Extended Essay

Computer Science

Title: Improving the efficiency of the A* algorithm

Research question: To what extent does a modification of the heuristic improve the efficiency of the A* algorithm on 8-connected undirected uniform-cost grid maps with search-space symmetry?

Word count: 3997

Table of Contents

1. Introduction

The field of artificial intelligence (AI) strives to understand and build intelligent entities.[1] Traditional AI now works with complex compute-intensive concepts such as neural networks. By contrast, in game AI, efficiency is prioritised over complexity, as decisions have to be made in real time.[2] Pathfinding, one of the major domains of game AI, finds its application beyond computer games, in areas such as network traffic, robot planning and military simulations.[3] Due to the applicability of research in pathfinding, this paper will study this domain of AI.

1.1. Search space representations

Pathfinding algorithms solve the following problem: Given points A and B, how to traverse the region between them?[4] Typically, a grid is superimposed over a region.[5] Square grids, which are favoured for their ease of implementation,[6] are most commonly used.[7] Square grids were used to represent playing areas in games such as *Civilization* and *Warcraft*. *Figure 1.1* displays an instance of a square grid with a highlighted tile. *Figure 1.2* shows a square grid superimposed over a region in the game *Skulls of the Shogun*.

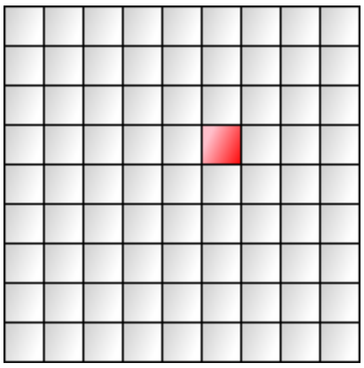
Figure 1.1: A square grid[8]

[1] (Russell & Norvig, 2010)
[2] (Madhav, 2014)
[3] (Yap, 2002)
[4] (Madhav, 2014)
[5] (Yap, 2002)
[6] (Champandard, 2011)
[7] (Patel, 2006)
[8] (Patel, 2006)

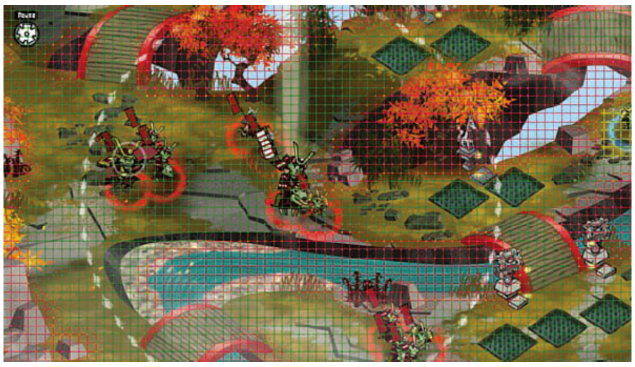

Figure 1.2: Square grid used for world representation in a game[9]

Nevertheless, a game's world representation does not have to match the underlying search space representation used in pathfinding.[10] Square grid maps often feature path symmetry. Path symmetry manifests itself as paths with a common start and end point and identical length.[11] The only difference is the order in which constituent moves forming a path occur. Path symmetry may force search algorithms to evaluate numerous equivalent paths, as outlined in *Figure 1.3*.

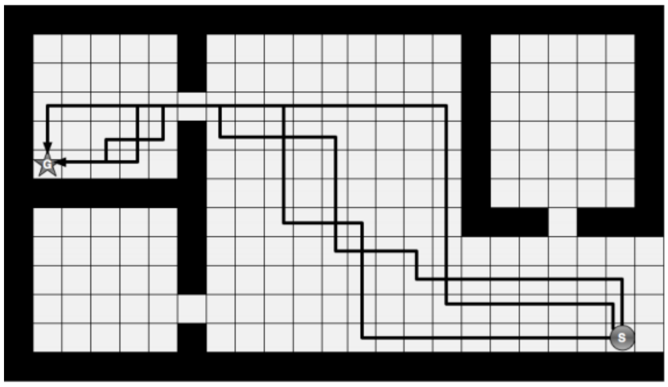

Figure 1.3: A grid map featuring path symmetry[12]

Thus, path symmetry may prevent progress towards the goal,[13] leading to an inefficient search. Under such circumstances, pathfinding algorithms can benefit from different search space representations. However, complex representations such as navigation meshes and polygonal maps, illustrated below, exceed the scope of this paper.

[9] (Madhav, 2014)
[10] (Patel, 2006)
[11] (Harabor & Grastien, 2011)
[12] (Harabor & Botea, 2010)
[13] (Harabor & Grastien, 2011)

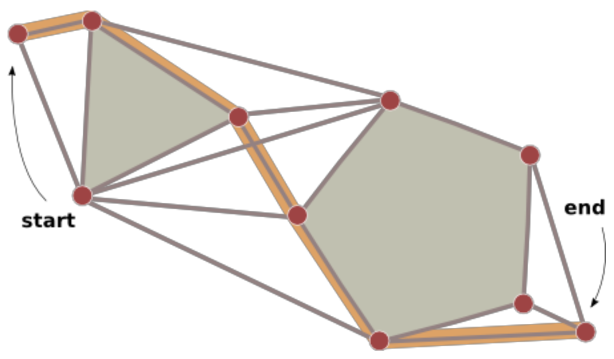

Figure 1.4: An instance of a polygonal map[14]

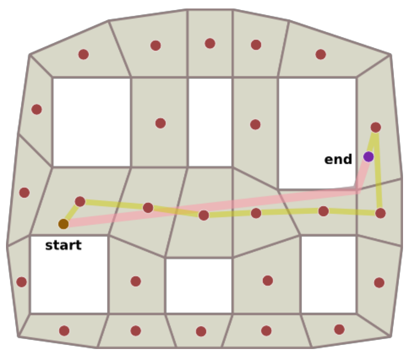

Figure 1.5: An instance of a navigation mesh[15]

This paper will focus on square grid maps as search space representations and investigate different strategies in fighting path symmetry. It is predominantly undirected uniform-cost grid maps that feature path symmetry. ‘Undirected’ denotes that movement between two arbitrary tiles on the grid can occur in either direction. ‘Uniform-cost’ means that for cardinal movement and diagonal movement respectively, a common cost is assigned.[16]

4-connected grid maps allow cardinal movement only, whereas the more popular 8-connected grid maps also allow diagonal movement.[17] 8-connected grid maps appear in popular areas such as robotics and game AI.[18] Therefore, investigating methods aimed at diminishing the cost incurred due to path symmetry on 8-connected undirected uniform-cost square grid maps may have significant implications.

1.2. Search space representations as input

Pathfinding algorithms take a graph as an input. A graph is a set of nodes and edges that connect the nodes.[19] An example is provided in *Figure 1.6*. A square grid is a special case of a graph.[20] In the domain of grids, nodes are represented by individual tiles on the grid. On 8-connected square grid maps, each node will have a maximum of eight edges leading to its neighbour nodes.

[14] (Patel, 2018b)
[15] (Patel, 2018b)
[16] (Harabor, 2018)
[17] (Harabor, 2011)
[18] (Harabor, et al., 2011)
[19] (Patel, 2010)
[20] (Patel, 2014a)

As outlined in *Figure 1.7*, this number will vary depending on whether the set of neighbour nodes includes obstacles or boundaries

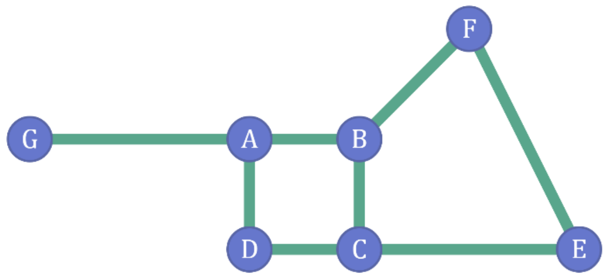

Figure 1.6: A graph; nodes marked violet and edges green[21]

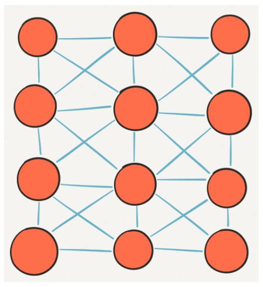
Figure 1.7: An 8-connected grid represented as a graph; nodes marked orange and edges blue[22]

1.3. Pathfinding algorithms

While numerous pathfinding algorithms exist, the A* algorithm, noted for its performance, accuracy and flexibility, has become a well-established standard in the realm of pathfinding.[23] Due to its relevance, this paper will focus on A*.

The algorithm's heuristic function can be used to control the algorithm's behaviour.[24] Therefore, this paper will discuss the question: **To what extent does a modification of the heuristic improve the efficiency of the A* algorithm on 8-connected undirected uniform-cost grid maps with search-space symmetry?**

1.4. Plan of investigation

This paper will first describe Dijkstra's and Greedy Best-First-Search algorithm, from which A* takes. Then, the connection of A* to either will be explained. Next, the notion of admissible and non-admissible heuristic will be introduced. Finally, this paper will elaborate on three respective modifications of A*'s heuristic function. During the experimental phase, these modifications will be implemented in a testing environment to assess their performance on a set of randomly generated grid maps.

[21] (Patel, 2014a)
[22] (Fischer, 2015)
[23] (An, et al., 2016)
[24] (Patel, 2009)

2. Dijkstra’s Algorithm

Dijkstra’s Algorithm considers the incremental cost of reaching a node from the start node. It expands outwards from a given start point, examining unvisited nodes closest to the start point until the goal node is reached.[25] As Dijkstra’s expands in all directions, there may be less progress towards the goal node. *Figure 2.1* exemplifies such a scenario, with nodes visited by the algorithm marked turquoise with hue, and the start and goal node represented by the pink and purple tile, respectively.

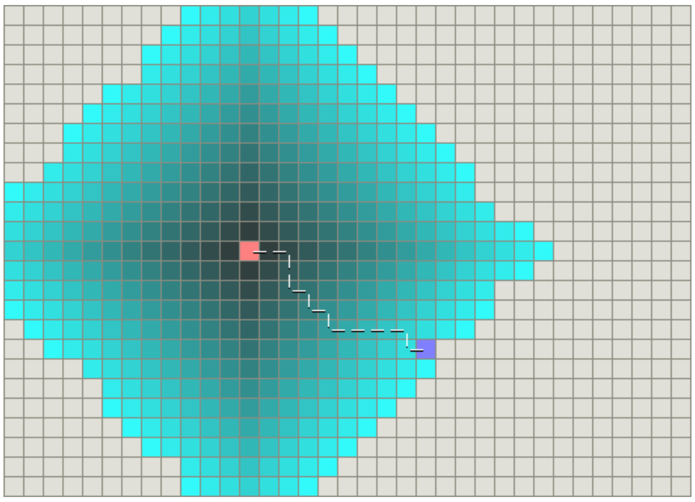

Figure 2.1: The frontier of expansion of Dijkstra’s Algorithm[26]

[25] (Patel, 2010)
[26] (Patel, 2010)

3. Greedy Best-First-Search algorithm

During search, Greedy BFS selects the node with the lowest estimated cost of reaching the goal node. This estimate is obtained using a heuristic function. This approach may allow Greedy BFS to progress towards the goal faster than Dijkstra's. An example is provided in *Figure 3.1*, where nodes visited by Greedy BFS are marked yellow with hue. Greedy BFS visited visibly less nodes than Dijkstra's, whose expansion in an identical scenario was shown in *Figure 2.1*.

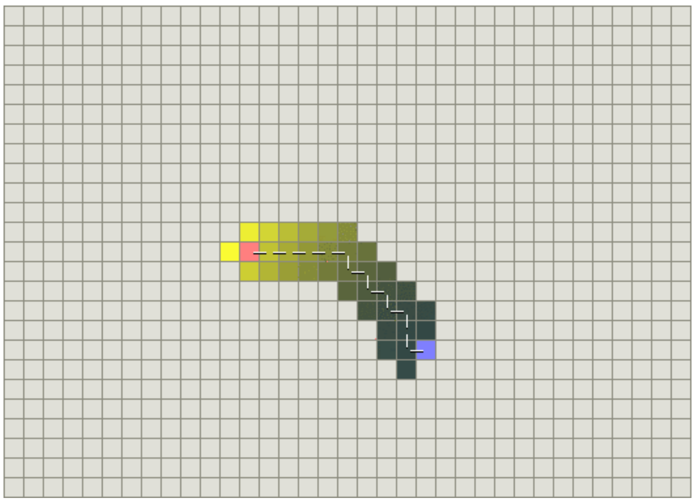

Figure 3.1: The expansion of Greedy BFS[27]

However, Greedy BFS ignores the distance from the starting node to the node being visited.[28] This may cause the algorithm to return a path that is suboptimal, *i.e.* not the shortest one possible. Consider the contrast between the expansion of Dijkstra's and Greedy BFS, shown in *Figure 3.2* and *Figure 3.3*. Although Greedy BFS visited less nodes than Dijkstra's, it returned a suboptimal path.

[27] (Patel, 2010)
[28] (Patel, 2010)

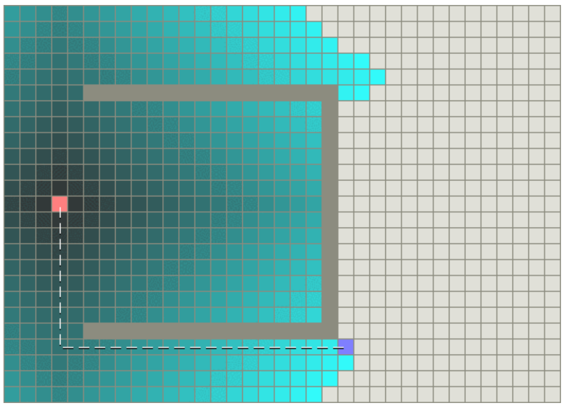

Figure 3.2: The expansion of Dijkstra's Algorithm in the presence of a concave obstacle[29]

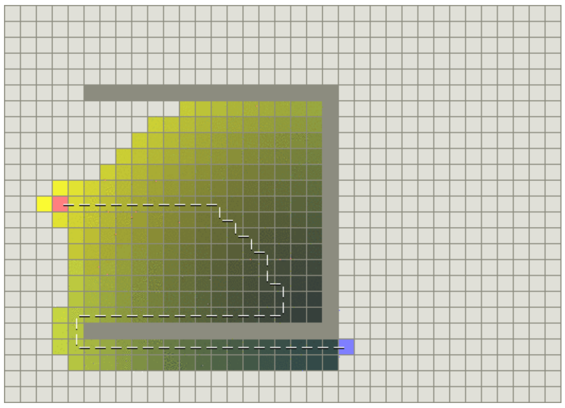

Figure 3.3: The expansion of Greedy BFS in the presence of a concave obstacle[30]

The fact that the search is guided and does not occur in all directions as in the case of Dijkstra's may reduce the time it takes to find a path by decreasing the number of visited nodes. However, due to Greedy BFS's ambivalence towards path length, the path found may be more costly.

[29] (Patel, 2010)
[30] (Patel, 2010)

4. A* search algorithm

The A* algorithm combines the formal approaches of Dijkstra's Algorithm while also relying on Greedy BFS' heuristic approach. When compared to Dijkstra's Algorithm, A* performs better with respect to time due to the use of heuristics[31,32] while, unlike Greedy BFS, guaranteeing that an optimal path will be returned.

4.1. The g cost

Akin to Dijkstra's Algorithm, the A* algorithm observes the incremental cost of moving to a given node from the start node. This information is stored in the node's g cost. For any arbitrary node n, the g cost is calculated using $g(n) = g(n.parent) + cost(n.parent, n)$,[33] where:

$$n.parent = parent\ node\ of\ n$$

$$cost(n_1, n_2) = cost\ of\ movement\ from\ n_1\ to\ n_2$$

4.2. The h cost

Similar to the heuristic used in Greedy BFS, the h cost is an estimate of the distance between a given node and the goal node. The h cost is calculated using a heuristic function, whose suitability will vary depending on the properties of the search space.[34] For any arbitrary node n, the h cost is calculated using $h(n) = heuristic(n, goal)$.

4.3. The f cost

When selecting the next node to visit, Dijkstra's Algorithm selected the node with the lowest movement cost from the start node and Greedy BFS selected the one with the lowest estimated movement cost of reaching the goal node. In the case of A*, the next node is picked according to the lowest f cost. For any arbitrary node n, the f cost is obtained using $f(n) = g(n) + h(n)$.

[31] (Patel, 2010)
[32] (Nosrati, et al., 2012)
[33] (*Growing with the Web*, 2012)
[34] (*Growing with the Web*, 2012)

5. Heuristic functions

5.1. Admissible heuristic

A* will always find an optimal path as long as it uses an admissible heuristic function.[35] A heuristic function is admissible if for any arbitrary node n, the condition holds: $h(n) \leq h^*(n)$,[36] where:

$$h(n) = cost\ approximated\ by\ heuristic\ to\ reach\ the\ goal\ node$$

$$h^*(n) = optimal\ cost\ to\ reach\ the\ goal\ node$$

That is, a heuristic function is admissible if for any given node n, it does not overestimate the cost of reaching the goal node.[37] As Patel argues, in many implementations of A*, a heuristic function is used that is inefficient on the implementation's search space representation.[38] In the following subsections, the three most widely used admissible heuristics will be evaluated to select the one most likely to provide greatest efficiency on 8-connected undirected uniform-cost grid maps.

5.1.1. Manhattan distance heuristic

The *Manhattan distance* heuristic is limited to 4-directional movement. Therefore, it does not consider the diagonal movement between adjacent nodes when estimating the cost of reaching the goal node. It is calculated using $Ds * (dx + dy)$[39], where:

$$dx = |node.x - goal.x|$$

$$dy = |node.y - goal.y|$$

$$Ds = cost\ of\ moving\ cardinally$$

[35] (Patel, 2014b)
[36] (López, 2013)
[37] (Taylor, n.d.)
[38] (Patel, 2018a)
[39] (Patel, 2009)

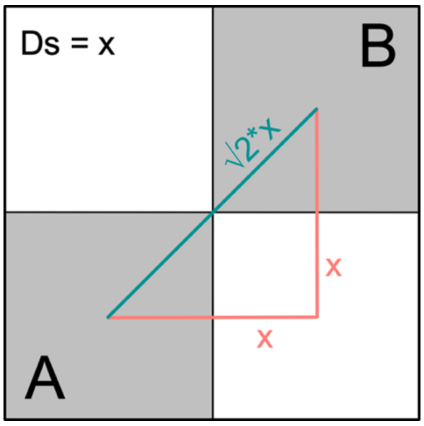

Figure 5.1: Manhattan distance heuristic's overestimation of movement cost

Figure 5.1 exemplifies the limitation of the *Manhattan distance* heuristic. The function estimated that the cost of reaching node A from node B at $h(n) = 2x$ whereas the optimal cost is $h^*(n) = \sqrt{2x^2} = \sqrt{2} * x$. Hence, as $h(n) > h^*(n)$, the *Manhattan distance* heuristic will overestimate the cost of reaching a goal node.[40] Therefore, this heuristic is non-admissible for 8-connected grid maps and its use would result in considerably suboptimal paths being yielded. Therefore, this heuristic function will not be implemented in the experimental phase.

5.1.2. Euclidean distance heuristic

The *Euclidean distance* heuristic assumes that movement is allowed in any direction. However, on 8-connected grid maps, movement is limited to cardinal or diagonal. It is calculated using $Ds * \sqrt{dx^2 + dy^2}$.[41]

Figure 5.2 shows the limitation of the *Euclidean distance* heuristic. The function approximated the cost of reaching node B from node A at $h(n) = \sqrt{5} * x$ while the optimal cost on an 8-connected grid map is $h^*(n) = (\sqrt{2} + 1)x$. Hence, as $h(n) < h^*(n)$, the *Euclidean distance* heuristic will underestimate the cost of reaching a goal node. Therefore, this heuristic is admissible, but inaccurate on 8-connected grid maps. For this reason, this heuristic function will not be implemented.

[40] (Patel, 2018a)
[41] (Patel, 2009)

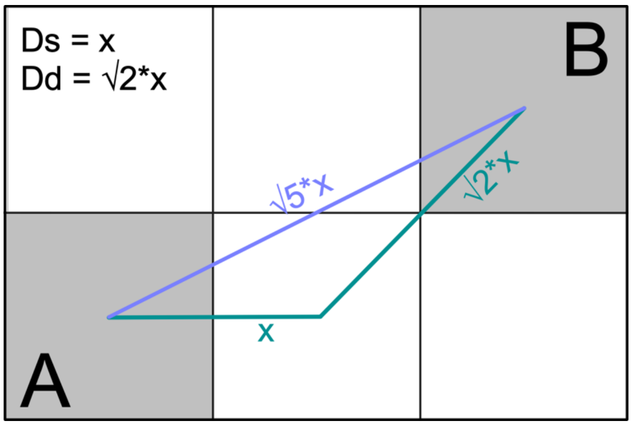

Figure 5.2: Euclidean distance heuristic's underestimation of movement cost

5.1.3. Diagonal distance heuristic

The *Diagonal distance* heuristic function first calculates the cost using *Manhattan distance* heuristic. This is illustrated in *Figure 5.3*. The *Manhattan distance* cost was estimated at $dx + dy = 3x$. Then, the number of steps that could potentially be taken diagonally between adjacent nodes is calculated using $\min(dx, dy)$. In the case given by *Figure 5.3*, the number is $\min(2,1) = 1$, meaning that one step could be taken diagonally. The cost that could thus be saved is obtained using $2 * Ds - Dd$, where $Dd = cost\ of\ moving\ diagonally$. In *Figure 5.3*, it is given by $2x - \sqrt{2}x$. The cost is then multiplied by the number of steps that could be taken diagonally and subtracted from the original value given by the *Manhattan distance* heuristic.

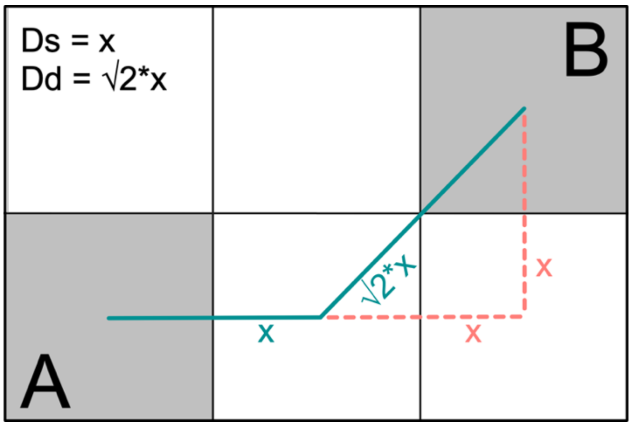

Figure 5.3: Diagonal distance heuristic

The complete equation used to calculate the *Diagonal distance* heuristic is $h(n) = Ds * (dx + dy) - (2 * Ds - Dd) * \min(dx, dy)$.[42] When applied to the situation given in *Figure 5.3*: $h(n) = 3x - (2x - \sqrt{2}x) * 1 = (1 + \sqrt{2})x$. Indeed, unlike *Manhattan* and *Euclidean distance* heuristic, which estimated the cost at $h(n) = 3x$ and $h(n) = \sqrt{5}x$ respectively, the cost estimated by the *Diagonal distance* heuristic exactly matches the optimal cost of reaching the goal node. **Due to its accuracy on 8-connected grid maps, the *Diagonal distance heuristic* will be implemented.**

5.2. Non-admissible heuristic in tie-breaking

It has been found that maintaining the admissibility of A* is restrictive in that it forces the algorithm to spend a disproportionately long time discriminating amongst candidate nodes with identical f costs.[43]

Figure 5.4 outlines a situation where multiple nodes share the same f cost. This causes path symmetry, *i.e.* the existence of multiple optimal paths. Path symmetry often occurs on grid maps with little variation in terrain. Under such circumstances, A* might explore all paths with an identical cost rather than only one. In *Figure 5.4*, five equivalent paths are highlighted using dotted lines.

[42] (Patel, 2009)
[43] (Dechter & Pearl, 1985)

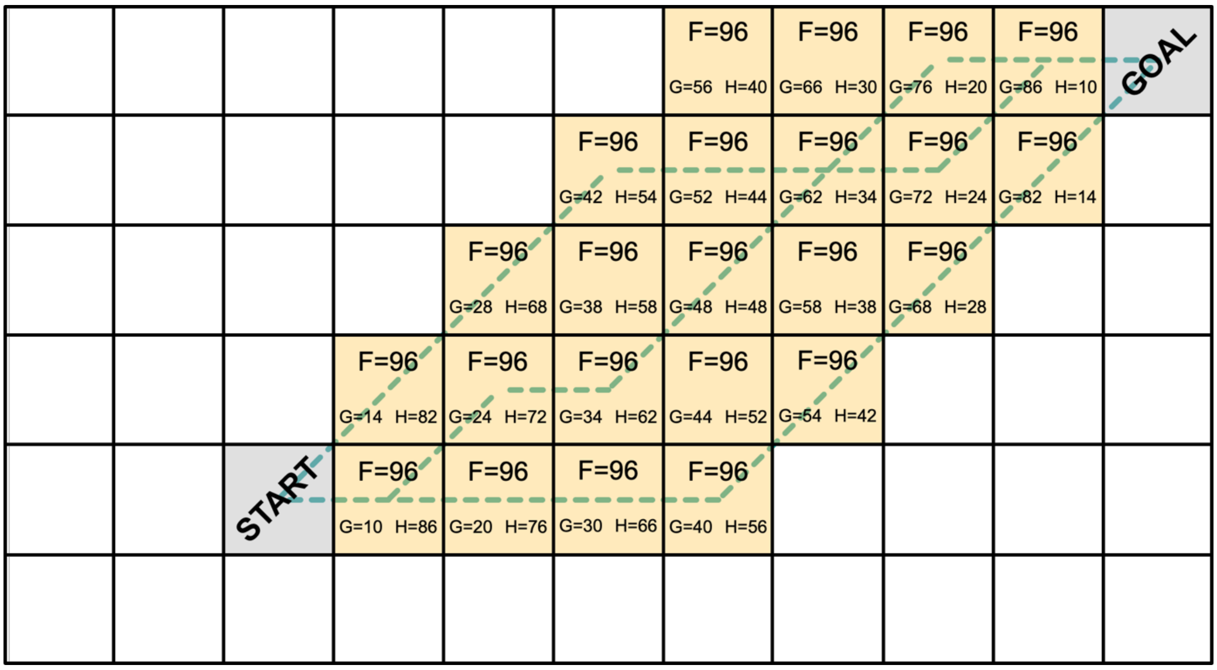

Figure 5.4: An expansion of A showing path symmetry*

In order to avoid such a situation, a tie-breaking rule can be set. The role of a tie-breaker is to decrease the number of nodes with an identical f cost.[44] A non-admissible heuristic is one example of a tie-breaker. The use of a non-admissible heuristic function may lead to suboptimal paths being found. However, as Ericson argues, calculating an optimal path is rarely necessary in game development.[45] In the experimental phase, the performance of three non-admissible modifications of the admissible *Diagonal distance* heuristic function will be analysed on grid maps featuring path symmetry. These modifications will be outlined in detail in the following subsections.

5.2.1. Multiplication by a constant

One way to adjust the heuristic is to multiply the h cost by a constant. For nodes with identical f values, the g and h values may differ. In the example given in *Figure 5.4*, no two nodes with an identical f cost share the same h cost. Therefore, multiplying each node's h cost by a constant

[44] (Patel, 2009)
[45] (Ericson, 2008)

would make the nodes' f costs differ. Thus, the tie between the nodes will be broken, rendering this method a viable tie-breaker[46].

The constant can be obtained using $C = 1.0 + p\ (i)$, where $0 < p < \frac{min\ cost\ of\ one\ step}{expected\ \max path\ length}$. The h cost is then calculated by multiplying the value given by the *Diagonal distance* heuristic function by C. In the experiment, the expected maximum path length will be obtained using $(map.width + map.height) * Ds$, as this provides sufficient approximation of the maximum path length.

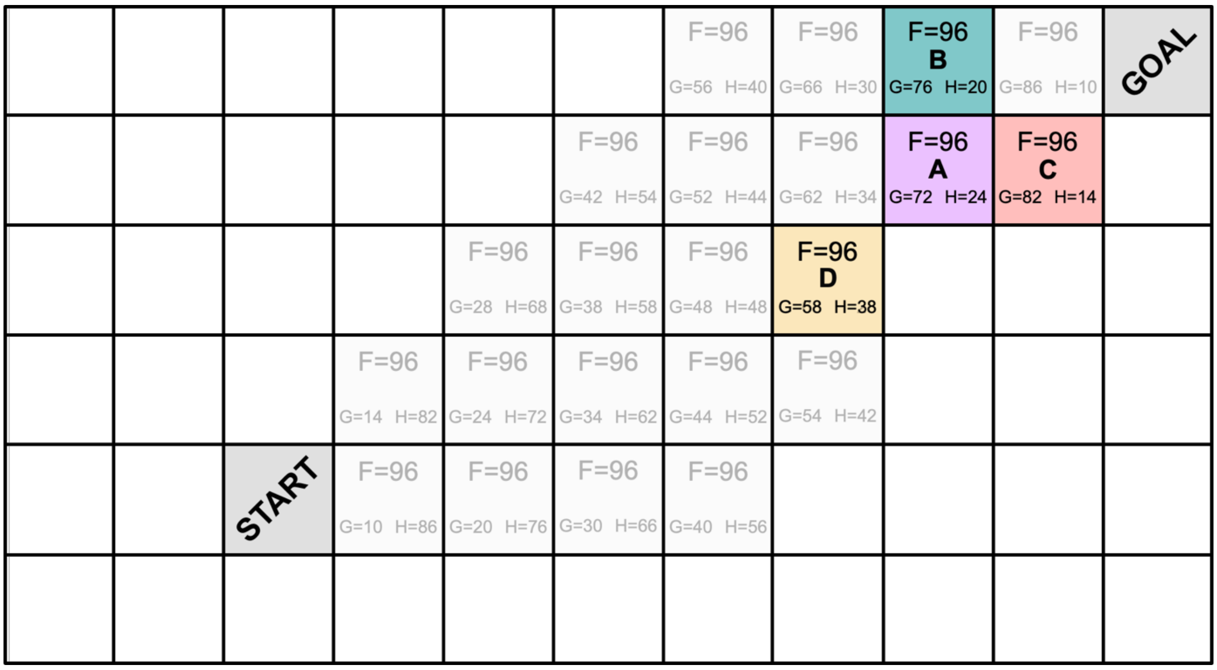

Figure 5.5:A tie between multiple nodes

This approach also creates a bias towards nodes with a lower h cost. Assume node A and its neighbour nodes B and C, marked in *Figure 5.5*. Nodes B and C that share the same f cost of 96. However, B's h cost is greater than C's. Therefore, upon multiplication by a constant C, obtained using i, C's h cost (and thus also its f cost) will be lower than that of node B. The algorithm will now prefer node C to node B due to its lower f cost when proceeding from node A. This means that when breaking ties, the algorithm will have a bias towards nodes with lower

[46] (Patel, 2009)

estimated cost to reach the goal node. This may allow it to progress towards the goal more rapidly.

5.2.2. Addition of a deterministic random number

A pseudo-random number generator is a program used to generate deterministic random numbers.[47] Assume the situation given in *Figure 5.5*. If for multiple nodes with an identical f cost, each node's f cost is incremented by a different pseudo-random number, the tie will be broken. Hence, incrementing the f cost by a deterministic random number is a viable means of tie-breaking.

However, as this number is generated pseudo-randomly, it does not provide any additional information to the algorithm when discriminating between candidate nodes with an identical f cost. Assume nodes B, C and D marked in *Figure 5.5*. Upon the addition of a deterministic random number to each node's f cost, the algorithm might expand node D rather than node B or C, which are closer to the goal. Therefore, this approach might provide questionable performance.

5.2.3. Cross product

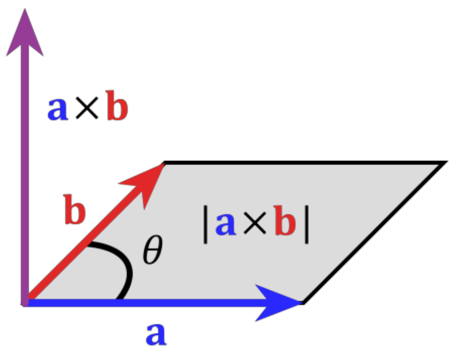

Figure 5.6: Graphical representation of a cross product[48]

[47] (Rouse, 2011)
[48] (*Wikipedia*, 2008)

As outlined in *Figure 5.6*, by computing the magnitude of the cross product of two vectors, the area they span can be measured. The area is given by[49] $|\vec{a} \times \vec{b}| = |\vec{a}| \cdot |\vec{b}| \cdot \sin\theta$. Indeed, the smaller the angle θ (*i.e.* the closer the vectors are to being aligned), the smaller the magnitude of their cross product.

Assume the situation given in *Figure 5.5*. If for either node B and C we take two vectors, the start-goal vector and the current-goal vector, which are shown using dotted lines, by computing the cross product of these two vectors and adding this value to the current node's h cost, the f cost will change. The algorithm will prefer nodes that are along the straight line to the goal,[50] as a lower value will be added to their f cost. In the case given in *Figure 5.5*, node C will be preferred to node B due to lower magnitude of cross product.

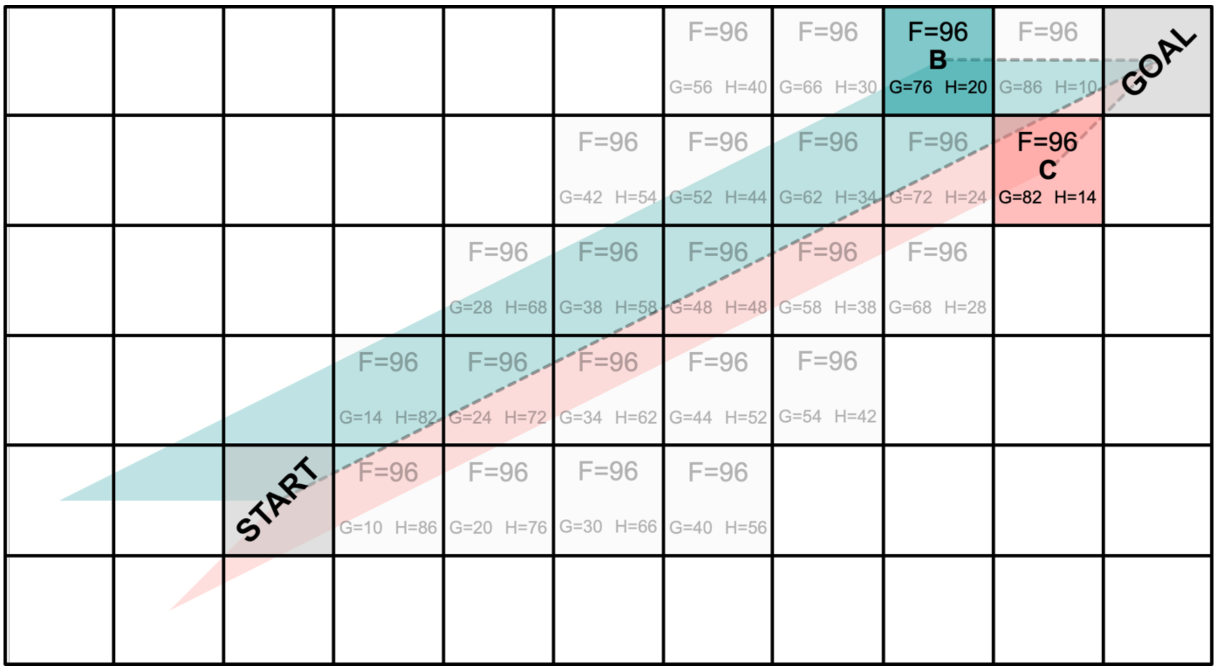

Figure 5.7: Visualisation of the Cross product heuristic

[49] (Weisstein, 2018)
[50] (Patel, 2009)

6. Experiment

The purpose of the experiment will be to compare the modifications of the *Diagonal distance* heuristic in terms of relevant attributes across a variety of path symmetry scenarios.

In pathfinding, computational time is of importance. This is due to its use cases' reliance on real-time computation. Therefore, analysis will be done in terms of this attribute. Moreover, as the previously outlined modifications break the *Diagonal distance* heuristic's admissibility, suboptimal paths may be found. Therefore, the cost of path found will also be analysed.

6.1. Implementation

6.1.1. Programming language

The programming language Python is amongst the leading languages used for developing AI projects. As the A* algorithm finds its use in the domain of AI, the testing system will be developed in this language.

The measurement will be provided using Python's cProfile library. The source code will be compiled by PyCharm Professional Edition and executed on a macOS Mojave computer system with 8 GB of RAM and Intel Core i5-8259U 4-core 8-threaded CPU clocked at 2.30 GHz.

6.1.2. Movement cost

On 8-connected uniform-cost grid maps, if each step in the cardinal direction is assigned a cost of 1, then according to the Pythagorean theorem, each diagonal step presents a cost of $\sqrt{2}$.[51] A common practice is to scale the costs up so that cardinal and diagonal movement have a cost of 10 and 14 respectively. This provides sufficient approximation of the movement costs, while avoiding the expensive square root. Therefore, a movement cost of 10 and 14 will be assigned to cardinal and diagonal movement, respectively.

[51] (Harabor, 2018)

6.1.3. Pseudocode

Outlined below is the core loop of the A* algorithm used in the experimental phase. The call to the heuristic function, highlighted in blue, changes based on the currently selected modification.

```
START ← start node
GOAL ← goal node
CANDIDATES = a min-heap storing candidate nodes with f cost set as priority
CANDIDATES.add(START, 0)
G_COST = a dictionary storing the g costs of added nodes
G_COST[START] ← 0
PARENT = a dictionary storing a reference to parent nodes of nodes added; used
in path reconstruction
PARENT[START] ← null
MAP ← a 2D array containing search space in 0s (passable) and 1s (obstacle)

while CANDIDATES contains nodes do
   CURRENT ← extract candidate node with the lowest f cost from CANDIDATES

   if CURRENT = GOAL then
      return G_COST[CURRENT]
   end if

   for each NEIGHBOR in neighbors(MAP, CURRENT) do
      NEW_G_COST ← G_COST[CURRENT] + cost to move from CURRENT to NEIGHBOR
      if NEIGHBOR ∉ VISITED or if NEW_G_COST < G_COST[NEIGHBOR] then
         G_COST[NEIGHBOR] ← NEW_G_COST
         F_COST ← NEW_G_COST + heuristic(NEIGHBOR, GOAL)
         CANDIDATES.add(NEIGHBOR, F_COST)
         PARENT[NEIGHBOR] ← CURRENT
      end if
   end loop

end loop
```

6.2. Datasets

6.2.1. Obstacle percentage

For the sake of objectivity, randomly generated maps with varying obstacle percentages will be used. For each obstacle percentage, 10 maps will be generated; for each map, 20 unique start-goal scenarios will be randomly generated to achieve sufficient diversity of the dataset. The percentage of obstacles will be within the range $\{0, 0.5, 1, \dots, 10\}\%$. The range will simulate increasing variation in terrain and thus decreasing path symmetry.

6.2.2. Grid map dimensions

In September 2018, 1080p was the second most frequently used desktop screen resolution and exhibited a sustained growth in popularity.[52] Due to the wide use and growing popularity of 1080p screens, the map dimensions will be set to 192×108 cells to approximate the size of a grid on which each cardinal step represents 10 pixels, displayed on a 1080p screen.

6.3. Data collection

For each map, for each scenario, for each modification of the *Diagonal distance* heuristic, the algorithm will run 10 times in order to increase the accuracy of measurement. Then, for each obstacle percentage, each modification's average running time and average path cost will be calculated. The results will be stored in a text file for further data processing.

6.4. Data processing

The results will then be processed and plotted to display each modification's performance relative to the unmodified *Diagonal distance* heuristic against increasing obstacle percentage. Thus, accurate insight into the benefits of using the respective modifications with respect to the admissible *Diagonal distance* heuristic will be obtained.

[52] (*StatCounter*, 2018)

6.5. Data analysis

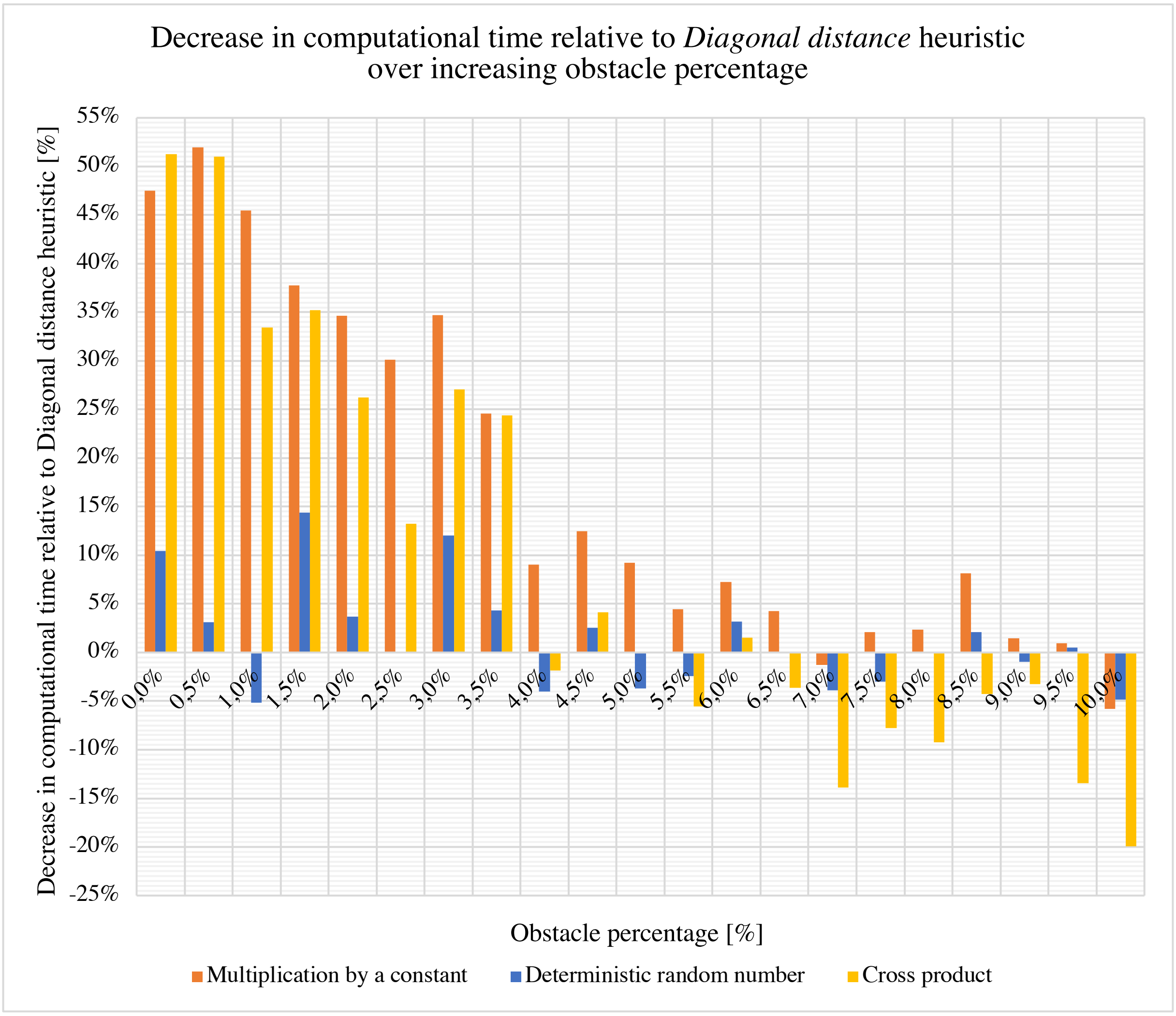

Figure 6.1

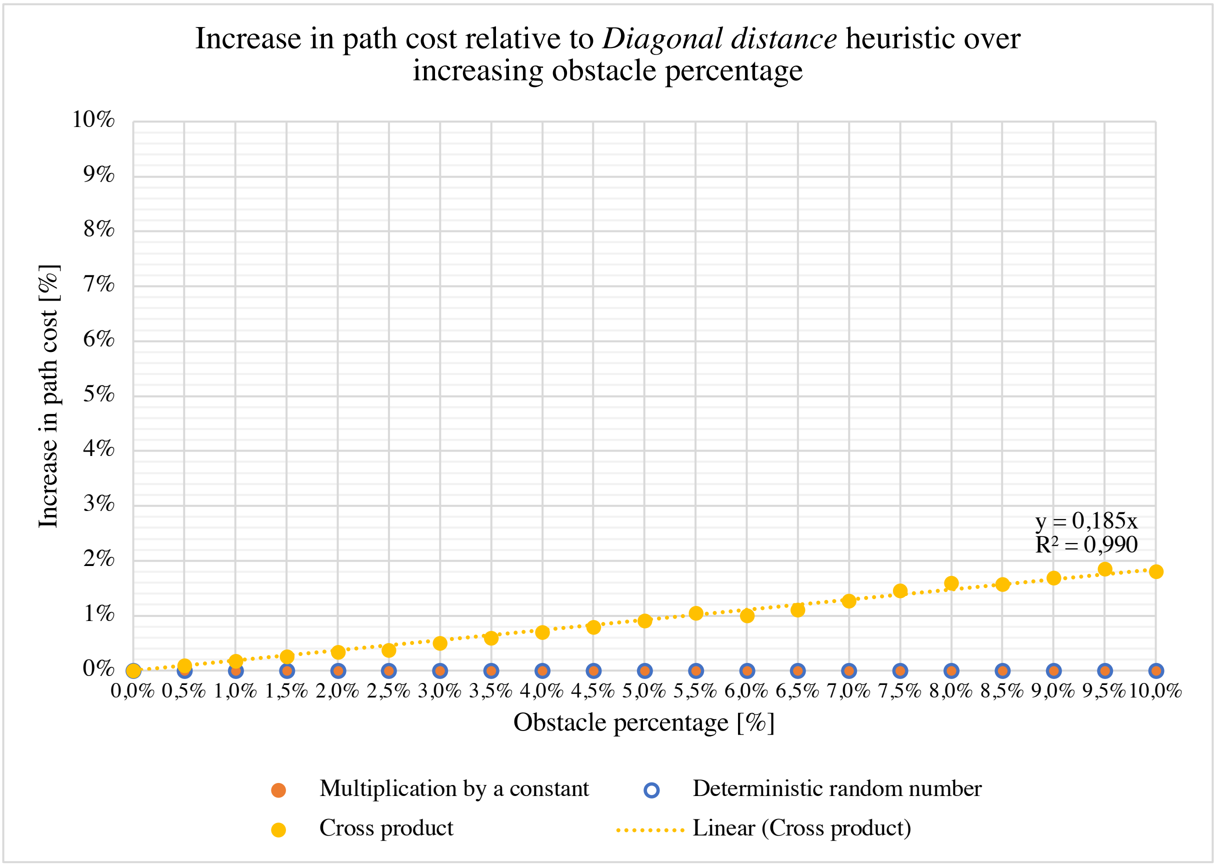

Figure 6.2

6.5.1. Multiplication by a constant

Figure 6.1 shows that, over the range 0.0% – 3.5% of obstacles, *Multiplication* provided large decreases in computational time relative to *Diagonal distance* heuristic. From 4% up to 6.5% of obstacles, the speed-ups were less significant. In fact, at 7% and 10%, *Multiplication* performed worse than *Diagonal distance* heuristic. A large speed-up was achieved at 8.5%. In the remaining cases, speed-ups were modest yet still superior to those achieved by the other modifications.

The gradual decrease in performance is likely to have been caused by the fact that *Multiplication* creates a bias towards nodes with lower estimated cost of reaching the goal. With increasing obstacle percentage, the paths consisting of such nodes are more likely to become blocked by obstacles. This may require the algorithm to evaluate alternative paths, resulting in a less effective search. Furthermore, as illustrated in *Figure 6.2*, the cost of paths found by

Multiplication was optimal. However, as this modification breaks admissibility, the modified algorithm is not guaranteed to find optimal paths.

6.5.2. Addition of a deterministic random number

As shown in *Figure 6.1*, *Deterministic* provided inconsistent performance. Over the whole range of obstacle percentages, this modification led to speed-ups and slow-downs with no clear relationship between obstacle percentage and computational time. This inconsistency is due to the fact that this method involves merely an addition of a pseudo-random number to the nodes' f costs in tie-breaking. Being randomly generated, such a number provides no additional information to the algorithm. This deems the search strategy haphazard and considerably less effective than the other modifications and, in some cases, the underlying admissible *Diagonal distance* heuristic itself. Similarly as in *Multiplication*, *Deterministic* returned optimal paths. However, this modification too breaks admissibility.

6.5.3. Cross product

Figure 6.1 shows that, over the range 0.0% – 3.5% of obstacles, *Cross product* provided large relative decreases in computational time. For 4% of obstacles and higher, *Cross*'s performance deteriorated, providing either a negligible speed-up in comparison with *Multiplication*, or a relative slow-down.

The rapid decrease in performance is most likely to have been caused by the bias created by this modification for nodes that lie along the start-goal vector. With increasing obstacle percentage, the paths lying along the start-goal vector may become blocked. Due to its bias for nodes contained in such paths, the algorithm may carry out a less effective search.

As exemplified by *Figure 6.2*, the cost of paths found by *Cross* was increasing approximately linearly with obstacle percentage. The average increase in path cost relative to the optimal *Diagonal distance* heuristic was 0.91%. However, it can be argued that such increase in path cost may be negligible in use cases such as Game AI, which are more likely to benefit from the speed-up offered by this modification. Nevertheless, the increasing trend in path cost indicates that for higher percentages, *Cross* may return paths that are too costly in comparison with optimal paths.

6.6. Evaluation

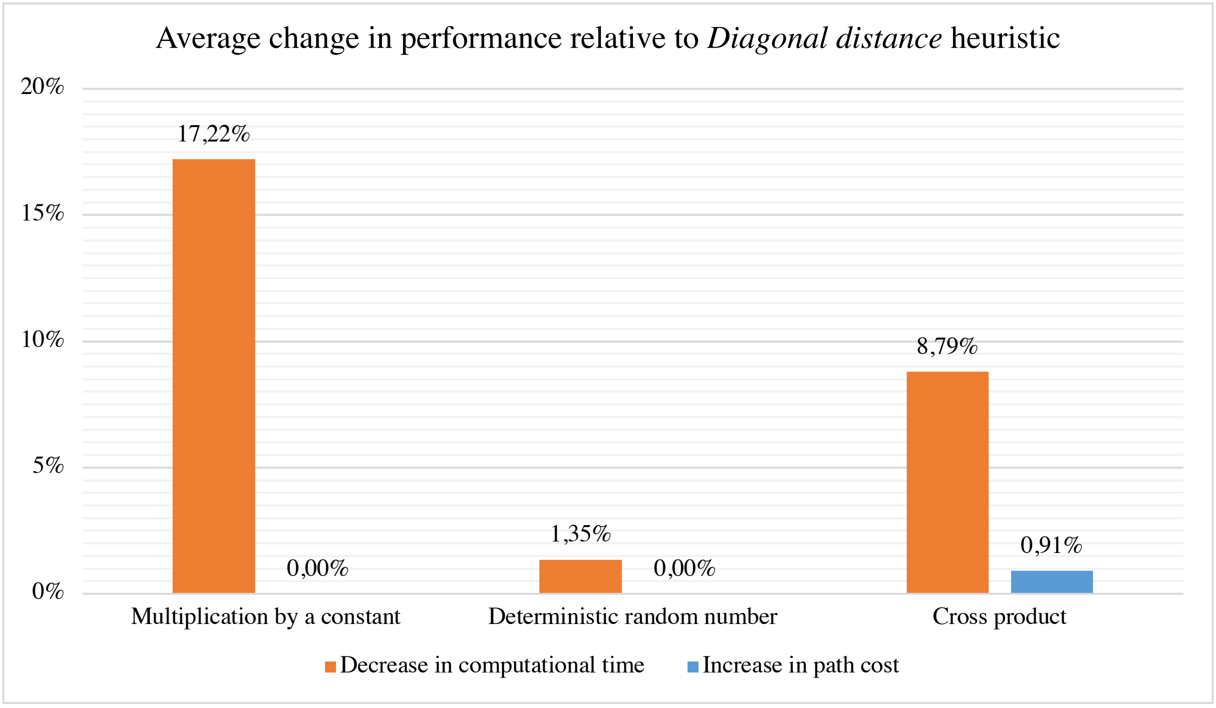

Figure 6.3

As illustrated in *Figure 6.3*, on average, *Multiplication* provided the greatest increase in performance of 17.22% while returning paths of identical cost to *Diagonal distance* heuristic. *Cross* provided approximately half the speed-up of *Multiplication* and returned paths that were on average 0.91% more costly than those found by the *Diagonal distance heuristic*. By contrast, *Deterministic random number addition* resulted in a little average speed-up of 1.35% while returning paths that were optimal. However, *Multiplication* also returned optimal paths and together with *Cross* provided a much higher increase in performance. This dichotomy in performance is most likely due to the fact that while *Multiplication* and *Cross* both created a specific bias for the algorithm, *Deterministic* did not provide any additional information, deeming the search strategy ineffective.

Therefore, on average, *Multiplication* and *Cross* are highly likely to provide a performance increase superior to that of *Deterministic*. Out of these two, *Multiplication* provided a greater speed-up relative to *Diagonal distance* heuristic over a wider range of path symmetry scenarios represented by increasing obstacle percentages, while returning optimal paths. Therefore, for obstacle percentages ranging up to 6.5%, *Multiplication by a constant* is a powerful modification of the slower, admissible *Diagonal distance* heuristic.

6.7. Possible limitations

Firstly, dataset generation was done using a pseudo-random number generator to determine whether a tile should be marked as passable or as an obstacle. This approach is very unlikely to generate a map containing the desired percentage of obstacles and the deviation in obstacle percentages in the testing system will have affected the accuracy of results. Secondly, the number of calls to the A* function for each start-goal scenario in the dataset only provides a rough average value.

Moreover, process priority for the testing system was not manipulated. An adjustment thereof could have led to different results being obtained. Furthermore, memory management was left to the built-in garbage collector, whose periodic activity may have affected the results by infringing upon performance overhead during computation. Moreover, the use of pre-implemented data structures might have led to unexpected behaviour and may thus have affected the results. In addition, the pseudo-random number generator used in the modification *Deterministic* was a function native to Python's random module. The use of a different function may have led to different results. In addition, temporary results were stored in dynamic arrays during testing. If such an array's capacity is used up, data migration to a new dynamic array follows. This process would have affected the algorithm's performance.

Lastly, grid dimensions were set to 192×108 tiles, and movement costs were set to 10 and 14, respectively. However, different use cases may require different values, possibly leading to different results.

7. Conclusion

The aim of this essay was to determine the extent to which a change in the heuristic may improve the efficiency of the A* algorithm on 8-connected undirected uniform-cost grid maps with search-space symmetry. The results of the experiment suggest that the greatest increase in efficiency can be achieved by multiplying the h cost by a constant. This method offered a considerable speed-up of 17.22% and showed consistent performance across a range of path symmetry scenarios, while returning optimal paths and maintaining A*'s popular ease of implementation.

7.1. Future study

Firstly, the behaviour of the three modifications could be studied over a range of obstacle percentages beyond 10%. Another option would be the use of the g cost in tie-breaking in a fashion similar to that discussed by this paper. A more advanced possibility would be the employment of different search space representations such as Triangular and Hexagonal grids. Yet another option would be to analyse recently-developed complex algorithms such as *Jump Point Search* and *Rectangle Expansion A**.

8. Bibliography

Dechter, R. & Pearl, J., 1985. Generalized Best-First Search Strategies and the. *Journal of the ACM,* July, 32(3), p. 507.

Ericson, C., 2008. *Don't follow the shortest path!.* [Online] Available at: http://realtimecollisiondetection.net/blog/?p=56 [Accessed 7 July 2018].

Fischer, J., 2015. *Let's build GameplayKit - Grid Based Pathfinding.* [Online] Available at: http://www.jonathanfischer.net/archives/ [Accessed 7 July 2018].

Growing with the Web, 2012. *A* pathfinding algorithm.* [Online] Available at: http://www.growingwiththeweb.com/2012/06/a-pathfinding-algorithm.html [Accessed 6 July 2018].

Harabor, D., 2011. Graph Pruning and Symmetry Breaking on Grid Maps. In: *IJCAI 2011 - 22nd International Joint Conference on Artificial Intelligence.* Barcelona: Association for the Advancement of Artificial Intelligence, pp. 2816-2817.

Harabor, D., 2018. *E-mail correspondence,* s.l.: s.n.

Harabor, D. & Botea, A., 2010. Breaking Path Symmetries on 4-Connected Grid Maps. In: M. G. Youngblood & V. Bulitko, eds. *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment.* s.l.:The AAAI Press.

Harabor, D., Botea, A. & Kilby, P., 2011. Path Symmetries in Undirected Uniform-Cost Grids. In: *SARA.* s.l.:s.n.

Harabor, D. & Grastien, A., 2011. *Online Graph Pruning for Pathfinding on Grid Maps.* San Francisco: National Conference on Artificial Intelligence.

Champandard, A. J., 2011. *The Mechanics of Influence Mapping: Representation, Algorithm & Parameters.* [Online] Available at: http://aigamedev.com/open/tutorial/influence-map-mechanics/#a2DGrids [Accessed 20 July 2018].

López, L. C., 2013. *How does an admissible heuristic ensure an optimal solution?.* [Online] Available at: https://cs.stackexchange.com/questions/16065/how-does-an-admissible-heuristic-ensure-an-optimal-solution [Accessed 7 July 2018].

Madhav, S., 2014. Artificial Intelligence. In: *Game Programming Algorithms and Techniques: A Platform-Agnostic Approach.* s.l.: s.n.

Nosrati, M., Karimi, R. & Hasanvand, H. A., 2012. Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches. *World Applied Programming (WAP) journal,* April, 2(4), pp. 251-256.

Patel, A., 2006. *Grids.* [Online] Available at: http://www-cs-students.stanford.edu/~amitp/game-programming/grids/
[Accessed 20 July 2018].

Patel, A., 2009. *Heuristics.* [Online] Available at: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#breaking-ties [Accessed 5 July 2018].

Patel, A., 2010. *Introduction to A*.* [Online] Available at: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html [Accessed 6 July 2018].

Patel, A., 2014a. *Grids and Graphs.* [Online] Available at: https://www.redblobgames.com/pathfinding/grids/graphs.html
[Accessed 6 July 2018].

Patel, A., 2014b. *Introduction to A*.* [Online] Available at: http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#dijkstras-algorithm-and-best-first-search [Accessed 6 July 2018].

Patel, A., 2018a. *E-mail correspondence,* s.l.: s.n.

Patel, A., 2018b. *Map representations.* [Online] Available at: http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html [Accessed 6 November 2018].

Rouse, M, 2011. *Pseudo-random number generator (PRNG).* [Online] Available at: https://whatis.techtarget.com/definition/pseudo-random-number-generator-PRNG [Accessed 14 September 2018].

Russell, S. J. & Norvig, P., 2010. Introduction. In: *Artificial Intelligence: A Modern Approach.* 3rd ed. Upper Saddle River, New Jersey: Pearson Education, Inc., p. 1.

StatCounter, 2018. *Desktop Screen Resolution Stats Worldwide.* [Online] Available at: http://gs.statcounter.com/screen-resolution-stats/desktop/worldwide [Accessed 31 October 2018].

Taylor, C., n.d. *Robotics: Computation Motion Planning - A* algorithm.* [Online] Available at: https://www.coursera.org/lecture/robotics-motion-planning/1-4-a-algorithm-Vv9fL [Accessed 7 July 2018].

Weisstein, E., 2018. *Cross Product.* [Online] Available at: http://mathworld.wolfram.com/CrossProduct.html [Accessed 9 July 2018].

Wikipedia, 2008. *Cross product.* [Online] Available at: https://en.wikipedia.org/wiki/Cross_product#/media/File:Cross_product_parallelogram.svg [Accessed 9 July 2018].

Yap, P., 2002. Grid-Based Path-Finding. In: R. Cohen & B. Spencer, eds. *Advances in Artificial Intelligence.* Edmonton: Department of Computing Science, University of Alberta, p. 44.

9. Appendix 1: Source code for the testing system

9.1. Constants

```
import numpy as np

LOCATION_OF_FILES = 'data/benchmarks/'
LOCATION_OF_RESULTS = 'data/results/'

MIN_OBSTACLE_PERCENTAGE = 0
MAX_OBSTACLE_PERCENTAGE = 10
STEP = 0.5
OBSTACLE_PERCENTAGES = np.arange(MIN_OBSTACLE_PERCENTAGE,
                                 MAX_OBSTACLE_PERCENTAGE + 0.01,
                                 STEP)

OBSTACLE = '1'
PASSABLE = '0'
DS = 10
DD = 14

NUM_OF_MAPS = 10
NUM_OF_SCENARIOS = 20
NUM_OF_RUNS = 10
MAP_WIDTH = 192
MAP_HEIGHT = 108

CONSTANT = 1.0 + 1/(MAP_WIDTH + MAP_HEIGHT)  # maximum expected path length is
given by DS/(DS*(MAP_WIDTH + MAP_HEIGHT))


offsets_cardinal = {
    'N': (0, -1),
    'E': (1, 0),
    'S': (0, 1),
    'W': (-1, 0),
}

offsets_intermediate = {
    ('N', 'W'): (-1, -1),
    ('N', 'E'): (1, -1),
    ('S', 'W'): (-1, 1),
    ('S', 'E'): (1, 1)
}
```

9.2. The A* algorithm and modifications of the heuristic function

```
# class PriorityQueue, function reconstruct_path and the core loop of a_star_search
function along with nested functions h_diagonal_distance and h_cross_product
retrieved from https://www.redblobgames.com/pathfinding/a-star/, published in 2014
by Red Blob Games

import heapq
from random import random

from Constants import *
from Visualization import visualize


class PriorityQueue:
    def __init__(self):
        self.elements = []

    def empty(self):
        return len(self.elements) == 0

    def put(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def get(self):
        return heapq.heappop(self.elements)[1]


def reconstruct_path(came_from, start, goal):
    current = goal
    path = []
    while current != start:
        path.append(current)
        current = came_from[current]
    path.append(start)
    return path


def a_star_search(graph, start, goal, heuristic_name):

    def h_diagonal_distance():
        # (x1, y1) = goal
        # (x2, y2) = next
        # dx = abs(x1 − x2)
        # dy = abs(y1 − y2)
        dx = abs(goal[0] − next[0])
        dy = abs(goal[1] − next[1])
        return DS ∗ (dx + dy) + (DD − 2 ∗ DS) ∗ min(dx, dy)

    def h_multiplication_by_constant():
        return h_diagonal_distance() ∗ CONSTANT

    def h_deterministic_random_number():
        return h_diagonal_distance() + random()

    def h_cross_product():
        # (x1, y1) = goal
        # (x2, y2) = next
        # (x3, y3) = start
        # dx1 = x2 − x1
        # dy1 = y2 − y1
        # dx2 = x3 − x1
        # dy2 = y3 − y1
        dx1 = next[0] − goal[0]
        dy1 = next[1] − goal[1]
        dx2 = start[0] − goal[0]
        dy2 = start[1] − goal[1]
        return h_diagonal_distance() + abs(dx1 ∗ dy2 − dx2 ∗ dy1)

    def neighbors(current_node):

        def is_passable(node):
            (x, y) = node
            return graph[y][x] == PASSABLE

        list_of_neighbors = []
        passable = {'N': False, 'E': False, 'S': False, 'W': False}  # offsets for
cardinal directions

        # process cardinal directions
        for key in offsets_cardinal.keys():
            offset = offsets_cardinal[key]
            offset_node = tuple([sum(x) for x in zip(current_node, offset)])
            if is_passable(offset_node):
                list_of_neighbors.append((offset_node, DS))
                passable[key] = True

        # process intermediate directions based on cardinal
        for dir1, dir2 in [('N', 'W'),
                           ('N', 'E'),
                           ('S', 'W'),
                           ('S', 'E')]:
            if passable[dir1] and passable[dir2]:
                offset = offsets_intermediate[(dir1, dir2)]
                offset_node = tuple([sum(x) for x in zip(current_node, offset)])
                if is_passable(offset_node):
                    list_of_neighbors.append((offset_node, DD))

        return list_of_neighbors

    heuristic_list = {
        'DIAGONAL': h_diagonal_distance,
        'MULTIPLICATION': h_multiplication_by_constant,
        'DETERMINISTIC': h_deterministic_random_number,
        'CROSS': h_cross_product
    }

    frontier = PriorityQueue()
    frontier.put(start, 0)
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0
    heuristic = heuristic_list[heuristic_name]

    while not frontier.empty():
        current = frontier.get()

        if current == goal:
            # alternatively request visualization
            return cost_so_far[goal]

        for next, cost_to_next in neighbors(current):
            new_cost = cost_so_far[current] + cost_to_next
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = new_cost + heuristic()
                frontier.put(next, priority)
                came_from[next] = current
    return 0
```

9.3. Dataset generation

```
from os import path, makedirs
from random import randint
from Astar import *
from Constants import *


def create_folders():
    for x in OBSTACLE_PERCENTAGES:
        try:
            path = 'data/benchmarks/OP_{}%'.format(x)
            makedirs(path)
        except FileExistsError:
            continue
```

```
def get_random_map(obstacle_percentage):
    obstacle_percentage /= 100
    line_of_obstacles = [OBSTACLE for _ in range(MAP_WIDTH + 1)]
    map_array = [line_of_obstacles]
    num_of_obstacles = 0

    for _ in range(MAP_HEIGHT):
        line = []
        for _ in range(MAP_WIDTH):
            if random() <= obstacle_percentage:
                line.append(OBSTACLE)
                num_of_obstacles += 1
            else:
                line.append(PASSABLE)
        line.append(OBSTACLE)
        map_array.append(line)

    map_array.append(line_of_obstacles)
    obstacle_percentage = num_of_obstacles*100/(MAP_HEIGHT * MAP_WIDTH)
    print('\tGenerated obstacle percentage: {}%'.format(round(obstacle_percentage,
2)))

    return map_array


def generate_benchmarks():
    for obstacle_percentage in OBSTACLE_PERCENTAGES:
        print('\nNow generating maps with {}% of
obstacles'.format(obstacle_percentage))
        for map_num in range(NUM_OF_MAPS):
            map_array = get_random_map(obstacle_percentage)
            scenarios = set()
            while len(scenarios) != NUM_OF_SCENARIOS:
                while True:
                    x1 = randint(0, MAP_WIDTH)
                    y1 = randint(0, MAP_HEIGHT)
                    start = (x1, y1)
                    x2 = randint(0, MAP_WIDTH)
                    y2 = randint(0, MAP_HEIGHT)
                    goal = (x2, y2)
                    if map_array[y1][x1] != OBSTACLE and map_array[y2][x2] !=
OBSTACLE and start != goal:
                        break
                path_cost = a_star_search(map_array, start, goal, 'DIAGONAL')
                if path_cost != 0:
                    scenarios.add((start, goal))

            file_name = 'OP_{}%/map_{}.txt'.format(obstacle_percentage, map_num)
            with open(path.join(LOCATION_OF_FILES, file_name), 'w') as out:
                for (x1, y1), (x2, y2) in scenarios:
                    out.write('{};{};{};{}\n'.format(x1, y1, x2, y2))
                for line in map_array:
                    out.write(OBSTACLE + ''.join([char for char in line]) + '\n')


if __name__ == '__main__':
    create_folders()
    generate_benchmarks()
```

9.4. Testing system

```
import cProfile
from os import path
import pstats
from Astar import *
from Constants import *

heuristic_functions = ['DIAGONAL', 'MULTIPLICATION', 'DETERMINISTIC', 'CROSS']

if __name__ == '__main__':
    for heuristic_function in heuristic_functions:
        file_name = 'results_{}.txt'.format(heuristic_function.lower())
        with open(path.join(LOCATION_OF_RESULTS, file_name), 'w') as out:
            out.write('%;Avg path cost;Avg comp time\n')
            for obstacle_percentage in OBSTACLE_PERCENTAGES:
                print('Obstacles: {}%'.format(obstacle_percentage))
                print('\tHeuristic: {}'.format(heuristic_function))
                running_times = []
                path_costs = []
                for map_number in range(NUM_OF_MAPS):
                    print('\t\tMap no.: {} '.format(map_number))
                    file_name = 'OP_{}%/map_{}.txt'.format(obstacle_percentage,
map_number)
                    with open(path.join(LOCATION_OF_FILES, file_name), 'r') as
raw_map:
                        starts = []
                        goals = []

                        for _ in range(NUM_OF_SCENARIOS):
                            coordinates = next(raw_map).split(';')
                            coordinates = list(map(int, coordinates))
                            start = tuple((coordinates[0], coordinates[1]))
                            goal = tuple((coordinates[2], coordinates[3]))
                            starts.append(start)
                            goals.append(goal)

                        map_array = [list(line) for line in raw_map]

                        for x in range(NUM_OF_SCENARIOS):
                            print('\t\t\tScenario: {}'.format(x+1))
                            for _ in range(NUM_OF_RUNS):
                                pr = cProfile.Profile()
                                pr.run('a_star_search(map_array, starts[x],
goals[x], heuristic_function)')
                                ps = pstats.Stats(pr)
                                running_times.append(ps.total_tt)
                            path_cost = a_star_search(map_array, starts[x],
goals[x], heuristic_function)
                            path_costs.append(path_cost)

                avg_running_time =
round(sum(running_times)/len(running_times)*1000, 3)
                avg_path_length = round(sum(path_costs) / len(path_costs), 3)
                out.write('{};{};{}\n'.format(obstacle_percentage, avg_path_length,
avg_running_time))
```

9.5. Visualisation

```
import matplotlib.pyplot as plt
from matplotlib import colors
import numpy as np
```

```python
def visualize(graph, heuristic, expanded_nodes, final_path, path_cost):
    # 0 - free space
    # 1 - obstacle
    # 2 - expanded node
    # 3 - node on path
    goal = final_path[0]
    start = final_path[-1]

    for node in expanded_nodes.keys():
        (x, y) = node
        graph[y][x] = '2'
    for node in final_path:
        (x, y) = node
        graph[y][x] = '3'
    for node in (start, goal):
        (x, y) = node
        graph[y][x] = '4'

    data = np.array(graph)
    data = np.delete(data, len(graph[0])-1, 1)
    data = data.astype(int)

    cmap = colors.ListedColormap(['white', 'black', 'yellow', 'green', 'purple'])
    bounds = [0, 1, 2, 3, 4, 5]
    norm = colors.BoundaryNorm(bounds, cmap.N)

    fig, ax = plt.subplots()
    ax.imshow(data, cmap=cmap, norm=norm)
    ax.set_title('Heuristic {}\n'
                 'Path cost {}\n'
                 'Visited nodes {}'
                 .format(heuristic.lower(), path_cost, len(expanded_nodes.keys())))
    ax.grid(which='major', axis='both', linestyle='-', color='k', linewidth=0.5)

    plt.show()
```

10. Appendix 2: Output data

10.1. Diagonal distance heuristic

Diagonal distance heuristic				
%	Avg path cost	Avg comp time	Inc in path cost	Dec in comp time
0.0%	769.69	26.21	0.00%	0.00%
0.5%	850.61	23.20	0.00%	0.00%
1.0%	843.73	24.47	0.00%	0.00%
1.5%	851.74	35.26	0.00%	0.00%
2.0%	844.93	30.53	0.00%	0.00%
2.5%	830.87	30.29	0.00%	0.00%
3.0%	847.09	38.15	0.00%	0.00%
3.5%	812.48	31.66	0.00%	0.00%
4.0%	785.00	39.26	0.00%	0.00%
4.5%	771.77	45.97	0.00%	0.00%
5.0%	833.32	41.83	0.00%	0.00%
5.5%	796.75	46.32	0.00%	0.00%
6.0%	779.09	51.47	0.00%	0.00%
6.5%	761.97	51.17	0.00%	0.00%
7.0%	724.75	53.24	0.00%	0.00%
7.5%	763.61	50.87	0.00%	0.00%
8.0%	812.35	78.13	0.00%	0.00%
8.5%	782.11	50.28	0.00%	0.00%
9.0%	814.55	60.06	0.00%	0.00%
9.5%	766.76	77.43	0.00%	0.00%
10.0%	739.58	73.42	0.00%	0.00%
Average	**799.18**	**45.68**	-	-

10.2. Multiplication by a constant modification

	Multiplication by a constant			
%	Avg path cost	Avg comp time	Inc in path cost	Dec in comp time
0.0%	769.69	13.75	0.00%	47.54%
0.5%	850.61	11.14	0.00%	51.99%
1.0%	843.73	13.34	0.00%	45.49%
1.5%	851.74	21.95	0.00%	37.74%
2.0%	844.93	19.95	0.00%	34.66%
2.5%	830.87	21.16	0.00%	30.15%
3.0%	847.09	24.92	0.00%	34.69%
3.5%	812.48	23.88	0.00%	24.58%
4.0%	785.00	35.72	0.00%	9.02%
4.5%	771.77	40.23	0.00%	12.48%
5.0%	833.32	37.97	0.00%	9.24%
5.5%	796.75	44.25	0.00%	4.47%
6.0%	779.09	47.75	0.00%	7.23%
6.5%	761.98	48.98	0.00%	4.26%
7.0%	724.75	53.90	0.00%	-1.25%
7.5%	763.61	49.79	0.00%	2.12%
8.0%	812.35	76.28	0.00%	2.36%
8.5%	782.11	46.17	0.00%	8.16%
9.0%	814.56	59.19	0.00%	1.45%
9.5%	766.76	76.70	0.00%	0.94%
10.0%	739.58	77.67	0.00%	-5.79%
Average	**799.18**	**40.22**	0.00%	17.22%

10.3. Deterministic random number modification

Deterministic random number				
%	Avg path cost	Avg comp time	Inc in path cost	Dec in comp time
0.0%	769.69	23.47	0.00%	10.45%
0.5%	850.61	22.48	0.00%	3.10%
1.0%	843.73	25.74	0.00%	-5.19%
1.5%	851.74	30.18	0.00%	14.40%
2.0%	844.93	29.40	0.00%	3.71%
2.5%	830.87	30.26	0.00%	0.09%
3.0%	847.09	33.56	0.00%	12.04%
3.5%	812.48	30.29	0.00%	4.32%
4.0%	785.00	40.84	0.00%	-4.02%
4.5%	771.77	44.79	0.00%	2.56%
5.0%	833.32	43.38	0.00%	-3.71%
5.5%	796.75	47.44	0.00%	-2.42%
6.0%	779.09	49.84	0.00%	3.18%
6.5%	761.97	51.15	0.00%	0.03%
7.0%	724.75	55.29	0.00%	-3.86%
7.5%	763.61	52.38	0.00%	-2.98%
8.0%	812.35	78.19	0.00%	-0.08%
8.5%	782.11	49.23	0.00%	2.09%
9.0%	814.55	60.62	0.00%	-0.94%
9.5%	766.76	77.05	0.00%	0.49%
10.0%	739.58	76.99	0.00%	-4.87%
Average	**799.18**	**45.36**	0.00%	1.35%

10.4. Cross product modification

Cross product				
%	Avg path cost	Avg comp time	Inc in path cost	Dec in comp time
0.0%	769.69	12.78	0.00%	51.26%
0.5%	851.44	11.37	0.10%	50.98%
1.0%	845.16	16.29	0.17%	33.41%
1.5%	853.89	22.84	0.25%	35.21%
2.0%	847.76	22.53	0.33%	26.22%
2.5%	833.92	26.28	0.37%	13.25%
3.0%	851.37	27.84	0.51%	27.05%
3.5%	817.28	23.94	0.59%	24.41%
4.0%	790.51	40.00	0.70%	-1.88%
4.5%	777.91	44.06	0.80%	4.16%
5.0%	840.89	41.82	0.91%	0.02%
5.5%	805.10	48.87	1.05%	-5.51%
6.0%	786.88	50.70	1.00%	1.50%
6.5%	770.43	53.04	1.11%	-3.66%
7.0%	733.96	60.63	1.27%	-13.89%
7.5%	774.73	54.81	1.46%	-7.75%
8.0%	825.34	85.34	1.60%	-9.23%
8.5%	794.42	52.41	1.57%	-4.24%
9.0%	828.31	62.01	1.69%	-3.25%
9.5%	780.94	87.84	1.85%	-13.46%
10.0%	752.90	88.06	1.80%	-19.94%
Average	**806.33**	**44.45**	0.91%	8.79%