Experimental report for the 2021 COM1005 Assignment: The Rambler's Problem

- GitHub: link
- Date: 2021/05/24

Description of my branch-and-bound implementation

Maintain two lists

- *open* is used to store the nodes to be processed
- *closed* is used to store the nodes that have been processed

When the *open* list is not empty, perform the following operations

1. Each time the node n with the smallest $n.GC$ is selected from the *open* list, and then the node n is removed from the *open* list.
2. If node n is the target node, then node n and its corresponding path are the optimal path
3. If the node n is not the target node
    1. Generate k successor nodes $s_1, s_2, \ldots, s_k$ and set the corresponding *Local cost* $s_i.LC$ and *Global cost* $s_i.GC$ ($s_i.GC = s_i.LC + n.GC$). In this experiment, the calculation of $s_i.LC$ uses the *Rambler's* algorithm.
    2. Traverse the *open* list, whether there is a node n_i and s_i that represent the same coordinate (y, x), if it exists and $s_i.GC < n_i.GC$, use the state of s_i Replace the status of n_i
    3. Add $s_{1...4}$ to the *open* list
    4. Add node n to the *closed* list
    5. Repeat step 1

Description of my A* implementation

Maintain two lists

-*open* is used to store the nodes to be processed
-*closed* is used to store the nodes that have been processed

When the *open* list is not empty, perform the following operations

1. Each time the node n with the smallest $n.RemRC$ is selected from the *open* list, then the node n is removed from the *open* list.
2. If node n is the target node, then node n and its corresponding path are the optimal path
3. If the node n is not the target node
    1. Generate k successor nodes $s_1, s_2, \ldots, s_k$ and set the corresponding *Estimated remained cost* $s_i.RemRC$ ($s_i.RemRC = H(s_i)$), *Local cost * $s_i.LC$, *Global cost* $s_i.GC$ ($s_i.GC = s_i.LC + n.GC$).
    2. Traverse the *open* list, whether there is a node n_i and s_i that represent the same coordinate (y, x), if it exists and $s_i.GC < n_i.GC$, use the state of s_i Replace the status

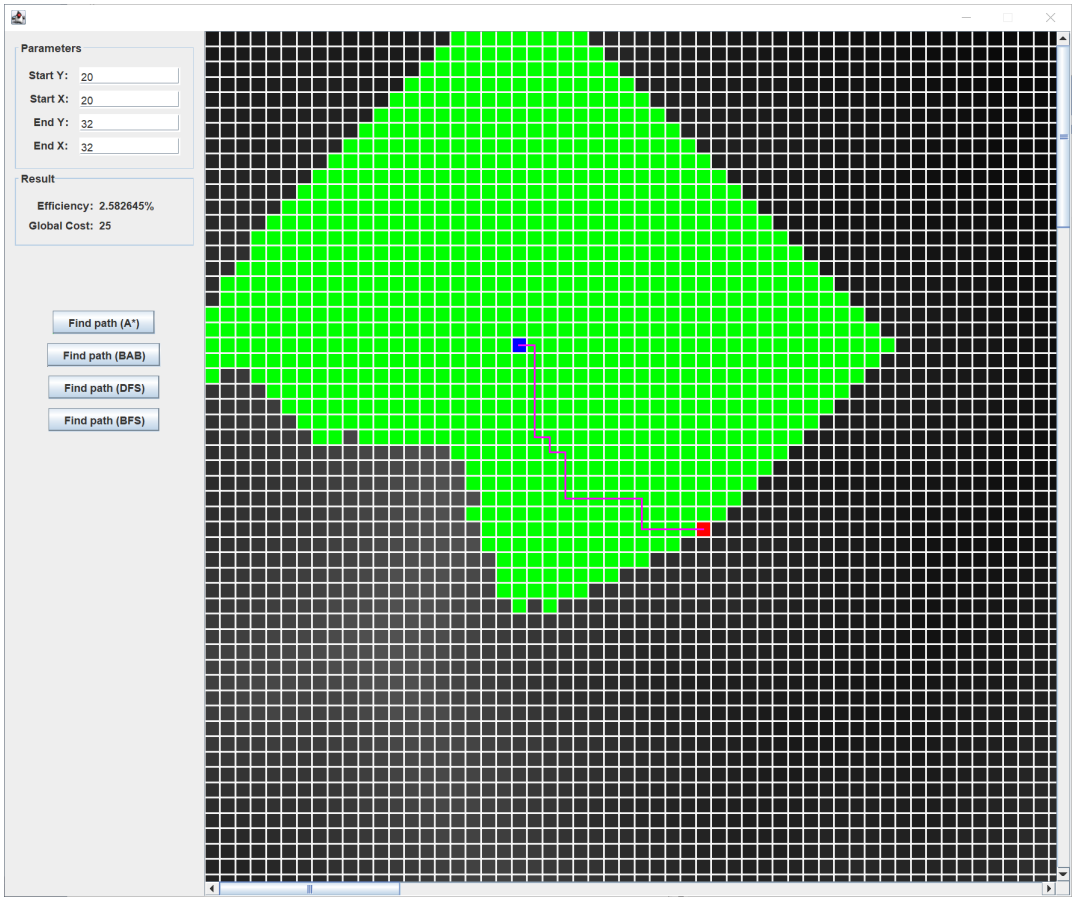
of n_i
3. Add $s_{1...4}$ to the *open* list
4. Add node n to the *closed* list
5. Repeat step 1

The realization of the *A** algorithm is the same as the basic steps of *branch-and-bound*, only a slight change is made when the node *n* is selected from *open*, according to the estimated result of the heuristic function *Estimated Remained Cost*.

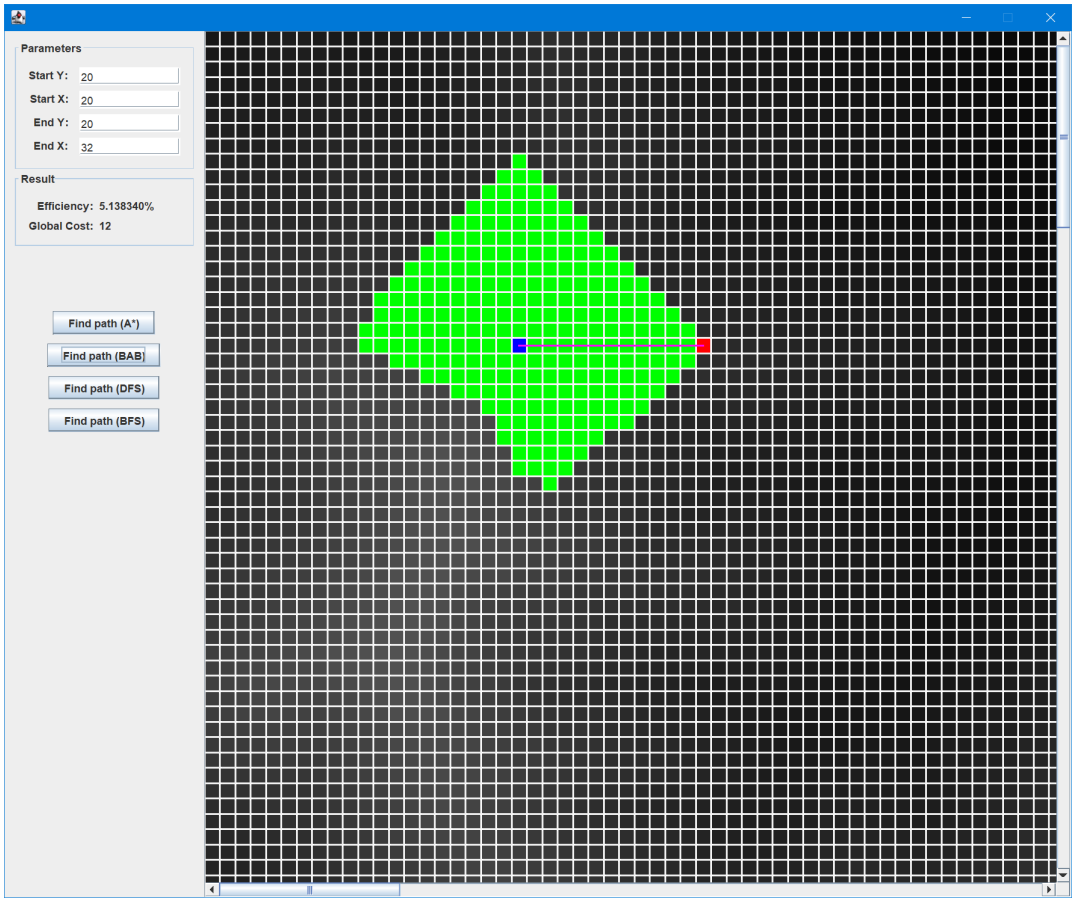
Assessing the efficiency of my branch-and-bound search algorithm

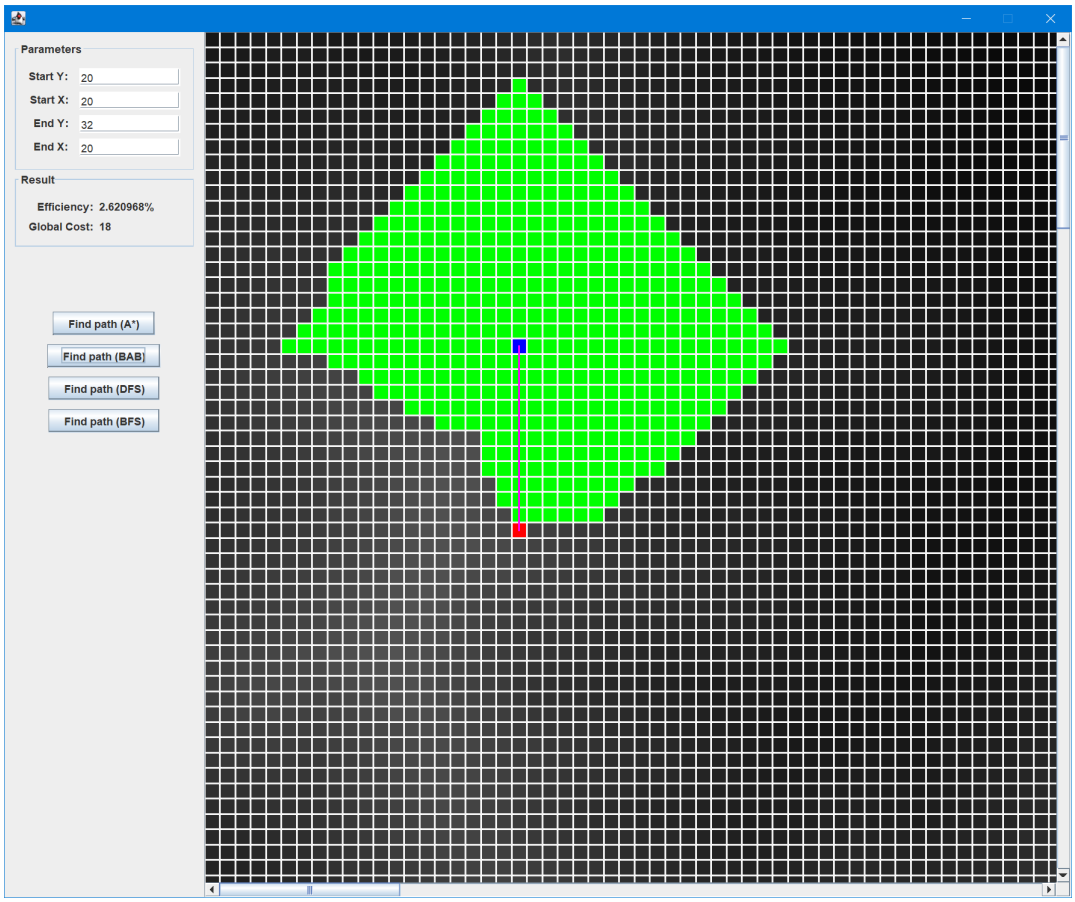
The following are three sets of experiments. The map data used in each set of experiments comes from the file "diablo.pgm". The software in the picture is located in the source code "RunRamblersSwing.java". In the picture, the **blue square** represents the starting point of the search, the **red square** represents the end of the search, the **purple line** represents the search path, **green The box** means that the state of a node in the *closed list* corresponds to a certain coordinate on the map.

- Experiment 1
  - Experiment content: When the path is a complex path, the situation of the *closed list* in the path finding process
  - Experimental results:

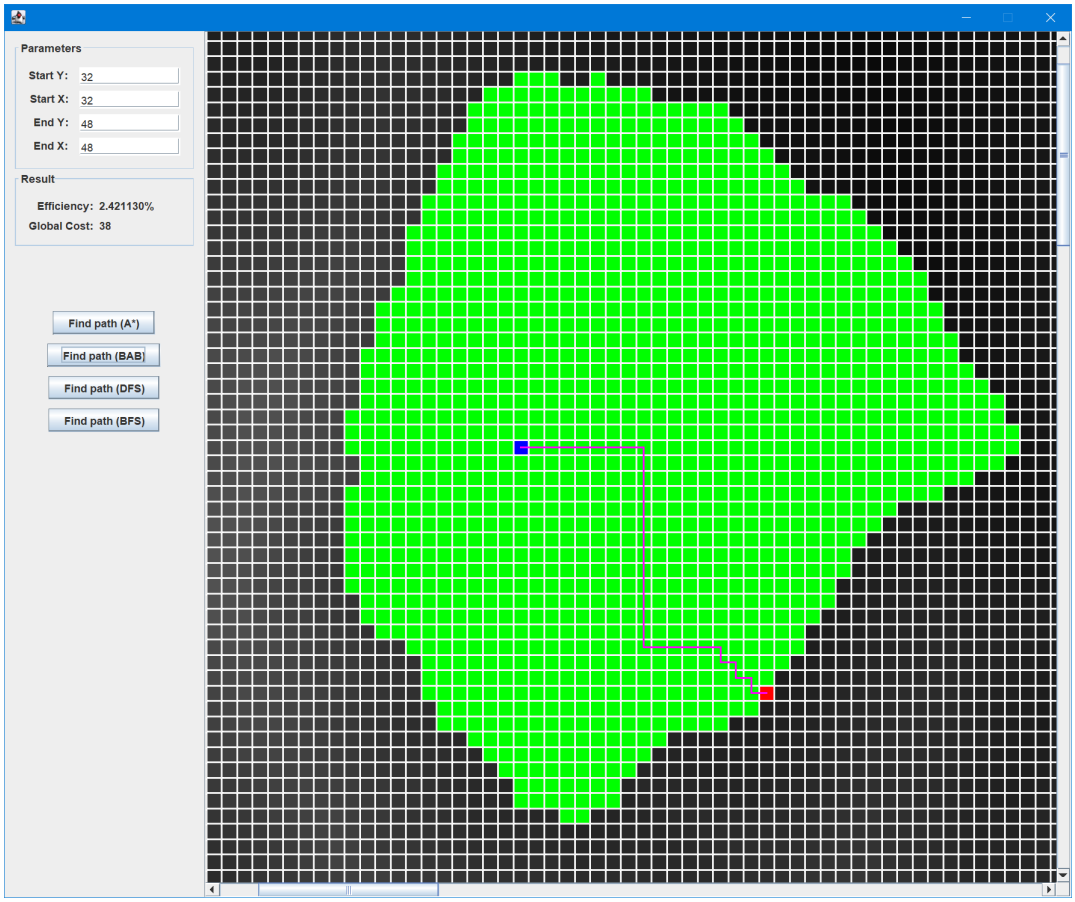


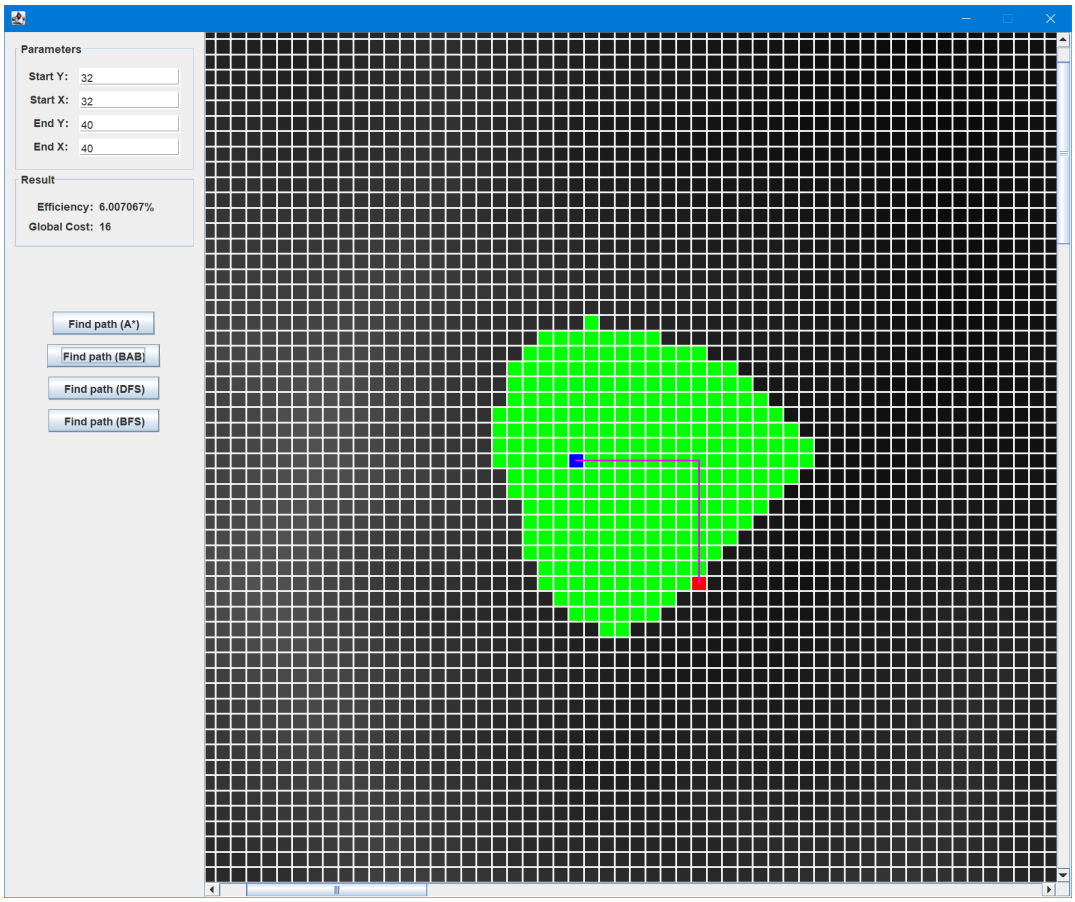
- Experiment 2
  - Experiment content: When the path is a straight path, the situation of the *closed list* in the path finding process
  - Experimental results:





- Experiment 3
  - Experiment content: the influence of path length on *closed list*
  - Experimental results:





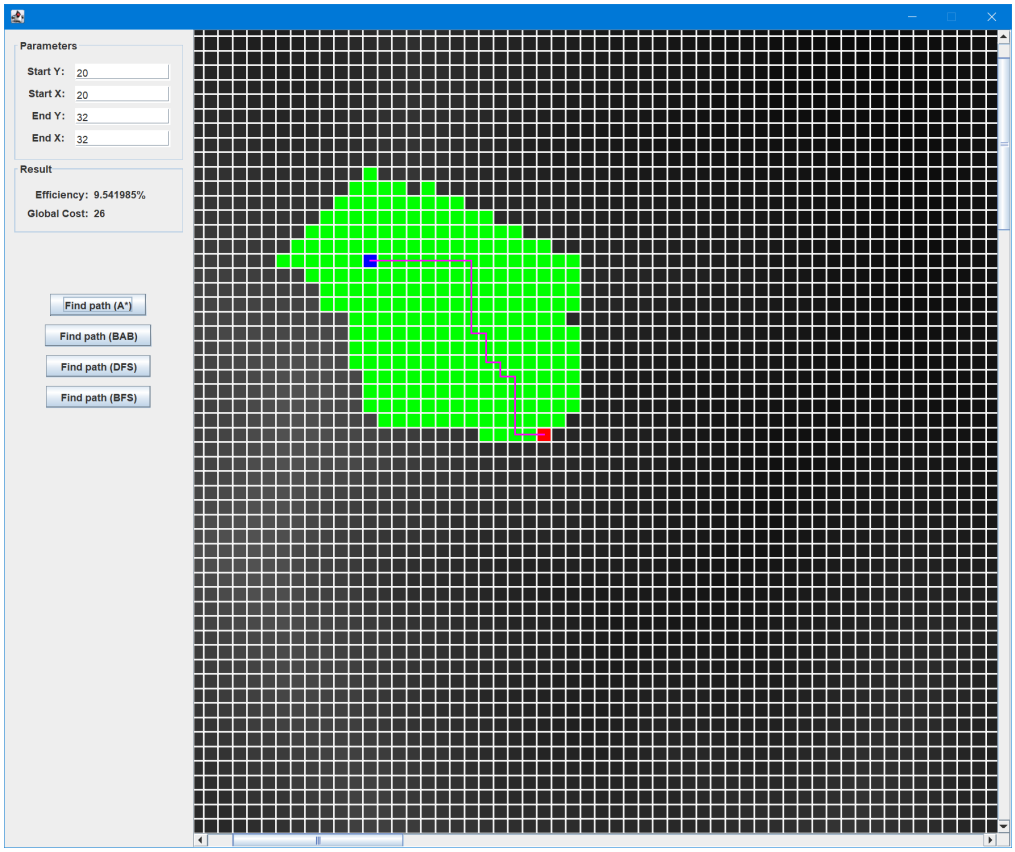
It can be seen from the above groups of experiments that the final *efficiency* of the *branch-and-bound* algorithm has little to do with the length of the path. Combining this *branch-and-bound* realization idea, the more complex the local shape, the lower the *efficiency*, which has a lot to do with the greedy idea used in this algorithm. The greedy algorithm is to obtain the global optimal solution by finding the local optimal solution. In our algorithm implementation, we give priority to the node with the smallest *local cost* in the *open list*. Under this condition, when there are multiple

local areas near the path, and each area has a small or the same cost difference with other areas, the greedy algorithm will be guided by this local optimal solution, thus performing a large number of additional calculations. , Although these local optimal solutions may be far from the global optimal solution.

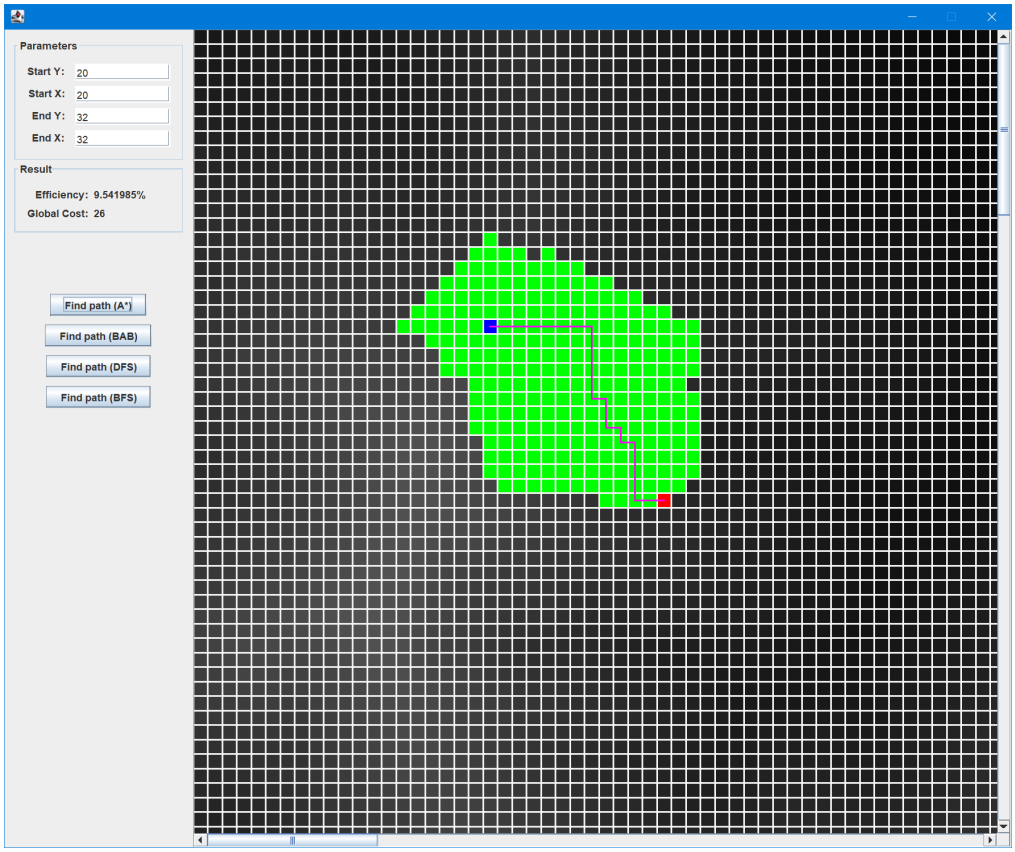
Assessing the efficiency of my A* search algorithm

Use the same experiment as the previous topic, but use the A* version of the algorithm

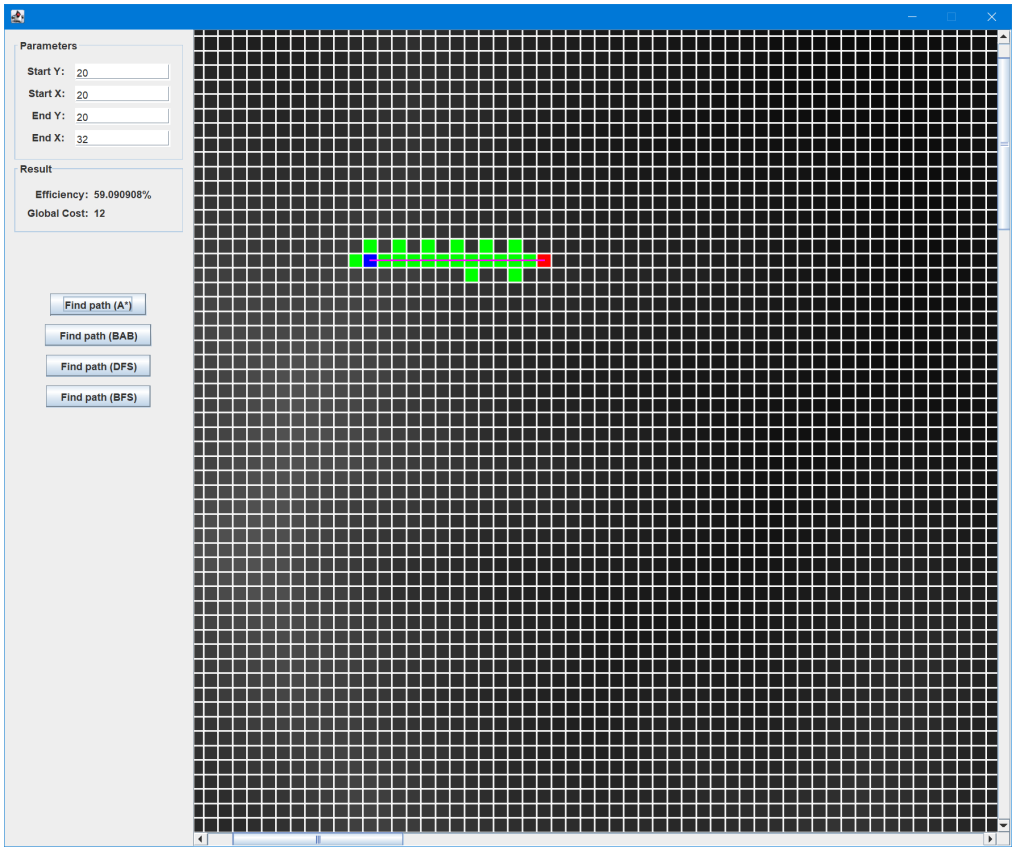
- Experiment 1
  - Experimental results:
    - Heuristic algorithm based on Manhattan distance

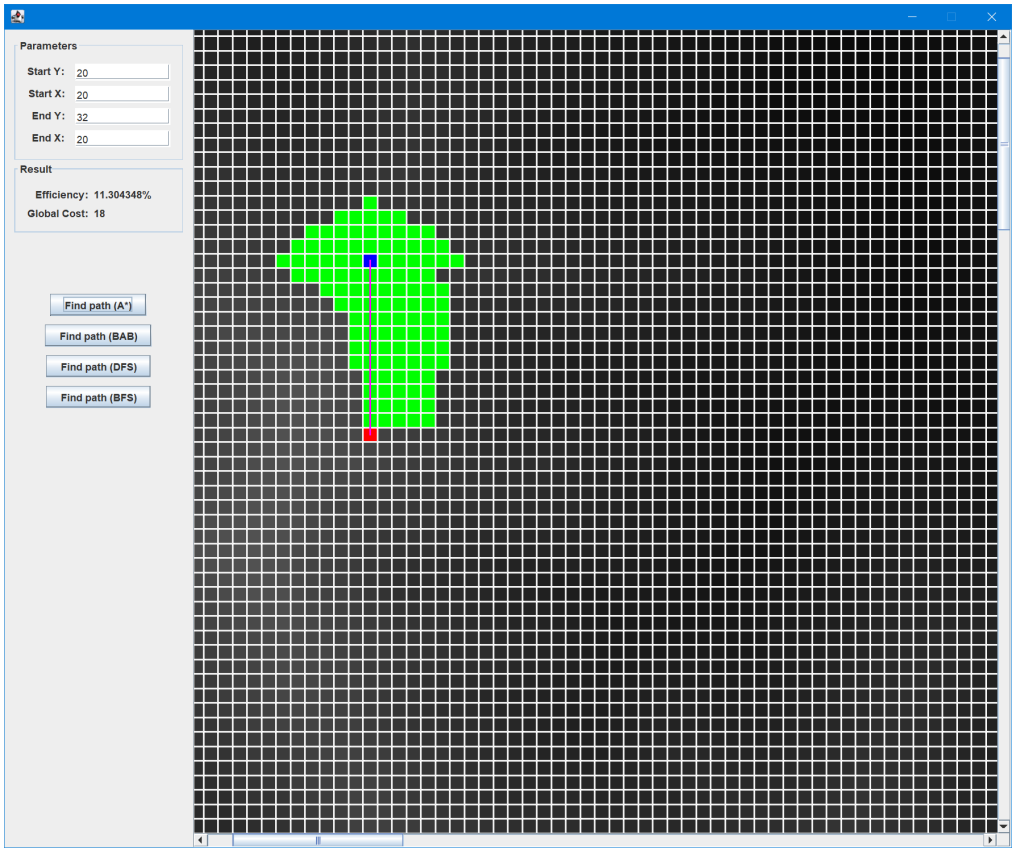


    - Heuristic algorithm based on Euclidean distance

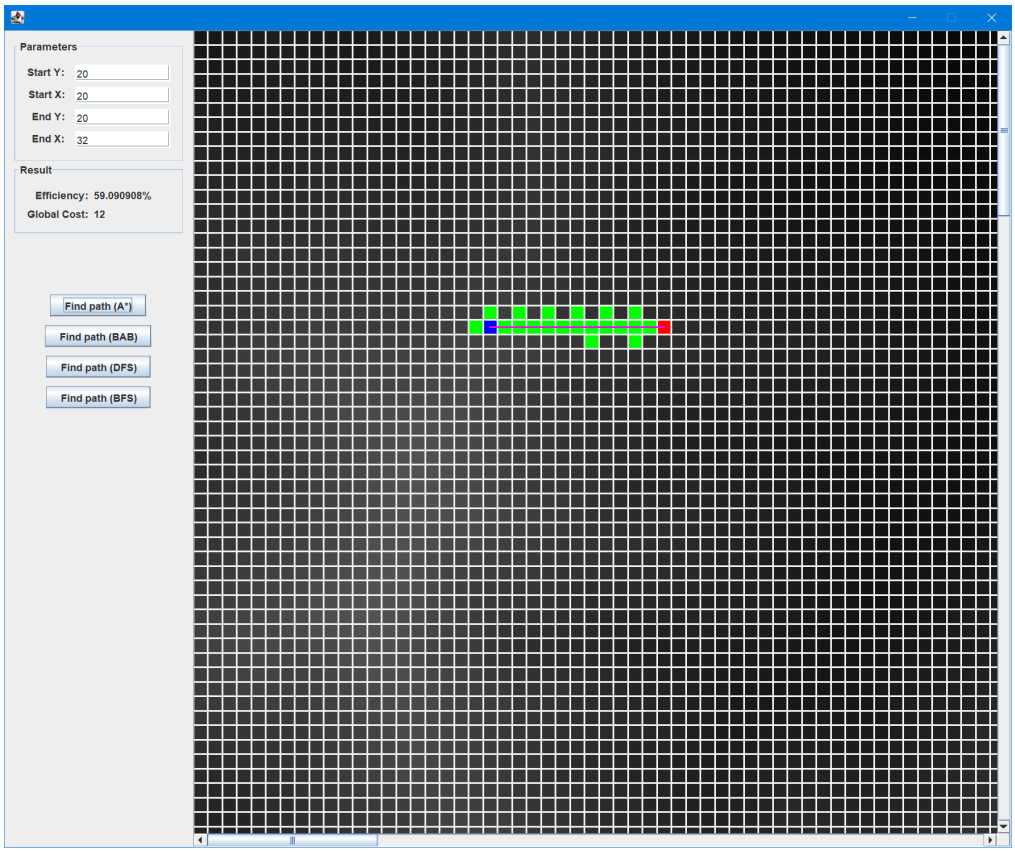


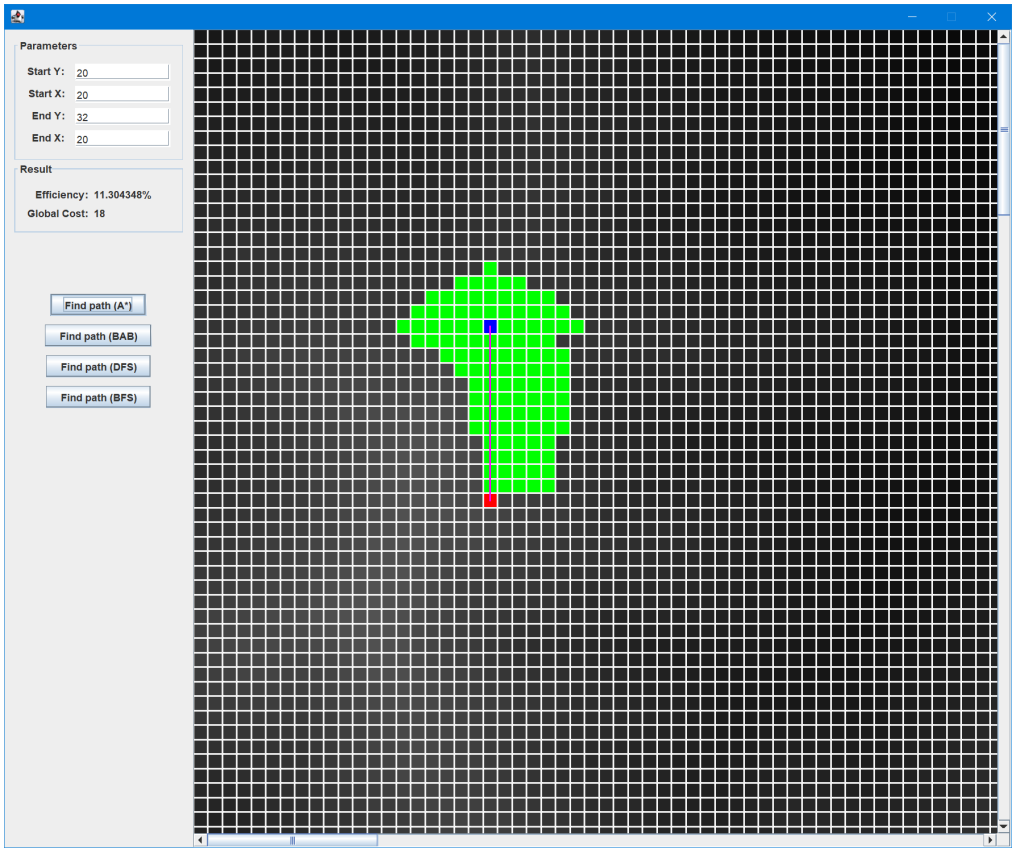
- Experiment 2
  - Experimental results:
    - Heuristic algorithm based on Manhattan distance



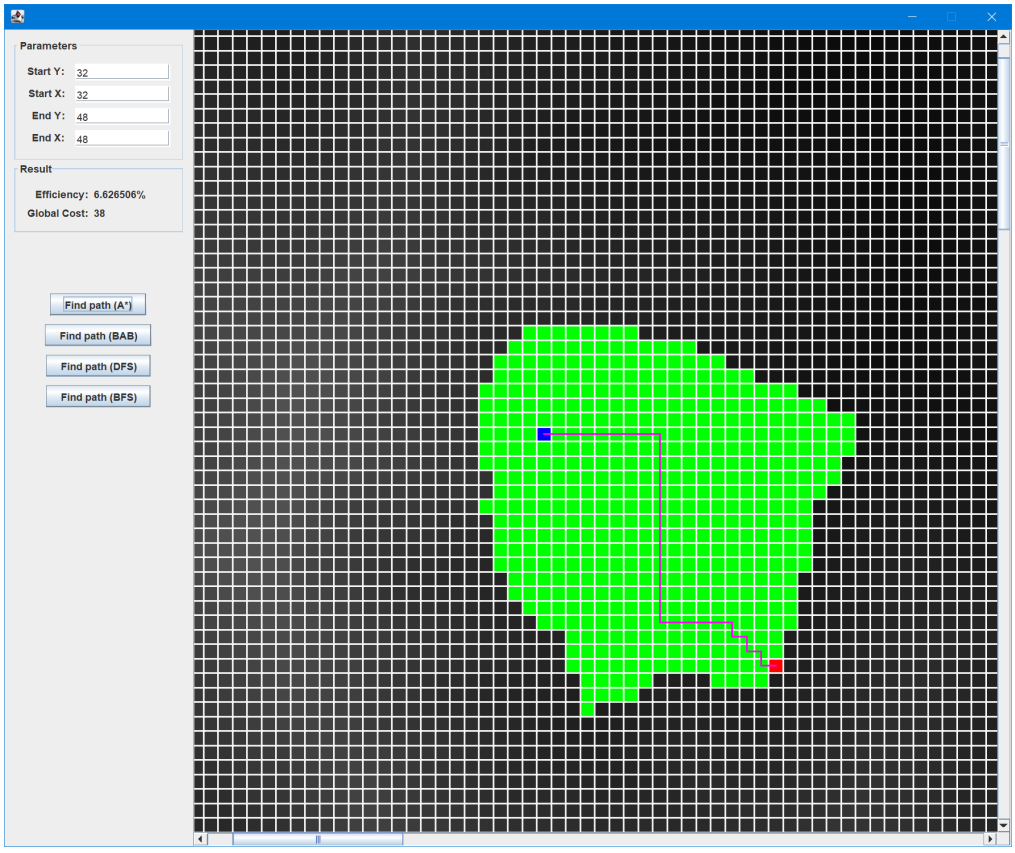


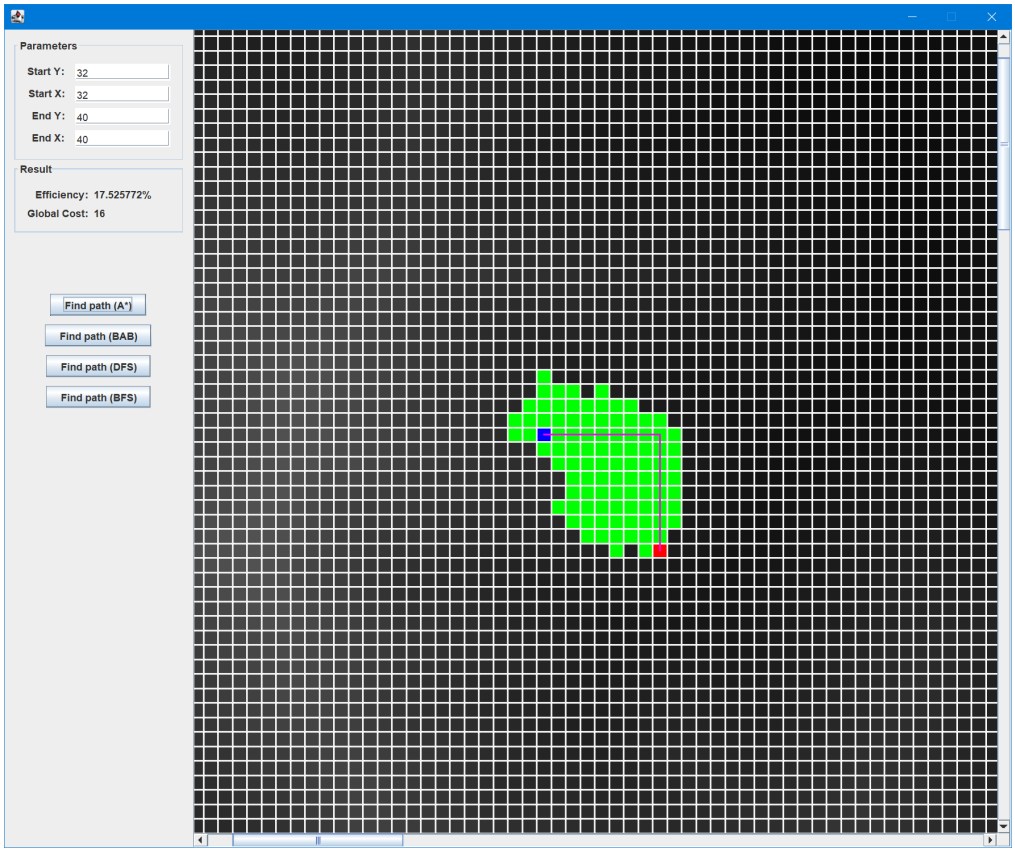
- Heuristic algorithm based on Euclidean distance



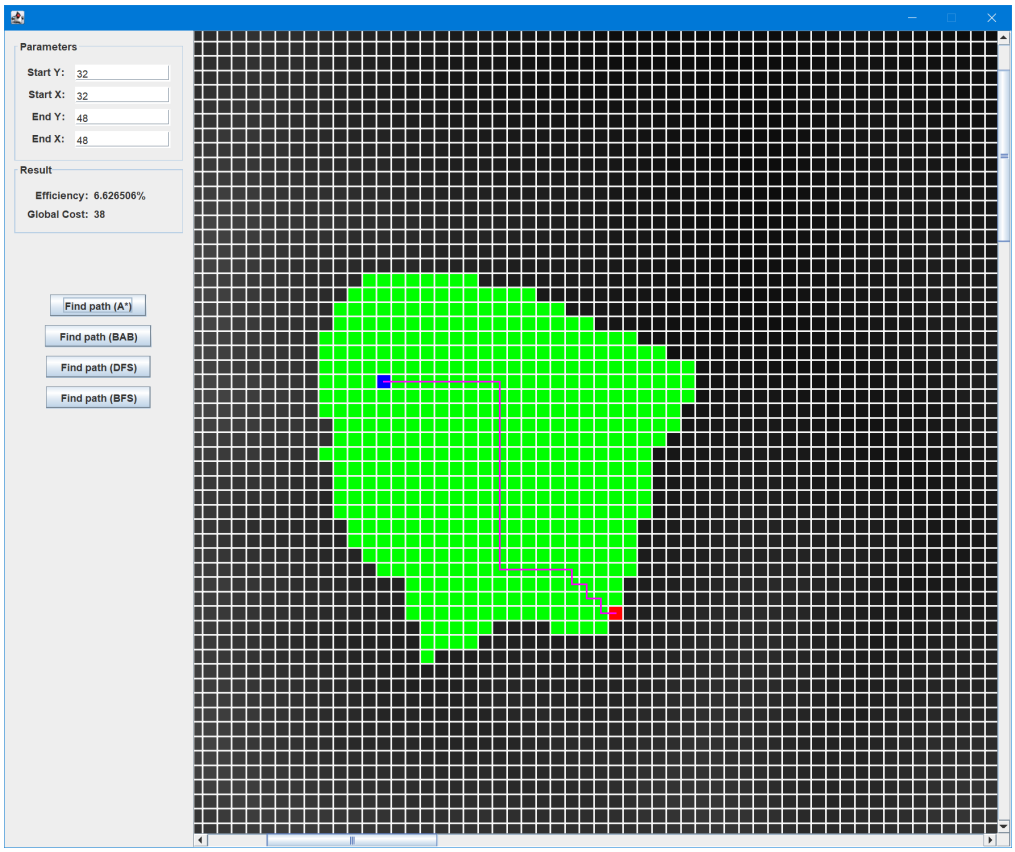


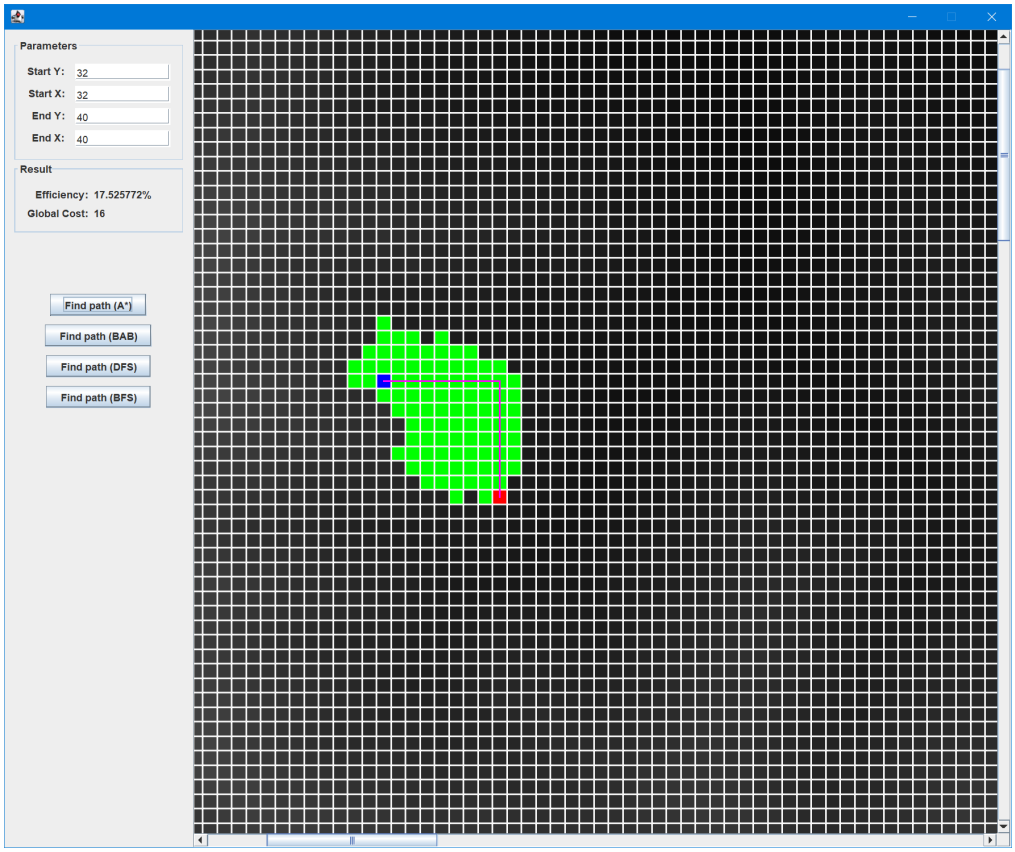
- Experiment 3
  - Experimental results:
    - Heuristic algorithm based on Manhattan distance





- Heuristic algorithm based on Euclidean distance





Through the above groups of experiments, it can be seen that the *efficiency* of the *A** algorithm is better than the *branch-and-bound* algorithm in most cases, and the green square is less than the *branch-and-bound* algorithm in most cases It can also be said that the *A** algorithm is closer to the target more accurately than the *branch-and-bound* algorithm. This is due to its *heuristic function*.

*A** When calculating the cost in the algorithm, the cost function can be expressed as $F = G + H$, where G represents the total cost from the starting point to the current node, and H is called the heuristic function, which means from the current node Estimate of the remaining cost to the target node. *A** When the algorithm selects the next node, the node with the smallest estimated value H will be selected first. Therefore, the selection of the heuristic function H has an absolute impact on the final efficiency of the *A** algorithm. The **A* algorithm will search along the gradient direction of the heuristic function H. A good heuristic function H will always enable the A* algorithm to approach the target faster**.

By analyzing the code implementation, it can be observed that the *A** algorithm is extremely dependent on the heuristic function H. If the heuristic function H is not selected properly, it may cause the *A** algorithm to give wrong targets when it stops.

If for the directed weight graph G, there is an optimal path from p to q $\overrightarrow{\omega}$ and the farthest path $\overrightarrow{\phi}$, for the path $\overrightarrow{\omega}$, the heuristic function H can always give a divergent gradient, for the path $\overrightarrow{\phi}$, the heuristic function H can always give a convergent gradient. Then the *A** algorithm always fails to find the optimal path $\overrightarrow{\omega}$ when it stops. Because for any two adjacent nodes on the path $\overrightarrow{\omega}$, the heuristic function H can always give a larger estimate. Therefore, the gradient direction will be away from the direction of the path $\overrightarrow{\omega}$.

Therefore, the choice of the heuristic function H will affect the correctness of the *A** algorithm.

We always hope that the pathfinding algorithm can reach the end point faster from the starting point. In the experiment, we chose the following heuristic functions:

- Heuristic function based on Manhattan distance
- Heuristic function based on Euclidean distance
- Heuristic function based on height change

Through the experimental results, we can see that the final *efficiency* difference between the Manhattan distance and the Euclidean distance-based heuristic function is extremely small, which shows that as long as the heuristic function H can accurately describe the gradient change from the node to the target, then * A** algorithm can accurately find the target.

Comparing the two search strategies

1. Code Implementation
    - The *branch-and-bound* algorithm is determined according to the smallest actual cost when selecting the next state.
    - *A** The algorithm is determined according to the smallest *heuristic function* estimated value when selecting the next state.

      There is not much difference in the actual implementation process of the two, only the conditions for selecting the next node are different.
2. Search efficiency
    - The *branch-and-bound* algorithm uses the adjacent minimum cost method to select the next node. This method will try every possibility and increase the amount of calculation due to multiple local optimal regions.
    - The *A** algorithm uses a method called *heuristic function* to select the next node, which will determine the search direction according to the heuristic function, and will approach the target result more purposefully.

      When the *heuristic function* is selected appropriately, the *A** algorithm will be faster than the ordinary *branch-and-bound* algorithm most of the time, because the A* algorithm will always be faster than the *branch-and-bound* algorithm under the appropriate *heuristic function* The branch-and-bound* algorithm searches for fewer nodes.
3. Algorithm results
    - The branch-and-bound algorithm will exhaust all possibilities, so the *branch-and-bound* algorithm can always find the global optimal solution at the end of the algorithm.
    - The *A** algorithm determines the search direction by the method of estimated value, so it depends on the estimated value given by the *heuristic function*. When the *heuristic function* is not selected properly, the wrong solution may be obtained at the end of the algorithm.

Conclusions

Through this lab homework, I learned

- *branch-and-bound* method.
- *A** algorithm and the influence of the heuristic function H on the *A** algorithm.
- Basic experimental methods, research methods and algorithm visualization methods.
- Dynamic planning ideas.