CARDIFF
UNIVERSITY

PRIFYSGOL
CAERDYDD

Student Name:	Geraint Harries
Number:	1100682
Module code:	CM3108
Module Title:	Computational Intelligence
Coursework Title:	Computational Intelligence Coursework
Lecture:	C. Mumford & S. Schockaert
Hours spent on exercise:	40
Special Provision:	n/a

Task 1.

See enclosed code for task 1.

Task 2.

Note

Along with this conclusion, there are comments within the code explaining the Bidirectional algorithm.

I did not use the same maze instance when testing the two searching algorithms. I instead ran more than 50 iterations of the algorithm on each turn. Instead of 50 I ran 2000. This gives an equally accurate average value as using the same maze fewer times.

Conclusion

From appendix 10, you can see that A* search is more efficient (in terms of nodes expanded) in mazes with fewer walls removed. Bidirectional is more efficient in mazes with more walls removed.

If we look at appendix 9, we see the mazes graphically compared against each other over the entire sample space (Mazes size 10 to 150). You will note there is no significant difference between the two. However, this is only on a maze with 10 walls removed.

The number of walls present affects the performance of the algorithms. If we look at appendix 10, we see that when the number of walls deleted is 10 and the maze size is 150, A* search is more efficient. However when the number of walls is decreased (or the walls deleted value increases) Bidirectional search becomes more efficient than A* search. If we Bidirectional compare when walls value is 10 and walls value is 1000, Bidirectional is more efficient at 1000.

The size of the maze affects the performance. If we look at appendix 10, we see that the nodes expanded for A* when the maze is size 10 is always less than Bidirectional. This shows that the performance is somewhat dependant on maze size. We notice however, that when the maze is bigger that Bidirectional does not always have less nodes expanded than A*. This means there is another factor in the performance of these algorithms.

Combining these two dependancies, we can conclude; when the maze is small, A* is best; when the maze is large but has many walls, A* is best and when the maze is large and with few walls Bidirectional is best.

With mazes with many walls, there are less potential shortest paths whereas with few walls there are several ways to explore. Using A* search, you have only the furthest frontier to choose and therefor in a maze with many walls, it is able to utilise the furthest frontier to find the direct path. The computational time it takes to run is O(n^d) where d is distance from goal. Whereas with Bidirectional, when there are fewer walls, there are more directs routes which means more frontiers. With more frontiers, the frontiers are more likely to meet. The computational runtime of this is O($n^{d/2}$). This explains why Bidirectional is more efficient with more walls.

Task 3

Overview

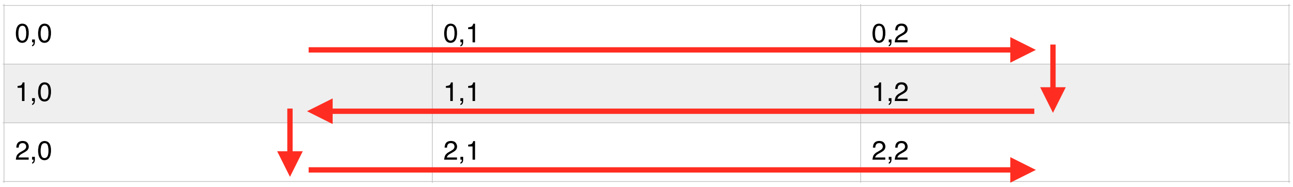
My genetic algorithm seeks to find a global optimal solution (or as near as you can with a genetic algorithm). It has a set of fitness criteria (whether a gene contains legal moves, if it hits the goal node, etc) which it the population is iterated through to evaluate. The population is then relatively ranked. The top 50% interbreed using single point crossover and the bottom 50% are randomly mutated. I have written about this algorithm in more detail below.

Representation

An array of x,y pairs which represent x, y values of the maze co-ordinates.

e.g.

The path in the below maze,



0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

would be represented as

([0,0], [0,1], [0,2], [1,2], [1,1], [1,0], [2,0], [2,1], [2,2])

The fitness would also need to be stored with each gene. You could create an object and store the fitness value within that, however, I have decided to store the value as my first array value

e.g. the above path would be represented as

(100, [0,0], [0,1], [0,2], [1,2], [1,1], [1,0], [2,0], [2,1], [2,2]) where 100 is the fitness value.

This saves coding complexity.

Population Size and Breeding proportion

The population is initially created by assigning random path values to a large population. It has to be a large population to secure that the randomness will throw out some “good” sequences and relatively fit genes.

The pseudocode:

```
createPopulation(populationSize)
    for i = 0 to populationSize
        geneLength = rand(ManhattenDistance, maze.size * maze.size -1)
        for j = 0 to genelength
            gene[i] = [random(0, maze.size), random(0, maze.size)]
        population.add(gene)
    return population
```

Population will remain the same, as a better gene is created an old worse one will die. When 2 child genes are generated from two parents, the un-fitest two of the four are deleted. This will gradually increase the fitness of the population but maintain it’s size. Adding to the population without deleting old genes will cause too much of a population increase and slow the process down.

Crossover/ Mutation

There will be two kinds of crossover/ mutation, a single point crossover for the fittest 50% of the population and random mutation for the un-fittest 50%. This is because the bottom 50% are unfit and need to be changed randomly.

The reason for the crossover is due to the dependancy on the element co-ordinate$_i$ in a gene on co-ordinate$_{i-1}$ i.e for the gene

[0,0], [0,1], [0,2], …

element [0,2] is only valid as the previous element is [0,1]. Therefor if you randomly chopped up each gene it would not be a valid gene.

If you split it at a matching number then you do not affect the validity of the gene.

e.g. for the two genes below

[0,0], [0,1], [0,2], [1,2], [1,1], [1,0], [2,0], [2,1], [2,2]

and

[0,0], [1,0], [2,0], [2,1], [1,1], [0,1], [0,2], [1,2], [2,2]

if you arbitrarily split the gene and swapped them over you wouldn't get valid genes.

e.g. if a split and swapped the genes where the green line is,

[0,0], [0,1], [0,2], [1,2], [1,1], [1,0], [2,0], [2,1], [2,2]

[0,0], [1,0], [2,0], [2,1], [1,1], [0,1], [0,2], [1,2], [2,2]

I would get

[0,0], [0,1], [0,2], [2,1], [1,1], [0,1], [0,2], [1,2], [2,2]

[0,0], [1,0], [2,0], [1,2], [1,1], [1,0], [2,0], [2,1], [2,2]

are not valid paths.

However, if you swapped them over at a match point (a point where the values are the same), they would create valid paths. (In the below example, I'm splitting the genes at the point [1,0]

[0,0], [0,1], [0,2], [1,2], [1,1], [1,0], [2,0], [2,1], [2,2]

[0,0], [1,0], [2,0], [2,1], [1,1], [0,1], [0,2], [1,2], [2,2]

This gives us

[0,0], [1,0], [2,0], [2,1], [2,2]

[0,0], [0,1], [0,2], [1,2], [1,1], [1,0], [2,0], [2,1], [1,1], [0,1], [0,2], [1,2], [2,2]

two valid paths.

The Pseudocode:

```
crossoverSinglePoint(x1, x2)
    for i = 1 to x1.length
        for j = x2.length to 1
            if x1[i] == x2[j]
                x3 = x1[0 - i] + x2[j - x2.length]
                x4 = x2[0 - i] + x1[j - x2.length]

    return x3, x4
```

To maintain population, each of the parent genes and child genes are assessed and given a fitness, then the 2 least fit are discarded.

This works for fit and healthy genes however, for unfit genes this will not work. For unfit genes, it is better to randomise values arbitrarily in the hope they will increase in fitness.

e.g. the below genes are unfit genes and rather than trying to cross them over, are being randomly mutated

[0,0], [2,2], [2,1], [1,0], [1,2], [1,2], [1,1], [0,2]

[0,0], [1,0], [2,1], [2,2], [1,2], [1,1], [1,1], [1,2]

Every other value of the gene is changed to add variety to the overall gene pool.

The pseudocode:

```
randomiseGene(x)
    for i = 0 to x.length
        randomGene  = [random(0, maze.size()), random(0, maze.size())]
        x[i] = randomGene
        i+= 2
    return x
```

Having the bottom 50% change genes regularly allows the chance of global optima to increase. The top 50% are converging on a similar gene, this gene could be a local optima. Occasionally, if the bottom 50% generates a fit gene which is merits to be in the top 50% it will add more variety to the gene pool and potentially add global optima genes. This will allow the top 50% to flourish and bottom 50% to support the top 50%.

Fitness

My fitness function for this algorithm would be set out similar to below.

The pseudocode:

```
SetFitnessOnGene(x)
    x = validHamming(x)
    x = checkLoops(x)
    x = meetsGoal(x)
    x = validStart(x)
    x = checkIllegalSquares(x)
    x = checkLegalLength(x)
    x = rankLength(x)
```

After a series of methods are run, each editing a gene's fitness, the true fitness value is known. Below is a breakdown of the fitness criteria and pseudocode for each.

Hamming Distance of 1

For a series of co-ordinates within a gene to be valid, it must have a hamming distance of 1. As each step in the complete path can only move one square at a time, it must only have one co-ordinate with a difference of 1.

e.g.

[0,0], [0,1], [0,2], [1,2], [1,1], [1,0], [2,0], [2,1], [2,2]

all only have a difference of 1 between then, and consequently each makes a step of only one to the next space

[0,1], [2,2], [1,0], [2,1], [0,0], [1,1]

have a hamming distance of greater than 1, which means they move more than one space each turn, this is invalid.

The pseudocode:

```
validHamming(x)
    for i = 0 to x.length
        if x[i+1] == null
            fitness++
        else if difference between x[i] and x[i+1] > 1
            fitness—
```

Avoiding Loops

To avoid a looping gene, e.g.

[0,0], [0,1], [1,1], [1,0], [0,0], [0,1], [1,1], [0,1]…

one of the fitness criteria need to evaluate whether a gene is looping. It will do so but iterating through itself and checking for any repeated co-ordinates

The pseudocode:

```
checkLoops(x)
    for i = 1 to x.length
        for j = i to x.length
            if x[j] == x[i]
                fitness--
```

Reaching Goal

If the final co-ordinate of the gene is the goal node then the fitness can be increased as it's achieving getting to the goal e.g.

if the goal node = [2,2]

a fit path would be

[0,0], [0,1], [0,2], [1,2], [2,2]

as the final node is the goal node, however if the final node is not the goal node then it would be considered unfit.

The pseudocode:

```
meetsGoal(x)
    if x[x.length] == maze.goalNode
        fitness++
    else
        fitness—
```

Valid Start

Similarly to the reaching goal criteria, the genes need to be checked to see if they start in the right place.

e.g.

[0,0], [0,1], [0,1] …

starts on a the “start node” (assuming the start node is [0,0])

[1,0], [1,1], [2,1] …

does not

The pseudocode:

```
validStart(x)
    if x[0] == maze.startNode
        fitness++
    else
        fitness—
```

Legal Moves

In the above examples, we are using totally open mazes, mazes which have no illegal squares (squares which are not valid to move to or through). Below is an imperfect maze.

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

In this maze, the square at co-ordinate [1,1] is not allowed to be moved into. I assume when the maze is created, a list of ‘illegal’ squares are given.

This means, to find whether a gene contains an illegal move you have to iterate through each gene checking if the co-ordinates are in the ‘illegal moves’ list

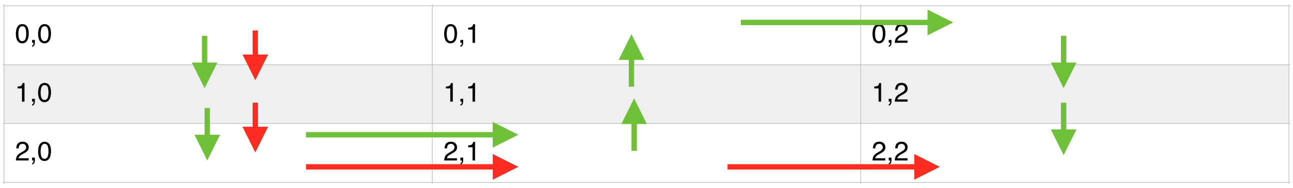
The pseudocode:

```
checkIllegalSquares(x)
    for i = 0 to x.length
        if illegalMoves.contains(x[i])
            fitness--
```

This would ensure that genes which contained illegal moves would be considered unfit.

Legal Length

The size of my genes would be between the Manhattan distance and total number of spaces -1.



0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

The green line indicates the longest possible path (without looping) and the red line shows the shortest possible path (Manhattan distance).

Valid genes would be between these two sizes as any shorter than the shortest path, wouldn't be a complete path. If a path is longer than the longest distance then it contains loops and therefore is an invalid path.

The pseudocode:

```
checkLegalLength(x)
    if x.length > (maze.size * maze.size) -1
        valid = false
    else x.length < maze.goalNode.getX + maze.goalNode.getY
        valid = false
    else
        valid = true
    return valid
```

Gene Length

The gene length would be a factor for how optimal a solution is. If a gene has a smaller length and meets the same fitness criteria (legal length, legal moves, etc) as another longer gene, it is considered a more optimal solution than the second gene

For example, the below genes are both valid genes, however the first is significantly shorter and therefore a more optimal solution.

[0,0], [1,0], [2,0], [2,1], [2,2]

[0,0], [1,0], [2,0], [2,1], [1,1], [0,1], [0,2], [1,2], [2,2]

The population is divided into two, the 50% shortest genes get a fitness increase and the longest 50% get a fitness decrease.

The pseudocode:

```
rankLength()
    populationSorted = quicksort(population)
    for i = 0 to populationSorted.length/2
        /*
        * The first value in a gene is the fitness (see representation
        * section) therefore populationSorted[i[0]] get’s the fitness
        * value
        */
        populationSorted[i[0]]++
    for i = population.length/2 to population.length
        populationSorted[i[0]]—
```

Summary

The disadvantages is the computational time this algorithm will take. It often iterates through the entire population with some parts having *$O(n^2)$* time. This is very time consuming and will take long on very large populations. This is unfortunate, as the randomised initial start requires a large population to increase the probability of relatively fit and healthy genes to be generated and be able to breed.

Overall, I believe that this is an effective algorithm for finding the best solution a genetic algorithm can find - albeit in a long time.

Task 4

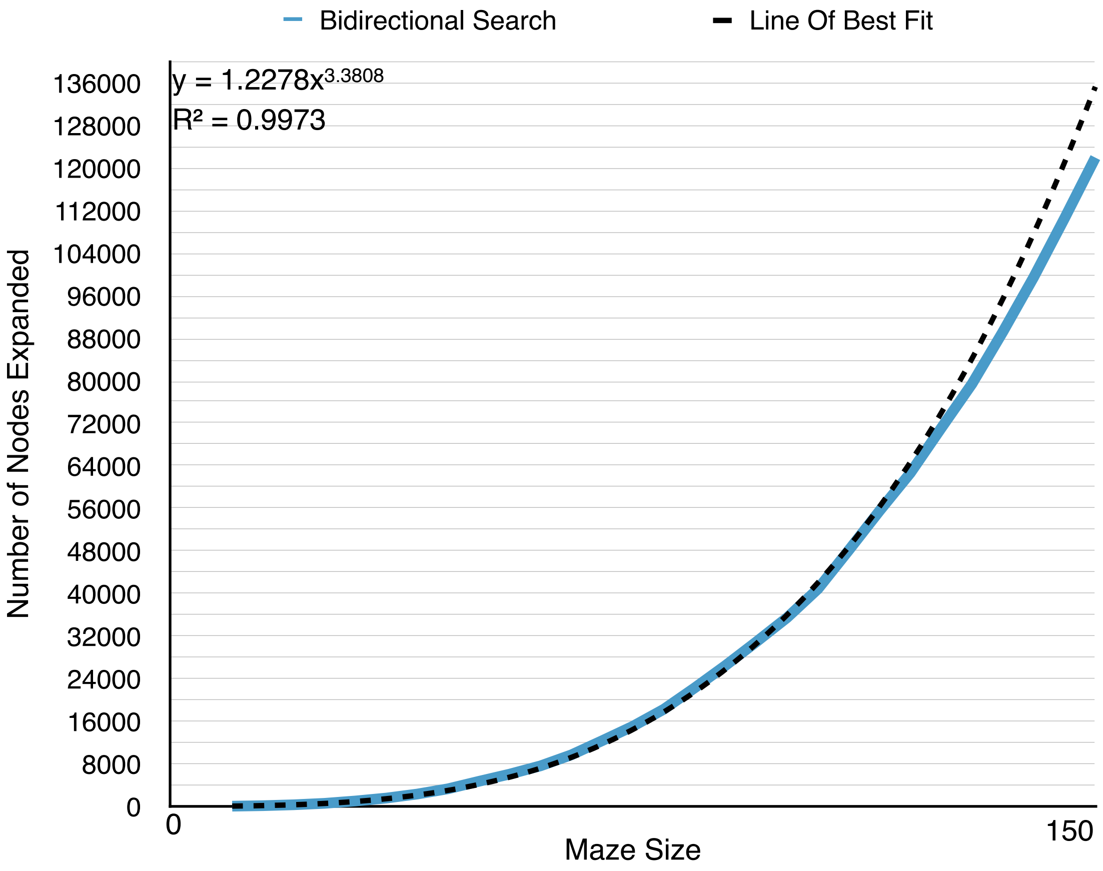
I did not attempt this question.

Appendix

Bidirectional	
Maze Size	Nodes Expanded
10	46
15	136
20	328
25	610
30	1010
35	1652
40	2441
45	3465

50	4649
55	6300
60	8391
65	10291
70	12571
75	15502
80	18694
85	21905
90	26381
95	30864
100	36262
105	41861
110	48863
115	56717
120	63502
125	70540
130	79127
135	88879
140	100311
145	110569
150	122375

Appendix 1: The number of nodes expanded using Bidirectional search between the maze size of 10 to 150 (increasing with steps of 5) with 10 walls being removed.

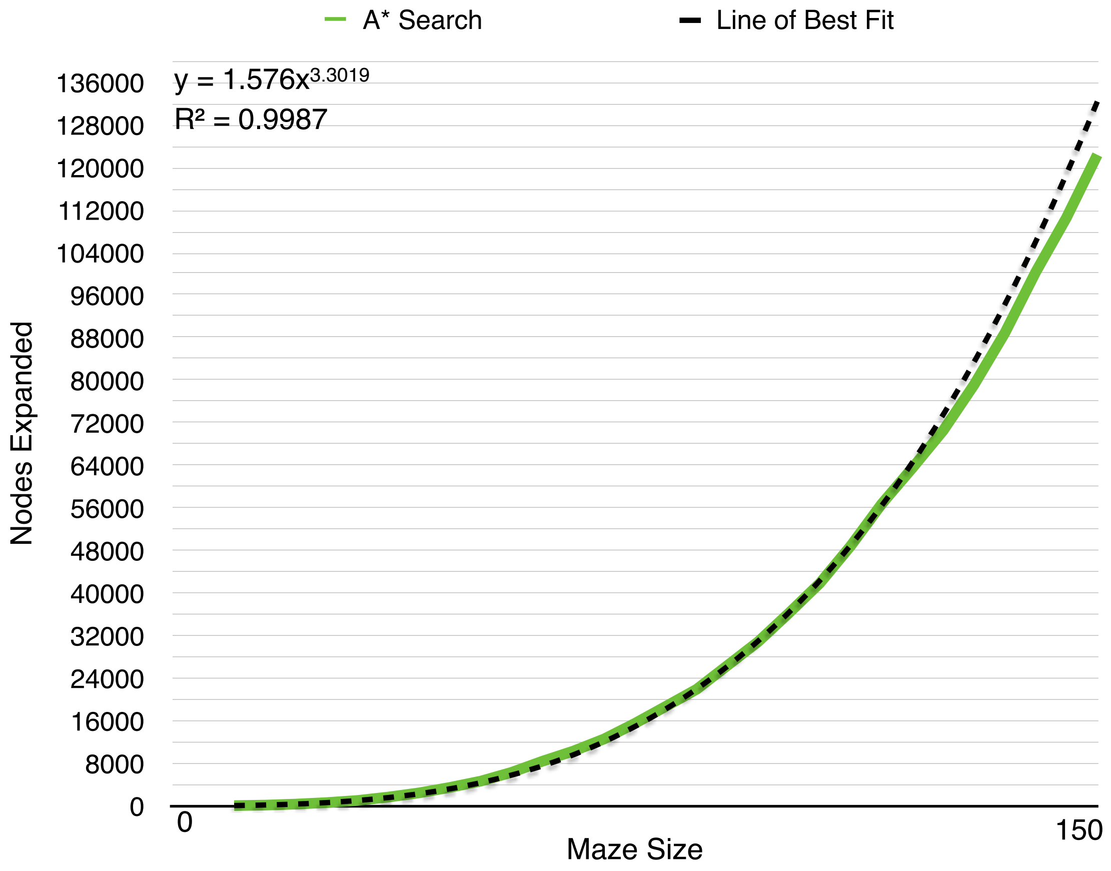


Appendix 2: A graph to visualise the data appendix 1

A*	
Maze Size	Nodes Expanded
10	32
15	125
20	295
25	566
30	1008
35	1555
40	2308
45	3298
50	4701
55	6056
60	7608
65	9676
70	12335

75	15061
80	18249
85	22262
90	26466
95	30909
100	35471
105	40866
110	48135
115	55548
120	62673
125	71108
130	79431
135	89207
140	99471
145	110539
150	121948

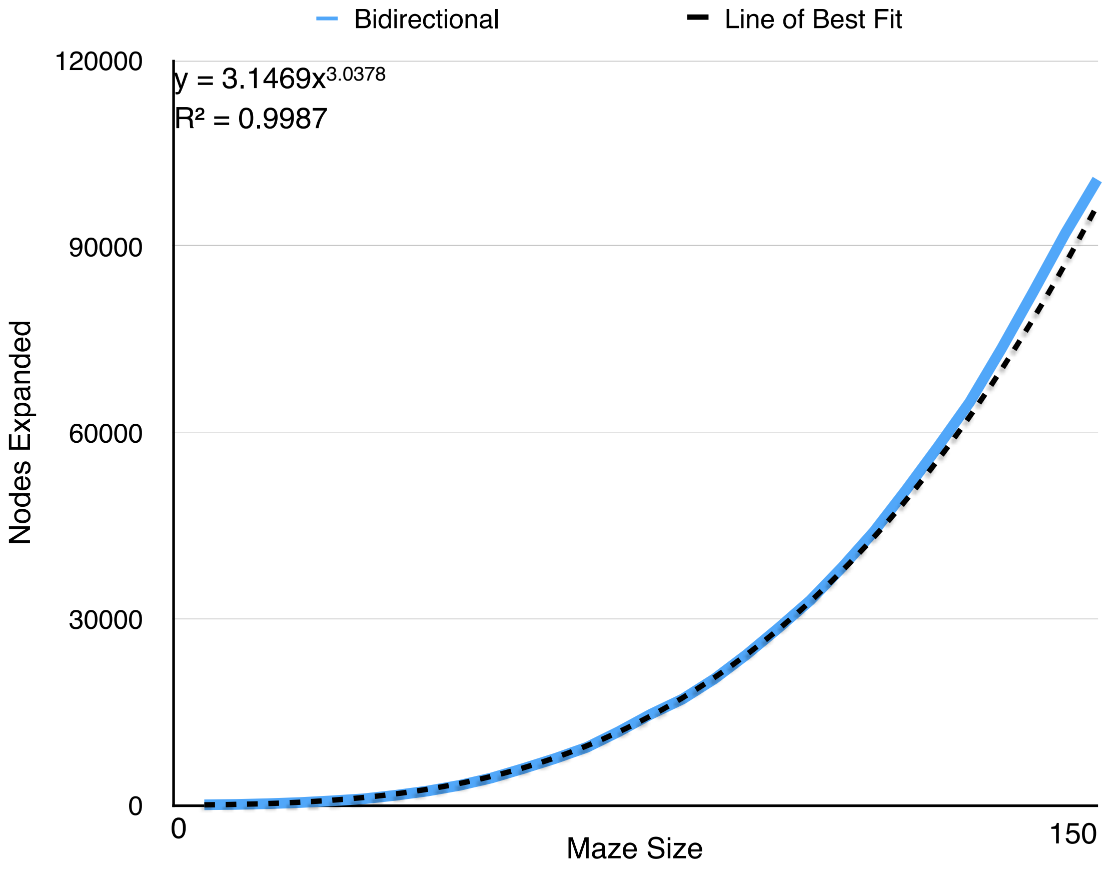
Appendix 3: The number of nodes expanded using A* search between the maze size of 10 to 150 (increasing with steps of 5) with 10 walls being removed.



Appendix 4: A graph to visualise the data of appendix 3

Bidirectional	
Maze Size	Nodes Expanded
10	35
15	91
20	211
25	393
30	667
35	996
40	1564
45	2251
50	3161
55	4350
60	5873
65	7493
70	9306
75	11840
80	14629
85	17081
90	20342
95	24222
100	28494
105	32966
110	38296
115	44069
120	50728
125	57684
130	64901
135	73555
140	82729
145	92127
150	100767

Appendix 5: The number of nodes expanded using A* search between the maze size of 10 to 150 (increasing with steps of 5) with 50 walls being removed.

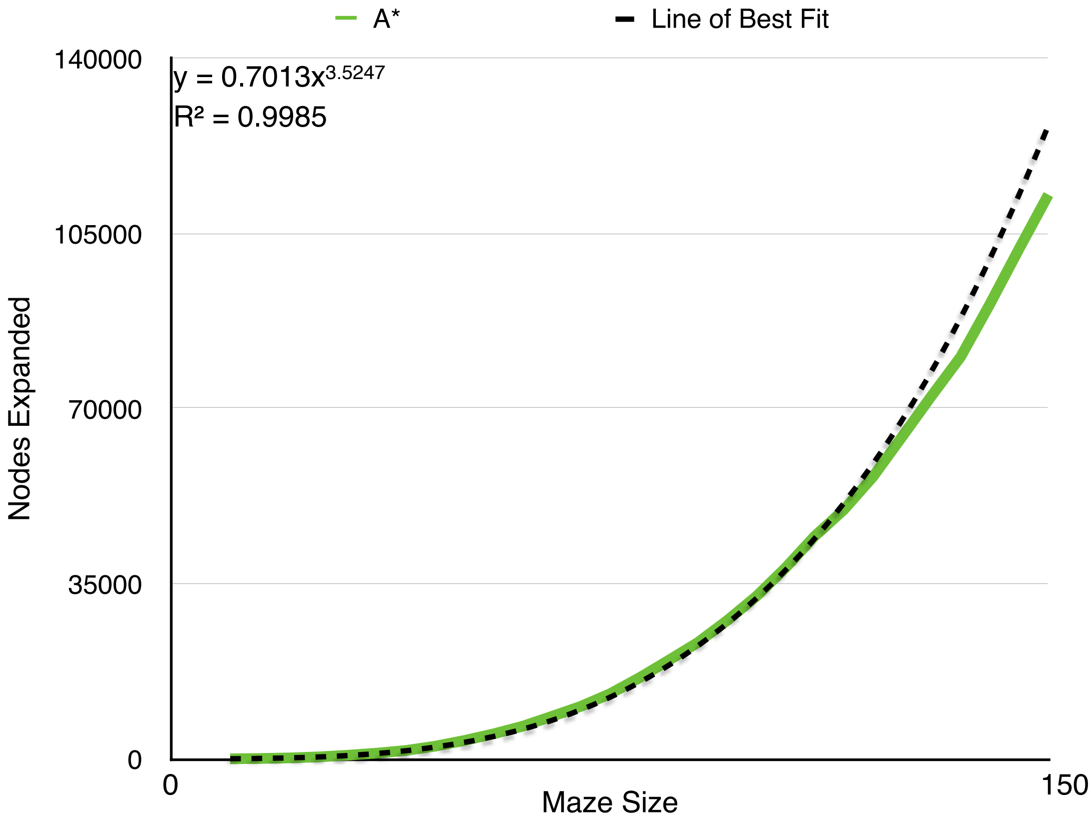


Appendix 6: A graph to visualise the data of appendix 5.

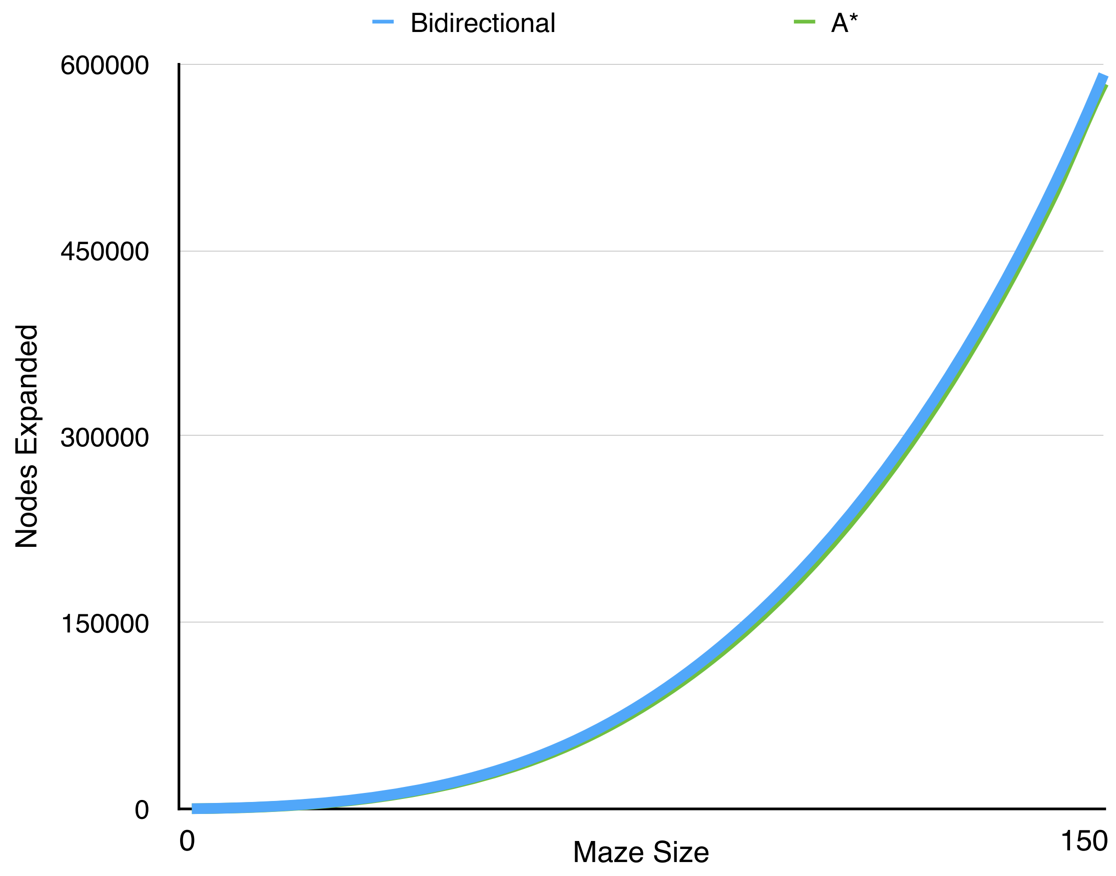
A*	
Maze Size	Nodes Expanded
10	26
15	84
20	195
25	397
30	728
35	1144
40	1675
45	2527
50	3665
55	4984
60	6516

65	8483
70	10505
75	13022
80	16203
85	19689
90	23255
95	27631
100	32458
105	38166
110	44508
115	49793
120	56406
125	64449
130	72540
135	80389
140	90925
145	101989
150	112788

Appendix 7: The number of nodes expanded using Bidirectional search between the maze size of 10 to 150 (increasing with steps of 5) with 50 walls being removed.



Appendix 8: A graph to visualise the data of appendix 7.



Appendix 9: A graph comparing the number of nodes expanded using Bidirectional and A* search between maze size of 10 to 150 (increasing in steps of 5) with 10 walls being removed.

	Walls Deleted	Maze Size	Nodes Expanded
Bidirectional	10	10	46 [Appendix 1]
A*	10	10	32 [Appendix 3]
Bidirectional	10	100	36262 [Appendix 1]
A*	10	100	35471 [Appendix 3]
Bidirectional	10	150	122375 [Appendix 1]
A*	10	150	121948 [Appendix 3]
Bidirectional	50	10	35 [Appendix 5]
A*	50	10	26 [Appendix 7]
Bidirectional	50	100	28494 [Appendix 5]
A*	50	100	38166 [Appendix 7]

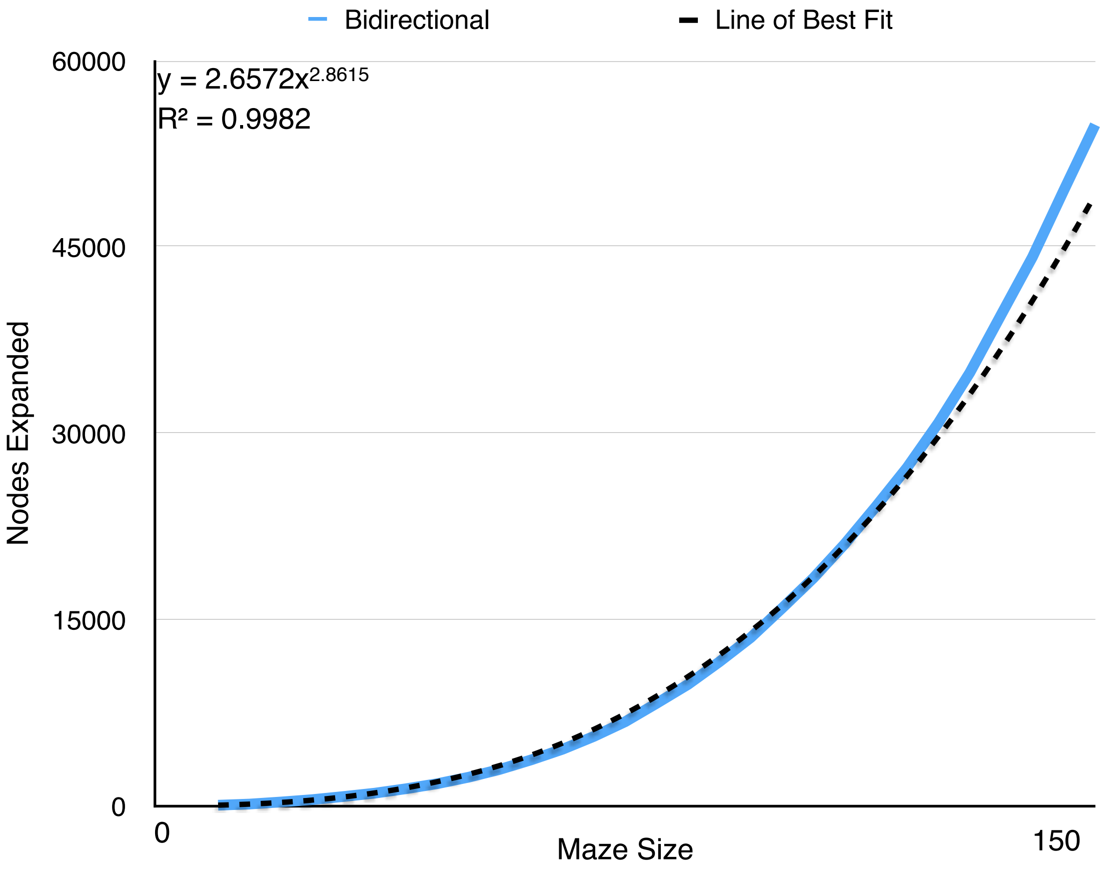
	Walls Deleted	Maze Size	Nodes Expanded
Bidirectional	50	150	100767 [Appendix 5]
A*	50	150	112788 [Appendix 7]
Bidirectional	1000	10	59 [Appendix 11]
A*	1000	10	41 [Appendix 13]
Bidirectional	1000	100	15905 [Appendix 11]
A*	1000	100	16556 [Appendix 13]
Bidirectional	1000	150	54837 [Appendix 11]
A*	1000	150	64609 [Appendix 13]

Appendix 10: A table to compare nodes expanded at the minimum, maximum and 100 maze size values with variable walls removed. The lowest (most efficient) values are highlighted in green and the least efficient in red.

Bidirectional	
Maze Size	Nodes Expanded
10	59
15	150
20	316
25	506
30	757
35	1028
40	1385
45	1792
50	2313
55	2946
60	3712
65	4565
70	5599
75	6783
80	8260
85	9778
90	11595
95	13544
100	15905
105	18332

110	21073
115	24099
120	27252
125	30852
130	34878
135	39545
140	44193
145	49560
150	54837

Appendix 11: The number of nodes expanded using Bidirectional search between the maze size of 10 to 150 (increasing with steps of 5) with walls being removed value of 1000

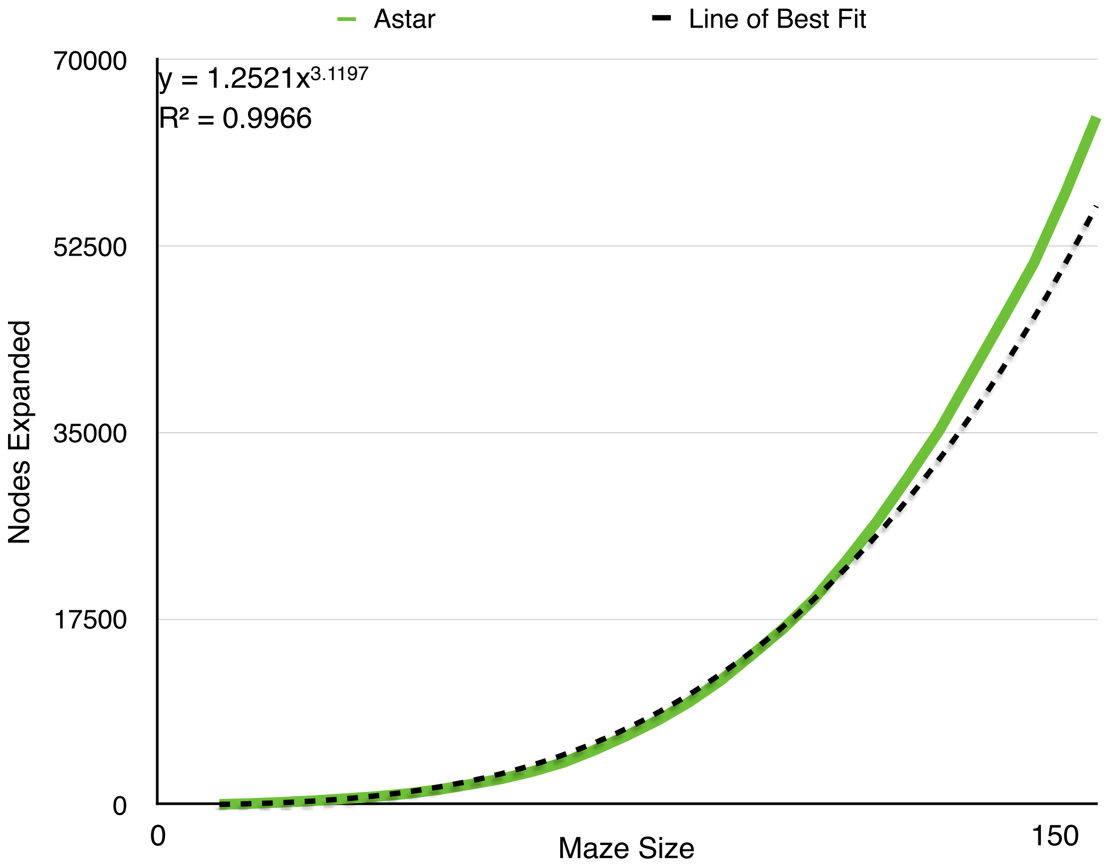


Appendix 12: A graph to visualise the data of appendix 11.

A*	
Maze Size	Nodes Expanded
10	41

15	115
20	236
25	378
30	567
35	770
40	1047
45	1425
50	1914
55	2433
60	3132
65	3992
70	5163
75	6486
80	7937
85	9649
90	11675
95	14070
100	16556
105	19374
110	22871
115	26641
120	30838
125	35236
130	40426
135	45606
140	50867
145	57437
150	64609

Appendix 13: The number of nodes expanded using A* search between the maze size of 10 to 150 (increasing with steps of 5) with walls being removed value of 1000



Appendix 14: A graph to visualise the data of appendix 13