AI CS520 & Project1 Report

Siyuan Zhong, Dongxiang Mao

Oct 3rd 2016

1 Phase1

1.1 Create Map

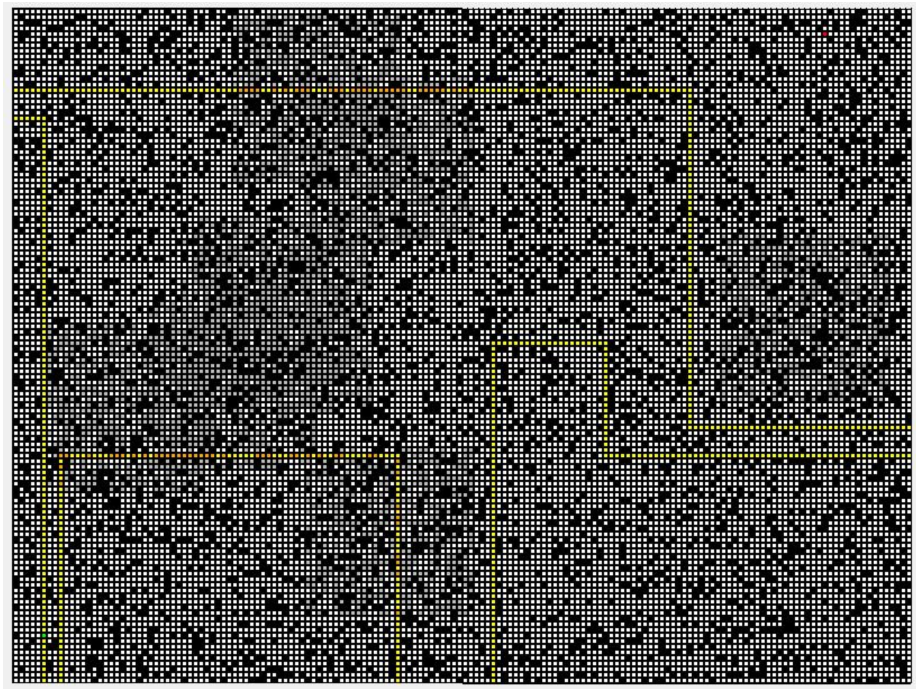
In this project, we create map with 120 rows and 160 cols. There are some different type of cell.

- Use 0 to indicate a blocked cell, color is black
- Use 1 to indicate a regular unblocked cell, color is white
- Use 2 to indicate a hard to traverse cell, color is gray
- Use ax to indicate a regular unblocked cell with a highway, color is yellow
- Use bx to indicate a hard to traverse cell with a highway, color is orange
- Use S to indicate a start cell, color is green
- Use G to indicate a goal cell with a highway, color is red

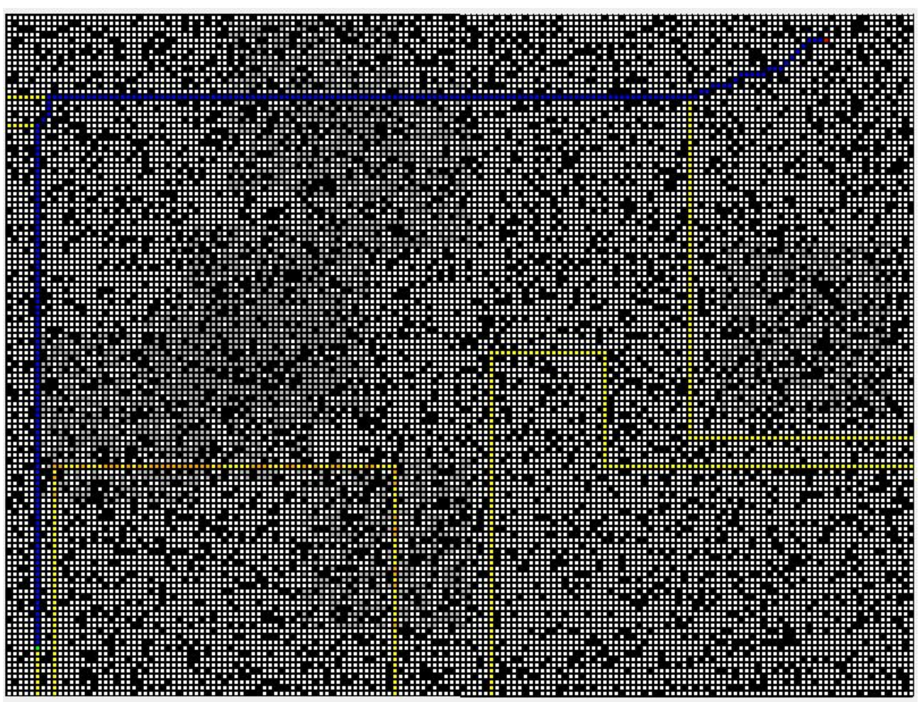
Figure.1 is an example of our map.

1.2 A* implementation

In this project, we need to implement three types search algorithm, Uniform-cost search, A* and Weighted A*. We can select to use which algorithm by adjust weight value. Because in normal A* cost function is $f = g + h$, in weight A* cost function is $f = g + w \cdot h$, in uniform search cost function is $f = g$. So we only need to set cost function is $f = g + w \cdot h$, w can be $0, 1, 1.25, 1, etc...$



(a) category 1



(b) category 2

Figure 1: the top map is a map with initial state, the bottom map is has a path compute by A* algorithm

1.3 Code Optimization

A practical optimization is to adopt dynamic weight for weighted A*. In the initial phrase of search, weight can be assigned with a relative large value, which can help the agent explore the graph more quickly. In the later phrase of search, since agent needs to find the accurate position of the goal, the weight can be set small to ensure the heuristic is admissible.
We implement the min-heap by our own. You can see it in Classes.py, there is a class heap. It is min heap with several function.

- insert
  Insert an element to the heap.

- pop
  Return an element with minimum value

- remove
  Delete an element from the heap.

- renew leaf and renew root function
  When we insert pop or remove an element we need to use this function to maintain the heap.

1.4 Different Heuristic Function

In this project, we use 1 best admissible heuristic function and 4 other heuristic function.

- The best admissible heuristic function

  $$\begin{aligned} h(s) = 0.25\times(&\sqrt{2}\cdot min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) \\ &+ max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) - min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|)) \end{aligned}$$

  We believe that this function is best admissible because in this grid world when agent and goal are in different row or column, agent can reduce its moving distance by moving diagonally.In this map, the minimum cost of travel between two vertical or horizon adjacent units is 0.25, so 0.25 times of diagonal distance will never overestimate the distance of two nodes. So we believe 0.25 times of diagonal distance is the best admissible heuristic function.

- Diagonal Distance

  $$h(s) = \sqrt{2} \cdot min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) + max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) - min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|)$$

  Using diagonal distance directly as a heuristic function is inadmissible. It estimate the average moving cost between two vertical or horizon adjacent units to be 1, which can be larger than the actual value.
  However, we still think diagonal distance is not a bad choice as it provide a good evaluation of distance between nodes and the computation cost is acceptable.

- Manhattan Distance

  $$h(s) = |s^x - s^x_{goal}| + |s^y - s^y_{goal}|$$

  Manhattan distance is not an admissible heuristic. It assumes agent cannot move in diagonal directions, which will definitely overestimate the moving cost. However, we still think diagonal distance is not a bad choice as it provide a rough evaluation of distance between nodes and the computation cost is quite cheap.

- Euclidean Distance

  $$h(s) = \sqrt{|s^x - s^x_{goal}|^2 + |s^y - s^y_{goal}|^2}$$

  Euclidean distance with a ratio of 1 is not admissible. When ratio is 0.25, it will be admissible.However compared with diagonal distance with a ratio of 0.25, it underestimate the moving cost, which can make the program run for a longer time.
  We choose it for it provide a rough evaluation of distance between nodes and the computation cost is acceptable.

- Advanced Diagonal Distance

  $$h(s) = \sqrt{2} \cdot min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) + \\ 0.25 \times (max(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|) - min(|s^x - s^x_{goal}|, |s^y - s^y_{goal}|))$$

  Another modification on diagonal distance is to multiply the diagonal part by 1 plus 0.25 times of straight part. We observed the graph and found the highways are always in vertical or horizontal directions.So we think agent may choose to move on highway a lot when moving vertically or horizontally, but seldom on highway when move diagonally.

We call it advanced diagonal distance. This heuristic is not admissible as there is a slight chance that it will overestimate the moving cost when agent travel via some highway nodes while moving diagonally.
We think advanced diagonal distance is the best heuristic to estimate the moving cost.And it is easy to compute.And it is admissible in most cases.

1.5 Experiment Result

Figure 2 is the average statistical results for 5 maps with 10 different start and end points.

			TOTAL AVG				TOTAL AVG
diagonal	1	length	137.1211013	admissible heuristic	1	length	119.607728
		actual length/optimal	1.149965337			actual length/optimal length	1
		nodes	2675.46			nodes	14275.84
		time	6.95580484			time	49.80907914
	1.25	length	156.7643265		1.25	length	119.6340988
		actual length/optimal length	1.331949136			actual length/optimal length	1.000198213
		nodes	883.82			nodes	13534.22
		time	2.110554356			time	47.85050453
	2	length	187.8848137		2	length	120.9974249
		actual length/optimal length	1.622249673			actual length/optimal length	1.010748114
		nodes	458.66			nodes	11889.06
		time	1.16683213			time	42.35785933
Euclidean	1	length	129.0130216	advanced heuristic	1	length	129.0120984
		actual length/optimal length	1.075071123			actual length/optimal length	1.077905371
		nodes	2881.34			nodes	11592.02
		time	7.675496291			time	37.61858351
	1.25	length	157.5537308		1.25	length	141.5648096
		actual length/optimal	1.341997166			actual length/optimal length	1.17706201
		nodes	744.06			nodes	7718.18
		time	1.815387331			time	20.965334
	2	length	179.4416521		2	length	156.7776621
		actual length/optimal	1.513425765			actual length/optimal length	1.30590372
		nodes	464.84			nodes	3424.4
		time	1.162598434			time	8.128748055
Manhattan	1	length	160.502488	uniform-cost		length	119.607728
		actual length/optimal	1.351687079			actual length/optimal length	1
		nodes	1443.64			nodes	16309.04
		time	2.979299063			time	28.26925517
	1.25	length	176.159102				
		actual length/optimal	1.524727311				
		nodes	678.24				
		time	1.589702638				
	2	length	193.5325754				
		actual length/optimal	1.671736345				
		nodes	471.24				
		time	1.197015028				

Figure 2: Experiment result

Figure 3 is the memory usage for different type heuristic function. We use memory_profile library in python to test memory usage. The possible reason why the memory usage for different algorithm near is that the expand much nodes don't cost lots of memory, but initialize will cost lots of memory. Because in our function, we first need to transfer the map, this will cost lots of memory compare to open list and close list status. So the memory doesn't change obviously.

heuristic	weight	memory(MiB)
	1	38.1
	1.25	38.1
diagonal	2	38.2
	1	38.2
	1.25	38.2
Euclidean	2	38.2
	1	38.4
	1.25	38.4
Manhattan	2	38.5
	1	38.5
	1.25	38.6
optimal(0.25)	2	37.2
	1	37.7
	1.25	37.7
advanced diagonal	2	37.9
uniform	1	40.5

Figure 3: Memory result

1.6 Result Discuss

From the Figure 2 we can see that different heuristic function have different result on time and path length. For some heuristic function with big value such as Manhattan distance, euclidean distance and diagonal distance, it will spend less time due to expand less nodes. But it cost may be larger compare to other function. The heuristic function with small value usually spend more time to run due to expand more nodes.

Uniform function have minimum path length, but it expand most nodes and spend most time, because this algorithm always try to select the minimum cost path length. Compare to Uniform search, admissible heuristic function have less path length path and expand nodes, but it is too still too slow. Because heuristic function value is too small compare to g value. It take longer time than uniform may be because it need to do more multiplication and add computation. While Manhattan distance, euclidean distance and diagonal distance have a larger heuristic value so it spend minimal time but path length may be larger.

Compare to above algorithm, advanced diagonal distance consider the actual game situation, so it combine the time and path length performance. So in general I prefer to choose advanced diagonal distance as my heuristic function because it balance the time and path length performance.

2 Phase2

2.1 Implement other two A*

In this phase, we implement the two algorithm sequential A* and integrated A* as Algorithm 2 (Figure 4) and Algorithm 3 (Figure 5) showed in pdf attachment. We run it on 5 different map and each map with 10 different start and end points. For both this two algorithm, we run the two algorithm with $w_1 = 1.25, 2$ and $w_2 = 1.25, 2$ on those benchmark.

```
Key(s,i)
    return g_i(s) + w_1 * h_i(s);
ExpandState(s,i)
    Remove s from OPEN_i;
    foreach s' ∈ Succ(s) do
        if s' was never generated in the i^th search then
            g_i(s') = ∞; bp_i(s') = NULL;
        if g_i(s') > g_i(s) + c(s, s') then
            g_i(s') = g_i(s) + c(s, s'); bp_i(s') = s;
            if s' ∉ CLOSED_i then
                Insert/Update s' in OPEN_i with Key(s', i);
Main()
    for i = 0, 1, ..., n do
        OPEN_i ← Ø;
        CLOSED_i ← Ø;
        g_i(s_start) = 0;g_i(s_goal) = ∞;
        bp_i(s_start) = bp_i(s_goal) = NULL;
        Insert s_start in OPEN_i with Key(s_start, i) as priority
    while OPEN_0.MinKey() < ∞ do
        for i = 1, 2, ..., n do
            if OPEN_i.Minkey() ≤ w_2 * OPEN_0.Minkey() then
                if g_i(s_goal) ≤ OPEN_i.Minkey() then
                    if g_i(s_goal) < ∞ then
                        Terminate and return path pointed by bp_i(s_goal)
                else
                    s ← OPEN_i.Top();
                    ExpandState(s, i);
                    Insert s in CLOSED_i ;
            else
                if g_0(s_goal) ≤ OPEN_0.Minkey() then
                    if g_0(s_goal) < ∞ then
                        Terminate and return path pointed by bp_0(s_goal)
                else
                    s ← OPEN_0.Top();
                    ExpandState(s, 0);
                    Insert s in CLOSED_0
```

Algorithm 2: Sequential Heuristic A*

Figure 4: Algorithm 2

```
Key(s,i)
    return g(s) + w_1 * h_i(s);
ExpandState(s)
    Remove s from OPEN_i, ∀ i = {0, 1, ⋯, n} ;
    v(s) = g(s);
    foreach s' ∈ Succ(s) do
        if s' was never generated then
            g(s') = ∞; bp(s') = NULL;
            v(s') = ∞;
        if g(s') > g(s) + c(s, s') then
            g(s') = g(s) + c(s, s'); bp(s') = s;
            if s' ∉ CLOSED_anchor then
                Insert/Update s' in OPEN_0 with Key(s', 0);
                if s' ∉ CLOSED_inad then
                    for i = 1, 2, ..., n do
                        if Key(s', i) ≤ w_2 * Key(s', 0) then
                            Insert/Update s' in OPEN_i with Key(s', i);
Main()
    g(s_start) = 0; g(s_goal) = ∞;
    bp(s_start) = bp(s_goal) = NULL;
    u(s_start) = u(s_goal) = ∞;
    for i = 0, 1, ..., n do
        OPEN_i ← ∅;
        Insert s_start in OPEN_i with Key(s_start, i)
    CLOSED_anchor ← ∅;
    CLOSED_inad ← ∅;
    while OPEN_0.Minkey() < ∞ do
        for i = 1, 2, ..., n do
            if OPEN_i.Minkey() ≤ w_2 * OPEN_0.Minkey() then
                if g(s_goal) ≤ OPEN_i.Minkey() then
                    if g(s_goal) < ∞ then
                        Terminate and return path pointed by bp(s_goal)
                else
                    s ← OPEN_i.Top();
                    ExpandState(s);
                    Insert s in CLOSED_inad;
            else
                if g(s_goal) ≤ OPEN_0.Minkey() then
                    if g(s_goal) < ∞ then
                        Terminate and return path pointed by bp(s_goal)
                else
                    s ← OPEN_0.Top();
                    ExpandState(s);
                    Insert s in CLOSED_anchor
```

Algorithm 3: Integrated Heuristic A*

Figure 5: Algorithm 3

The experiment result is as Figure 6 and Figure 7 show.

uniform-cost		length	119.607728
		actual length/optimal	1
		nodes	16309.04
		time	28.26925517
Sequential Heuristic A*	w1=1.25, w2=1.25	length	119.9688401
		actual length/optimal	1.00274189
		nodes	15316.86
		time	38.62201389
	w1=1.25, w2=2	length	162.5577946
		actual length/optimal	1.348571055
		nodes	4590.88
		time	9.836304965
	w1=2, w2=1.25	length	121.7391459
		actual length/optimal	1.016297776
		nodes	12869.02
		time	35.42154099
	w1=2, w2=2	length	168.7162253
		actual length/optimal	1.378682922
		nodes	5500.8
		time	11.87081609
Integrated Heuristic A*	w1=1.25, w2=1.25	length	132.3191616
		actual length/optimal	1.113950156
		nodes	12086.3
		time	21.027424
	w1=1.25, w2=2	length	137.1826219
		actual length/optimal	1.148440408
		nodes	14219.12
		time	57.89303996
	w1=2, w2=1.25	length	138.0986251
		actual length/optimal	1.183792784
		nodes	9526.92
		time	15.5285876
	w1=2, w2=2	length	143.9389231
		actual length/optimal	1.211378519
		nodes	8881.8
		time	21.56102743

Figure 6: phase 2 algorithms' performance comparison

		memory
uniform-cost		40.5
Sequential Heuristic A*	w1=1.25, w2=1.25	85.5
	w1=1.25, w2=2	103.2
	w1=2, w2=1.25	104.4
	w1=2, w2=2	104
Integrated Heuristic A*	w1=1.25, w2=1.25	85.6
	w1=1.25, w2=2	76.5
	w1=2, w2=1.25	68.5
	w1=2, w2=2	68.6

Figure 7: phase 2 algorithms' memory consumption

2.2 Result Discussion

Our implementation generally follows the structure of given pseudocode. To expand the open list efficiently, we created a list called action to represent all the possible extensions of each node.

Observing Figure 6 , we can find some interesting facts:

1. For each algorithm, w1 and w2 directly relates to the number of nodes expanded, length of result and time consumption.

2. The larger w1 is, the less nodes expanded, the longer time consumed and the better result we get, which fits the result in section e. Because as w1 increases, algorithm is more likely to expand nodes remote to the source node.

3. As w2 grows, algorithm will spend less time returning a longer path. That is because increment of w2 makes algorithm tolerates in admissible heuristics more, leading to a faster speed but worse performance.

4. The performances of sequential heuristic A* are more sensitive to the value of w2. As w2 increases, the number of nodes expanded decreases rapidly and the result it returns degenerates significantly.

5. The performances of integrated heuristic A* are more stable compared with that of sequential heuristic A*. When w1 and w2 increase by the same extent, the performance of integrated heuristic A* degenerates slower than that of sequential heuristic A*.

6. Uniform-cost search spends less time than sequential heuristic A* when w1 and w2 are all equal to 1.25 in spite of uniform-cost search expands more nodes. That means if you want a path whose length is less than 2% longer than the optimal one's, you may choose uniform-cost search. The reason is when expanding the same amount of nodes, uniform-cost search takes less time than sequential heuristic A* due to the overhead of switching among different heuristics.

7. Compared with admissible search(weight=1) in section e, sequential heuristic(w1=w2=1.25) expands a little more nodes and actual length/ optimal length is slightly greater than 1. But run time is less. While in other weights, sequential A* use less time and expand less node obviously.

Figure 7 shows that uniform-cost search consumes least memories for it cashes only one g,h,f value for each state and maintains one open list and one open list. Sequential heuristic A* takes most memories as it maintains n+1 g,h,f values for each state and maintains n+i open list and close list. Integrated heuristic A* requires intermediate memories.It uses only one g,h,f value for each state. But maintains two closed list and n+1 open list. Figure 3 shows admissible heuristic's memory usage is almost equal to uniform-cost's as they use exact the same data structures.

2.3 Question i proof

Algorithm 2 is as Figure 4 showed. So we have given $g_0(s) \le w_1 * c^*(s)$, we need to prove that $Key(s, 0) \le w_1 * g^*(s_{goal})$.
Proof:
We prove this by contradiction. Let us assume, $Key(s, 0) = g_0(s) + w_1 * h_0(s) > w_1 * g^*(s_{goal})$.
Consider a least cost path from start to goal is given as $P(s_0 = s_{start}, \ldots, s_k = s_{goal})$. In this path, we select the first state $s_i \in OPEN_0$, and s_i must be s_0 due to initialization part as algorithm 2 Line 18 showed. So if the state $s_j \in P$ in the anchor search, $s_{j+1} \in P$ is always inserted in $OPEN_0$. Besides $s_k = s_{goal}$ will never expanded in the anchor search, otherwise, s_{goal} has the least key in $OPEN_0$ and the search will terminate.
If i = 0, we have $g_0(s_{start}) = 0 \le w_1 * g^*(s_0)$. If $i \ne 0$, by the choice of s_i we know s_{i-1} has already been expand in anchor search. And we have $g_0(s_{i-1}) \le w_1 * g^*(s_{i-1})$.
So we have

$$\begin{aligned} g_0(s_i) &\le g_0(s_{i-1}) + c_(s_{i-1}, s_i)(Line9) \\ &\le w_1 * g^*(s_{i-1}) + c_(s_{i-1}, s_i) \\ &\le w_1 * (g^*(s_{i-1}) + c_(s_{i-1}, s_i)) \qquad (s_i, s_{i-1} \in P) \\ &= w_1 * g^*(s_i) \end{aligned}$$

Then we can use to prove that

$$\begin{aligned} Key(s_i, 0) &= g_0(s_i) + w_1 * h_0(s_i) \\ &\le w_1 * g^*(s_i) + w_1 * h_0(s_i) \\ &\le w_1 * g^*(s_i) + w_1 * c^*(s_i, s_{goal}) \qquad h_0 \textit{ is admissible} \\ &= w_1 * g^*(s_{goal}) \end{aligned}$$

Next we need to prove that $g_i(s_{goal}) \le w_1 * w_2 * c^*(s_{goal})$.
If this algorithm terminate in Algorithm 2 Line 24 in inadmissible search h_i. Then

we have

$$\begin{aligned} g_i(s_{goal}) &\le w_2 * OPEN_0.Minkey() \\ &\le w_2 * w_1 * g^*(s_g) \qquad from\ previous\ proof \end{aligned}$$

If this algorithm terminate in Algorithm 2 Line 32 in admissible search, we have that

$$\begin{aligned} g_0(s_{goal}) &\le w_1 * g^*(s_{goal}) \\ &\le w_2 * w_1 * g^*(s_g) \qquad w_2 \ge 1.0 \end{aligned}$$

So in both situation, we can prove that $g_i(s_{goal}) \le w_1 * w_2 * c^*(s_{goal})$.

2.4 Question j

- no state is expanded more than twice
  In this algorithm node can be only expand in Algorithm 3 Line 35 and 43. And from statement 3, we can see that if s expand in inadmissible search it can only be reexpand in anchor search. From the statement 2 we can see that if s expand in admissible search, it can not be expand in another search. So no state is expanded more than twice.

- a state being expanded in the anchor search is never re-expanded.
  When a state is expand in anchor search at line 43, s is removed from all $OPEN_i$ in line 4. s can only be expanded in inadmissible search or in anchor search, if it is re-inserted in any of $OPEN_i$.However, the line 14 will insured it can't inserted $OPEN_i$. So a state expanded in the anchor search is never re-expanded.

- a state expanded in an inadmissible search can only be re-expanded in the anchor search if its g-value is lowered.
  If it expands in line 35, it removes the state from $OPEN_i$. Now a state can be only inserted in $OPEN_i, i \neq 0$ in line 17. If a state s has already been expand in inadmissible search, the line 14 will insured it can't inserted $OPEN_i$. If s has been inserted in $OPEN_0$ in the future and if line 10 is true. Which means this g-value is lower than the earlier g-value. Thus, a state s whose g has not been lowered after its expansion in any inadmissible search will never satisfy the condition at line 10 and will not be re-inserted in $OPEN_0$ and can never be expanded in the anchor search.

References

[1] Aine S, Swaminathan S, Narayanan V, et al. Multi-Heuristic A[J]. The International Journal of Robotics Research, 2016, 35(1-3): 224-243.