HO CHI MINH UNIVERSITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

- Introduction to Artificial Intelligent –

Search and A algorithm for greyscale image*

Class: 19CLC5

Instructor:

- Dr. Châu Thành Đức
- MSc. Ngô Đình Hy

Student:

- Đỗ Vương Phúc
- Hoàng Như Thanh

Contents

1. Theory of Adversarial Search

Before jumping to the definition of the Adversarial Search, let us discuss Search. We the human do make decisions every second or minute on the daily-basis. Therefore, computers have to be set able to decide toward a problem or event, in order to react and act like human. For that reason, search algorithm is a critical part of AI since it is a step-by-step procedure to solve a problem, each step the computer must make a decision what action to take.

1.1. What is Adversarial Search?

Unlike uninformed search algorithms, where one agent tries to find the solution for a problem, adversarial search is the kind of search that multiagents are included and they all try to search for the solution which help beat other agents, we can find them in games that have more than one player, each of them tries to win against others. In other words, adversarial search takes place where there is a competitive environment that agents' goals are in conflict.

We can name many games where adversarial search is applied like chess, tic-tac-toe, Othello, Poker, etc.

1.2. Game theory

For adversarial search is usually known as games, it may cause many people to think that it is overdo to have a kind of search only for games. However, let us clarify a little more. Our life is full off situations, and is an environment with multiple agents. For example, we have a daily-life situation like the girls waiting for sales and try to get the limited edition of Dior lipstick before anyone else buy all of them. Game theory views any multi-agent environment like that as a game. The impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive.

For example, two wedding decoration companies give their ideas in order to win a wedding, the customer will choose 1 over them. And that situation can be seen as a game with 2 players fighting to win, the 2 agents are competitive.

1.3. Zero-sum game

Zero -sum game is a situation in game theory that can be quickly described as the following statements:

- A zero-sum game is the situation where one agent's gain is other's loss
- It can have more than just 2 players fighting against other
- It is not a win-win or lose-lose situation, there must be the winner and the loser

- Total loss of the loser is equal to total gain of the winner

For example, Monopoly is zero-sum game if it intends to have only one winner, who collect all money of others. On the contrary, Monopoly is not recognized as a zero-sum game if it intends to have many winners by reaching the specific the money goal.

Besides, we have chess, poker and gambling as a popular zero-sum game.

One situation that usually be thought as a zero-sum game is, the buyer go to the store to buy shoes, they bargain the price. The conflict between the buyer and the seller is clear: The buyer wants to have the shoes with the price as low as possible while the seller wants to sell them with the possible highest price. However, it is not a zero-sum game. If the price is 20$ and the buyer bargain it down to 15$, and the seller accept the price, so the buyer gets the shoes with affordable price, and the buyer also gain advantage because he better sell them rather than not able to sell any shoes at all. This is a non-zero-sum game.

1.4. Game tree and Minimax algorithm

1.4.1. Game problem formulation

In the present days we play lot of two-player games with the computer, like tic-tac-toe. In order for the computer to be able to play a game, it has to define the game as a search problem which can be formalized with:

- Initial state: How the game is set up and start
  E.g: a blank 3x3 table
- Player: Player to move
- Successor function: A list of legal (move, state) pairs
- Goal test: Is the game over?
- Utility function: A numeric value of a terminal state (win, lose, draw)

1.4.2. Game tree

Game tree is a directed graph which shows every possible state of the game. Concretely, each of its node represents the game state, each of its edge is a move by player and its root is the initial state. Game tree also shows the terminal state and state utility.

The figure below is an image of the game tree, we are X and the opponent is O.

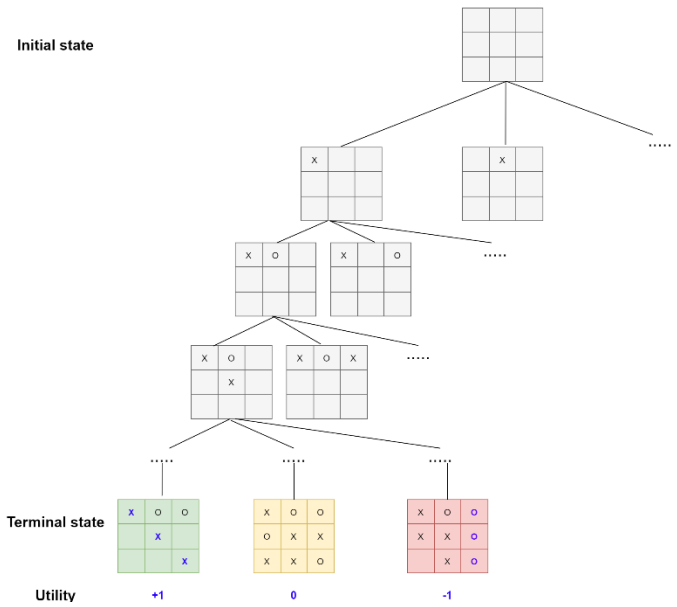


Figure 1. Game tree

1.4.3. Minimax algorithm

Minimax is one of the adversarial search algorithm. The Minimax algorithm tries to maximize the ability to win by predicting that the opponent will choose the move which makes the worst affect to us. If we keeps move to the best state, and the opponent keeps move to the worst state for our viewpoint (both player is playing optimally), then we can see that the way Minimax algorithm predicts the opponent behavior works right. Therefore, in deterministic games, the Minimax algorithm is the perfect play.

- **Complete:** Yes, if the tree is finite
- **Optimal:** Yes, if the opponent plays optimally
- **Time complexity:** $O(b^m)$
- **Space complexity:** $O(bm)$
    - b: the legal moves at each point
    - m: the maximum depth of the tree

Quick demonstration

The Minimax algorithm can be demonstrated basically like below. We are MAX, try to maximize our score, so we keep choosing the maximum value among all. The opponent is MIN, try to minimize our score, so he keeps choosing the minimum value among all.

Firstly, to minimize the score, MIN choose the minimum MAX's utility values.

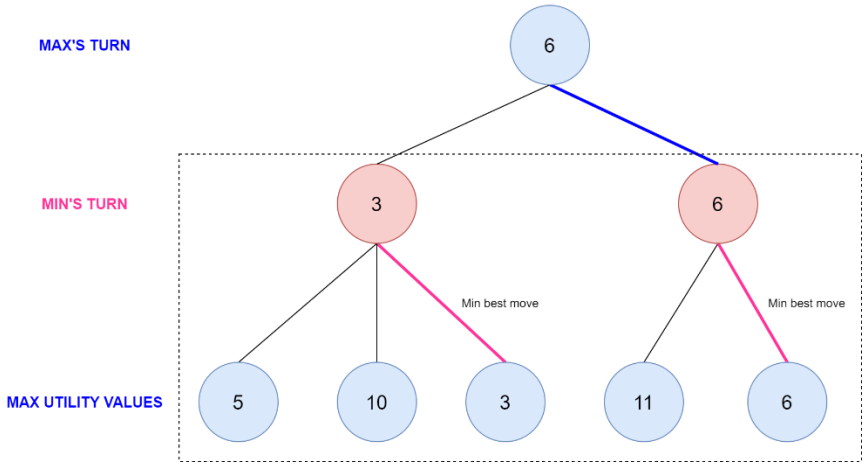


Figure 2. First move Minimax

Then, when it is MAX's turn, to maximize the score, it chooses the maximum value among 3 and 6, which is 6.

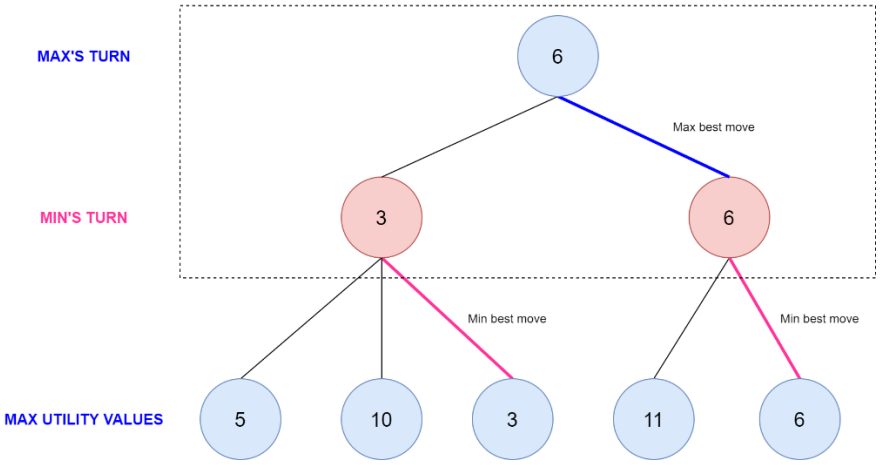


Figure 3. Second move Minimax

1.5. Improvement by alpha-beta pruning

From above, we can see that the state space of the Minimax search tree is exponential in depth, which means very large. Therefore, it consumes too much memory and time. For that reason, the evolution for Minimax algorithm has to be figured out, it is when alpha-beta pruning came up.

Alpha-beta pruning will eliminate the branch that does not affect the final result.

Quick demonstration: The figure below show that Minimax algorithm using alpha-beta pruning has done with the left-side branch and is considering the one on the right where an arrow points to. The nodes in red represent MIN's turn and the nodes in blue represents MAX's turn.

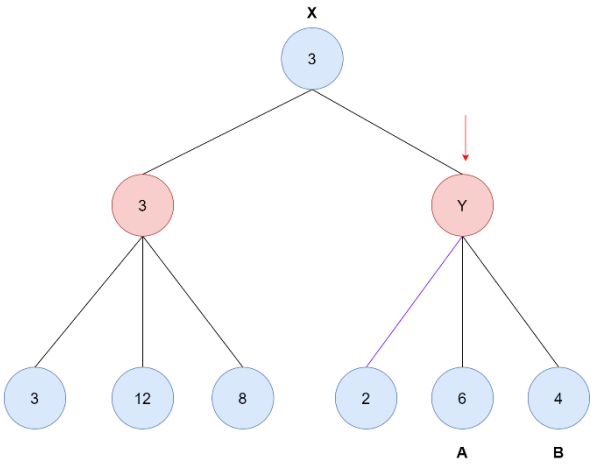


Figure 4. Alpha beta pruning

At node Y, MIN tries to get minimum value from the MAX's utility values. It first goes down on the left-most edge and find 2. Then, what is next? We will get the other branches to A and B be pruned.

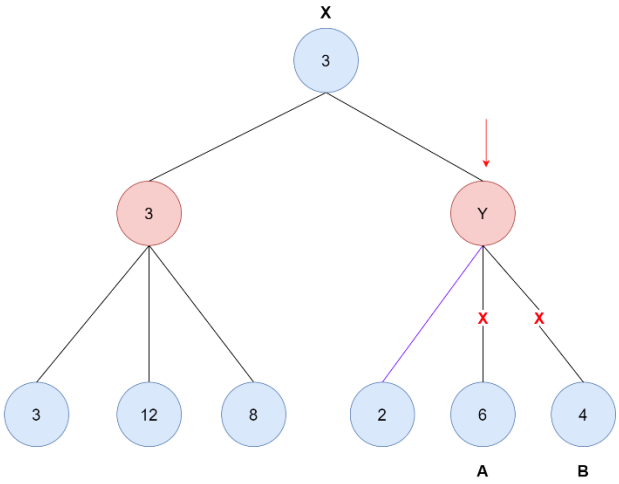


Figure 5. Pruning branches

Now we discuss how the algorithm knows that it must prune the two branches above. As we can see, the node X is in blue, which means it is MAX's turn, and it will get the maximum value among 3 and Y. With Y, it tries to get the min value among 2, 6, 4. It came on the left-most edge and see 2.

- However, 2 is less than the 3 so it is useless to get that value, because X will end up choosing 3.
- About A and B, their values are 6 and 4, all greater than 3. However, remember that this is MIN's turn, Y only chooses the minimum value which is 2.
- What if A and B are not 6 and 4 but are 0 and 1, all smaller than 2. The minimum value is now 0. However, 0 is less than 3. So again, X ends up choosing 3.

Therefore, we see that whatever A and B are, is not matter anymore, so the branches to them have been pruned.

Now we see that, with games, the Minimax search tree is surely enormous. Thanks to alpha-beta pruning, we can avoid visiting every node in the tree. Besides, it is interesting that the later the step is, the more branches we can prune, which means we can save lot of works.

1.6. Application of Adversarial Search

In conclusion, adversarial search is known as game search and is applied to enable a computer to play game against human or solve the problems which are formalized into a game. In the present days, human has successfully enable AI agents to play game. We can name many of them like:

- *AlphaGo* | DeepMind Technologies: a computer program that plays "Go"
- Leela Chess Zero | Gary Linscott: and open-sourced neural network-based chess engine
- TD-Gammon | IBM: a computer program that plays backgammon

2. A* alogrithm

As we know that, to solve a problem we have many search algorithm. Each algorithm has themselves advantages and disadvantages. For instance, in uninformed search (also know Blind search) we have Breadth-first search, Uniform-cost search, etc. This kind of search does not base on our knowledge about the world, it just try to find the solution on the current running state. Another kind of search is informed search which is based on the additional knowledge about the world. One of the most famous search algorithm is A* (A-Star) algorithm.

2.1. Idea

The first thing we need to detemine is how can we represent our knowledge about the world in the program? The additional knowledge about the problem is imparted as a function which is called heuristic function. However, if we only search base on our knowledge, in some situation it will cause subjective. The searching with only the additional knowledge called Greedy best-first search. To improve the subjection, we must use the knowledge of the explored part of our problem. This is the idea of A-Star algorithm.

To represent our knowledge, we use heuristic function called $h(n)$ which is the estimated cost from node n to the goal by ourselves. Also, to represent the knowledge of the explored part, we do the same what BFS algorithm does. We call $g(n)$ is the shortest-cost from the start to node n. Finally, the evaluate function is calculated by adding those two $f(n) = g(n) + h(n)$.

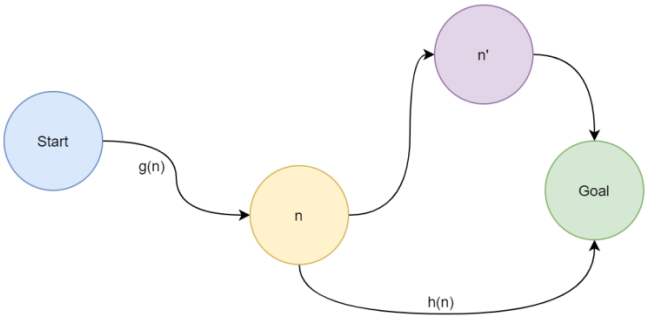


Figure 6. A* algorithm

We must clear that the heuristic function is the estimated cost by ourselves, it not the optimal cost. After that, we use the function $f(n)$ to evaluate and run in the same way as BFS does.

2.2. Problem description

Given an greyscale image represent a map. Each pixel contains the grey value from 0 to 255 provide the high of the current position in our map. Called the function $a(x, y)$ is the height (grey value). To move between two positions $A(x_1, y_1)$ and $B(x_2, y_2)$ we must ensure that $\begin{cases} |x_1 - x_2| \le 1 \\ |y_1 - y_2| \le 1 \\ |\Delta a| \le m \end{cases}$ with the difference height is $\Delta a = a(x_1, y_1) - a(x_2, y_2)$. The cost to move is:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + (\frac{1}{2}.sign(\Delta a) + 1)\,|\Delta a|$$

Input the parameter m and find the optimal path to move from two given points. Moreover, we also need to print out the number of the we touched.

2.3. Problem analysis

As we can easily see that, we can understand the condition to move between two points that is we only able to move eight directions to the adjacent node and the different height between our nodes must not greater than m:

	n*	n*	n*	
	n*	n	n*	
	n*	n*	n*	

For example, with node n above we only can move to the orange node $n*$ only when the different height of them is not higher than m.

Then we look at the cost function between two nodes. We can see that the first part is the euclidean distance $\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$. The second part can create three situations: $\begin{cases} 1.5|\Delta a| & (\Delta a > 0) \\ 0 & (\Delta a = 0) \\ 0.5|\Delta a| & (\Delta a < 0) \end{cases}$.

This mean that, on the first situation ($\Delta a > 0$), we move down from a node to another lower node, it cost 150% of the difference. On the second case, if we move from two point whose height are equal ($\Delta a = 0$), the cost is the euclidean distance (does not affacted by Δa). And the last case is the inverse of the first one, it cost only 50% of the difference.

2.4. Implementation

Our team choose the Python programming language to implement the algorithm to solve the problem above. Because of the requirement, we must manipulate with the image, so our team use the PIL (Pillow) library. You can easily install the library by using the command:

```
$ pip install Pillow
```

And to calculate complex mathematics function, we use the math library which is the standard library of Python. For easier to understand and configure, we create ourselves classes and files: constant, MyImg, MyMath, MyNode. This is the purpose of each file:

- Constant: contains the constant variable for easier to configure
- MyImg: the class to manipulate with image
- MyMath: define ourselves math formula
- MyNode: the class represent a pixel as a node in our algorithm

2.5. Function specification

Figure 7. Class and component diagram

2.5.1. Class MyImg

- Attribute
  - path: the path direct to the image
  - img: the Image class of Pillow library
  - width: the width of the picture
  - height: the height of the picture
- Method:
  - MyImg(path): Create the class with the image path
    - Input: the path of the image
  - printInfo(): Print out the information about path, width and height of the image
  - at(pos): Get the pixel at a position
    - Input: pos is a tuple contains two integer numbers which is x and y coordinate
    - Output: a tuple contains red, green, blue value and alpha (transparency) if the image has.
  - greyAt(pos): Get the greyscale value at a position (Average of R, G and B)
    - Input: pos is a tuple contains two integer numbers which is x and y coordinate
  - copy(path): Copy the current image to a new image whose path is input
    - Input: the path of image we want to create

        - Output: the class MyImg contains the new image
    - write(pos,color): Draw a pixel with the input color on the input position
        - Input: the position (tuple of x and y) we want to draw and the color (tuple of R, G and B value)
    - save(): Save the current image after call write method.
    - show(): Using the default application of operating system to open the image

2.5.2. MyMath

- abs(x): Get the absolute value of input number
    - Input: the number we want to take its absolute
    - Ouput: the absolute value
- sign(x): Get the sign value of input number
    - Input: the number we want to get the sign
    - Output: $\begin{cases} 1 & (x > 0) \\ 0 & (x = 0) \\ -1 & (x < 0) \end{cases}$

2.5.3. Class MyNode

- Attribute:
    - x: x-coordinate of the pixel
    - y: y-coordinate of the pixel
    - g: the minimum cost from the start node to this node
    - h: the heuristic value, estimated cost from this node to goal
    - f: sum of g and h
    - a: greyscale value of the pixel
    - parent: the parent node whose this node move from
- Method:
    - Node(pos,a): Construct the node with the position and greyscale value
        - Input: the position tuple of x and y; and the greyscale value
    - printInfo(): Print the information of the node to screen
    - toStr(): Cast the information of the current node to string
        - Output: A string after casted
    - euclidDistance(node): Get the euclidean distance ($\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$) to move from the current to the input node
        - Input: The next node
        - Output: The value of euclidean distance
    - deltaA(node): Get the different height (greyscale) value when it move from the current node to the input one
        - Input: The next node
        - Output: The difference height Δa

- distanceTo(node): Get the cost to move from the current node to the input node
  - Input: The next node
  - Output: The cost to move by

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} + (\frac{1}{2}.sign(\Delta a) + 1) | \Delta a |$$

- calculateF(): Calculate the evaluate function by taking the sum of g and h
- setParent(node): set the parent node of the current to the new node and re-calculate all the value of g and f
  - Input: The parent node
- setH(node): set the heuristic value for the current node and re-calculate f
  - Input: The heuristic value
- pos(): Get the position of the current as a tuple
  - Output: Tuple of x-coordinate and y-coordinate

2.5.4. Main source driver

The idea of A* algorithm is we choose the node whose $f(n)$ value is minimum in our queue (some document may call it the open-set) out and begin to add (or update) the node it can move to into the queue. By this way, we do until we take out the goal node.

Because our problem is the graph search problem, so we also need an explored set (closed-set) to avoid repeated node. With the input image, we create a matrix of the node for each pixel of the image. After that, we take out the starting and goal node to begin the algorithm.

When we add a node to the queue, we must check whether it is already in the queue. If it already sit inside, we calculate the new cost from the starting to that node with the old cost. If the new cost is less than the old one, we must update the parent of the node. In the other case, if the node was not stay inside, we easily calculate the heuristic value and put it inside the queue.

Now, let begin the algorithm. First, we will put the starting node to the queue. Then we do until we found the path (take out the goal node) or there is no more node to pop out (cannot found solution). Each time, we pop out the node whose $f(n)$ is lowest. With the node, we check whether it is the goal. If it does, we found the solution. Otherwise, we remove that node from queue, add it to explored set; and for each neighboor node who is not in the explored set, we add it to the queue.

When we found the goal node, we can take out the solution by tracing the parent node from the goal one. We use a list to put the tracing node, the result now must be found reversedly. So we do a reverse to get the path from the starting point.

Because of the requirement of problem, we must use one more set called touch to store all the touched node whenever we add it to queue. Here is the pseudo code of the algorithm described above:

```
getResult(start,goal){
	result = list()
	while (goal!=start){
		result.append(goal)
		goal=goal.parent
	}
	result.append(start)
	return reverse(result)
}

A*(start,goal){
	explored = set()
	queue = set()
	touch = set()

	queue.add(start)
	while (!queue.empty()) {
		node = min(queue,key=f)
		if (node==goal){
			return getResult(start,goal)
		}
		queue.pop(node)
		explored.add(node)
		for each neighboor in neighboorOf(node) {
			if (neighboor not in explored) {
				queue.add(neighboor)
```

```
                    }

                }

            }

}
```

To be able to implement the algorithm above, we must have some utility functions. The table below desciption the functions defined in main source:

- priority(node): Get the value to compare between nodes
  - Output: The f value of the node
- getNeighboor(nodeMat, m, node): Get all the node the can move from the current node
  - Input: Matrix of nodes, value m (maximum different height), the query node
  - Output: The list of the movable node
- createNode(img): Create the node matrix from the input image
  - Input: A MyImg object (described above)
  - Output: The matrix of node created from the image
- addNode(queue, parent, node, goal, hFunction): Add a node to the queue. If it already existed, calculate the new cost and update if it is better. If it not existed, calculate the heuristic value and add to queue
  - Input: the queue, parent node, inserting node, goal node and a pointer to the heuristic function
- drawNode(nodes,img,path,color): Copy the input image to the new path and draw all the node in the nodes list with the specified color
  - Input: the list of nodes, the image we want to copy, the path of new image, the drawing color
  - Output: A MyImg direct to the result image
- run(img, hFunc, startPos, endPos, m): Run the A* algorithm begin from startPos and the goal at endPos with the input image and parameter m. This algorithm use a specified input heuristic function
  - Input: the MyImg object direct to the image, a pointer point to the heuristic function, starting and goal position (as a tuple), the parameter m
  - Output: the list of node we must follow (path), the cost and the touched set.

With all the above function, we can do for many heuristic function by recalling the “run” function. First, we read the input file and image. Then with each heuristic function we call “run” function and write down the result to the image and output file.

3. Heuristic function

3.1. Admissibility and consistency

The interesting thing of the A* algorithm is the heuristic function. The function is imparted our knowledge about the problem. Because of the subjective, sometimes our knowledge may create the path which is not optimal. For example, we take one image to try two different heuristic functions:

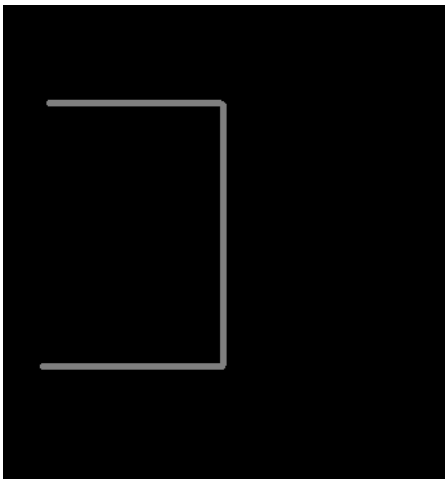


Figure 8. Experiment map 1

Now, with the first heuristic function we choose the cost function and multiple the second part by 1000 times:

$$h(n) = 1000 \,|\, \Delta a \,|$$

This means that, we estimate that the cost to move from current node to the goal is about 1000 times the difference between there height. So with the starting point on the left hand side and the goal on the right hand side. The result of the algorithm provide the path with the cost is 466.05:

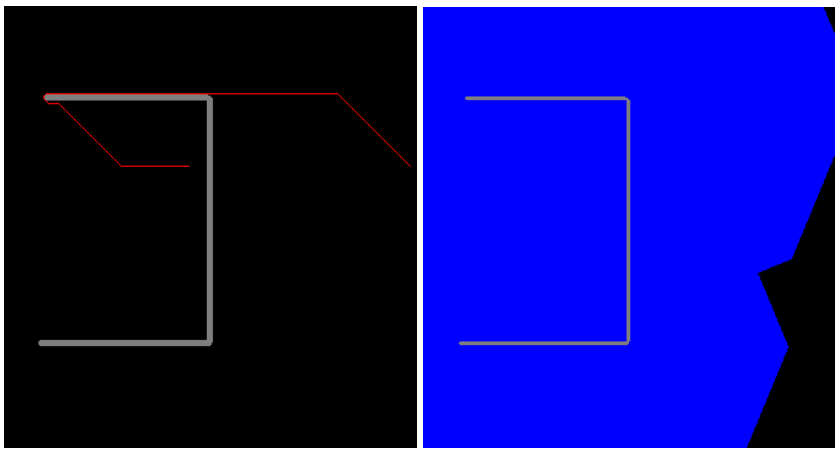


Figure 9. Path result and touched nodes

However, if we choose another heuristic function such as $h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ (Euclidean distance). The algorithm will provide the better solution with the cost is 434:

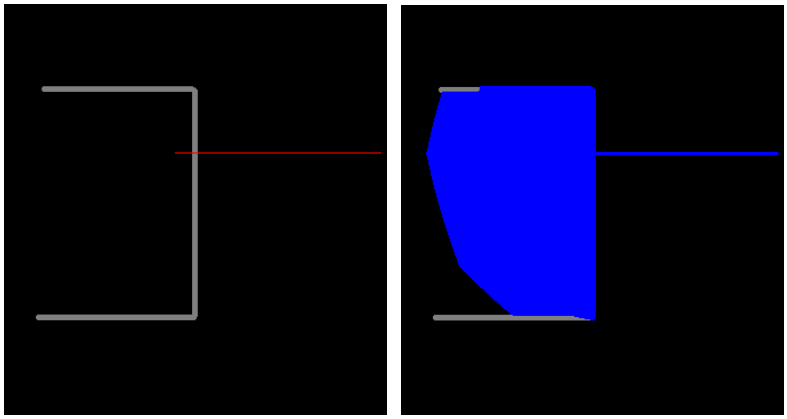


Figure 10. Path finding and touched nodes

So why the first heuristic function causes incorrectness? With the first function, we can see that the node on the grey pixel may have the estimate cost to goal very high (1000 times of the difference height). The algorithm will skip those nodes and does not try to expand those nodes. And on the black pixel, the heuristic function only estimates zero. This is why it does not try to climb up the grey zone. Moreover, if we look at the right image, the blue zone is the node it touched. On the first function, it does not touch the grey pixel. In contrast, the second function does.

To provide an "good" heuristic function, we must understand the "admissibility" of the function. A heuristic function that is admissible whenever it does not "overestimate" the real cost. Particularly, our heuristic function must satisfied: $h(n) \le h^*(n)$ for all the node n with $h^*(n)$ is the real cost to reach the goal. If the heuristic function is not admissible, it may skip the node that guide to the solution.

However, admissibility is just only ensure for the tree-search problem. On the graph-search problem, we must ensure our heuristic function is "consistent" (or can be called as monotone). To satisfied the consistency, the heuristic function must has that $h(n) \le h(n^*) + c(a, n, n^*)$. It means that for all the neighboor node n^* of n, the heuristic function of n must not higher than the heuristic function of the neighboor add the cost to move between those two nodes by doing action a. If the function is consistent, it also admissible (can prove by induction).

The best heuristic function gives the "exact" cost to move from the current node to the goal will have the algorithm run very fast. It explored the queue in the optimal path directly because $f(n) = g(n) + h(n)$ is constant and $f(n)$ of the node sit on the path is smallest.

If we do not provide the heuristic function that satisfied two properties above, it may not find out the optimal solution. However, in practice, we do not need to find the optimal one. Usually, we trade-offs

the optimality with the speed in order to in the path rapidly. The A* algorithm is mostly used in game programming. The following section we will take about some heuristic functions.

3.2. Relax problem

To give a consistent and admissible heuristic function, we need to "relax" the problem. It means that we make the problem to be easier. For example, in our problem the cost is depend on two components: euclidean distance and the difference height. We can relax the problem by skipping the difference height. By this way, we have the heuristic function $h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ (we will prove the admissibility and consistency on the follow section).

3.3. Zero heuristic

The easiest way to satisfy the admissibility and consistency is to using the "zero" heuristic function:

$$h_1 = h(n) = 0$$

We relax the problem as at any position, we can move directly to the goal with no cost. With this function, we can easily see it is admissble because $h^*(n)$ is positive ($0 \le h^*(n)$). It is also consistent because $h(n) = 0$ lead to $h(n) \le h(n^*) + c(a, n, n^*)$, both $h(n)$ and $h(n^*)$ is zero and the cost to move between two node is positive.

However, this heuristic function does not give us any information about the estimated cost to move from the current node to the goal. It makes the algorithm turn into Uniform-cost search algorithm $f(n) = g(n)$.

3.4. Euclid distance

The second heuristic function is using the Euclidean distance:

$$h_2 = h(n) = \sqrt{(x_n - x_{goal})^2 + (y_n - y_{goal})^2}$$

With this function, we are doing the relaxation as we do not consider the height. We do the problem as searching on the grid only. As we can see that on the second component of the cost function, we have

$$\begin{cases} 1.5|\Delta a| & (\Delta a > 0) \\ 0 & (\Delta a = 0) \\ 0.5|\Delta a| & (\Delta a < 0) \end{cases}$$

as we discussed in the Problem analysis section. This property give that our function is admissible and consistent. We just only need to prove that it is consistent then it also admissible.

To prove the consistency, we prove the condition $h(n) \le h(n^*) + c(a, n, n^*)$ is always true (valid). The cost to move between two nodes n_i and n_{i+1} is

$$c(n_i, n_{i+1}) = ED(n_i, n_{i+1}) + SC(n_i, n_{i+1})$$

With $ED(a,b)$ is the euclidean distance between node a and b; $SC(a,b) = (\frac{1}{2}.sign(\Delta a)+1)|\Delta a|$ is the value of second component of two nodes a and b; n_k is the goal node.

For any n_i node, we have the $\begin{cases} h(n_i) = ED(n_i, n_k) \\ h(n_{i+1}) = ED(n_{i+1}, n_k) \\ c(n_i, n_{i+1}) = ED(n_i, n_{i+1}) + SC(n_i, n_{i+1}) \end{cases}$

Easily, we can see that $h(n_{i+1}) + c(n_i, n_{i+1})$ has $ED(n_i, n_k) \le ED(n_i, n_{i+1}) + ED(n_{i+1}, n_k)$ because the eculidean distance between two points is the smallest. This lead to that the heuristic function is consistent, for all node n_i we have $h(n_i) \le h(n_{i+1}) + c(n_i, n_{i+1})$.

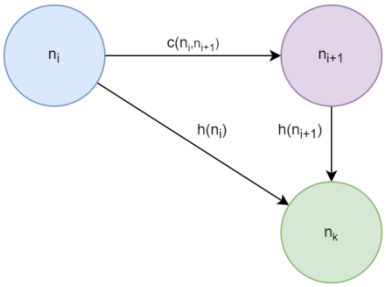


Figure 11. Consistent property

The function run very fast when the optimal path is sit on the skyway path. We can see it more deeply in the Experiment section.

3.5. Manhattan distance

The next heuristic function we may discuss is the Mahattan distance:

$$h_3 = h(n) = |x_n - x_{goal}| + |y_n - y_{goal}|$$

Mahanttan distance is one of the most famous heuristic function used in A* algorithm. The heuristic function estimate that the distance from a node to goal is calculated by take the difference of x-coordinate and y-coordinate.

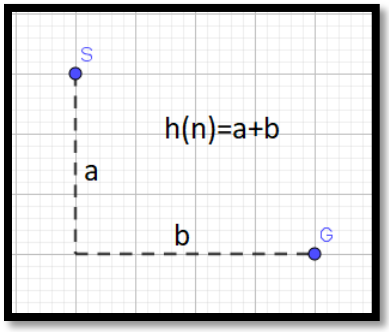


Figure 12. Manhattan distance

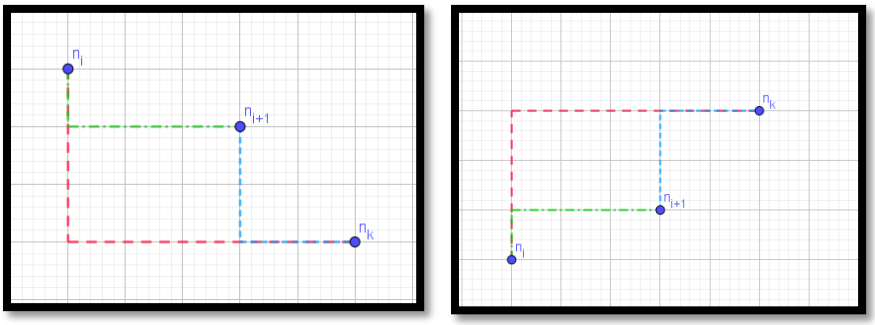
However, if we take a look at the consistency of this function, we may find out an interesting problem. As the same way we prove the consistency of the euclidean distance function. We call $MH(n_i, n_{i+1}) = | x_{n_i} - x_{n_{i+1}} | + | y_{n_i} - y_{n_{i+1}} |$ is the Mahattan distance between node n_i and n_{i+1}.

For any node n_i, we have that $\begin{cases} h(n_i) = MH(n_i, n_k) \\ h(n_{i+1}) = MH(n_{i+1}, n_k) \\ c(n_i, n_{i+1}) = ED(n_i, n_{i+1}) + SC(n_i, n_{i+1}) \end{cases}$

Now, we consider whether it is consistent or not. To prove it consistent must prove $h(n_i) \le h(n_{i+1}) + c(n_i, n_{i+1})$ is true for all nodes. The formula in this situation equivalent to

$$MH(n_i, n_k) \le MH(n_{i+1}, n_k) + c(n_i, n_{i+1})$$

If we take a look at the Mahattan function, we can see that $MH(n_i, n_k) - MH(n_{i+1}, n_k) \le MH(n_i, n_{i+1})$ and specially, the equal is happen when the node n_{i+1} sit on the rectangle constructed by node n_i and goal n_k:



The red line is $MH(n_i, n_k)$, blue line is $MH(n_{i+1}, n_k)$ and the green one is $MH(n_i, n_{i+1})$. In that situation, our inequation become:

$$MH(n_i, n_{i+1}) \le c(n_i, n_{i+1})$$

But $c(n_i, n_{i+1}) = ED(n_i, n_{i+1}) + SC(n_i, n_{i+1})$ and in the case our map nodes have the same height, the cost become $c(n_i, n_{i+1}) = ED(n_i, n_{i+1}) = \sqrt{\Delta x^2 + \Delta y^2}$ with $\begin{cases} \Delta x = \mid x_{n_i} - x_{n_{i+1}} \mid \\ \Delta y = \mid y_{n_i} - y_{n_{i+1}} \mid \end{cases}$. Now the inequation become $\Delta x + \Delta y \le \sqrt{\Delta x^2 + \Delta y^2}$. We can see that for all the positive numbers Δx and Δy, the inequation is wrong:

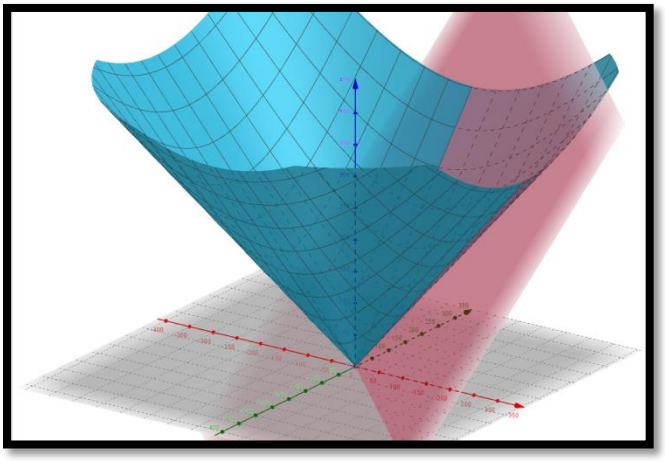


Figure 13. Visualize plane and surface

The red plane is $z_p = x + y$ and blue surface $z_s = \sqrt{x^2 + y^2}$. So on the positive part of x-coordinate and y-coordinate, we can see that $z_p \ge z_s$. By using the above graph, we can conclude $\Delta x + \Delta y \ge \sqrt{\Delta x^2 + \Delta y^2}$. **So the Manhattan heuristic function is not consistent in this problem**.

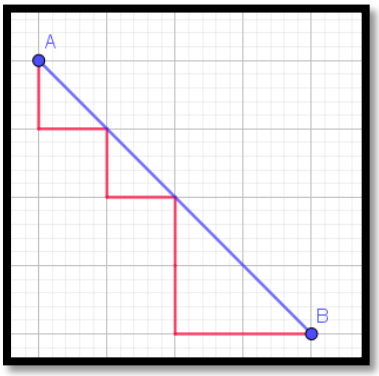


Figure 14. Manhattan distance and Euclidean distance

The blue line is the real cost and the red one is the estimated cost by Manhattan distance, we can see that it is overestimated. In some document, you may find that it say "On a square grid that allows 4 directions of movement, use Manhattan distance (L_1)"[1]. To understand the reason why, you can read about Taxicab geometry for futher explaination.

3.6. The cost function

Now, we already have two consistent heuristic functions. However, to make the algorithm run faster, we must found another heuristic function that estimated the cost closer to the real cost. We try to use the cost function as a heuristic function:

$$h_4 = h(n_i) = \sqrt{(x_{n_i} - x_{n_{goal}})^2 + (y_{n_i} - y_{n_{goal}})^2} + (\frac{1}{2}.sign(\Delta a) + 1)\,|\,\Delta a\,|$$

We can see that $h_4(n) \geq h_2(n)(\forall n)$. If the heuristic function h_4 is admissible also this called dominance. In the case it is dominance, we can choose the function h_4 because it is closer to the real cost than the h_2:

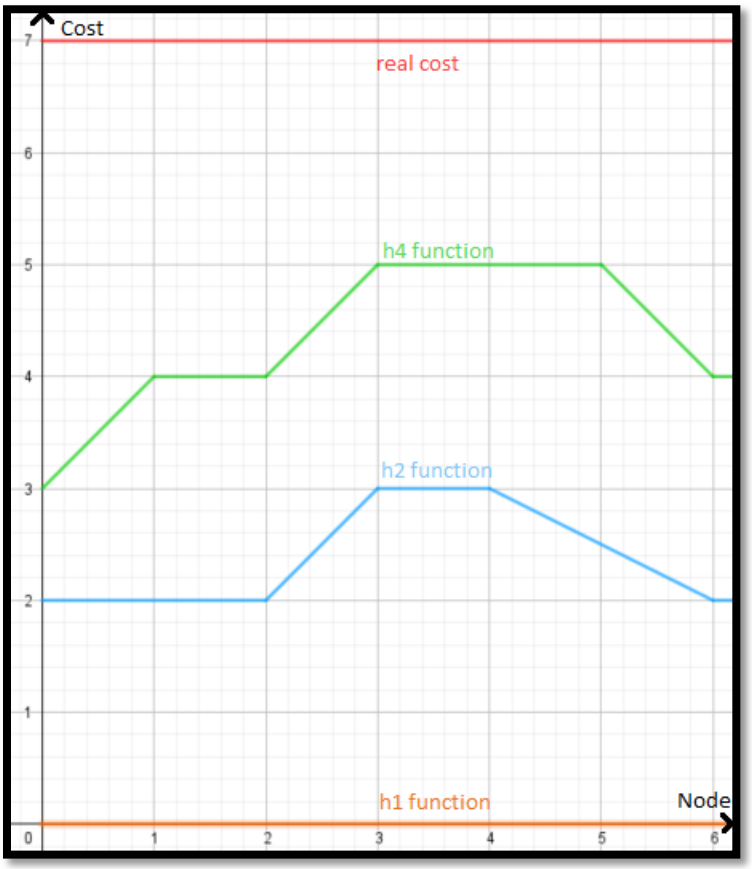


Figure 15. Dominance

In this section we try to prove that this heuristic function is consistent. We prove that $h(n_i) \le h(n_{i+1}) + c(n_i, n_{i+1})$:

$$ED(n_i, n_k) + SC(n_i, n_k) \le [ED(n_{i+1}, n_k) + SC(n_{i+1}, n_k)] + [ED(n_i, n_{i+1}) + SC(n_i, n_{i+1})]$$

If we look at the euclidean distance only we can see that $ED(n_i, n_k) \le ED(n_{i+1}, n_k) + ED(n_{i+1}, n_k)$ is always true for all the nodes (skyway is the shortest path). Now, if we have that $SC(n_i, n_k) \le SC(n_{i+1}, n_k) + SC(n_i, n_{i+1})$, the consistency is proved.

In our problem, we have two cases that we must consider. The first one is when the node n_{i+1} has the height between the height of n_i and the height of n_k:

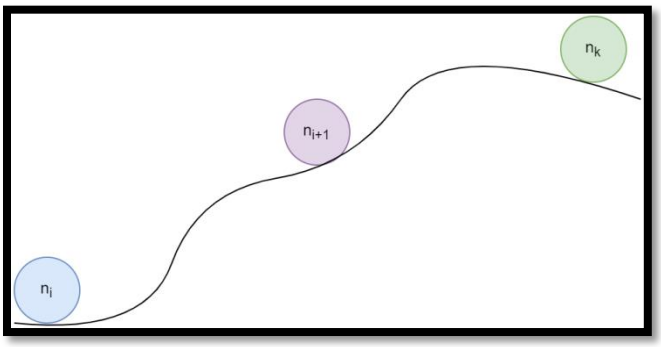


Figure 16. Situation 1 of height

To prove this, we must make clear the different of $SC(n_1, n_2) \ne SC(n_2, n_1)$, we can see that $SC(n_i, n_{i+1}) + SC(n_{i+1}, n_k) = c.|\Delta a(n_i, n_{i+1})| + c.|\Delta a(n_{i+1}, n_k)|$ with $c = \begin{cases} 1.5 & (\Delta a > 0) \\ 0 & (\Delta a = 0) \\ 0.5 & (\Delta a < 0) \end{cases}$. Because the node n_{i+1} sitting between n_i and n_k, the sign of $\Delta a(n_i, n_{i+1})$ and $\Delta a(n_{i+1}, n_k)$ is the same. By that property, we can expand the absolute function which makes our formula become:

$$SC(n_i, n_{i+1}) + SC(n_{i+1}, n_k) = c.|\Delta a(n_i, n_{i+1}) + \Delta a(n_{i+1}, n_k)|$$

$$SC(n_i, n_{i+1}) + SC(n_{i+1}, n_k) = c.|a(n_i) - a(n_{i+1}) + a(n_{i+1}) - a(n_k)|$$

$$SC(n_i, n_{i+1}) + SC(n_{i+1}, n_k) = c.|a(n_i) - a(n_k)| = SC(n_i, n_k)$$

Finally, we can conclude that this function is consistent if the node n_{i+1} has the height between n_i and n_k. So how about if the node n_{i+1} does not sit between them? Now, we look at the picture below:

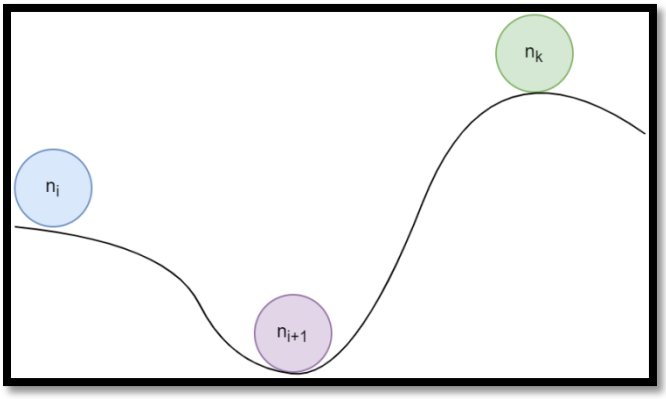


Figure 17. Situation 2 of height

In this situation, we split the formula by inserting an "imagine" node n_x whose height is same as n_i. Do the same way, we have $SC(n_i, n_{i+1}) + SC(n_{i+1}, n_k) = c_1.|\Delta a(n_i, n_{i+1})| + c_2.|\Delta a(n_{i+1}, n_k)|$. Then we insert n_x:

$$SC(n_i, n_{i+1}) + SC(n_{i+1}, n_k) = c_1.|\Delta a(n_i, n_{i+1})| + c_2.(|\Delta a(n_{i+1}, n_x)| + |\Delta a(n_x, n_k)|)$$

We can insert the node n_x like this because we have already prove that if n_x sit between n_{i+1} and n_k, we can split the formula by the above situation. Now, we distribute the constant c_2 into:

$$SC(n_i, n_{i+1}) + SC(n_{i+1}, n_k) = c_1.|\Delta a(n_i, n_{i+1})| + c_2.|\Delta a(n_{i+1}, n_x)| + c_2.|\Delta a(n_x, n_k)|$$

And we know that the height of n_x is same as n_i so that $\begin{cases} \Delta a(n_x, n_{i+1}) = \Delta a(n_i, a_{i+1}) \\ \Delta a(n_x, n_k) = \Delta a(n_i, n_k) \end{cases}$

$$SC(n_i, n_{i+1}) + SC(n_{i+1}, n_k) = c_1.|\Delta a(n_i, n_{i+1})| + c_2.|\Delta a(n_{i+1}, n_i)| + c_2.|\Delta a(n_i, n_k)|$$

The component $c_2.|\Delta a(n_i, n_k)|$ is also equal to $SC(n_i, n_k)$. So in this situation, it also consistent because:

$$SC(n_i, n_k) \le SC(n_{i+1}, n_k) + SC(n_i, n_{i+1})$$

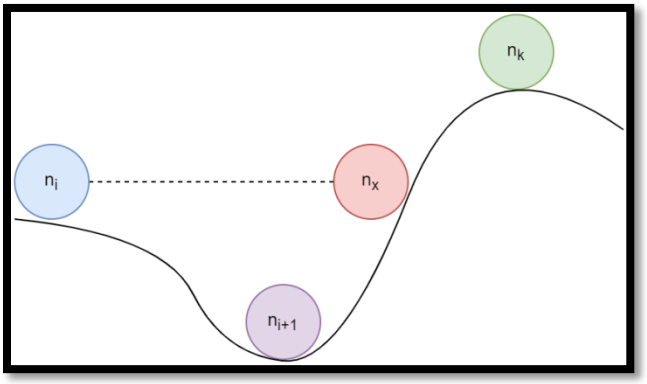


Figure 18. Imagine for situation 2

4. Experiment

To see the effeciency between those heuristic function, we will do some experiment. The first experiment is find the path from (74;213) to (96,311) with the parameter $m = 10$ on the below image:

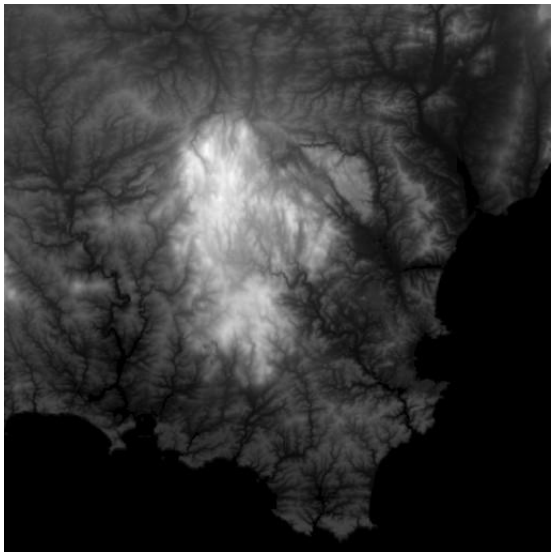


Figure 19. Experiment map

This is the result of our four functions:

Function	h1	h2	h3	h4
Path cost	317.54	317.54	317.54	317.54
Number of node	42400	19531	15782	9619

On this first experiment, all our functions give the optimal path. However, we can see that the function $h_1 = 0$ runs very slow, it must calculate about 42400 nodes. Moreover, as we know that the function h_4 is dominance than h_2 so it runs much faster. The heuristic function h_4 has the component to evaluate the height (greyscale) value of nodes. We can see it does not check the white zone as the h_2 does:

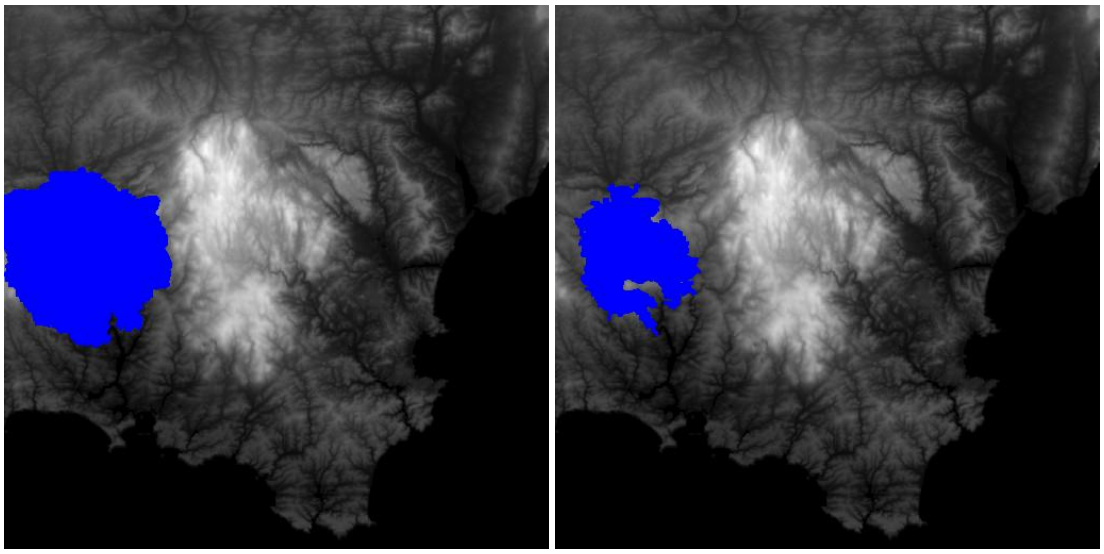


Figure 20. Touched nodes of h_2 (left) and h_4 (right)

But with the above experiment, we cannot see that the heuristic Manhattan h_3 does not consistent. Now, we take another experiment to see more clearly that h_3 does not give the optimal solution. We find the path from (150;130) to (180,160) with the parameter $m = 255$ means that we can go to any node neighboor does not matter the difference of height:

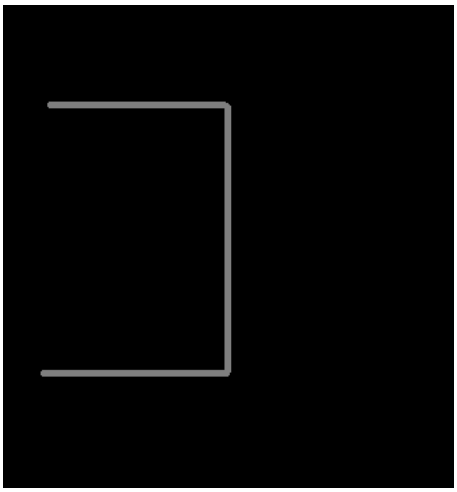


Figure 21. Experiment map

And this is the result:

Function	h1	h2	h3	h4
Path cost	296.43	296.43	301.11	296.43
Number of node	57874	26821	23117	25458

We can see that the path cost given by h_3 is not optimal. However, h_3 is the fastest heuristic function in this case. In pratice, people usually take this trade-off. They do not need to find the "optimal" path, they just need to find a "good" path in a acceptable time.

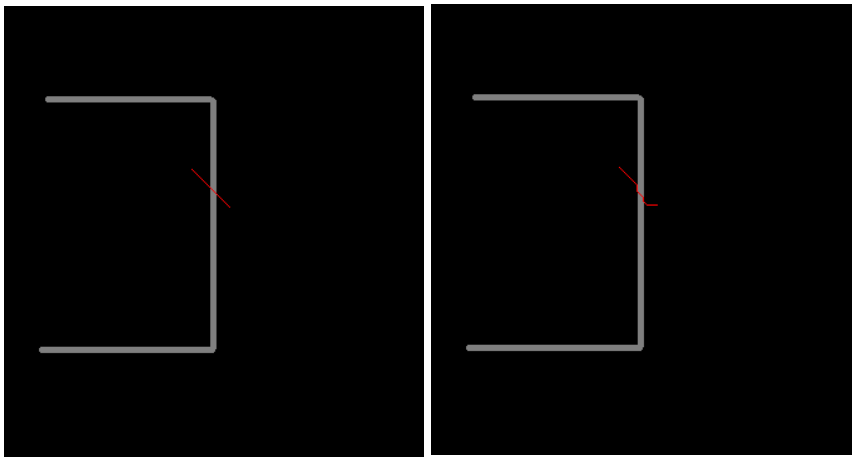


Figure 22. Path result of h1 (left) and h3 (right)

5. Improvement study

From the discussion of the heuristic number 4, we know about the dominance between the heuristic. To improve the efficency of heuristic function, we can use two techniques: dominance and semi-lattice.

5.1. Dominance

If we have two heuristic functions that already admissible, the dominance is occured when $\forall n : h_a(n) \geq h_b(n)$. This is case, we should choose h_a as it is closer to the exact cost.

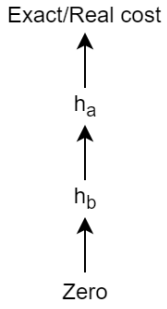


Figure 23. Dominance

This is the situation of heuristic function number 2 and number 4.

5.2. Semi-lattice

But how about if we have two heuristic functions that does not dominance? In some case, the h_a is closer than h_b and some case $h_b > h_a$?

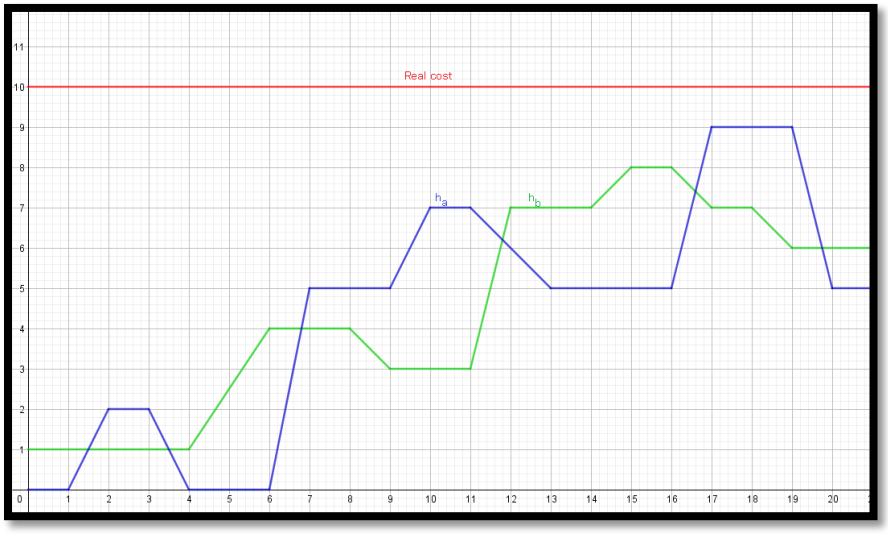


Figure 24. Heuristic functions that does not dominance

In this case, we will take a new heuristic whose value is the maximum of our two heuristic function. We can easily see that the new function is also admissible as it lower than the exact cost:

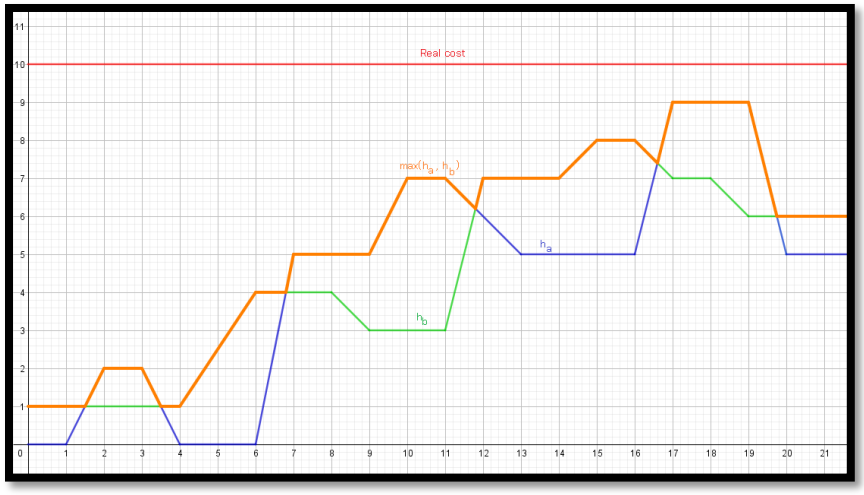


Figure 25. Max value of two heuristic functions

The new heuristic function is the closest heuristic function. If we have k heuristic functions $h_1, h_2, \cdots, h_k$. We can create a heuristic function using semi-lattice $h(n) = max(h_1(n), h_2(n), \cdots, h_k(n))$:

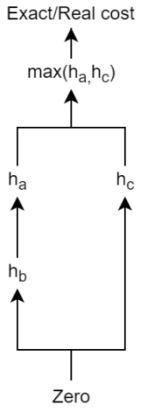


Figure 26. Semi-lattice

5.3. Other optimization

There are many other optimization you can apply into this A* algorithm. For example, we have the Iterative-deeping A* (called as IDA*) by mixing the Iterative-deeping Search and A* algorithm. In this algorithm, instead of limit the depth of search tree, we limit the $f(n)$.

Or when the memory is limit, we cannot run normal A*. We can use the Memory-bound A* (called MA*). It runs like A* does but it will delete the worst node (means the node whose $f(n)$ is largest) when the memory is full.

Or if you want to speed up the calculation, you can read about the ALT A* algorithm. ALT A* algorithm uses "landmarks" to preprocess the pathfinding graph. The paper talks about ALT A* algorithm will be leave at the reference [6].

For more complex problem, the data would be huge (especially using "landmarks" in ALT A*). To using the algorithm in pratice, we need to compress the data. The paper shows the result when compress the landmarks also references below [7].

6. References

1. Theory.Stanford.Edu
2. Brilliant.org
3. S. J. Russell and P. Norvig - Artificial Intelligence: A Modern Approach
4. A. Felner, U. Zahavi, R. Holte, J. Schaeffer, N. Sturtevant and Z.Zhang - Inconsistent Heuristics in Theory and Practice
5. D. Suter, Q. Li – University of Adelaide – Informed Search course
6. A. V. Goldberg, C. Herrelson – Microsoft Research – Computing the shortest path: A* Search Meets Graph Theory
7. M. Goldenberg, N. Sturtevant, A. Felner, J. Schaeffer - The Compressed Differential Heuristic*