves , and the other concerns the diagonal steps. The objective of pruning is to eliminate nodes that are involved in a symmetric path from the parent node p(x) of the current node x. This is illustrated in Figure 4 below, borrowed from [2]. The first grid shows pruning of unnecessary nodes when the parent of x is node 4, and the direction of travel is horizontal. The plain arrow indicates the direction of travel. The second grid demonstrates pruning in case of a diagonal move. The parent if x is node 6. All nodes in grey are pruned nodes. They can be reached optimally without going through node x. The nodes that remain after pruning are named the *natural neighbours (white coloured nodes in figure 4).*

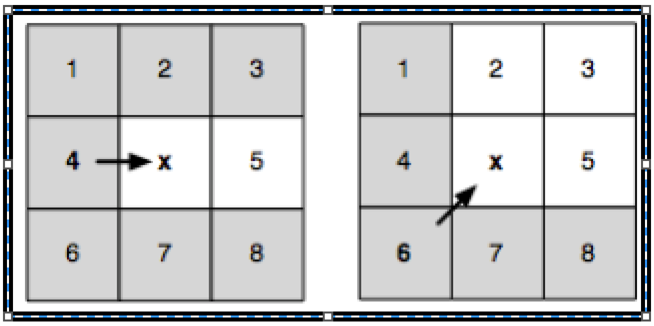


Figure 4 – Pruning nodes in action (no obstacle)

There is an additional complexity, relating to pruning in the presence of obstacles. An example of this is shown in Figure 5, borrowed from [6]. The nodes 3 and 1, cannot be pruned by the above pruning rule as they are blocked by obstacles. Therefore, it is not possible to deduce any other alternative optimal path from x. Consequently, these nodes are added to the list of nodes to be considered during the tree search expansion. They are called *forced neighbours*. The pruning algorithm then reclusively prunes each node in the vicinity of forced and natural neighbours.

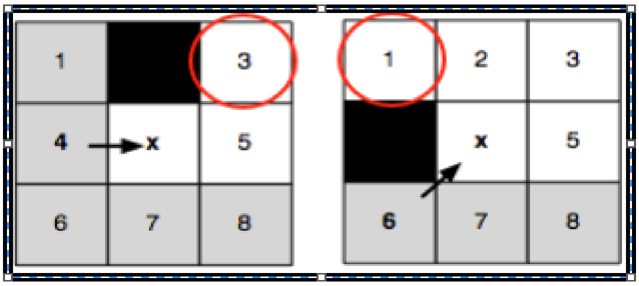


Figure 5 – Pruning nodes in action in presence of obstacle

Pseudo Code

The JPS core implementation is the same as the *A** implementation discusssed above. The only difference lies in the selection of the neighbours. The GET_ALL_NEIGHBOURGS(currentLocation) delegates into INDENTIFY_SUCCESSORS(current, startLocation,goal) function. Instead of returning the height cardinal neighbours, it returns the jump points successors. Consequently, this section will only detail the functions relating to the identifying successors, all the rest being equal.

```
Function INDENTIFY_SUCCESSORS(locationOrigins,current, startLocation,goal)  returns  list of jump points successors

successors <- []
neighbours <- []
parent <- locationOrigins.GET(current)
dRow <- 0 /* vertical direction of travel */
dCol <- 0 /* horizontal direction of travel */

if parent is null then
   /* Case 1: At the start location, add all non-boundary neighbours  to the neighbours list */
   neighbours <- ADD_ALL_NON_BOUNDARY_NEIGHBOURS (current, neighbours)
end if
else
    /*Case 2: any location that is not the start location*/
   /*Find the direction of travel along the path. The abs is used to keep direction of move
   dRow <-(current.row - parent.row) / abs( current.row - parent.row)
   dCol <-(current.col - parent.col) / abs( current.col -parent.col)
  /*Get the list of natural and forced neighbours */
  prunedNeighbours <- GET_PRUNED_NEIGHBOURS(current,dRow, dCol)
  /*Add them to the neighbours list */
  neighbours.ADD(neighbours)
end if

 for each neighbour in neighbours
   /* Direction of travel between the neighbour and the current location */
   dRow <-  neighbour.row – current.row
   dCol <-  neighbour.col – current.col
  jumpPoint <- JUMP(dRow, dCol, current, startLocation, goal)
  jumpPoint is not null
 successors.ADD(jumpPoint)
end for
return successors
```

```
/*Implemented in [Code Block 4]*/
Function ADD_ALL_NON_BOUNDARY_NEIGHBOURS (current, neighbours)  returns  list of non-boundary neighbours around the start node

/*Implemented in [Code Block 5]*/
Function GET_PRUNED_NEIGHBOURS(current,dRow, dCol) returns list of natural and forced neighbours
/*initialise the list of natural and forced neighbours*/
neighbours <- []
/*Identify a forced neighbour*/
forcedNeighbours <- null

/*Default the next, previous row and col coordinate from the current location*/
nextRow <- dRow +1
precRow <- dRow -1
nextCol <-dCol +1
prevCol <-dCol -1


/*Diagonal Travel*/
if (dRow !=0 and dCol !=0) then
    /*this set the next row and col depending on the direction of travel*/
    nextRow <- dRow + dRow
    precRow <- dRow  - dRow
    prevCol <-dCol – dCol
    nextCol <-dCol +  dCol

   /* Attempt to add natural neighbours for the following locations*/
  ADD_NATURAL_NEIGHBOURG(neighbours, dRow, nextCol)
  ADD_NATURAL_NEIGHBOURG(neighbours, nextRow, dCol)
  ADD_NATURAL_NEIGHBOURG(neighbours, nextRow, nextCol)

   /*Attempt to add forced neighbours on the east/west diagonal of the current location. Depending on the travel direction */
  If  there is an obstacle on the location with the coordinate (currRow,  prevCol ) and
     the location with the coordinate (nextRow,  prevCol ) is not an obstacle and
     the location with the coordinate (nextRow,  prevCol ) is not a boundary then
       forcedNeighbours.ADD(node with the coordinate (nextRow,  prevCol ))
 endif

  If  there is an obstacle on the location with the coordinate (currRow,  nextCol ) and
     the location with the coordinate (prevRow,  nextCol ) is not an obstacle and
     the location with the coordinate (prevRow,  nextCol) is not a boundary then
       forcedNeighbours.ADD(node with the coordinate (prevRow,  nextCol))
 endif

/*Vertical Travel Case*/
else if (dRow !=0)
   /* Attempt to add natural neighbours for the following locations*/
   ADD_NATURAL_NEIGHBOURG(neighbours, nextRow, currCol)

    /********* Forced Neighbour: Obstacle/Boundary East/West ************/
   /*Attempt to add forced neighbours on the south east diagonal of the current location*/
   If  there is an obstacle on the location with the coordinate (currRow,  nextCol ) and
     the node with the coordinate (nextRow,  nextCol) is not an obstacle and
     the node with the coordinate (nextRow,  nextCol) is not a boundary then
        forcedNeighbours.ADD(node with the coordinate (nextRow,  nextCol))
  endif
```

```
/*Attempt to add forced neighbours on the north east diagonal of the current location*/
If  there is an obstacle on the location with the coordinate (currRow,  nextCol ) and
   the node with the coordinate (prevRow,  nextCol) is not an obstacle and
   the node with the coordinate (prevRow,  nextCol) is not a boundary then
      forcedNeighbours.ADD(node with the coordinate (prevRow,  nextCol))
endif

/*Attempt to add forced neighbours on the south east diagonal of the current location*/
If  there is an obstacle on the location with the coordinate (currRow,  prevCol ) and
   the node with the coordinate (nextRow,  prevCol) is not an obstacle and
   the node with the coordinate (nextRow,  prevCol) is not a boundary then
      forcedNeighbours.ADD(node with the coordinate (nextRow,  prevCol))
endif

/*Attempt to add forced neighbours on the north west diagonal of the current location*/
If  there is an obstacle on the location with the coordinate (currRow,  prevCol ) and
   the node with the coordinate (prevRow,  prevCol) is not an obstacle and
   the node with the coordinate (prevRow,  prevCol) is not a boundary then
      forcedNeighbours.ADD(node with the coordinate (prevRow,  prevCol))
endif

/********* Forced Neighbour: Obstacle/Boundary North/South  (2 steps ahead) ************/
/*Attempt to add forced neighbours on the south east diagonal of the current location*/
If  there is an obstacle on the location with the coordinate (nextRow+dRow,  currCol ) and
   the node with the coordinate (nextRow,  nextCol) is not an obstacle and
   the node with the coordinate (nextRow,  nextCol) is not a boundary then
      forcedNeighbours.ADD(node with the coordinate (nextRow,  nextCol))
endif

/*Attempt to add forced neighbours on the south west diagonal of the current location*/
If  there is an obstacle on the location with the coordinate (nextRow+dRow,  currCol ) and
   the node with the coordinate (nextRow,  prevCol) is not an obstacle and
   the node with the coordinate (nextRow,  prevCol) is not a boundary then
      forcedNeighbours.ADD(node with the coordinate (nextRow,  prevCol))
endif

/*Attempt to add forced neighbours on the north east diagonal of the current location*/
If  there is an obstacle on the location with the coordinate (prevRow+dRow,  currCol ) and
   the node with the coordinate (prevRow,  nextCol) is not an obstacle and
   the node with the coordinate (prevRow,  nextCol) is not a boundary then
      forcedNeighbours.ADD(node with the coordinate (prevRow, nextCol))
endif

/*Attempt to add forced neighbours on the north west diagonal of the current location*/
If  there is an obstacle on the location with the coordinate (prevRow+dRow,  currCol ) and
   the node with the coordinate (prevRow,  prevCol) is not an obstacle and
   the node with the coordinate (prevRow,  prevCol) is not a boundary then
      forcedNeighbours.ADD(node with the coordinate (prevRow, prevCol))
endif

endif
```

```
/*Horizontal Travel Case*/
else if (dCol !=0)
   /* Attempt to add natural neighbours for the following locations*/
   ADD_NATURAL_NEIGHBOURG(neighbours, currRow, nextCol)

   /********* Forced Neighbour: Obstacle/Boundary North/South  ************/
   /*Attempt to add forced neighbours on the west diagonal of the current location*/
    If  there is an obstacle on the location with the coordinate (prevRow,  currCol ) and
       the node with the coordinate (prevRow,  nextCol ) is not an obstacle and
       the node with the coordinate (prevRow,  nextCol) is not a boundary then
          forcedNeighbours.ADD(node with the coordinate (prevRow,  nextCol))
   endif
   If  there is an obstacle on the location with the coordinate (prevRow,  currCol ) and
      the node with the coordinate (prevRow,  prevCol ) is not an obstacle and
      the node with the coordinate (prevRow,  prevCol) is not a boundary then
         forcedNeighbours.ADD(node with the coordinate (prevRow,  prevCol))
  endif
   If   there is an obstacle on the location with the coordinate (nextRow,  currCol ) and
       the location with the coordinate (nextRow,  nextCol ) is not an obstacle and
       the location with the coordinate (nextRow,  nextCol) is not a boundary then
         forcedNeighbours.ADD(node with the coordinate (nextRow,  nextCol))
  endif
  If   there is an obstacle on the location with the coordinate (nextRow,  currCol ) and
       the location with the coordinate (nextRow,  prevCol ) is not an obstacle and
       the location with the coordinate (nextRow,  prevCol) is not a boundary then
         forcedNeighbours.ADD(node with the coordinate (nextRow, prevCol))
  endif

   /********* Forced Neighbour: Obstacle/Boundary East/West (2 steps ahead) ************/
  /*Attempt to add forced neighbours on the south west diagonal of the current location*/
   If   there is an obstacle on the location with the coordinate (currRow, prevCol+dCol ) and
       the node with the coordinate (nextRow,  prevCol ) is not an obstacle and
      the node with the coordinate (nextRow,  prevCol) is not a boundary then
         forcedNeighbours.ADD(node with the coordinate (nextRow,  prevCol))
  endif
  /*Attempt to add forced neighbours on the north west diagonal of the current location*/
   If  there is an obstacle on the location with the coordinate (currRow, prevCol+dCol ) and
      the node with the coordinate (prevRow,  prevCol ) is not an obstacle and
      the node with the coordinate (prevRow,  prevCol) is not a boundary then
         forcedNeighbours.ADD(node with the coordinate (prevRow,  prevCol))
  endif
  /*Attempt to add forced neighbours on the north west diagonal of the current location*/
   If  there is an obstacle on the location with the coordinate (currRow, nextCol+dCol ) and
      the node with the coordinate (nextRow,  nextCol ) is not an obstacle and
      the node with the coordinate (nextRow,  nextCol) is not a boundary then
         forcedNeighbours.ADD(node with the coordinate (nextRow,  nextCol))
  endif
  /*Attempt to add forced neighbours on the north west diagonal of the current location*/
   If  there is an obstacle on the location with the coordinate (currRow, nextCol+dCol ) and
      the node with the coordinate (prevRow,  nextCol ) is not an obstacle and
      the node with the coordinate (prevRow,  nextCol) is not a boundary then
        forcedNeighbours.ADD(node with the coordinate (prevRow,  nextCol))
  endif

endif

/*Implemented in [Code Block 7]*/
```

```
function ADD_NATURAL_NEIGHBOURG(neighbours, nextRow, currCol) returns list of natural neighbours /
/* The function checks if  location with the coordinate (nextRow, currCol) is not null and is not a boundary and is not an obstacle,
then it adds the location to the list of neighbours*/

/*Implemented in [Code Block 8]*/
```

```
Function Jump(dRow, dCol, current, start, goal) returns the next location (if it exists) that could be the next jump point
nextRow <- current .row + dRow
nextCol <- current .col +dCol

If  the node with the coordinate (nextRow,  nextCol ) is not an obstacle and
  the node with the coordinate (nextRow,  nextCol) is not a boundary then
 return NULL;
endif

/*The next location is the goal, then return the next location…*/
if  nextRow == goal.Row and nextCol == goal.Col then
   return new LocationWrapper nextRow , nextCol)
endif

if (dRow != 0 and dCol != 0)
   /*Diagonal forced neighbours check*/
   rowCheck =  there is an obstacle on the location with the coordinate (current.Row,  current.col-dCol) and
                    the location with the coordinate (nextRow,  current.col-dCol ) is not an obstacle and
                    the location with the coordinate (nextRow,  current.col-dCol) is not a boundary
  colCheck =  there is an obstacle on the location with the coordinate (current.Row -dRow,  current.col) and
                    the location with the coordinate (current.row-dRow,  nextCol) is not an obstacle and
                    the location with the coordinate (current.row-dRow,  nextCol) is not a boundary

   /*If any of these are true, then this is a jump point*/
   if (rowCheck || colCheck)
       return next
   endif

    /*Need to check the vertical and horizontal directions to ensure we can continue on the diagonal else this is the jump point
    If (jump(dRow, 0, next, start, end) is not NULL or jump(0, dCol, next, start, end) is not NULL) then
        return next
    endif

else

   if (dRow != 0)
       /*Vertical forced neighbours check*/
      northNeighbourIsForced =  there is an obstacle on the location with the coordinate (nextRow,  nextCol-1) and
                          the location with the coordinate (nextRow+dRow,  nextCol-1) is not an obstacle and
                          the location with the coordinate (nextRow+dRow,  nextCol-1) is not a boundary
       southNeighbourIsForced =  there is an obstacle on the location with the coordinate (nextRow, nextCol+1) and
                          the location with the coordinate (nextRow+dRow,  nextCol+1) is not an obstacle and
                          the location with the coordinate (nextRow+dRow,  nextCol+1) is not a boundary

        /*If any of these are true, then this is a jump point*/
        if (northNeighbourIsForced || southNeighbourIsForced)
            return next
        endif
 else
```

```
/*Horizontal forced neighbours check*/
westNeighbourIsForced =  there is an obstacle on the location with the coordinate (nextRow-1,  nextCol) and
                    the location with the coordinate (nextRow-1,  nextCol+dCol) is not an obstacle and
                    the location with the coordinate (nextRow-1,  nextCol+dCol) is not a boundary
eastNeighbourIsForced =  there is an obstacle on the location with the coordinate (nextRow+1, nextCol) and
                    the location with the coordinate (nextRow+1,  nextCol+dCol) is not an obstacle and
                    the location with the coordinate (nextRow+1,  nextCol+dCol) is not a boundary

        /*If any of these are true, then this is a jump point*/
        if (westNeighbourIsForced || eastNeighbourIsForced)
            return next
        endif
endif

return jump(dRow,dCol,next,start,end)

/*Implemented in [Code Block 6]*/
```

Problem & Limitations

The *A** is an optimal algorithm; however it has a large memory overhead due to the tree node expansion at each step of the process. Therefore it is not practical to resolve large scale problems [4]. The aim of this report is to compare the current implementation performance of the *A** and *JPS* against a set of defined scenarios, to establish the performance gains of the *JPS* algorithm and whether both algorithms reach the target.

Performance Measurements

This analysis will confirm whether the proposed implementation of each algorithm is complete and optimal in the listed scenarios. In order to establish the potential memory performance gain, the number of steps used to reach destination, the number of expansions as well as the number of neighbours/jump points will be used for comparison between the *A** and JPS. Time to completion will also be analysed to establish which algorithm is more time efficient.

Model & Experiment Assumptions

We will study an ant agent which aim is to depart from its nest and move in any of the quadrant directions to reach its food supply. The quadrant directions means any one horizontal/vertical or diagonal step move, represented by the enumeration; North (N), South (S), East (E), West (W), NE (North East), NW(North West), SE(South East),SW (South West).

The problem domain is defined and constrained by these assumptions:

- The environment is represented by a uniform grid system that sub-divides the space in equal sized cell. Each section is represented by a square. It also contains a unique agent (the 'ant') that departs from one cell of red colour (the nest) and aims at finding the food supply; its final destination. This is represented by a blue cell on the grid. The environment is delimited by a boundary and contains a number of obstacles.
- The Boundary is a line delimiting the edges of the environment. Any object existing outside the environment borders is assumed to have no direct or indirect influence on the agent. The environment is defined by the shape of the boundary, the space within the boundary as well as any elements located within this space. The boundary is assumed to be a parallelogram with right angles. The line joining any two angles is continuous (i.e. there is no gap). Therefore, in this model, the boundary can only represent a square or a rectangle. No other shape is allowed.
- The agent is rational and autonomous. It is assumed the agent can move one or more steps at a time in any of these directions; North (N), South(S), East (E) or West (W) and one step away diagonally. There is only one single agent.
- The Obstacle is an element located in the environment that prevents the agent to move at the location of the Obstacle.
- A run is represented by the path the ant takes from its nest to either i) its food supply (successful run) or ii) when all cells have been visited and the nest has not been reached (a.k.a. unreachable run). It is assumed there is no time limit to attain the food supply. It is also allowed to visit a cell more than once for a given run.
- There is one unique goal (i.e. the food supply), that is static for a given scenario.
- No other external factor disturbs the environment, the agent or the goal.

All these assumptions are summarised in the Table 1 below.

Table 1 below, inspired from [4], summaries the environment and the agent properties.

Environment	**Observable/ Partially Observable/ Unobservable?**	**Deterministic/ Stochastic/ Uncertain?**	**Episodic/ Sequential?**	**Static/Dynamic/Semi-Dynamic?**
The grid with the nest, food supply and potential obstacle	Partially – the agent has only a partial view of the environment at any point in time.	Uncertain – the environment is neither fully observable nor deterministic, usually the agent has more than one available choice when deciding to make the next step	The agent's experience is divided into atomic episodes	The environment configuration is static, i.e. the boundaries, obstacles, food supply and nest cannot move during a run.

Environment state	**Discrete/Continuous**
The environment	Discrete – The grid has a finite number of cells, and the agent's move at a given time comes from a discrete enumeration (N,W,E,S, NE, NW, S,SW, NE, NW)

Agent	**Single/Multiagent**
The 'ant'	Single –only one ant crosses the grid during a run

The Goal	**Single/Multigoals**
The 'ant'	Single – there is only one goal and it is static

Externalities	**Presence and behaviours**
External factors	No other factor environment, the agent or the goal.

Table 1 – Environment and Agent Summary

Methodology

The experiment involves running the *A** and the *JPS* algorithms against a set of environments, and gathering measurements relating to the number of steps, search tree expansions, neighbours/jump points generated as well as time to completion for each scenario run. We will also establish whether the algorithms are optimal and complete in these scenarios. An environment is defined by a grid shape. There are currently two shapes available: a 16x16 grid and a 32x8 grid. Both grids contain 256 cells. Each environment contains 7 identical scenarios, defined below:

- Scenario 1 – this is the vanilla case. The nest and food supply are set diametrically opposed and there is no obstacle in the environment.
- Scenario 2 – there is still no obstacle, but this time both the nest and the food supply are adjacent.
- Scenario 3 – Similar to *Scenario 1* with a large obstacle separating the diagonal.
- Scenario 4 – the food supply is totally surrounded by an obstacle, i.e. the goal is unreachable.
- Scenario 5 – the *ant* needs to travel along a defined path.
- Scenario 6 – the obstacle is a straight horizontal line that separates the nest and food. However, the obstacle does touch the boundary at its extremity. There is a gap of one cell width available on each side, between and obstacle and the boundary.
- Scenario 7 – The *food* is surrounded by an obstacle on each side of the food cell, bar one side.

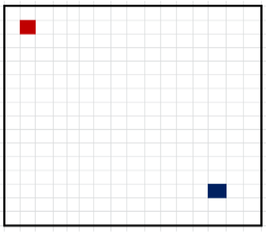
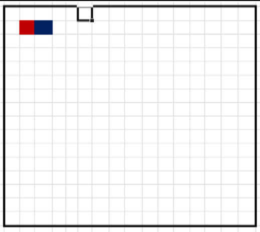
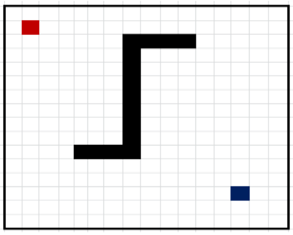
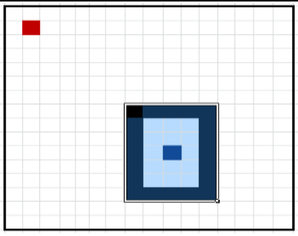
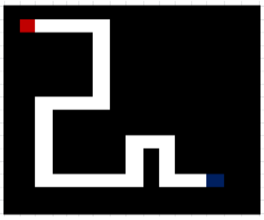
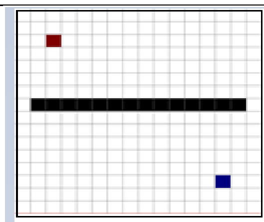
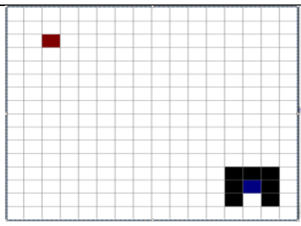
 Scenario A1	 Scenario A2	 Scenario A3	 Scenario A4
 Scenario A5	 Scenario A6	 Scenario A7	

Table 2 – Scenario results on a 16x16 grid environment.

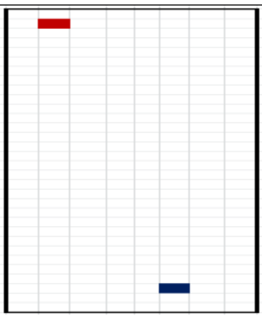
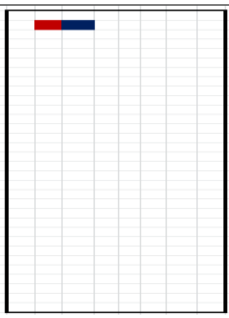
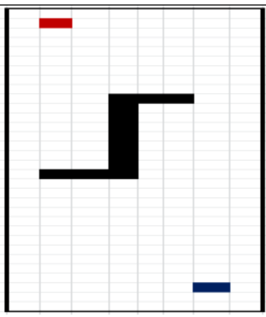
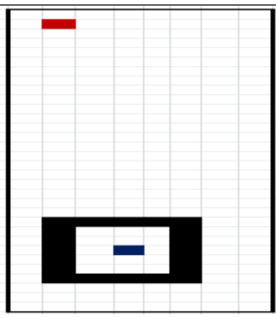
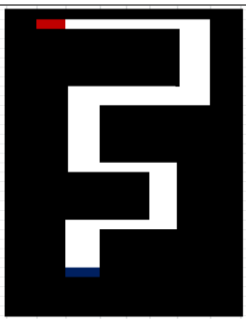
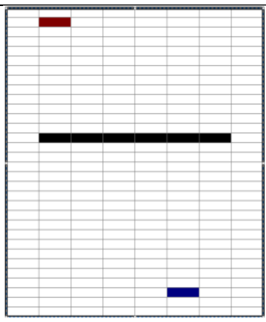
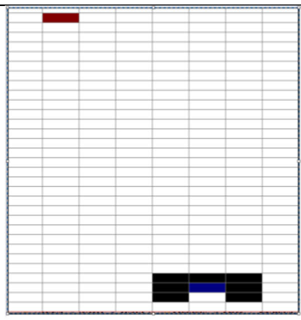
 Scenario B1	 Scenario B2	 Scenario B 3	 Scenario B4
 Scenario B5	 Scenario B6	 Scenario B7	

Table 3 – Scenario results on a 16x16 grid environment.

Results

A* Search Algorithm Path Discovery

Tables 4 and 5 show the paths taken by the ant from its net (red cell) to the food supply (blue cell), with different obstacle settings.

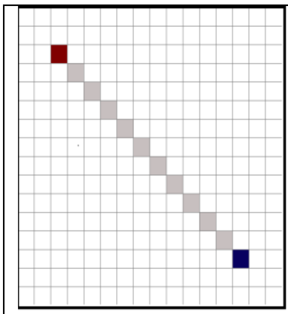
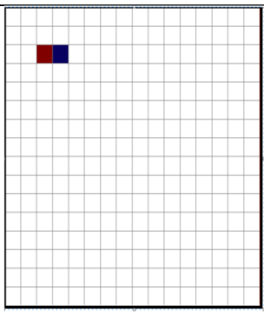
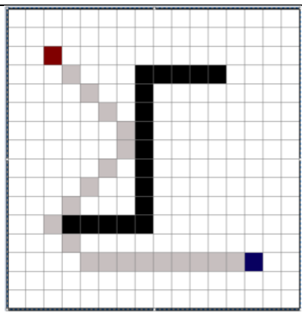
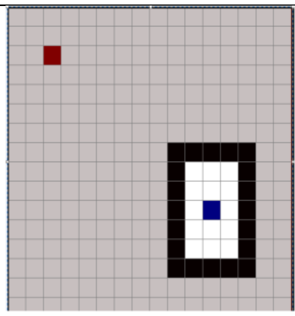
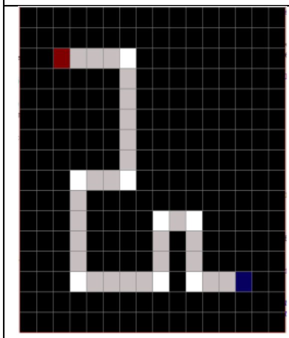
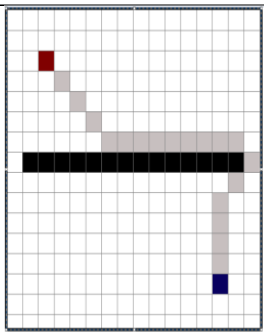
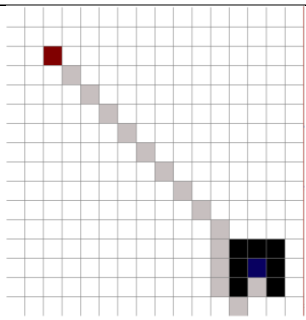
 Scenario A1	 Scenario A 2	 Scenario A 3	 Scenario A 4
 Scenario A 5	 Scenario A 6	 Scenario A 7	

Table 4 – Scenario results on a 16x16 grid environment.

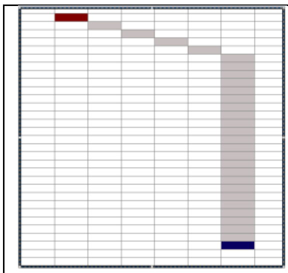
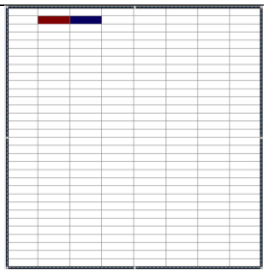
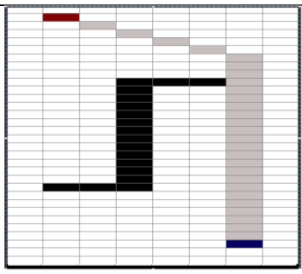
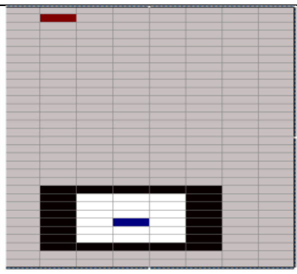
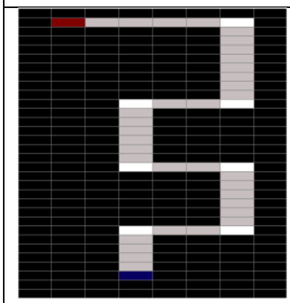
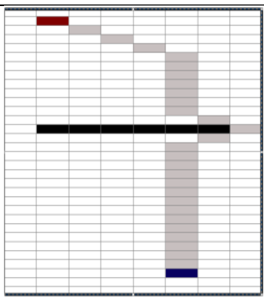
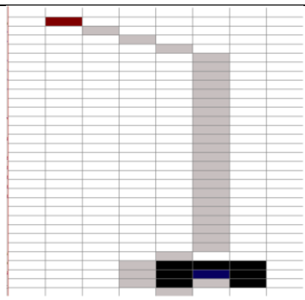
 Scenario B1	 Scenario B2	 Scenario B3	 Scenario B4
 Scenario B5	 Scenario B6	 Scenario B7	

Table 4 – Scenario results on a 32x8 grid environment.

It appears that the ant reaches its destination (the food supply) across all scenarios. The only exceptions are *Scenario_A4* and *Scenario_B4*, where the destination is unreachable. The diagonal path is used as a short-cut to reach destination. The ant does not cross boundaries or obstacles. There is no case where the ant visits a cell more than once or get into an infinite loop.

A* Search Algorithm Performance Summary

AlgoId	A_STAR

Row Labels	Number of Steps	Number of Expansions	Number of Neighbours	Time to Completion (microseconds)
FALSE				
SCENARIO_A4	222	222	1520	23
TRUE				
SCENARIO_A1 (*)	11	12	88	14
SCENARIO_A2 (*)	1	2	8	13
SCENARIO_A3	20	48	338	16
SCENARIO_A5	27	34	80	16
SCENARIO_A6	19	40	274	16
SCENARIO_A7	15	41	286	16

Average/Standard deviation of the number of Expansions in the presence of obstacles (when the destination is reached) is 41 and 6
Average/Standard deviation of the time to completion in the presence of obstacles (when the destination is reached) is 16 and 0

AlgoId	A_STAR

Row Labels	Number of Steps	Number of Expansions	Number of Neighbours	Time to Completion (microseconds)
FALSE				
SCENARIO_B4	217	217	1426	26
TRUE				
SCENARIO_B1 (*)	28	29	224	17
SCENARIO_B2 (*)	1	2	8	13
SCENARIO_B3	28	29	221	16
SCENARIO_B5	35	41	92	19
SCENARIO_B6	28	59	431	19
SCENARIO_B7	32	203	1484	21

Average/Standard deviation of the number of Expansions in the presence of obstacles (when the destination is reached) is 83 and 81
Average/Standard deviation of the time to completion in the presence of obstacles (when the destination is reached) is 18.75 and 1.63

(*) Scenarios which do not contain obstacles

Table 6a/6b – *A** detailed results

Tables 6a/6b lists the scenarios of types A (16x16 grid) and B (32x8 grid) in the first column. Each scenario type is split in two categories; the ones i) where the destination is reachable vs ii) non reachable destinations. The second column shows the number of steps taken to reach destination, or when the search is stopped (all cells have been visited due to destination unreachability). The third column indicates the number of expansions that is generated at each step. At each iteration of the most outer loop of the *GET_SHORTEST_PATH()* function, the *A** algorithm determines which ones of its partial paths need to be expanded into one or more longer paths. This is achieved by minimising the heuristic function f(n) defined in section 'High Level Description'. The fourth column displays the number of neighbours that need to be checked upon at each expansion point. The time to completion (in microseconds) of each scenario run is present in the last column.

Observation 1: When the target is not reachable, the number of expansions/neighbours to consider is the largest of all the use cases. This expected as all cells need to be visited.

Observation 2: As expected in *Scenario_A1* and *Scenario_B1*, the number of steps is 8 times the number of neighbours, as there are no obstacles along the way (respectively 88 = 11*8 and 224 = 28*8). As the ant takes the shortest path (i.e. the diagonal), it needs to check each of the 8 neighbours at each step. Note - the number of expansions (in the absences of obstacle) is equal to the number of steps+1, as the nest location needs to be expanded but it is not counted as a step.

Observation 3: The presence of obstacles along the way increases the number of steps to reach the destination, as both the number of steps and number of neighbours to consider rise with the increased number of obstacles. However, for *Scenarios_A3/_A5/_A6* and *_A7* the number of steps required to attain destination is not proportional to the number of tree expansions. Indeed, *Scenario_A5* has 27 steps for 34 expansions. The other scenarios have respectively 20, 19 and 15 steps for 48, 40 and 41 expansions. This behaviour is also shown in *Scenario_B1/_B3/_B6,* where for a constant number of steps 28 in all cases, the number of neighbours varies from 221 to 431. Scenario_A5 is particular, as each cell is surrounded by obstacles which force the ant to only move west/east (north/south), with an extra diagonal move at the corners. Therefore, for a constant number of obstacles, the obstacles position on the gird have a greater or lesser impact on the number of neighbours to consider, and by extension on memory usage.

Observation 4: A comparison of Scenario_A7 and Scenario_B7, shows that the doubling of steps to reach destination (from 15 to 32) has potentially far greater than expected impact on memory, in this case five times (from 286 neighbours to consider, to 1484) . This results is worrying in particular when Scenario_B6 (28 steps to destination) and *Scenario_B7* (32 steps to destination) are compared, the number of expansions/neighbours jump respectively from 59/431 to 203/1484. Consequently, it looks like the relationship between the number of steps/obstacles and memory is not linear but potentially exponential.

Jump Point Search Path Discovery

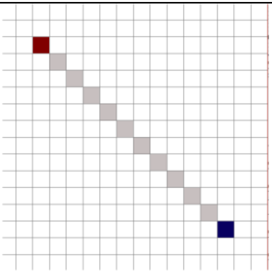
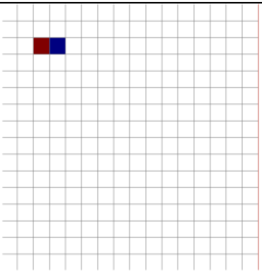
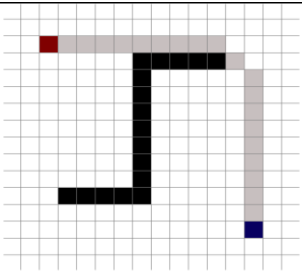
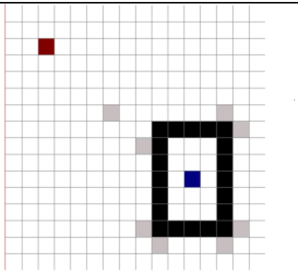
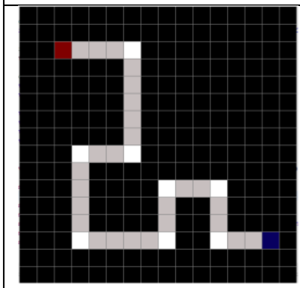
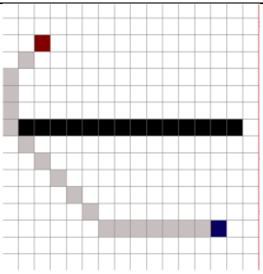
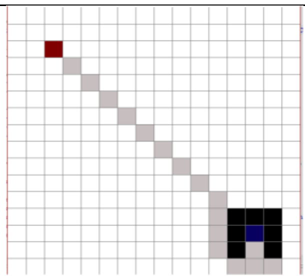
 Scenario A1	 Scenario A2	 Scenario A3	 Scenario A4
 Scenario A5	 Scenario A6	 Scenario A7	

Table 7 – Scenario results on a 16x16 grid environment.

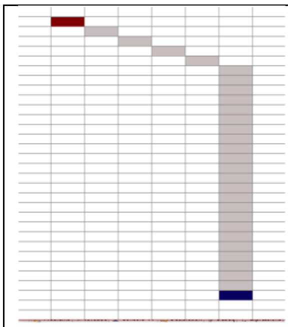
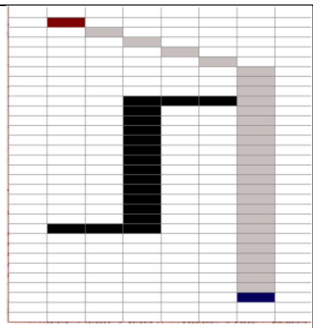
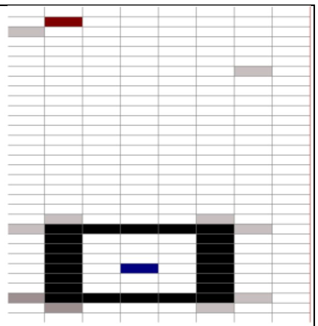
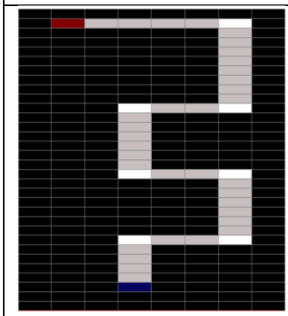
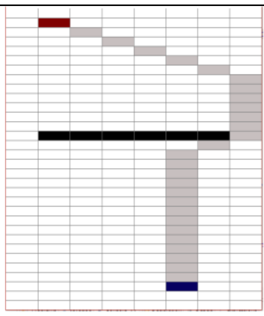
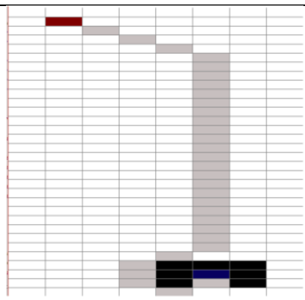
 Scenario B1	 Scenario B2	 Scenario B3	 Scenario B4
 Scenario B5	 Scenario B6	 Scenario B7	

Table 8 – Scenario results on a 32x8 grid environment.

The observations are the same as the ones made for the A* algorithm.

Jump Point Search Performance Summary

AlgoId	JUMP_POINT_SEARCH

Scenario Name	Number of Steps	Number of Expansions	Jump Points	Time to Completion (microseconds)
DESTINATION IS REACHED: FALSE				
SCENARIO_A4	9	9	10	15
DESTINATION IS REACHED: TRUE				
SCENARIO_A1 (*)	11	2	1	16
SCENARIO_A2 (*)	1	2	1	13
SCENARIO_A3	20	4	4	15
SCENARIO_A5	27	18	27	15
SCENARIO_A6	18	9	9	15
SCENARIO_A7	17	10	12	13

Average/Standard deviation of the number of Expansions in the presence of obstacles (when the destination is reached) is 10 and 6
Average/Standard deviation of the time to completion in the presence of obstacles (when the destination is reached) is 15 and 1

AlgoId	JUMP_POINT_SEARCH

Scenario Name	Number of Steps	Number of Expansions	Jump Points	Time to Completion (microseconds)
DESTINATION IS REACHED: FALSE				
SCENARIO_B4	13	13	16	14
DESTINATION IS REACHED: TRUE				
SCENARIO_B1 (*)	28	3	2	15
SCENARIO_B2 (*)	1	2	1	12
SCENARIO_B3	28	4	6	15
SCENARIO_B5	35	16	24	15
SCENARIO_B6	28	6	8	15
SCENARIO_B7	32	7	9	15

Average/Standard deviation of the number of Expansions in the presence of obstacles (when the destination is reached) is 8 and 5
Average/Standard deviation of the time to completion in the presence of obstacles (when the destination is reached) is 15 and 1

(*) Scenarios which do not contain obstacles

Table 9a/9b – *Jump Point Search* detailed results

Tables 9a/9b display the same information as Tables 4a/4b. However, the number of neighbours to be considered is replaced by the number of jump points. There are indeed equivalent concepts. A jump point is a selected natural or forced neighbours generated by pruning where an expansion is considered. The number of steps in this case has been reconstructed from the final list of evaluated jump points.

Observation 5: When the target is not reachable, the number of expansions is small (9). This is a small number compared to the number of expansions (222) to consider for the A* algorithm.

Observation 6: It is difficult to use statistical measures to compare *Scenario_A* and *Scenario*_B against each algorithm, as the list of samples is very small and the standard deviation is large around the means. However, it is safe to say that the JPS algorithm significantly reduces the number of expansions and therefore, the memory footprint. When the A scenarios are compared for A* and JPS, it emerges that the average number of expansions, for reachable scenarios with obstacles is 10 (std dev =6) for the JPS, whereas it is 41 (std dev = 6) for the A*.

Observation 7: It is also interesting to note that scenarios with obstacles which are reachable have a standard deviation for the JSP's A and B scenarios are respectively 6 and 5, whereas they are 18 and 72 for the A*.

Observation 8: In terms of speed, the time performance gain is more visible with longer paths. With the A scenarios, the average and standard deviation indicate that no conclusion can be drawn in terms of time gain/loss between the two algorithms. However, the picture is clearer with the B scenarios; where there is on average is 4 microseconds gain between the two algorithms for scenarios that are complete with obstacles. It also shows that JSP can cope better with increasing paths, as the time to completion remains constant between scenarios A and B. The experiment uses a very small grid, so it is difficult to state on the time speed up gains per path length. However, it is probably safe to assume the time to completion will increase at a slower pace compared to the A*.

Evaluation

From this analysis of the results, it seems that the JSP provides the same characteristics in terms of optimality and completeness as the A* algorithm. They both complete reach destination using the shortest path when the destination is reachable. JSP shows a real advantage in terms of memory usage, with a small variation of memory usage with the increased distance and complexity (i.e. additional obstacles). The time to completion also seems to be improved, with the *JSP* algorithm and it seems that it would increase at a slower pace than the A*.

Conclusion

JPS is the clear winner in terms of memory size and time to completion across all the proposed scenarios. On an 16x16 grid, the *JPS* algorithm generated approximately 4 times less expansion tree nodes compared to the *A**, and examined approximately 20 times less jump points/neighbours. The time to completion also improved on average by 4 micro seconds on scenarios with longer paths (B scenarios). By nature, the *JPS* algorithm reduces the list of nodes in the search tree, making each list operation cheaper. It also prunes nodes online with no extra preprocessing or memory overhead. However, this test is somewhat limited by i) the grid size, ii) the fact that the grid is uniform, iii) all scenarios imply that the *JSP* can take full advantage of the environment symmetry. It would be interesting to compare these algorithms in a non-symmetrical environment. For example, where moving to a set of defined path implies a penalty. It would also be interesting to investigate the performance gain in environments where obstacles are not static, but appear/disappear during a scenario run. In this case, pre-processing jump points may be inadequate, as a number of jump points could become incorrect moves from a period of time to the next (when a jump change from being a moveable cell to an obstacle). An improvement of the current *JSP* incarnation could make use of a jump point store that then saves all jump points per scenario. This would imply the jump points pre-processing is performed once online, and then stored against a scenario name in a data store (at time t). The list of jump points per scenario could be reloaded, alleviating the need of pre-processing jump points and therefore improving the time to completion of a run, for time strictly greater than t.

Appendix

The proposed implementation of the *A** and *JSP* algorithms have been inspired from [1] and [7].

```
package Algorithms.JumpPointSearch;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Vector;

import Shared.IAntWordAccessor;
import Shared.Location;

public class InformedSearch {

	private JumpPointGrid grid;
	private PriorityQueue<LocationWrapper> openSet;
	private HashSet<LocationWrapper> closedSet, jumpPoints, jumpPointsOnPath;
	private HashMap<LocationWrapper, LocationWrapper> locationOrigins;
	private int numRows;
	private int numCols;
	private Boolean isGoalReached;
	private static LocationWrapper startLocation;
	private static LocationWrapper destinationLocation;
	private static IAntWordAccessor antWordAccessor;
	private int expansionCount;
	private int neighbourCount;
	private InformedSearchType informedSearchType;

	public InformedSearch(		InformedSearchType informedSearchType,
						IAntWordAccessor antWordAccessor,
						JumpPointGrid grid,
						Location startLocation, Location destinationLocation,
						int numRows,
						int numCols) {

		/*The underlying grid*/
		this.grid = grid;
		/*The set of nodes already evaluated.*/
		this.closedSet = new HashSet<LocationWrapper>();
		/*The set of currently discovered nodes still to be evaluated.*/
		this.openSet = new PriorityQueue<LocationWrapper>();
		/*The map key corresponds to a given location and
		the value relates to the most efficient location the key (location) can be reached from*/
		this.locationOrigins = new HashMap<LocationWrapper, LocationWrapper>();
		this.jumpPoints = new HashSet<LocationWrapper>();
		this.jumpPointsOnPath = new HashSet<LocationWrapper>();
		/*Set the start point*/
		InformedSearch.startLocation = new LocationWrapper(0,0,startLocation.row, startLocation.col);
		/*Set the goal*/
		InformedSearch.destinationLocation = new LocationWrapper(0,0,destinationLocation.row,  destinationLocation.col);
		/*Set the antWordAccessor*/
		InformedSearch.antWordAccessor = antWordAccessor;
		/*Set the grid row number*/
		this.numRows = numRows;
		/*Set the grid col number*/
		this.numCols = numCols;
		/*Count the number of times, the tree has been expanded*/
		this.expansionCount = 0;
		/*Count the number of jump points*/
		this.neighbourCount = 0;
		/*Establishes whether the goal has been reached or not*/
		this.isGoalReached = false;
		this.informedSearchType = informedSearchType;
	}
```

```
//********************************* Start Code Block 1 *********************************************
	//The aim of this search is to find the nearest jump point
	/* this method was copied from [7]*/
	@SuppressWarnings("unchecked")
	public List<Location> getShortestPath() {

		this.openSet.add(InformedSearch.startLocation);
		/*A list only used in case of failure of the search to discover the goal. It contains all discovered node (evaluated or not)*/
		LinkedList<LocationWrapper> retainedLocations = new LinkedList<LocationWrapper>();

		LocationWrapper current = null;
		while(!this.openSet.isEmpty()) {
			this.expansionCount++;
			//Get the current location
			current = this.openSet.poll();
			//This keeps in memory all the neighbours that have been retained at some point in the search
			//This is only used in case the goal cannot be reached. It provides the bread crums path along the search path
			retainedLocations.add(current);

			//Define the Goal - the destination is reached
			//Return the shortest path
			if(current.equals(InformedSearch.destinationLocation)) {
				this.isGoalReached = true;
				List<? extends Location> shortestPath = getPath();
				return (List<Location>) shortestPath;
			}

			//Add the current node to the set of nodes already evaluated.
			this.closedSet.add(current);

			//Get the list of possible neighbours.
			List<LocationWrapper> neighbours= this.getAllNeighbours(current);

			for(LocationWrapper neighbour : neighbours) {
				this.neighbourCount++;
				//g score corresponds to the  shortest distance from the startLocation to the currenetLocation
				//+1 corresponds to a discrete move (one step)
				double gScore = current.getGValue()+1;
				if(this.closedSet.contains(neighbour) &&
						gScore >= neighbour.getGValue()) {
					continue;
				}

				if(!this.openSet.contains(neighbour) || gScore < neighbour.getGValue()) {
					//remove a relationship for a neighbour where the current location gSCore is the neighbour gSCore
					this.locationOrigins.remove(neighbour);
					//set the relationship between the neighbour and the current location
					this.locationOrigins.put(neighbour, current);
					//Set the gValue and fValue of the selected best location for the next move
					neighbour.setGValue(gScore);
					neighbour.setFValue(neighbour.getGValue() + getHeuristic(neighbour));
					//update the neighbour in the openSet
					this.openSet.remove(neighbour);
					this.openSet.add(neighbour);
				}
			}
		}

		//The goal is not reach... Return the partial found path.
		isGoalReached = false;
		List<? extends Location> partialPath =  retainedLocations;
		return (List<Location>) partialPath;
	}

//********************************* End Code Block 1 *********************************************
```

```
	//Returns whether the search algorithm has attained the target or not
	public Boolean isGoalReached(){

		return this.isGoalReached;
	}

	//Get the number of expansions
	public int getExpansionCount() {

		return this.expansionCount;
	}

	//Get the number of jump points
	public int getNeighbourCount() {

		return this.neighbourCount;
	}

//********************************* Start Code Block 2 **********************************************

	//This method wraps two use cases:
	//i) AStar algorithm: returns the list of neighbours one step away from the current location
	//ii)JUMP POINT SEARCH: returns the list of successors. A successor node is the node that is just as
	// good (or better) than the best node in the open list.
	private List<LocationWrapper> getAllNeighbours(LocationWrapper current){

		List<LocationWrapper> neighbours = null;
		if (this.informedSearchType == InformedSearchType.A_STAR){
			//Identify all potential neighbours for the A STAR case
			neighbours = getOneStepNeighbours(current);
		}
		else if (informedSearchType == InformedSearchType.JUMP_POINT_SEARCH){
			//Identify individual jump point successors for JUMP POINT SEARCH case
			neighbours = identifySuccessors(current, InformedSearch.startLocation, InformedSearch.destinationLocation);
		}
		else {
			System.out.println("Informed Search type not supported" + informedSearchType.toString());
			neighbours = null;
		}
		return neighbours;
	}
//********************************* End Code Block 2 **********************************************
```

```
//********************************* Start Code Block 3 *********************************************

	//This is the path reconstruction
	//Return the path from the initial location to the goal as a list of LocationWrapper
	private List<LocationWrapper> getPath() {

		LinkedList<LocationWrapper> path = new LinkedList<LocationWrapper>();

		LocationWrapper current = InformedSearch.destinationLocation;
		jumpPointsOnPath.add(current);
		LocationWrapper next = null;
		for (LocationWrapper key: this.locationOrigins.keySet()){
			if (key.equals(current)){
				next = this.locationOrigins.get(key);
				break;
			}
		}
		//section copied from [7]
		while(next != null && !current.equals(InformedSearch.startLocation)){
			Location nextLoc = (Location) next;
			Location currentLoc = (Location) current;
			int dRow = nextLoc.row - currentLoc.row;
			if(dRow != 0) {
				dRow /= Math.abs(dRow);
			}
			int dCol = nextLoc.col - currentLoc.col;
			if(dCol != 0) {
				dCol /= Math.abs(dCol);
			}
			while(!current.equals(next)) {
				path.add(current);
				currentLoc = (Location) current;
				int curX = currentLoc.row, curY = currentLoc.col;
				int nextRow = curX + dRow, nextCol = curY + dCol;
				current = this.grid.get(nextRow, nextCol);
				if(current == null || Location.isBoundary(InformedSearch.antWordAccessor, current)) {
					return null;
				}
			}
			current = next;
			next = this.locationOrigins.get(next);
			this.jumpPointsOnPath.add(current);
		}
		return path;
	}

//********************************* End Code Block 3 *********************************************
```

```
//********************************** Start Code Block 4 ************************************************

	//Returns a list of all found successors
	private List<LocationWrapper> identifySuccessors(LocationWrapper current, LocationWrapper start, LocationWrapper end) {

		int dRow, dCol;
		List<LocationWrapper> successors = new ArrayList<LocationWrapper>();
		Location currentLocation = (Location)current;

		List<LocationWrapper> neighbours = new ArrayList<LocationWrapper>();
		//Get the parent of the current location
		LocationWrapper parent = this.locationOrigins.get(current);
		//At the start location, add all possible neighbours to the neighbours list
		if(parent == null) {
			neighbours = addAllNonBoundayNeighours(current, neighbours);
		}
		//This is a location along the path
		else {
			//Find the vertical direction of travel
			dRow = currentLocation.row - ((Location)parent).row;
			if(dRow != 0) {
				//this is necessary to keep the correct direction (north/south)
				dRow = dRow/Math.abs(dRow);
			}
			//Find the horizontal direction of travel
			dCol = currentLocation.col - ((Location)parent).col;
			if(dCol != 0) {
				//this is necessary to keep the correct direction (east/west)
				dCol = dCol/Math.abs(dCol);
			}
			neighbours.addAll(getPrunedNeighbours(current, dRow, dCol));
		}

		for(LocationWrapper neighbour : neighbours) {
			//This is the direction between the current location and its neighbour
			Location neightborLocation = (Location)neighbour;
			dRow = (neightborLocation.row - currentLocation.row);
			dCol = (neightborLocation.col - currentLocation.col);

			//Get jump, when it exists, following the neighbour direction
			LocationWrapper jumpPoint = jump(dRow, dCol, current, start, end);

			//Add jump point to the successors list
			if(jumpPoint != null) {
				successors.add(jumpPoint);
			}
		}

		//Add the jumpPoints to the set
		this.jumpPoints.addAll(successors);

		//Return the list of jump points
		return successors;
	}

//********************************** End Code Block 4 ************************************************

//********************************** Start Code Block 5 ************************************************

	//Add all allowed cardinal location to the neighbour list
	private List<LocationWrapper> addAllNonBoundayNeighours(LocationWrapper current, List<LocationWrapper> neighbours){

		for(LocationWrapper location : this.grid.getNeighbours(current, this.numRows, this.numCols)) {
			if(!Location.isBoundary(InformedSearch.antWordAccessor, location)) {
				neighbours.add(location);
			}
		}
		return neighbours;
	}
//********************************** End Code Block 5 ************************************************
```

```
//********************************** Start Code Block 6 ***********************************************
	//Returns the next location (if it exists, i.e. not null) that could be the next jump point
	private LocationWrapper jump(int dRow, int dCol, LocationWrapper current, LocationWrapper start, LocationWrapper end) {

		//Produce the next row and column coordinate
		Location currentLocation = (Location) current;
		int nextRow = currentLocation.row + dRow;
		int nextCol = currentLocation.col + dCol;

		int currRow = currentLocation.row;
		int currCol = currentLocation.col;

		//If the next location is a boundary or an obstacle, then return null
		if(Location.isBoundary(InformedSearch.antWordAccessor,nextRow, nextCol) ||
			Location.isObstacle(InformedSearch.antWordAccessor,nextRow, nextCol)) {
			return null;
		}

		//Get the next cell location from the grid
		LocationWrapper next = this.grid.get(nextRow, nextCol);
		//THe next move is the goal
		if(next.equals(end)) {
			return next;
		}

		if(dRow != 0 && dCol != 0) {
			//Diagonal Case
			//Diagonal forced neighbours check
			boolean rowCheck = Location.isObstacle(InformedSearch.antWordAccessor,currRow, currentLocation.col-dCol) &&
					!Location.isObstacle(InformedSearch.antWordAccessor,nextRow, currentLocation.col-dCol) &&
					!Location.isBoundary(InformedSearch.antWordAccessor,nextRow, currentLocation.col-dCol);
			boolean colCheck = Location.isObstacle(InformedSearch.antWordAccessor,currRow-dRow, currentLocation.col) &&
					!Location.isObstacle(InformedSearch.antWordAccessor,currentLocation.row-dRow, nextCol) &&
					!Location.isBoundary(InformedSearch.antWordAccessor,currentLocation.row-dRow, nextCol);

			//if any of the these are true, then this is the jump point
			if(rowCheck || colCheck) {
				return next;
			}

			//Need to check the vertical and horizontal directions to ensure we can continue on the diagonal
			//else this is the jump point
			if(jump(dRow, 0, next, start, end) != null ||
			  jump(0, dCol, next, start, end) != null) {
				return next;
			}

		} else {
			//Vertical Case
			if(dRow != 0) {
				//Vertical forced neighbours check
				boolean northNeighbourIsForced = Location.isObstacle(InformedSearch.antWordAccessor,nextRow, nextCol-1) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,nextRow+dRow, nextCol-1) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,nextRow+dRow, nextCol-1);
				boolean southNeighbourIsForced = Location.isObstacle(InformedSearch.antWordAccessor,nextRow, nextCol+1) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,nextRow+dRow, nextCol+1) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,nextRow+dRow, nextCol+1);

				//If any of the these are true, then this is the jump point
				if(northNeighbourIsForced || southNeighbourIsForced){
					return next;
				}
			} else {
				//Horizontal Case
				//Horizontal forced neighbours check
				boolean westNeighbourIsForced = Location.isObstacle(InformedSearch.antWordAccessor,nextRow-1, nextCol) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,nextRow-1, nextCol+dCol) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,nextRow-1, nextCol+dCol);
				boolean eastNeighbourIsForced = Location.isObstacle(InformedSearch.antWordAccessor,nextRow+1, nextCol) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,nextRow+1, nextCol+dCol) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,nextRow+1, nextCol+dCol);

				//If any of the these are true, then this is the jump point
				if(westNeighbourIsForced || eastNeighbourIsForced) {
					return next;
```

```
                    }
                }
            }

            //We continue in the same direct and we check for another jump point
            return jump(dRow, dCol, next, start, end);
        }

//********************************* End Code Block 6 **********************************************

//********************************* Start Code Block 7 **********************************************

        //Returns the list of natural and forced neighbours
        private List<LocationWrapper> getPrunedNeighbours(LocationWrapper current, int dRow, int dCol) {

            //Instantiate the  neighbours list
            List<LocationWrapper> neighbours = new ArrayList<LocationWrapper>();
            Location currentLocation = (Location) current;
            //Get the next location coordinates
            int currRow = currentLocation.row;
            int currCol = currentLocation.col;

            int nextRow = currentLocation.row + 1;
            int prevRow = currentLocation.row - 1;
            int nextCol = currentLocation.col + 1;
            int prevCol = currentLocation.col - 1;

            //Initialise the list of neighbours to check.
            LocationWrapper forcedNeighbour = null;
            boolean forceNeighbour =false;
            if(dRow != 0 && dCol != 0) /*Diagonal Travel Case*/ {

                nextRow = currentLocation.row + dRow;
                prevRow = currentLocation.row - dRow;
                prevCol = currentLocation.col - dCol;
                nextCol = currentLocation.col + dCol;

                //Attempt to add natural neighbours
                addNaturalNeighbour(neighbours, currentLocation.row,nextCol);
                addNaturalNeighbour(neighbours, nextRow,currentLocation.col);
                addNaturalNeighbour(neighbours, nextRow,nextCol);

                //Attempt to add forced neighbours on the east/west diagonal side on current location. Depending on the direction of travel.
                forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,currRow, prevCol) &&
                                !Location.isObstacle(InformedSearch.antWordAccessor, nextRow, prevCol) &&
                                !Location.isBoundary(InformedSearch.antWordAccessor,nextRow, prevCol);
                if(forceNeighbour) {
                    forcedNeighbour = this.grid.get(nextRow,prevCol);
                    neighbours.add(forcedNeighbour);
                }

                //Attempt to add forced neighbours on the east/west diagonal side on current location. Depending on the direction of travel.
                forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,currRow, nextCol) &&
                                !Location.isObstacle(InformedSearch.antWordAccessor,prevRow, nextCol) &&
                                !Location.isBoundary(InformedSearch.antWordAccessor,prevRow, nextCol);
                if(forceNeighbour) {
                    forcedNeighbour = this.grid.get(prevRow, nextCol);
                    neighbours.add(forcedNeighbour);
                }

            } else if(dRow != 0) /*Vertical Travel Case*/ {

                //Attempt to add natural neighbour
                addNaturalNeighbour(neighbours, nextRow,currentLocation.col);

                /********* Forced Neighbour: Obstacle/Boundary East/West ************/
                //Attempt to add forced neighbours on the south east diagonal of the current location
                forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,currRow, nextCol) &&
                                !Location.isObstacle(InformedSearch.antWordAccessor,nextRow, nextCol) &&
                                !Location.isBoundary(InformedSearch.antWordAccessor,nextRow, nextCol);
                if(forceNeighbour) {
                    forcedNeighbour = grid.get(nextRow, nextCol);
                    neighbours.add(forcedNeighbour);
                }
```

```
				//Attempt to add forced neighbours on the north east diagonal of the current location
				forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,currRow, nextCol) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,prevRow, nextCol) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,prevRow, nextCol);
				if(forceNeighbour) {
					forcedNeighbour = grid.get(prevRow, nextCol);
					neighbours.add(forcedNeighbour);
				}

				//Attempt to add forced neighbours on the south east diagonal of the current location
				forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,currRow, prevCol) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,nextRow, prevCol) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,nextRow, prevCol);
				if(forceNeighbour) {
					forcedNeighbour = grid.get(nextRow, prevCol);
					neighbours.add(forcedNeighbour);
				}

				//Attempt to add forced neighbours on the north west diagonal of the current location
				forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,currRow, prevCol) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,prevRow, prevCol) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,prevRow, prevCol);
				if(forceNeighbour) {
					forcedNeighbour = grid.get(prevRow, prevCol);
					neighbours.add(forcedNeighbour);
				}

				/********* Forced Neighbour: Obstacle/Boundary North/South (2 steps ahead) ************/
				//Attempt to add forced neighbours on the south east diagonal of the current location
				forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,nextRow+dRow, currCol) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,nextRow, nextCol) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,nextRow, nextCol);
				if(forceNeighbour) {
					forcedNeighbour = grid.get(nextRow, nextCol);
					neighbours.add(forcedNeighbour);
				}

				//Attempt to add forced neighbours on the south west diagonal of the current location
				forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,nextRow+dRow, currCol) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,nextRow, prevCol) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,nextRow, prevCol);
				if(forceNeighbour) {
					forcedNeighbour = grid.get(nextRow, prevCol);
					neighbours.add(forcedNeighbour);
				}

				//Attempt to add forced neighbours on the north east diagonal of the current location
				forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,prevRow+dRow, currCol) && /*here*/
						!Location.isObstacle(InformedSearch.antWordAccessor,prevRow, nextCol) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,prevRow, nextCol);
				if(forceNeighbour) {
					forcedNeighbour = grid.get(prevRow, nextCol);
					neighbours.add(forcedNeighbour);
				}

				//Attempt to add forced neighbours on the north west diagonal of the current location
				forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,prevRow+dRow, currCol) && /*here*/
						!Location.isObstacle(InformedSearch.antWordAccessor,prevRow, prevCol) &&
						!Location.isBoundary(InformedSearch.antWordAccessor,prevRow, prevCol);
				if(forceNeighbour) {
					forcedNeighbour = grid.get(prevRow, prevCol);
					neighbours.add(forcedNeighbour);
				}
				/***************************************************/

			} else if(dCol != 0) /*Horizontal Travel Case*/ {

				//Attempt to add natural neighbours
				addNaturalNeighbour(neighbours, currentLocation.row,nextCol);


				/********* Forced Neighbour: Obstacle/Boundary North/South  ************/

				forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,prevRow, currCol) &&
						!Location.isObstacle(InformedSearch.antWordAccessor,prevRow, nextCol) &&
```

```
                    !Location.isBoundary(InformedSearch.antWordAccessor,prevRow, nextCol);
        if(forceNeighbour) {
            forcedNeighbour = grid.get(prevRow, nextCol);
            neighbours.add(forcedNeighbour);
        }

        forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,prevRow, currCol) &&
                    !Location.isObstacle(InformedSearch.antWordAccessor,prevRow, prevCol) &&
                    !Location.isBoundary(InformedSearch.antWordAccessor,prevRow, prevCol);
        if(forceNeighbour) {
            forcedNeighbour = grid.get(prevRow, prevCol);
            neighbours.add(forcedNeighbour);
        }


        forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,nextRow, currCol) &&
                    !Location.isObstacle(InformedSearch.antWordAccessor,nextRow, nextCol) &&
                    !Location.isBoundary(InformedSearch.antWordAccessor,nextRow, nextCol);
        if(forceNeighbour) {
            forcedNeighbour = grid.get(nextRow, nextCol);
            neighbours.add(forcedNeighbour);
        }

        forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,nextRow, currCol) &&
                    !Location.isObstacle(InformedSearch.antWordAccessor,nextRow, prevCol) &&
                    !Location.isBoundary(InformedSearch.antWordAccessor,nextRow, prevCol);
        if(forceNeighbour) {
            forcedNeighbour = grid.get(nextRow, prevCol);
            neighbours.add(forcedNeighbour);
        }

        /********* Forced Neighbour: Obstacle/Boundary East/West (2 steps ahead) ************/

        //Attempt to add forced neighbours on the south west diagonal of the current location
        forceNeighbour =
                    Location.isObstacle(InformedSearch.antWordAccessor,currRow, prevCol+dCol) &&  /*here*/
                    !Location.isObstacle(InformedSearch.antWordAccessor,nextRow, prevCol) &&
                    !Location.isBoundary(InformedSearch.antWordAccessor,nextRow, prevCol);
        if(forceNeighbour) {
            forcedNeighbour = grid.get(nextRow, prevCol);
            neighbours.add(forcedNeighbour);
        }

        //Attempt to add forced neighbours on the north west diagonal of the current location
        forceNeighbour =
                    Location.isObstacle(InformedSearch.antWordAccessor,currRow, prevCol+dCol) &&  /*here*/
                    !Location.isObstacle(InformedSearch.antWordAccessor,prevRow, prevCol) &&
                    !Location.isBoundary(InformedSearch.antWordAccessor,prevRow, prevCol);
        if(forceNeighbour) {
            forcedNeighbour = grid.get(prevRow, prevCol);
            neighbours.add(forcedNeighbour);
        }

        //Attempt to add forced neighbours on the south east diagonal of the current location
        forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,currRow, nextCol+dCol) && /*here*/
                    !Location.isObstacle(InformedSearch.antWordAccessor,nextRow, nextCol) &&
                    !Location.isBoundary(InformedSearch.antWordAccessor,nextRow, nextCol);
        if(forceNeighbour) {
            forcedNeighbour = grid.get(nextRow, nextCol);
            neighbours.add(forcedNeighbour);
        }

        //Attempt to add forced neighbours on the north east diagonal of the current location
        forceNeighbour = Location.isObstacle(InformedSearch.antWordAccessor,currRow, nextCol+dCol) && /*here*/
                    !Location.isObstacle(InformedSearch.antWordAccessor,prevRow, nextCol) &&
                    !Location.isBoundary(InformedSearch.antWordAccessor,prevRow, nextCol);
        if(forceNeighbour) {
            forcedNeighbour = grid.get(prevRow, nextCol);
            neighbours.add(forcedNeighbour);
        }

        /**************************************************/
    }
    return neighbours;
} //******************************** End Code Block 7 **********************************************
```

```
//********************************* Start Code Block 8 ***********************************************

        //Add natural neighbour to the neighbours list
        private void  addNaturalNeighbour( List<LocationWrapper> neighbours, int row, int col){

                LocationWrapper neighbour= this.grid.get(row, col);
                if(neighbour != null &&
                  !Location.isBoundary(InformedSearch.antWordAccessor, neighbour) &&
                  !Location.isObstacle(InformedSearch.antWordAccessor, neighbour)) {
                        neighbours.add(neighbour);
                }
        }

//********************************* End Code Block 8 ***********************************************

//********************************* Start Code Block 9 ***********************************************

        //Returns the  Manhattan distance between the current location to the goal
        private int getHeuristic(LocationWrapper locationWrapper) {

                Location goalLocation = (Location)InformedSearch.destinationLocation;
                Location cellLocation = (Location)locationWrapper;

                int dx = Math.abs(goalLocation.row - cellLocation.row);
                int dy = Math.abs(goalLocation.col - cellLocation.col);

                return  dx + dy;
        }
//********************************* End Code Block 9 ***********************************************

//********************************* Start Code Block 10 ***********************************************
        //This is the list of neighbours to which the a move can be made to without being blocked
        private List<LocationWrapper> getOneStepNeighbours(LocationWrapper cell) {

                LinkedList<LocationWrapper> ret = new LinkedList<LocationWrapper>();
                for(LocationWrapper c : this.grid.getNeighbours(cell, this.numRows, this.numCols)) {
                        if(!Location.isObstacle(InformedSearch.antWordAccessor, ((Location)c).row, ((Location)c).col)){
                                ret.add(c);
                        }
                }
                return ret;
        }
//********************************* End Code Block 10 ***********************************************
}
```

```
//******************************** Start Code Block 11*********************************************

public class Location {
	int row, col;
	Location(int r, int c) {
		row = r;
		col = c;
	}
	Location(Location rc){
		row = rc.row;
		col = rc.col;
	}
	public String toString(){
		return "[" + row + " " + col + "]";
	}
	public boolean equals(Location rc){
		return row == rc.row && col == rc.col;
	}
}
//******************************** End Code Block 12*********************************************

//******************************** Start Code Block 13*********************************************

public class LocationWrapper extends  Location implements Comparable<LocationWrapper> {

	private double f, g;
	private int row, col;

	/* Constructs a new LocationWrapper that is initialized as an empty*/
	public LocationWrapper(double f, double g, int r, int c) {
		super(r, c);
		this.f = f;
		this.g = g;
	}

	/*Implements the Comparable interface  (necessary for the priority queue ordering)
	The ordering of location is based on its f score.
	@return Returns -1 if this location has a smaller F value than the other location, otherwise it returns 1.*/
	public int Comparable(LocationWrapper other) {
		if(this.f < other.f) {
			return -1;
		} else if(other.row == super.row && other.col == super.col){
			return 0;
		} else {
			return 1;
		}
	}

	/*Implements the Comparable interface  (necessary for the priority queue ordering)
	The ordering of location is based on its f score.
	@return Returns -1 if this location has a smaller F value than another location, otherwise it returns 1.*/
	public int compareTo(LocationWrapper arg0) {
		try {
		    return Comparable((LocationWrapper) arg0);
		} catch(Exception e) {
			return -1;
		}
	}

	/*Returns true when the locations have the same position, else false*/
	public boolean equals(LocationWrapper other) {
		Location otherLocation = (Location) other;
		return super.row == otherLocation.row && super.col == otherLocation.col;
	}

	/*Get the gValue for the current location*/
	public double getGValue() {
		return g;
	}


	/*Set the gValue for the current location*/
	public void setGValue(double val) {
		this.g = val;
	}
```

```
	/*Get the fValue for the current location*/
	public double getFValue() {
		return f;
	}

	/*Set the fValue for the current location*/
	public void setFValue(double val) {
		this.f = val;
	}

	/*Uniquely identify a location object  (necessary for the priority queue ordering)*/
	public int hashCode() {
		return (1000 * super.row) + super.col;
	}

}
//********************************* End Code Block 13**********************************************
```

References

[0] Blackwell T (2016) [Online], First coursework, https://learn.gold.ac.uk/mod/folder/view.php?id=379256 [Accessed 01 November 2016]

[1] Grinstead B. (2009) [Oneline] , A* Search Algorithm in JavaScript, Available at: *http://www.briangrinstead.com/blog/astar-search-algorithm-in-javascript*, [Accessed 01 November 2016]

[2] Harabor, D., and Grastien, A. (2011). *Online graph pruning for pathfinding on grid maps*, In AAAI-11. Ishida, T., and Korf, R. E. 1992. Moving Target Search. In IJCAI, 204–210.

[3] Patel A., Heuristics [Online], Available at: *http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html,* [Accessed 01 November 2016]

[4] Russek,S., Norvig,P. (2010) *Artificial Intelligence A Modern Approach*, 3rd ed, New Jersey, Prentice Hall Series in Artificial Intelligence, pp. 42-45, p80, p93-108

[5] Wikipedia [Online], Available at: A* search algorithm, Available at: *https://en.wikipedia.org/wiki/A*_search_algorithm*, [Accessed 02 January 2016]

[6] Shortest Path [Online], Available at: https://harablog.wordpress.com/2011/09/07/jump-point-search/, [Accessed 02 January 2016]

[7] Available at: https://www.cs.hmc.edu/~mmann/CS151/ [Accessed 09 December 2016]