Chris Zachariah | cvz2
Jason Chou | jc2280
Myles Chou | mc1998
Tyler Hong | th517
CS440: Introduction to Artificial Intelligence
Project 1: Heuristic Search

Report

1. **Graphical User Interface Overview:**

The Graphical User Interface, that we designed and built, displays the grid and all the functions needed in order to run multiple searches with heuristics and weights. The application is pre-loaded with all 50 benchmark grids that were used to obtain results for other portions of this report. It allows for the user to load saved maps (i.e. the benchmark grids), generate new maps and save the new maps.

The grid accurately displays all the different terrains using a range of colors that are all described in the legend at the bottom right corner once the application starts running. Once the user has called for a specific A*-family search to be run on the grid, the resulting path and the cells visited are clearly displayed. The user then has the ability to click on each cell and obtain information such as: the coordinates of the cell, the type of the cell and the g-cost, h-cost and f-cost of the cell.

2. **Abstract Search Implementation Overview and Description:**

A modular approach was taken in order to develop the code to run A* , Weighted A* and Uniform-Cost Search. Using the inheritance properties of Java, an abstract class called SearchAlgo was used in order to store most of the implementation for each of the search algorithms. A* has the ability to override and choose a heuristic and obtain a h-cost that would help determine the next steps for the search. Weighted A* has the ability to override and choose a heuristic and also choose a weight in order to use a greedy approach to obtain the most optimal path using the heuristic. Lastly, Uniform-Cost Search does not choose a heuristic and simply uses the g-cost in order to find the path to the goal.

For these algorithms to run, find and create the path from start to goal, a heap (priority queue in Java) was used as the fringe in order to store cells to be looked at and a

hashset was used as a closed set in order to store the cells that have been already looked at and visited. At every iteration of the algorithm, the lowest element (the top of the heap) is popped off and looked at in order to make decisions on the course of the path to take. If the current cell being looked at is not the goal cell, then the new g-cost is determined from the current cell to the new neighbor and put into the fringe if and only if it the new g-cost is less than the g-cost of that new neighbor (each cell has an initial g-cost of Integer.MAX_VALUE). The heap is initialized with a compare function that compares the f-costs of each cell in order to place the cells in the correct place and keep the lowest f-cost at the top of the heap for quick and easy retrevial.

Weighted A* takes a weight greater than 1.0 which will be multiplied by the h-value that trades off optimality for speed. A* does not do this. Uniform-Cost search does not use a heuristic but goes by the g-cost (cost between neighboring cells).

3. Optimizations:

After finishing with our implementation of the search algorithms we identified an area in which we could improve the runtime of our code. We used hashsets whenever possible in order to have an O(1) search, insert, and deletion. This is exemplified in our grid traversal, where all visited cells are stored into a hash set. Given the lengthy and diverse requirements for the project, we had to carefully choose a universal way to optimize our code to stay within the given time constraints.

To further optimize we used a priority queue (heap) in our implementation in order to maintain O(1) access to the minimum f-cost value stored in the heap. Our implementation of the grid used a 120x160 2D-array of Cells that allows for O(1) search, insert and deletion given the right coordinates. This also keeps the runtime shorter at the cost of the adding to the space complexity of the program.

4. Heuristics:

Regarding our group's choices of heuristics, we selected the Manhattan Distance, Euclidean Distance, Chebyshev Distance (also known as the the Diagonal Distance), Manhattan Distance by Four, and the Euclidean Distance by Four.

Of these five, we proposed, through our research, the Manhattan Distance would be the most consistent and admissible heuristic for this grid world. We believe this is the case because it is hypothesized to find the shortest path with the least amount of cells visited. Computationally, it finds the sum of the distance of the current cell to the goal cells which makes it inexpensive to compute and saves on runtime and space in comparison to the other selected heuristics.

Regarding the four other chosen heuristics, the Chebyshev and Euclidean Heuristics are hypothesized to also obtain the shortest path, but at the cost of runtime and space because of the slightly more expensive computation and greater number of visits to cells. The Euclidean Distance is known to be a good heuristic to use when the metrics are small, but for the metrics of our grid, we believe that the heuristic would not be as optimal as compared to the Manhattan Distance. Essentially, it is equivalent to taking the coordinates of the current cell and the goal cell and finding the direct distance to it. In other words, one can think of it as making a triangle on the grid and finding the hypotenuse.

The Chebyshev Distance is also similar to the Euclidean Distance. It measures the difference in magnitude of both the x and y axis and uses the greater of the two. Again, going back to the idea of making a triangle on the grid, the Chebyshev will find the distance of one of the other 2 sides of the triangle which will be less than or equal to the hypotenuse. This may not be the best approximation for our grid world. This is our justification for including the two heuristics in order to compare to the proposed optimal Manhattan heuristic.

The last two chosen heuristics, the Manhattan Distance by Four and the Euclidean Distance by Four, are deemed inadmissible because we propose that it would be grossly inefficient in finding the shortest path to the target, requiring the algorithm to search almost every cell in the grid. We believe this is the case because the values obtained

would be much less (lower h-cost) and it would not help the algorithm distinguish between better cells to add to the fringe and explore.

Increasing the number of heuristics gives more data, and particularly inefficient heuristics highlight the strengths of computationally inexpensive heuristics, which was our reason for including the last two heuristics. In concluding remarks, we chose three heuristics (Manhattan, Euclidean, Chebyshev) which were comparative in efficiency and two heuristics (Manhattan By Four, Euclidean By Four) which were decidedly less efficient and to have more diverse comparisons and trials.

Heuristic Equations:

Manhattan Distance:

Given coordinates: current cell = (a,b) and goal cell = (c,d)

Manhattan Distance = | a - c | + | b - d |

Euclidean Distance:

Given the coordinates: current cell = (a,b) and goal cell = (c,d)

Euclidean Distance = $\sqrt{(a - c)^2 + (b - d)^2}$

Chebyshev (Diagonal) Distance:

Given the coordinates: current cell = (a,b) and goal cell - (c,d)

Chebyshev Distance = max(| a - c | , | b - d |)

Manhattan Distance By Four:

Given coordinates: current cell = (a,b) and goal cell = (c,d)

Manhattan Distance By Four = 0.25 * (| a - c | + | b - d |)

Euclidean Distance By Four:

Given the coordinates: current cell = (a,b) and goal cell = (c,d)

Euclidean Distance By Four = $0.25 * \sqrt{(a - c)^2 + (b - d)^2}$

5. A* Family Search Algorithm Results (Average Across 50 Benchmarks)

A* : Manhattan Distance

Average Runtime : 2.14 milliseconds

Average Path Length : 129.8 Cells

Average Number of Cells Visited : 709.7 Cells

Average Cost of Path : 117.5619641

A* : Euclidean Distance

Average Runtime : 4.44 milliseconds

Average Path Length : 130.4 Cells

Average Number of Cells Visited : 1257.26 Cells

Average Cost of Path : 106.2101157

A* : Chebyshev Distance

Average Runtime : 6.86 milliseconds

Average Path Length : 133.7 Cells

Average Number of Cells Visited : 2393.82 Cells

Average Cost of Path : 106.2959741

A* : Manhattan Distance By Four

Average Runtime : 8.34 milliseconds

Average Path Length : 138.54 Cells

Average Number of Cells Visited : 2306.32 Cells

Average Cost of Path : 106.3930044

A* : Euclidean Distance By Four

Average Runtime : 21.6 milliseconds

Average Path Length : 145.68 Cells

Average Number of Cells Visited : 9407.78 Cells

Average Cost of Path : 95.57532434

Weighted A* : Manhattan Distance : 1.5

Average Runtime : 0.96 milliseconds

Average Path Length : 110.64 Cells

Average Number of Cells Visited : 196.32 Cells

Average Cost of Path : 154.48685546875

Weighted A* : Manhattan Distance : 3.0

Average Runtime : 0.60 milliseconds

Average Path Length : 109.36 Cells

Average Number of Cells Visited : 111.18 Cells

Average Cost of Path : 158.512412109375

Weighted A* : Euclidean Distance : 1.5

Average Runtime : 0.90 milliseconds

Average Path Length : 110.90 Cells

Average Number of Cells Visited : 162.26 Cells

Average Cost of Path : 125.951162109375

Weighted A* : Euclidean Distance : 3.0

Average Runtime : 0.64 milliseconds

Average Path Length : 105.4 Cells

Average Number of Cells Visited : 106.68 Cells

Average Cost of Path : 141.95734375

Weighted A* : Chebyshev Distance : 1.5

Average Runtime : 2.04 milliseconds

Average Path Length : 112.68 Cells

Average Number of Cells Visited : 590.0 Cells

Average Cost of Path : 141.48126953125

Weighted A* : Chebyshev Distance : 3.0

Average Runtime : 0.68 milliseconds

Average Path Length : 111.14 Cells

Average Number of Cells Visited : 117.56 Cells

Average Cost of Path : 148.8850390625

Weighted A* : Manhattan Distance By Four Distance : 1.5

Average Runtime : 5.16 milliseconds

Average Path Length : 124.34 Cells

Average Number of Cells Visited : 1795.58 Cells

Average Cost of Path : 142.466474609375

Weighted A* : Manhattan Distance By Four Distance : 3.0

Average Runtime : 1.3 milliseconds

Average Path Length : 114.0 Cells

Average Number of Cells Visited : 341.48 Cells

Average Cost of Path : 148.598525390625

Weighted A* : Euclidean Distance By Four Distance : 1.5

Average Runtime : 19.08 milliseconds

Average Path Length : 146.3 Cells

Average Number of Cells Visited : 7702.42 Cells

Average Cost of Path : 95.44998046875

Weighted A* : Euclidean Distance By Four Distance : 3.0

Average Runtime : 6.88 milliseconds

Average Path Length : 139.86 Cells

Average Number of Cells Visited : 2757.08 Cells

Average Cost of Path : 97.575908203125

Uniform Cost Search

Average Runtime : 24.16 milliseconds

Average Path Length : 145.84 Cells

Average Number of Cells Visited : 12135.86 Cells

Average Cost of Path : 95.960341796875

6. A* Family Search Algorithm Results Observation

Given the results recorded in the previous section, many different observations can be made. To start off, let's discuss the results from the A* Search trials.

It can observed that using the Manhattan Distance with the A* produced the best result for all categories except for the Average Cost of Path which was the highest of all the heuristics. This was a very surprising result that our group had not expected, as we hypothesized that the Manhattan Distance would outperform in every category as it computes the sum of the distance from the current cell to the goal cell, making it inexpensive. Retrospectively, the inexpensive cost of the heuristic may be the reason why the Average Cost of Path was not the most optimal. On the other hand, it is not surprising to see that it had the fastest runtime, lowest path length and lowest number of visited cells. The Euclidean Distance and Chebyshev Distance heuristics were also close in comparison to the Manhattan Distance with some slightly higher values for the runtime, path length and cells visited. Finally, the Manhattan Distance By Four and Euclidean Distance By Four Heuristics had some of the worst averages of all the other heuristics, however, to our surprise, the Euclidean Distance By Four had achieved the lowest average Cost of Path. This was a really interesting and astonishing result that we wanted to see if it would persist with Weighted A* Search as well.

With the Weighted A* Search our group used two different weights: 1.5 and 3.0. We compared the results separately and saw many differences. It is obvious that increasing the weight had an adverse effect on the results of the search. In all the cases presented, when increasing the weight from 1.5 to 3.0, the results were undoubtedly better which worked out exactly as expected. By using a bigger weight, it increases the h-cost for each cell which helps the algorithm distinguish between which cells will be the most optimal to use and put into the fringe to be explored (hence a greedy approach to

finding the optimal path). Among the different results, it was again surprising that some of the results from the A* Search trials persisted and also a new observation came about. For Weighted A*, for either weight, the Euclidean Distance Heuristic gave better results than the Manhattan Distance which was not expected. Again, as last time, the Euclidean Distance By Four gave the lowest overall Cost of Path for both weights.

After the series of trials, we came to the conclusion that the Manhattan Distance and Euclidean Distance was the two best overall Heuristics to use for the A* and Weighted A* Searches respectively. It seems that the Euclidean Distance works best with a weight multiplied to it and it is clearly shown in the results. On a surprising note, the Euclidean Distance By Four seems to keep finding the lowest Cost of Path for both A* and Weighted A*. These observations were definitely not what we hypothesized beforehand and it was an interesting turn of events.

Finally, the Uniform Cost Search was run on the benchmark girds and to no surprise, it was the worst of A*-family algorithms used in order to find the path from start to the goal. In all categories except for the Cost of Path, it had really high values, but the cost of path was relatively close to the Cost of Path computed by using the Euclidean Distance By Four Heuristic which is yet another interesting observation made in our trials.

7. Sequential Heuristic A* Search Algorithm Results (Average Across 50 Benchmarks)

Sequential Heuristic A* : Manhattan Distance : 1.5 : 1.5

Average Runtime : 2.42 milliseconds

Average Path Length : 111.0 Cells

Average Number of Cells Visited : 2250.2 Cells

Average Cost of Path : 125.3992578125

Sequential Heuristic A* : Manhattan Distance : 1.5 : 3.0

Average Runtime : 2.38 milliseconds

Average Path Length : 111.0 Cells

Average Number of Cells Visited : 2250.2 Cells

Average Cost of Path : 125.3992578125

Sequential Heuristic A* : Manhattan Distance : 3.0 : 1.5

Average Runtime : 1.98 milliseconds

Average Path Length : 105.24 Cells

Average Number of Cells Visited : 2112.86 Cells

Average Cost of Path : 142.29482421875

Sequential Heuristic A* : Manhattan Distance : 3.0 : 3.0

Average Runtime : 2.14 milliseconds

Average Path Length : 105.24 Cells

Average Number of Cells Visited : 2112.86 Cells

Average Cost of Path : 142.29482421875

Sequential Heuristic A* : Euclidean Distance : 1.5 : 1.5

Average Runtime : 2.52 milliseconds

Average Path Length : 110.4 Cells

Average Number of Cells Visited : 2273.74 Cells

Average Cost of Path : 143.13033203125

Sequential Heuristic A* : Euclidean Distance : 1.5 : 3.0

Average Runtime : 2.5 milliseconds

Average Path Length : 110.4 Cells

Average Number of Cells Visited : 2264.66 Cells

Average Cost of Path : 143.13033203125

Sequential Heuristic A* : Euclidean Distance : 3.0 : 1.5

Average Runtime : 2.3 milliseconds

Average Path Length : 107.52 Cells

Average Number of Cells Visited : 2206.22 Cells

Average Cost of Path : 147.989775390625

Sequential Heuristic A* : Euclidean Distance : 3.0 : 3.0

Average Runtime : 1.96 milliseconds

Average Path Length : 107.52 Cells

Average Number of Cells Visited : 2197.18 Cells

Average Cost of Path : 148.116982421875

Sequential Heuristic A* : Chebyshev Distance : 1.5 : 1.5

Average Runtime : 2.1 milliseconds

Average Path Length : 109.34 Cells

Average Number of Cells Visited : 2197.9 Cells

Average Cost of Path : 135.370546875

Sequential Heuristic A* : Chebyshev Distance : 1.5 : 3.0

Average Runtime : 2.24 milliseconds

Average Path Length : 109.34 Cells

Average Number of Cells Visited : 2197.9 Cells

Average Cost of Path : 135.370546875

Sequential Heuristic A* : Chebyshev Distance : 3.0 : 1.5

Average Runtime : 1.94 milliseconds

Average Path Length : 105.24 Cells

Average Number of Cells Visited : 2067.3 Cells

Average Cost of Path : 144.15998046875

Sequential Heuristic A* : Chebyshev Distance : 3.0 : 3.0

Average Runtime : 2.04 milliseconds

Average Path Length : 105.24 Cells

Average Number of Cells Visited : 2067.3 Cells

Average Cost of Path : 144.15998046875

Sequential Heuristic A* : Manhattan Distance By Four : 1.5 : 1.5

Average Runtime : 2.22 milliseconds

Average Path Length : 109.36 Cells

Average Number of Cells Visited : 2200.92 Cells

Average Cost of Path : 136.128349609375

Sequential Heuristic A* : Manhattan Distance By Four : 1.5 : 3.0

Average Runtime : 2.24 milliseconds

Average Path Length : 109.36 Cells

Average Number of Cells Visited : 2200.92 Cells

Average Cost of Path : 136.128349609375

Sequential Heuristic A* : Manhattan Distance By Four : 3.0 : 1.5

Average Runtime : 2.1 milliseconds

Average Path Length : 105.32 Cells

Average Number of Cells Visited : 2178.16 Cells

Average Cost of Path : 142.513134765625

Sequential Heuristic A* : Manhattan Distance By Four : 3.0 : 3.0

Average Runtime : 1.98 milliseconds

Average Path Length : 105.32 Cells

Average Number of Cells Visited : 2178.16 Cells

Average Cost of Path : 142.513134765625

Sequential Heuristic A* : Euclidean Distance By Four : 1.5 : 1.5

Average Runtime : 2.28 milliseconds

Average Path Length : 108.9 Cells

Average Number of Cells Visited : 2399.08 Cells

Average Cost of Path : 136.62513671875

Sequential Heuristic A* : Euclidean Distance By Four : 1.5 : 3.0

Average Runtime : 2.28 milliseconds

Average Path Length : 108.9 Cells

Average Number of Cells Visited : 2399.08 Cells

Average Cost of Path : 136.62513671875

Sequential Heuristic A* : Euclidean Distance By Four : 3.0 : 1.5

Average Runtime : 1.96 milliseconds

Average Path Length : 105.22 Cells

Average Number of Cells Visited : 2394.5 Cells

Average Cost of Path : 142.5554296875

Sequential Heuristic A* : Euclidean Distance By Four : 3.0 : 3.0

Average Runtime : 1.94 milliseconds

Average Path Length : 105.22 Cells

Average Number of Cells Visited : 2394.5 Cells

Average Cost of Path : 142.5554296875

8. Sequential Heuristic A* Implementation Description and Results Observation Description:

Our Sequential Heuristic A* takes in an array of heuristics and 2 weights in order to find the path from the start cell to the goal cell. It assumes the first heuristic is the admissible anchor to use. Different heuristics are explored independently where each heuristic gets its own heap to store its own fringe. The g-cost of the goal cell is initialized to infinity (Integer.MAX_VALUE for the purpose of the algorithm).

At every iteration, the algorithm performs the best expansion in a round robin fashion by looking at all the minimum values across all the heaps for each heuristic to choose the most optimal. For each of the heuristics, the algorithm pops from that heap and compares it to the cell from the anchor heuristic. While the g-cost of the other heuristic is less or equal to the g-cost of the anchor heuristic multiplied by weight2, the algorithm continues to use the anchor heuristic to create the most optimal path. If the check is violated, then it compares the g-cost of between the goal and the current cell to

see if the current cell is greater. If it is the case and the g-cost of the goal cell is less than infinity (Integer.MAX_VALUE), then that means the goal has been reached and the algorithm can terminate. If the g-cost of the current cell is not greater than that of the goal cell, then it simply raises (gets the next lower value from the anchor heap) and continues the algorithm.

The expansion is similar to that of the A* search where all the states are expanded at most once for each search. As stated before, the way to terminate from the algorithm is to find the cell with the g-cost that is greater than or equal to the g-cost of the goal cell. In our implementation, we created a hashmap that stores the cells explored as keys and the value would be an array that corresponds to each of the fringes and is used in order to save different states of the same cell with different values depending on the heuristic that explores it. A linked list and is used to store the cells that make up the path just like the A*-family searches. An ArrayList was used in order to hold the different fringes for each of the heuristics. The grid is composed of all the cells that would be traversed by the search. We believe that the data structures used helped achieve quick search, insert and deletion since most of the time, the indices, coordinates and values are known.

Based on the results of the Sequential Heuristic A* Search, some interesting observations can be made. Similar to the effects of increasing the weight for Weighted A*, by increasing the first weight (which inflates the h-value for each of the search procedures) an interesting turn of events takes place. Just like in Weighted A* Search, the overall runtime becomes less as the first weight gets increased, however, surprisingly the average path length and the average path cost increase across the board. This was not the case with Weighted A* which we find very interesting.

Through the test, we used different heuristics as the anchor to observe changes and found that the Manhattan Distance as the anchor seems to obtain the best overall values by using weights 1.5 and 3.0 respectively which allowed for low runtime (2.38 milliseconds), small path length (111 cells), smaller number of explored cells (2250) and lower cost of the path (125). We believe that this heuristic as the anchor, with the weights will be the most optimal in finding the shortest path with the lowest cost in a reasonably low runtime.

Comparing this value to our best A*-family search (Weighted A* with weight of 3.0, using the Euclidean Distance), we feel like the Weighted A* is better by comparison. With an average runtime of 0.64 ms, average path length of 105 cells, average number of cells visited being 106 cells and the average cost being 141, it seems like a better algorithm. Comparatively, the runtime is much lower for the Weighted A* than that of the Sequential Heuristic A* based on our testing. Next, the average path length is slightly lower than that of the Sequential Heuristic A*. Most importantly, the average number of cells is significantly lower for Weighted A* than that of the Sequential (almost 21 times less !). However, the Sequential Heuristic A* does have the upper hand when obtaining the least cost path because of its foundational use of multiple heuristics and it's better greedy approach when choosing cells to add to the fringe. Regardless, the obvious choice in our trials is the Weighted A* as the ideal search algorithm for our grid world.

We are definitely surprised by the values we have observed from our trials and feel like we have a better understanding of the strengths and weaknesses of the different algorithms and the effects of different weights and heuristics have on them. We do still believe that the algorithms themselves can be tweaked further and much more testing can be done to strengthen the previous observations. (Though given the time constraint, these are the best results and observations that our team decided upon.)

9. Sequential Heuristic A* Proof:

Given the property of Weighted A* : (Theorem 1)

In Weighted-A* search, the g value of any state s expanded by the algorithm is at most w_1-suboptimal. The same is true for the anchor search of Algorithm 2, i.e., for any state s for which it is true that $Key(s, 0) \leq Key(u, 0)$ $\forall$ $u \in OPEN0$, it holds that $g_0(s) \leq w_1 * c^*(s)$, where $c^*(s)$ is the optimum cost to state s.

We then need to prove that the minimum key in the admissible fringe is less than or equal to the g-cost of the goal bounded by w_1 (weight 1 used for inflating the h-cost values).

This may be able to be proved by a proof of contradiction. Let's say that there exists a cell that is greater than g-cost of the goal bounded by w1 and call this cell K. We will then show that there exists a cell C where the cost of C less than cost of K.

Now suppose there is a least cost path called P where we will take the first state in that path P, s_i, that has not been expanded yet but is in the fringe. Suppose i = 0, then the g-cost of the s_i is 0 since it is at the start and as mentioned above, this would be less than or equal to w_i ($g^*(s_i)$). Because the g cost strictly decreases as i increases, we have:

$g_o(s_i) \leq g_o(s_{i-1}) + c(s_{i-1}, s_i)$ where $c(s_{i-1}, s_i)$ represents the cost from s_{i-1} to s

$\Rightarrow g_o(s_i) \leq w_1(g^*(s_{i-1})) + c(s_{i-1}, s_i)$ (as mentioned above)

$\Rightarrow g_o(s_i) \leq w_1(g^*(s_i))$

$Key(s_i, 0) = g_o(s_i) + w_1(h_o(s_i))$

$\leq w_1(g^*(s_i)) + w_1(h_0(s_i))$

$\leq w_1(g^*(s_i)) + w_1(c^*(si, s_{goal}))$ h_0 is consistent, thus admissible

$\leq w_1(g^*(s_{goal}))$

Given this we get: $Key(s_i, 0) \leq w_1(g^*(s_{goal})) \leq K$ which is a contradiction to the beginning statement. Thus, minimum key in the admissible fringe is less than or equal to the g-cost of the goal bounded by w_1. (We get Theorem 2 here)

To prove that for any state s for which it is true that $Key(s_i, 0) \leq Key(u, 0)$ $\forall u \in OPEN_0$, it holds that $Key(s, 0) \leq w_1(g^*(s_{goal}))$, we need to evaluate the three scenarios that the algorithm terminates.

1. When the anchor search terminates, we have :

   $g_0(s_{goal}) \leq w_1 (g^*(s_{goal}))$

   $\leq w_1 * w_2 (g^*(s_{goal}))$ (when $w_2 \geq 1.0$) (by using Theorem 1)

   $\Rightarrow Key(s, 0) \leq w_1(g^*(s_{goal}))$

2. When the non-admissible search terminates, we have:

   $g_i (s_{goal}) \leq w_2 * OPEN_0.Minkey$

   $\leq w_1 * w_2 (g^*(s_{goal})$ (From Theorem 2)

   $\Rightarrow Key(s, 0) \leq w_1(g^*(s_{goal}))$

3. If the search terminates unsuccessfully: No solution

Thus from the terminations scenarios we can see clearly that when either the admissible

or inadmissible heuristic terminates, the solution is still bounded by $w_1 * w_2 (g^*(s_{goal}))$.