Table of Contents

Path Planning by good-old fashioned AI Approach

Fereshteh Daheshmoghadam

fd2e11@ecs.soton.ac.uk

Abstract. This report conducts a comprehensive survey on classical AI search algorithms applied to a real world problem, path planning. We will start by describing the problem and then continue with explaining why classic search methods are suitable to adopt for this type of problem. Finally, we present the results and evaluate different taken approaches to better understand the effectiveness and usability of each.

1. Introduction

Path planning is a common problem, usage of which can be found in many fields of research, especially in computer science. It is an ideal candidate to apply classic AI to; since we need to to build an agent that moves around the environment to achieve the goal of arriving at a destination, ideally with lowest possible cost. This can be perfectly mapped to the general AI scenario where an agent acts upon an environment to achieve its goal. If we solve this problem, we can address domains such as computer games, industrial vehicles, robotics, computer networks and, in general, any system that would need path planning. Users can utilise our solution for satisfying the requirements for a various range of problems, from simple maze solving games to dynamic pathfinding robot.

The motivation is to adopt classical AI algorithms to address the general scenario and develop a package that can be adopted regardless of the methods used. To achieve this goal, we take the pure AI problem solving approach described in (Russell and Norvig, 1998), starting by encoding the problem in a form compatible with general search algorithm, then we continue with describing the algorithms and presenting the results.

2. Abstraction

The abstraction must be strong enough for search algorithms to be useful. For example, we would need a very complex hardware for a robot to implement turning half a degree clockwise. Therefore, an abstraction that needs turning half a degree will not be of much help. Another thing that must be taken to account is the domain. Since the domain is broad, the abstraction must be as generic as the domain is. A suitable abstraction which benefits from extensive mathematical support is the geometrical abstraction of complex shapes: Cube. Any complex three-dimensional environment can be formed by having enough small cubes attached to eachother. It is also quite intuitive since when you watch those cubes from an enough far distance, it will look like the original environment (this is the way in which graphics in computers, e.g. in games, work). In addition, we need an abstraction of our agent

in a physical (or virtual) space. Hence, we can think of the agent being in one particular cube at each time step.

While the cube model is reasonably acceptable, it is not necessary the best one to adopt when the agent is limited to move on a surface (if the agent was moving in a three-dimensional space, e.g. an airplane, we would need to stick with the cube abstraction). Therefore, we can increase the level of abstraction to 'Square', which will have following benefits:

- Search states can be defined as squares, which is also beneficial from usability point of view since it is not that hard to provide the real-world agent with a mechanism of transport between two neighbouring squares.
- Flexibility; we have a trade-off between smoother moves and search speed by changing the scale of squares.
- Abstraction of obstacles, by considering the squares that an obstacle is covering as 'not walkable'.
- Mapping from real-world to the search algorithm's inputs will be easy, since we can pass the search states to the algorithm by a small amount of computation (in linear time).

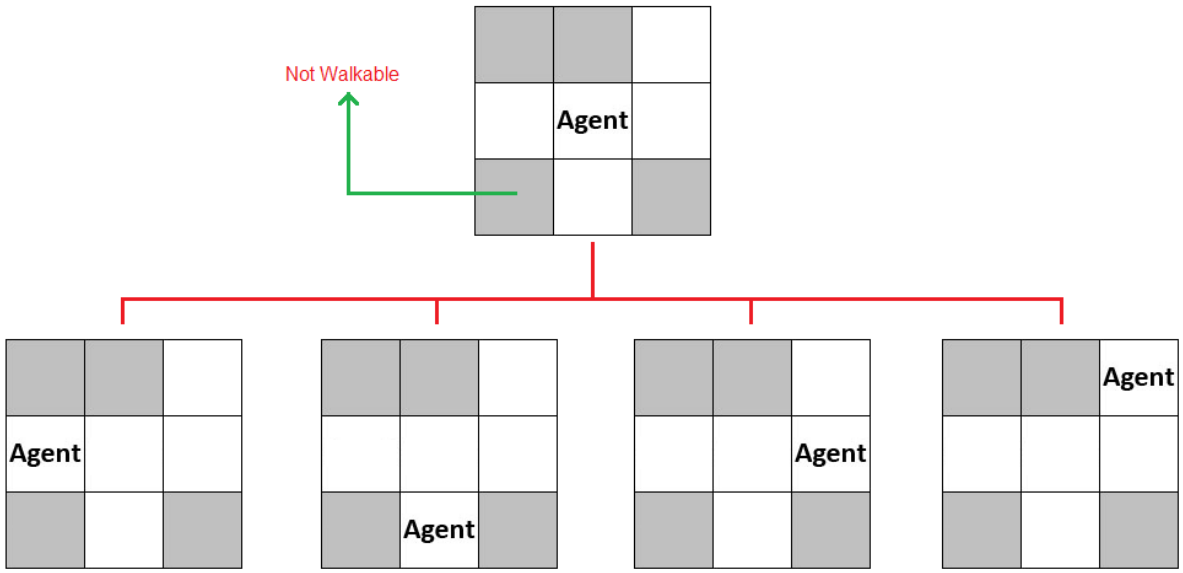

Figure 1: Generating successors of a state in Square abstraction

Using square abstraction relives us of having to deal with questions like "how in practice the agent is supposed to move in the environment" (e.g. turning on 'motor left' and turning off 'motor right'). Users should adapt our square abstraction and implement mechanisms for addressing questions as such. Instead, our system generates outputs like "move from the current square to the square in left".

3. Search

This section briefly reviews each of the search algorithms that have been used in this work. There are a number of elements that all these search algorithms share, which are as follows:

- Node, denoted as n: A structure used in search algorithms, which is the basic unit in forming search tree.

- Cost: A number for representing the cost of reaching one state from another. For the purpose of generating successors, the cost of a up, down, left and right moves are set as 10 whereas the cost of any diagonal move is 14, based on Pythagoras equation.
- Path Cost, denoted as g(n): The cost so far (in the search tree) for reaching node n.
- Heuristic Cost, denoted as h(n): The estimated cost of reaching the goal state from node n. We will discuss this later in section 3.2.
- Fringe: A data structure that holds the active nodes in a search tree. The way that fringes have been implemented in this work is that they are also responsible for prioritizing the order of expansion. Therefore, each search strategy will have its own fringe (e.g. BFS fringe, A* fringe, etc.)

This work tries to address different search algorithms, thereby, we have implemented the general graph search algorithm described in (Russell and Norvig, 1998) and different fringe structures for each search strategy. Figure 2 and Figure 3 show the pseudo-code of the graph search and the actual code from the implementation, respectively. We have tried to write the code in a way that matches the pseudo-code as much as possible.

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Figure 2: Graph Search, adapted from (Russell and Norvig, 1998)

```
function [solution, cost] = GraphSearch(problem, fringe)
    closed = State.empty;
    fringe.Insert(MakeNode(problem.InitialState));
    while(true)
        if(fringe.IsEmpty)
            solution = Node.empty; % Failure
            cost = 0;
            break;
        end
        node = fringe.RemoveFront();
        if(problem.GoalTest(node.State))
            solution = Solution(node);
            cost = node.PathCost;
            break;
        end
        if Contains(closed, node.State) == false
            closed = [node.State closed];
            fringe.InsertAll(Expand(node, problem));
        end
    end
end
```

Figure 3: Actual code from the implementation

3.1. Blind Search

A blind agent is one that does not have any sensors. Hence, it cannot perceive any knowledge from the environment. Any search algorithm that does not utilise the available knowledge about the environment (if there is any) can be confirmed as a blind search algorithm. Thereby, a blind search does not consider g(n) in the process of prioritizing the nodes in the fringe, so the agent will not be able to differentiate a non-diagonal move from a diagonal one.

The first blind search algorithm is Bread First Search. BFS starts by expanding the root node and continues by expanding all nodes in each depth before it continues to the next depth. This has been implemented by BFS fringe where nodes are added to the fringe in a first-in-first-out manner. Depth First Search (DFS) is the next algorithm that we implemented. DFS starts by expanding the root node and then expands the first node in each level untill it either finds the goal or reaches a node withot any children. If it reaches the bottom of the tree (i.e. no mor child to expand), it removes the the node in the last depth from the memory and continues with expanding the next node in previous depth. A DFS fringe has been adopted to apply DFS which prioterizes nodes in last-in-first-out order.

Figure 4 illustrates the results obtained by DFS and BFS. As can be drawn from the results, DFS has proven to be blind and is not anywhere near being optimal. This perfectly mathches with theory and the reason is that it does not have any idea of the goal's position. All it does is going far down in depth and obviously, the provided solution is not acceptable. BFS on the other hand shows a very reasonable result. However, it does not guaranty to give us the optimal solution since it does not take into accont different step-costs (a step-cos can be either 10 or 14, if we did not have different step costs, BFS would have been optimal), but it shows a huge progress over DFS with respect to the solution cost.

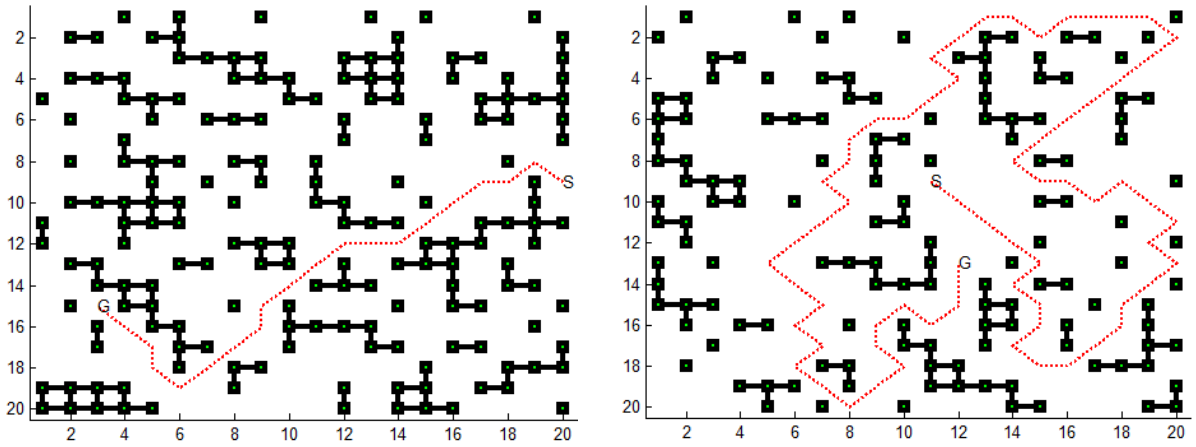

Figure 4: Results from Breadth First (left) and Depth First (right) search

3.2. Heuristic Search

The results from blind search algorithms did not seem to be promising, the reason is that they do not utilise the knowledge that we have about the environment. In contrast, heuristic search algorithms are designed to use any available knowledge. For this purpose, they are equipped with 'heuristic' mechanism that gives them some understanding about the goal that they try to achieve. In our model, the goal is a square somewhere in the grid, and we have the information on where it is (i.e. the row number and the column number). Hence, we can use this information to have an estimation of the remained cost for reaching the goal. This could

be easily calculated by 'optimized straight line distance heuristic', demonstrated in Figure 5. By simple linear calculations, we can identify how many diagonal and non-diagonal moves are needed to reach a square from another. This provides us with a 'relaxed' version of the original problem by not considering the 'not walkable' squares in between.

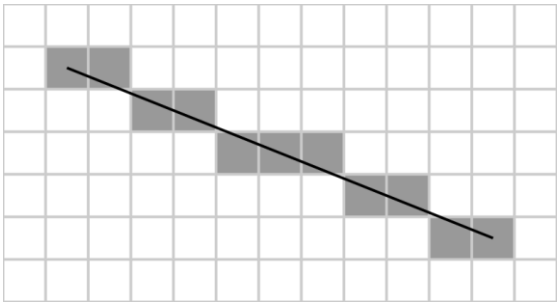

Figure 5: Optimized Straight Line Heuristic

Heuristic search algorithms expand nodes based on their f(n) cost, that is, a measure for evaluating how beneficial a node is with respect to achieving the goal as compared to the other nodes in the fringe (i.e. nodes are prioritised based on their f(n) values). Two search algorithms have been implemented in this category, Best First Search and A* search. For Best First, we have f(n) = h(n) which by intuition means "expand the node that looks more promising". A* however, considers "the cost so far as well as the estimated remaining cost", so we have f(n) = g(n) + h(n). Figure 6 shows the results obtained from Best First and A* search. Best First's solution is acceptable, but obviously not optimal. There are specific locations in the grid (marked in the figure) where Best First does not follow the rational path. The reason for this is that Best First decides on where to go merely based on the straight line heuristic values. This forces the agent to move towards the straight line as far as it can and may lead it to follow an irrational direction. A* shows the best results so far, and it also guaranties to give us the optimal solution.

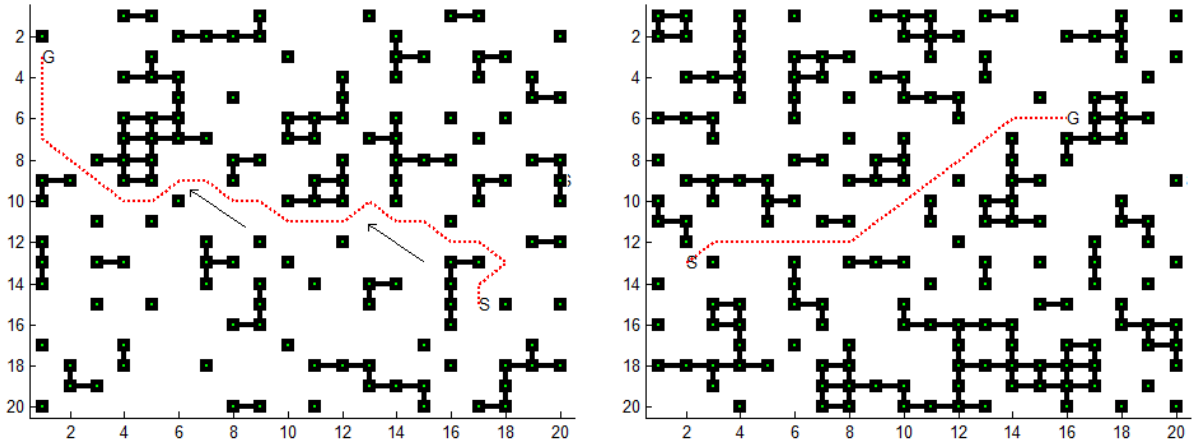

Figure 6: Results from Best First (left) and A* (right) search

The last algorithm is Bidirectional Search, which is not exactly a new search algorithm, but instead, a method of conducting two searches in parallel. It does that by having one algorithm to search from the start state to the goal state and another for the other way around. An interesting finding by implementing this search was that it does not guaranty to give you the optimal solution when there are different step costs. This has been demonstrated in Figure 7.

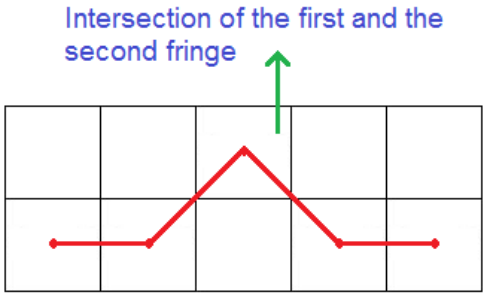

Figure 7: Sample case where Bidirectional Search does not give us the optimal solution

4. Quantitative Evaluation

We have already evaluated the quality of the result for each search in section 3. In this section, we perform a quantitative evaluation on the results. We do so by presenting the number of expanded nodes for each of the reviewed examples (Table 1). By using graph search as the basis of this work, the worst possible search strategy will expand $N * M$ nodes (where N is the number of rows and M is the number of columns in the grid), that is when the search expands all of the states in the problem and all of the squares are walkable.

Reference / Parameter	**DFS** Figure 4 Right	**BFS** Figure 4 Left	**Best First** Figure 6 Left	**A*** Figure 7 Right	**BFS** Sample Run	**Bidirectional** Same Sample Run
Solution Cost	885	246	247	168	114	118
Expanded Nodes	99	215	26	23	161	70

Table 1: Quantitative comparison of previous examples

For a better comparison of different search algorithms, we generated the graphs in Figure 8. Since the grids are generated randomly, one experiment for each value of N was not sufficient to provide a true measure for comparison. Therefore, for each value of N, we conducted 30*N experiment. What you see below is the averaged out values.

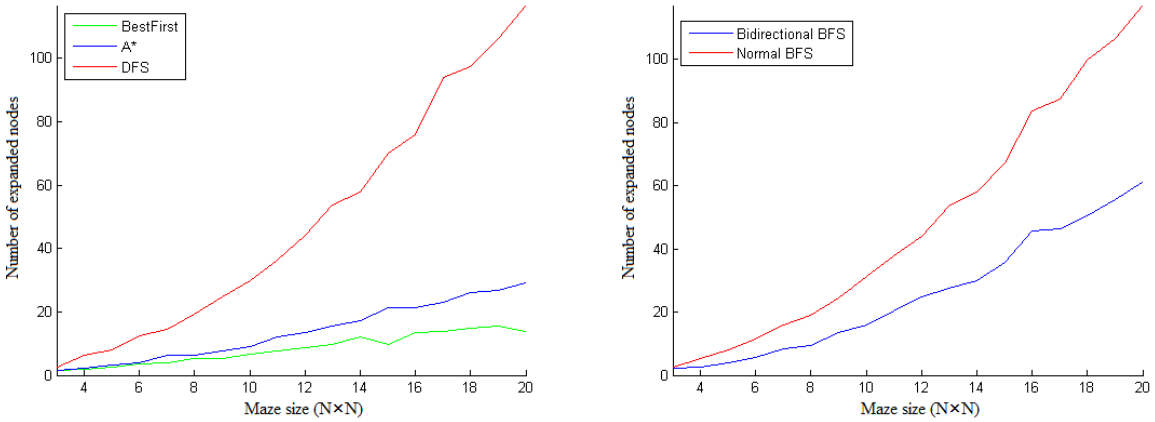

Figure 8: Quantitative Evaluation

Reference

RUSSELL, S. J. & NORVIG, P. 1998. *Artificial intelligence: A Modern Approach*, Prentice hall.

5. Appendix

You can find the code from the implementation in this section. Please note that the code has been only tested with MATLAB R2011b (the code might not work in earlier versions since it contains m files which need object orientation support). For the ease of assessment, I have provided a number of m files with the names such as 'ExampleAStar.m'. Basically, every file with name starting by 'Example' is a function that generates a random maze and then solves it using a specific strategy (you will need to pass two arguments to this function, the number of rows and the number of columns; Figure 4 and Figure 6 have been generated by calls to these functions). There are some scripts for evaluation, names of which starts by 'Evaluate'. You can open these files, change the parameters and simply call them; however, the results are already combined in Figure 8.

Word Count: 2014

5.1. GraphSearch.m

```
function [solution, cost] = GraphSearch(problem, fringe)
    closed = State.empty;
    fringe.Insert(MakeNode(problem.InitialState));
    while(true)
        if(fringe.IsEmpty)
            solution = Node.empty; % Failure
            cost = 0;
            break;
        end
        node = fringe.RemoveFront();
        if(problem.GoalTest(node.State))
            solution = Solution(node);
            cost = node.PathCost;
            break;
        end
        if Contains(closed, node.State) == false
            closed = [node.State closed];
            fringe.InsertAll(Expand(node, problem));
        end
    end
end

function solution = Solution(node)
    solution = node;
    currentNode = node;
    rootReached = false;

    while(rootReached == false)
        if isempty(currentNode.ParentNode)
            rootReached = true;
        end
        currentNode = currentNode.ParentNode;
        solution = [currentNode solution];
    end

end
```

5.2. Expand.m

```
function successors = Expand(node, problem)
    successors = Node.empty;
    [actions, results] = problem.SuccessorFunction(node.State);
    for i=1:length(results)
       s = Node;
       s.ParentNode = node;
       s.Action = actions(i);
       s.State = results(i);
       s.PathCost = node.PathCost + StepCost(actions(i));
       s.Depth = node.Depth + 1;
       successors(i) = s;
    end
end

function cost = StepCost(action)
   if(action == 4 || action == 6 || action == 8 || action == 2)
       % left, right, up, down (based on keyboard's numeric keypad)
       cost = 10;
   else % diagonals
       cost = 14;
   end
end
```

5.3. State.m

```
classdef State < handle

    properties
        Blocked = false;
        Row
        Column
    end

    methods
        function state = State(row, column, blocked)
            if nargin >= 2
                state.Row = row;
                state.Column = column;
            end
            if nargin == 3
                state.Blocked = blocked;
            end
        end

        function result = eq(state1, state2)
            if(state1.Row == state2.Row && state1.Column == state2.Column)
                result = true;
            else
                result = false;
            end
        end
    end

end
```

5.4. Node.m

```
classdef Node < handle

    properties
        Action
        State
        Depth
        ParentNode
        PathCost % g(n)
        EstimatedCost % f(n)
    end

    methods
        function result = eq(node1,node2)
            if node1.State == node2.State
                result = true;
            else
                result = false;
            end
        end
    end

end
```

5.5. MakeNode.m

```
function node = MakeNode(state)
   node = Node;
   node.State = state;
   node.ParentNode = Node.empty;
   node.Depth = 1;
   node.PathCost = 0;
end
```

5.6. SLHeuristic.m

```
classdef SLHeuristic < Heuristic

    methods
        function hn = CalculateHeuristic(heuristic, node)
            rowsInBetween = abs(node.State.Row-heuristic.GoalState.Row)+1;
            columnsInBetween = abs(node.State.Column-
heuristic.GoalState.Column)+1;

            diameterLength = min(rowsInBetween,columnsInBetween);
            if diameterLength > 1
                hn = (diameterLength - 1) * 14 + abs(rowsInBetween-
columnsInBetween) * 10;
            else
                hn = abs(rowsInBetween-columnsInBetween) * 10;
            end
        end
    end

end
```

5.7. Heuristic.m

```
classdef Heuristic < handle

    properties
        GoalState
    end

    methods  (Abstract=true)
        hn = CalculateHeuristic(node)
    end

end
```

5.8. Fringe.m

```
classdef Fringe < handle

    properties
        Nodes = Node.empty;
        NumberOfExpandedNodes = 0;
    end

    methods
        function node = RemoveFront(fringe)
            node = fringe.Nodes(1);
            fringe.Nodes = fringe.Nodes(2:length(fringe.Nodes));
        end

        function InsertAll(fringe, nodes)
            for node = nodes
                fringe.Insert(node);
            end
            fringe.NumberOfExpandedNodes = fringe.NumberOfExpandedNodes+1;
        end

        function result = IsEmpty(fringe)
            if isempty(fringe.Nodes)
                result = true;
            else
                result = false;
            end
        end
    end

    methods (Abstract=true)
        Insert(node)
    end

end
```

5.9. DFSFringe.m

```
classdef DFSFringe < Fringe

    methods
        function Insert(fringe, node)
            fringe.Nodes = [node fringe.Nodes]; % LIFO
        end
    end

end
```

5.10. BFSFringe.m

```
classdef BFSFringe < Fringe

    methods
        function Insert(fringe, node)
            fringe.Nodes(length(fringe.Nodes) + 1) = node; % FIFO
        end
    end

end
```

5.11. BestFirstFringe.m

```
classdef BestFirstFringe < HeuristicFringe

    methods
        function fn = EstimateCost(fringe, node)
            fn = fringe.HeuristicUnit.CalculateHeuristic(node); % f(n)=h(n)
        end
    end

end
```

5.12. AStarFringe.m

```
classdef AStarFringe < HeuristicFringe

    methods
        function fn = EstimateCost(fringe, node)
            fn = node.PathCost +
fringe.HeuristicUnit.CalculateHeuristic(node); % f(n)=g(n)+h(n)
        end
    end

end
```

5.13. HeuristicFringe.m

```
classdef HeuristicFringe < Fringe

    properties
        HeuristicUnit
    end

    methods (Abstract = true)
        fn = EstimateCost(node)
    end

    methods
        function Set(fringe,problem,heuristic)
            heuristic.GoalState = problem.GoalState;
            fringe.HeuristicUnit = heuristic;
        end

        function Insert(fringe, node)
            node.EstimatedCost = fringe.EstimateCost(node);
            inserted = false;
            for i=1:length(fringe.Nodes)
                if fringe.Nodes(i).EstimatedCost > node.EstimatedCost
                    InsertAt(fringe,i,node);
                    inserted = true;
                    break;
                end
            end
            if inserted == false
                InsertAtLast(fringe,node);
            end
        end
    end

end

function InsertAt(fringe,index,node)

    numberOfNodes = length(fringe.Nodes);
    newNodes(1,1:numberOfNodes+1) = Node;
    for i=1:index-1
        newNodes(i) = fringe.Nodes(i);
    end
    newNodes(index) = node;
    for i=index+1:numberOfNodes+1
        newNodes(i) = fringe.Nodes(i-1);
    end
    fringe.Nodes = newNodes;
end

function InsertAtLast(fringe,node)

    numberOfNodes = length(fringe.Nodes);
    if numberOfNodes == 0
        fringe.Nodes = node;
    else
        fringe.Nodes(numberOfNodes+1) = node;
    end

end
```

5.14. Contains.m

```
function result = Contains(collection, item)

    result = false;
    for element = collection
        if item == element
            result = true;
            break;
        end
    end
end
```

5.15. BidirectionalSearch.m

```
function [solution, cost] = BidirectionalSearch(problem, fringe1, fringe2)

    closed = State.empty;
    fringe1.Insert(MakeNode(problem.InitialState));
    fringe2.Insert(MakeNode(problem.GoalState));

    while(true)
        if fringe1.IsEmpty || fringe2.IsEmpty
            solution = Node.empty; % Failure
            cost = 0;
            break;
        end

        [intersectingNodeFromFring1, intersectingNodeFromFring2] =
FindIntersection(fringe1,fringe2);
        if isempty(intersectingNodeFromFring1) == false
            [solution, cost] = Solution(intersectingNodeFromFring1,
intersectingNodeFromFring2);
            break;
        end

        node1 = fringe1.RemoveFront();
        node2 = fringe2.RemoveFront();

        if Contains(closed, node1.State) == false
            closed = [node1.State closed];
            fringe1.InsertAll(Expand(node1, problem));
        end
        if Contains(closed, node2.State) == false
            closed = [node2.State closed];
            fringe2.InsertAll(Expand(node2, problem));
        end
    end
end

function [intersectingNodeFromFring1, intersectingNodeFromFring2] =
FindIntersection(fringe1,fringe2)

    intersectingNodeFromFring1 = Node.empty;
    intersectingNodeFromFring2 = Node.empty;
    for node1 = fringe1.Nodes
        for node2 = fringe2.Nodes
            if node1 == node2
                intersectingNodeFromFring1 = node1;
                intersectingNodeFromFring2 = node2;
```

```
                break;
            end
        end
    end

end

function [solution, cost] = Solution(node1,node2)

    solution = node1;
    currentNode = node1;
    rootReached = false;

    while(rootReached == false)
        currentNode = currentNode.ParentNode;
        solution = [currentNode solution];
        if isempty(currentNode) || isempty(currentNode.ParentNode)
            rootReached = true;
        end
    end

    currentNode = node2;
    rootReached = false;
    while(rootReached == false)
        currentNode = currentNode.ParentNode;
        solution = [solution currentNode];
        if isempty(currentNode) || isempty(currentNode.ParentNode)
            rootReached = true;
        end
    end

    cost = node1.PathCost + node2.PathCost;
end
```

5.16. Problem.m

```
classdef Problem < handle

    properties
        InitialState
        GoalState
        States = State.empty;
        ArrayVersion
    end

    methods
        function problem = Problem(numberOfRows, numberOfColumns)
            problem.States(1:numberOfRows, 1: numberOfColumns) = State;
            for i = 1:numberOfRows
                for j = 1:numberOfColumns
                    if rand < 0.4
                        blocked = true;
                    else
                        blocked = false;
                    end
                    problem.States(i,j) = State(i,j,blocked);
                    problem.ArrayVersion(i,j) = blocked;
                end
            end
```

```
            index = round(rand*(i*j-1))+1;
            problem.InitialState = problem.States(index);
            problem.InitialState.Blocked = false;
            problem.ArrayVersion(index) = 0;
            index = round(rand*(i*j-1))+1;
            problem.GoalState = problem.States(index);
            problem.GoalState.Blocked = false;
            problem.ArrayVersion(index) = 0;
        end

        function result = GoalTest(problem, state)
            if state == problem.GoalState
                result = true;
            else
                result = false;
            end
        end

        function [actions, results] = SuccessorFunction(problem, state)
            [numberOfRows, numberOfColumns] = size(problem.States);
            row = state.Row;
            column = state.Column;
            actions = double.empty;
            results = State.empty;
            index = 0;

            if(column>1 && problem.States(row,column-1).Blocked == false)
                index = index + 1;
                actions(index) = 4; % left (according to keyboard's numeric
keypad!)
                results(index) = problem.States(row,column-1);
            end

            if(column<numberOfColumns &&
problem.States(row,column+1).Blocked == false)
                index = index + 1;
                actions(index) = 6; % right
                results(index) = problem.States(row,column+1);
            end

            if(row>1 && problem.States(row-1,column).Blocked == false)
                index = index + 1;
                actions(index) = 8; % up
                results(index) = problem.States(row-1,column);
            end

            if(row<numberOfRows && problem.States(row+1,column).Blocked ==
false)
                index = index + 1;
                actions(index) = 2; % down
                results(index) = problem.States(row+1,column);
            end

            if(column>1 && row>1 && problem.States(row-1,column-1).Blocked
== false)
                index = index + 1;
                actions(index) = 7; % upper left
                results(index) = problem.States(row-1,column-1);
            end
```

```
            if(column<numberOfColumns && row>1 && problem.States(row-
1,column+1).Blocked == false)
                index = index + 1;
                actions(index) = 9; % upperRight
                results(index) = problem.States(row-1,column+1);
            end

            if(column>1 && row<numberOfRows && problem.States(row+1,column-
1).Blocked == false)
                index = index + 1;
                actions(index) = 1; % bottomLeft
                results(index) = problem.States(row+1,column-1);
            end

            if(column<numberOfColumns && row<numberOfRows &&
problem.States(row+1,column+1).Blocked == false)
                index = index + 1;
                actions(index) = 3; % bottomRight
                results(index) = problem.States(row+1,column+1);
            end

        end
    end
end
```

5.17. SolveMaze.m

```
function [cost , numberOfExpandedNodes] = SolveMaze(problem, drawMaze,
fringe, heuristic)

    if isa(fringe,'HeuristicFringe')
        fringe.Set(problem,heuristic);
    end

    [solution, cost] = GraphSearch(problem,fringe);
    numberOfExpandedNodes = fringe.NumberOfExpandedNodes;

    if drawMaze
        DrawMaze(problem, solution);
    end
end
```

5.18. SolveMazeBidirectional.m

```
function [cost , numberOfExpandedNodes] = SolveMazeBidirectional(problem,
drawMaze, fringe1, fringe2, heuristic1, heuristic2)

    if isa(fringe1,'HeuristicFringe')
        fringe1.Set(problem,heuristic1);
    end
    if isa(fringe2,'HeuristicFringe')
        if nargin == 5
            fringe2.Set(problem,heuristic1);
        else
            fringe2.Set(problem,heuristic2);
        end
    end

    [solution, cost] = BidirectionalSearch(problem, fringe1, fringe2);
    numberOfExpandedNodes = fringe1.NumberOfExpandedNodes +
fringe2.NumberOfExpandedNodes;

    if drawMaze
        DrawMaze(problem, solution);
    end
end
```

5.19. DrawMaze.m

```
function DrawMaze(problem, solution)

    arrayVersion = problem.ArrayVersion;
    [numberOfRows, numberOfColumns] = size(problem.States);

    figure;
    hold on;
    axis ij
    axis([0.5 numberOfColumns+0.5 0.5 numberOfRows+0.5]);

    for i=1:numberOfRows
        x=zeros(0);
        y=zeros(0);
        for j=1:numberOfColumns
            if(arrayVersion(i,j) == 1)
                x = [x j];
                y = [y i];
            else
                plot(x,y,'-ks','LineWidth',3,...
                    'MarkerEdgeColor','k',...
                    'MarkerFaceColor','g',...
                    'MarkerSize',5)
                x=zeros(0);
                y=zeros(0);
            end
        end
        plot(x,y,'-ks','LineWidth',3,...
                    'MarkerEdgeColor','k',...
                    'MarkerFaceColor','g',...
                    'MarkerSize',5)
    end
```

```
    for i=1:numberOfColumns
        x=zeros(0);
        y=zeros(0);
        for j=1:numberOfRows
            if(arrayVersion(j,i) == 1)
                x = [x i];
                y = [y j];
            else
                plot(x,y,'-ks','LineWidth',3,...
                    'MarkerEdgeColor','k',...
                    'MarkerFaceColor','g',...
                    'MarkerSize',5)
                x=zeros(0);
                y=zeros(0);
            end
        end
        plot(x,y,'-ks','LineWidth',3,...
                    'MarkerEdgeColor','k',...
                    'MarkerFaceColor','g',...
                    'MarkerSize',5)
    end

    if isempty(solution) == false
        [x, y] = ConvertToArray(solution);
        plot(x,y,':r', 'LineWidth',2);
    else
        text(numberOfColumns/2,0,'FAILURE');
    end
    text(problem.InitialState.Column, problem.InitialState.Row, 'S');
    text(problem.GoalState.Column, problem.GoalState.Row, 'G');
    hold off
end

function [x, y] = ConvertToArray(path)

    x=zeros(0);
    y=zeros(0);
    for node = path
        x = [x node.State.Column];
        y = [y node.State.Row];
    end

end
```

5.20. ExampleAStar.m

```
function ExampleAStar(numberOfRows, numberOfColumns)
    disp('A* Search, Straight Line Distance Heuristic');
    problem = Problem(numberOfRows, numberOfColumns);
    close all;
    [cost , numberOfExpandedNodes] = SolveMaze(problem, true, AStarFringe,
SLHeuristic);
    disp('Solution cost:');
    disp(cost);
    disp('Number of expanded nodes:');
    disp(numberOfExpandedNodes);
end
```

5.21. ExampleBestFirst.m

```
function ExampleBestFirst(numberOfRows, numberOfColumns)
    disp('Best First Search, Straight Line Distance Heuristic');
    problem = Problem(numberOfRows, numberOfColumns);
    close all;
    [cost , numberOfExpandedNodes] = SolveMaze(problem, true,
BestFirstFringe, SLHeuristic);
    disp('Solution cost:');
    disp(cost);
    disp('Number of expanded nodes:');
    disp(numberOfExpandedNodes);
end
```

5.22. ExampleBFS.m

```
function ExampleBFS(numberOfRows, numberOfColumns)
    disp('Breadth First Search');
    problem = Problem(numberOfRows, numberOfColumns);
    close all;
    [cost , numberOfExpandedNodes] = SolveMaze(problem, true, BFSFringe);
    disp('Solution cost:');
    disp(cost);
    disp('Number of expanded nodes:');
    disp(numberOfExpandedNodes);
end
```

5.23. ExampleDFS.m

```
function ExampleDFS(numberOfRows, numberOfColumns)
    disp('Depth First Search');
    problem = Problem(numberOfRows, numberOfColumns);
    close all;
    [cost , numberOfExpandedNodes] = SolveMaze(problem, true, DFSFringe);
    disp('Solution cost:');
    disp(cost);
    disp('Number of expanded nodes:');
    disp(numberOfExpandedNodes);
end
```

5.24. ExampleNormalVsBidirectional.m

```
function ExampleNormalVsBidirectional(numberOfRows, numberOfColumns)

    problem = Problem(numberOfRows, numberOfColumns);
    close all;
    [cost , numberOfExpandedNodes] = SolveMaze(problem, true, BFSFringe);
    disp('Solution cost with normal BFS:');
    disp(cost);
    disp('Number of expanded nodes with normal BFS');
    disp(numberOfExpandedNodes);
    [cost , numberOfExpandedNodes] = SolveMazeBidirectional(problem, true,
BFSFringe,BFSFringe);
    disp('Solution cost with bidirectional BFS:');
    disp(cost);
    disp('Number of expanded nodes with bidirectional BFS:');
    disp(numberOfExpandedNodes);

end
```

5.25. EvaluateAStar.m

```
upperBound = 10;
lowerBound = 3;
numberOfExperiments = 10;
x=1:upperBound;
y=zeros(1,upperBound);
for i=lowerBound:upperBound
    for j=1:i*numberOfExperiments
        problem = Problem(i, i);
        [cost , numberOfExpandedNodes] = SolveMaze(problem, false,
AStarFringe, SLHeuristic);
        y(i) = y(i) + numberOfExpandedNodes;
    end
    y(i) = y(i)/(i*numberOfExperiments);
end

figure('name','A* Evaluation');
hold on
axis([lowerBound upperBound 0 max(y)]);
plot(x,y,'b');
clear;
```

5.26. EvaluateBestFirst.m

```
upperBound = 10;
lowerBound = 3;
numberOfExperiments = 10;
x=1:upperBound;
y=zeros(1,upperBound);
for i=lowerBound:upperBound
    for j=1:i*numberOfExperiments
        problem = Problem(i, i);
        [cost , numberOfExpandedNodes] = SolveMaze(problem, false,
BestFirstFringe, SLHeuristic);
        y(i) = y(i) + numberOfExpandedNodes;
    end
    y(i) = y(i)/(i*numberOfExperiments);
end

figure('name','Best First Evaluation');
hold on
axis([lowerBound upperBound 0 max(y)]);
plot(x,y,'g');
clear;
```

5.27. EvaluateDFS.m

```
upperBound = 10;
lowerBound = 3;
numberOfExperiments = 10;
x=1:upperBound;
y=zeros(1,upperBound);
for i=lowerBound:upperBound
    for j=1:i*numberOfExperiments
        problem = Problem(i, i);
        [cost , numberOfExpandedNodes] = SolveMaze(problem, false,
DFSFringe, SLHeuristic);
        y(i) = y(i) + numberOfExpandedNodes;
    end
```

```
    y(i) = y(i)/(i*numberOfExperiments);
end

figure('name','DFS Evaluation')
hold on
axis([lowerBound upperBound 0 max(y)]);
plot(x,y,'r');
clear;
```

5.28. EvaluateNormalVsBidirectionalBFS.m

```
upperBound = 10;
lowerBound = 3;
numberOfExperiments = 5;
x=1:upperBound;
y1=zeros(1,upperBound);
y2=zeros(1,upperBound);
for i=lowerBound:upperBound
    for j=1:i*numberOfExperiments
        problem = Problem(i, i);
        [cost , numberOfExpandedNodes] = SolveMaze(problem, false,
BFSFringe);
        y1(i) = y1(i) + numberOfExpandedNodes;
        [cost , numberOfExpandedNodes] = SolveMazeBidirectional(problem,
false, BFSFringe, BFSFringe);
        y2(i) = y2(i) + numberOfExpandedNodes;
    end
    y1(i) = y1(i)/(i*numberOfExperiments);
    y2(i) = y2(i)/(i*numberOfExperiments);
end

figure('name','Normal vs Bidirectional BFS');
hold on
axis([lowerBound upperBound 0 max(max(y1),max(y2))]);
plot(x,y1,'r');
plot(x,y2,'b');
clear;
```