UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY
CLASS 19CLC5

ARTIFICIAL INTELLIGENCE
PROJECT 01 - OPTIMAL PATH FINDING

INSTRUCTOR
Mr. Ngo Dinh Hy

Group member
19127268 – Nguyễn Ngọc Thanh Tâm
19127511 – La Ngọc Hồng Phúc (Leader)

Contents

1. Adversarial Search

a. Introduction

In many applications, there might be more than one agent looking for a goal. This leads to a competitive environment in which agents' goals are in conflict.
In simple words, adversarial search is a search where two or more players with contrary goals are exploring the same search space for solution, often known as games.
The strategy or game approach of each player varies as the opponent's move. A player in Adversarial Search may take a maximiser role or minimizer role depending on the game situation and the intent of the opponent. Maximiser will always try to maximize the gain, while Minimizer will try to control the damage and minimize the loss.

b. Importance

Games are too hard to be solved. For example, chess has an average branching factor of about 35, and games often go 50 moves by each player, so the search tree has about 35^{100} or 10^{154} nodes while the graph has about 10^{40} nodes.
This causes a problem: making some decision even when calculating the optimal decision is impractical. To solve game efficiently, we formulate it as a search problem.
Adversarial Search is used in games to figure out a strategy. The algorithm searches through the possibilities and picks the best move. It considers all possible actions available at a time and then evaluate future moves based on these options.
Every game has a different set of winning conditions and the algorithm use it to find the set of moves which leads to the optimal solution.

c. Different game scenarios using Adversarial Search

Games with perfect information: the environment is fully observable, and all players know the game structure. They can clearly see each other moves and is perfectly informed of all the events which have previously occurred. Examples are Chess, Checkers, Go, …
Games with imperfect information: players have partial knowledge about the current state of the game. They need to balance all possible outcomes when making a decision while in perfect information games, the optimal move for each player is clearly defined. Examples are Poker, Bridge, …
Deterministic games: games in which player actions lead to predictable outcomes. Examples are Chess, Tic-tac-toe, …
Non-deterministic games: also called stochastic games, where a factor of chance or luck is involved which leads to unpredictable events. Examples are Backgammon, Poker, …
Zero-sum games: the total payoff to all players is the same for every game instance. Depending on the game environment and game situation, each player will either try to maximize the gain or minimize the loss. Examples are Chess, Tic-tac-toe, …

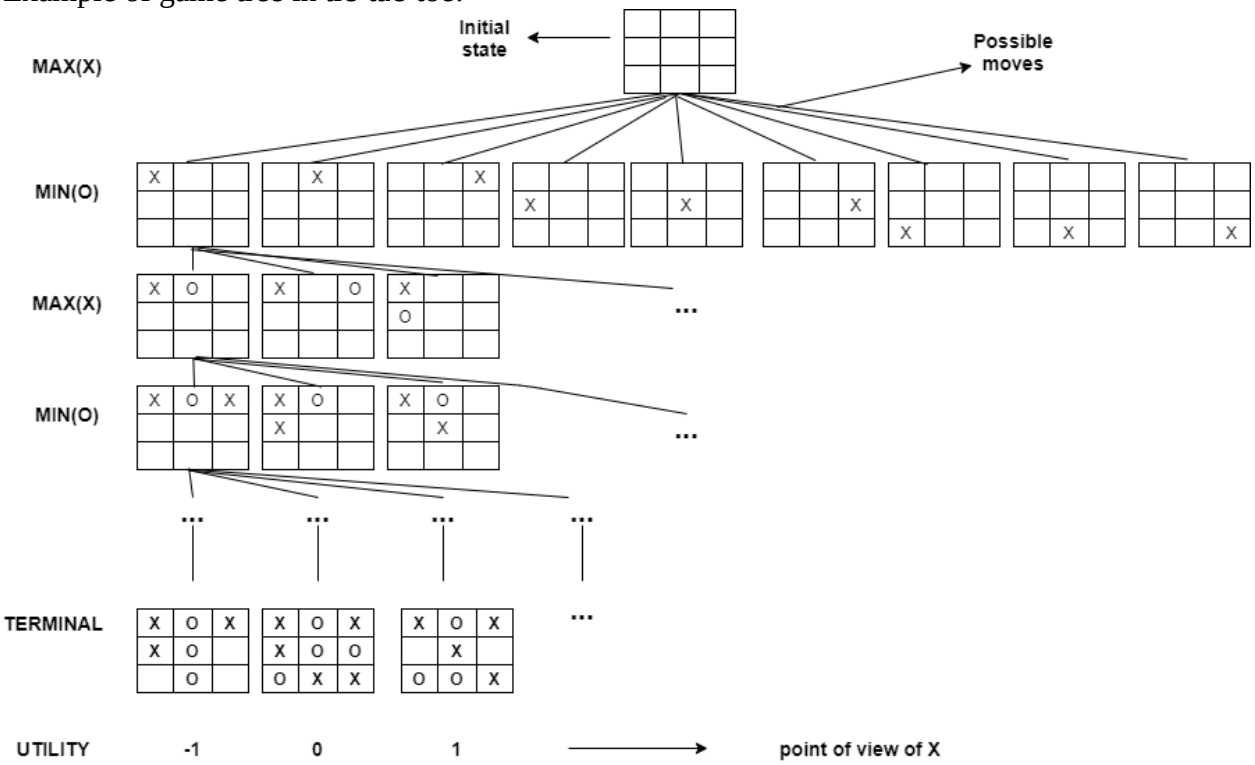
d. How it works

Formulated as a search problem with the following elements:

- S_0: initial state of the game along with configurations.
- $Player(s)$: defines which player has the move in a state.
- $Action(s)$: returns the set of legal moves in a state.
- $Result(s, a)$: defines the outcome of a move made in a state.
- $Terminal\ test(s)$: checks whether the game is over (win, lose, draw) or not, returns true when the game is over and false otherwise.
- $Utility(s, p)$: defines the final numeric value for a game that ends in terminal state s for a player p.

Game tree: a tree where the node are game states and the edges between nodes corresponds to moves. The difference between game tree and search tree is that in game tree, there are opponents. Root node represents the current configuration, player decides the best next single move to make based on the evaluator value. Each level of the tree has nodes which are all MAX or all MIN.

Example of game tree in tic-tac-toe:



Assume that our player is X, so we want X to achieve the best result. When it is X's turn to move, the root is labelled a "MAX" node, otherwise it is labelled a "MIN" node, indicating the opponent's turn. At the beginning, X has 9 possible moves to choose and after that it is the opponent's turn. The process is repeated until the game reaches a terminal state. The utility value can be -1, 0 or 1 from the point of view of each player. In this example, -1 means that X loses the game, 0 means that it is a draw and 1 means that X wins.

e. Algorithms

There are 2 algorithms used in Adversarial Search: *minimax algorithm* and *alpha-beta pruning*.

Minimax algorithm is a backtracking algorithm. It uses a simple recursive computation of the minimax values at each successor states and performs a depth-first search algorithm for the exploration of the complete game tree. The minimax values are backed up through the tree as the recursion unwinds.
Alpha-beta pruning is an improved version of minimax algorithm. This technique prunes away branches which cannot lead to the final decision.

2. A* Searching Algorithm

a. Prerequisite

Require library: Pillow, NumPy.

Source code:

- *astar.py*: standard implementation of A*.
- *alt.py*: ALT, a modified A*, using precomputed data and triangle inequality for faster and more efficient pathfinding.
- *util.py*: Utility functions and heuristic functions.
- *test.py:* test for both A* and ALT
- *landmarks.npz*: precomputed data with 8 landmarks and limit = 10. Use `ALT.init_landmarks` to generate.

b. Pseudocode with explanation

A* search algorithm

```
1  # A* finds a path from start to goal.
2  # h is the heuristic function. h(n) estimates the cost to reach goal from node n.
3  # limit is an integer number. A node can walk to its neighbor if delta height is smaller
4  # than the limit
5  function a_star(start, goal, map, h, d, limit):
6      # The set of discovered nodes that may need to be (re-)expanded.
7      # Initially, only the start node is known.
8      # This is usually implemented as a priority queue.
9      open_set = {start}
10
11     # For node n, came_from[n] is the node immediately preceding it on
12     # the cheapest path from start to n currently known.
13     # Initially, came_from contain start node
14     came_from = {start: None}
15
16     # For node n, g_score[n] is the cost of the cheapest path from
17     # start to n currently known.
18     g_score = map with default value of Infinity
19     g_score[start] = 0
20
21     # For node n, f_score[n] := g_score[n] + h(n). f_score[n] represents our current
22     # best guess as to how short a path from start to finish can be if it goes
```

```
    # through n.
    f_score = {start: h(start)}

    while open_set is not empty:
        current = get node in open_set with lowest f_score
        if current == goal:
            return reconstruct_path(came_from, current)
        open_set.remove(current)
        for each neighbor of current: # See function get_neighbors.
            if abs(map[current] - map[neighbor]) > limit:
                continue
            # d(current, neighbor) is the cost from current to neighbor
            # new_g_score is the distance from start to the neighbor through current
            new_g_score = g_score[current] + d(current, neighbor)
            if new_g_score < g_score[neighbor]:
                # This path to neighbor is better than any previous one. Record it!
                g_score[neighbor] = new_g_score
                # h(neighbor, goal) is the estimate cost from neighbor to the goal
                f_score[neighbor] = new_g_score + h(neighbor, goal)
                came_from[neighbor] = current
                # If open_set is a priority queue, checking for an element can be
                # expensive so we can skip the checking and just push into the queue.
                # Note that this will make the algorithm revisit some locations more
                # than necessary.
                if neighbor not in open_set:
                    open_set.add(neighbor)
    # Open set is empty, but goal was never reached
    return failure
```

```
function reconstruct_path(came_from, current):
    path = empty array
    while current != start:
        path.prepend(current)
        current = came_from[current]
    return path
```

```
function get_neighbors(current, map, limit):
    # Eight directional movement
    moveset = ((-1, -1), (0, -1), (1, -1), (-1, 0), (1, 0), (-1, 1), (0, 1), (1, 1))
    neighbors = []
    for each move in moveset:
        potential_neighbor = (current.x + move.x, current.y + move.y)
        if potential_neighbor is in_bounds and under_limit:
            neighbors.append(potential_neighbor)
```

```
9       return neighbors
```

c. Advantages and disadvantages

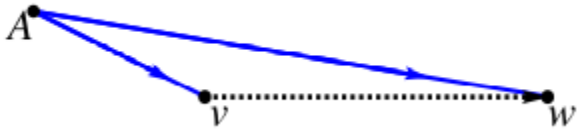
A* search algorithm is an informed search algorithm because it chooses the best direction to explore at each stage rather than blindly searches every available path. So, it is relatively quick comparing to uninformed search algorithms. A* can be used to solve very complex problems: scheduling problem. Although A* is complete, it is not guaranteed to find the shortest path since it heavily depends on the heuristic function $h(n)$ used. A* has a drawback which is related to complexity issues. It keeps all the generated nodes in the memory; therefore, it is not practical for large-scale problems.

d. ALT

ALT (**A*** + **L**andmarks + **T**riangle inequality) is a variant of A* search, whose main idea is to use landmarks and triangle inequality to compute a feasible potential function. This algorithm is divided into two phases.

On preprocessing phase, ALT first selects a set of *k* landmarks and precomputes the distance to and from these landmarks for every node in the map. **Landmark location and quantity can affect quality and optimality of the algorithm**. More landmarks give better heuristic cost on runtime but require more memory (each landmark needs *O(n)* space). You can generate landmarks by picking **random**, using **greedy farthest selection** (pick the node with greatest shortest-path distance as next landmark), or **predefined location** (see Experiment result).

On executing phase, ALT behave exactly like A*, except the heuristic cost is determined by differential of distance from node *v* to the *best landmark A* and distance from *that landmark* to node *w*, hence can be written as $h(v,w) = |dist(A,v) - dist(A,w)|$. A good landmark appears "before" *v* or "after" *w*, which will give the maximum heuristic cost.



ALT preprocessing phase

```
1   # Compute distances between k landmarks and all nodes
2   function precompute_landmarks(landmark_nodes):
3       dist_arr = empty array
4       for each landmark in landmark_nodes:
5           # Compute distances between a landmark and all nodes
6           landmark_dist = dijkstra(landmark)
7           dist_arr.append(landmark_dist)
8       return dist_arr
```

ALT heuristic

```
function landmark_heuristic(from_node, to_node):
    h_cost = empty array
    for each dist in dist_arr:
        h_cost.append(abs(dist[from_node] - dist[to_node]))
    return max(h_cost) # Return the best distance
```

3. Heuristic functions

Consider the true distance as follow, with $\Delta a = a(x_1, y_1) - a(x_2, y_2)$ (the height difference between 2 states):

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

Let $\Delta x = x_1 - x_2,\ \Delta y = y_1 - y_2$. We can rewrite the formula:

$$G = \sqrt{\Delta x^2 + \Delta y^2} + \left(\frac{1}{2}.sgn(\Delta a) + 1\right).|\Delta a|$$

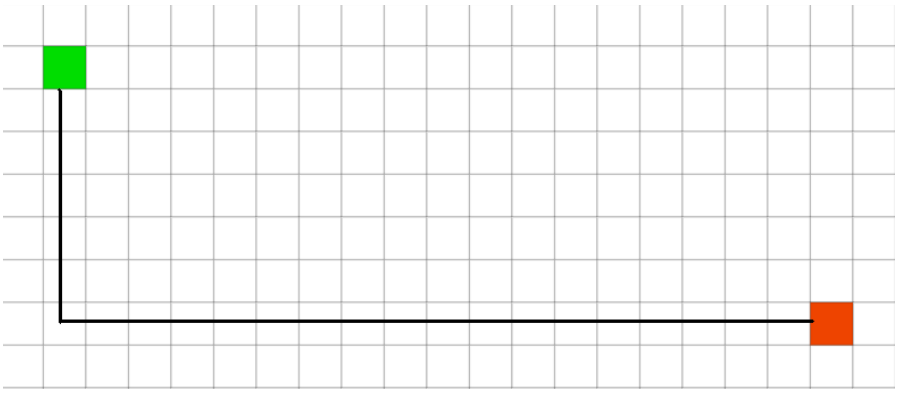
a. Manhattan distance

$$D_{Manhattan} = |\Delta x| + |\Delta y|$$

The Manhattan distance computes the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates, respectively. It is not admissible for 8 directional movement.

Prove: Assume that $|\Delta a| = 0$, Manhattan distance will overestimate the distance between a state and the states diagonally accessible to it.

Example: $Manhattan_{(0,0)\to(1,1)} = 2 > G_{(0,0)\to(1,1)} = \sqrt{2}$



b. Euclidean distance

$$D_{Euclidean} = \sqrt{\Delta x^2 + \Delta y^2}$$

$\rightarrow D_{Euclidean} \leq$ true cost in all states $\rightarrow$ Euclid distance is admissible

The Euclidean distance simply returns the straight-line distance between current cell and goal.

c. Chebyshev distance

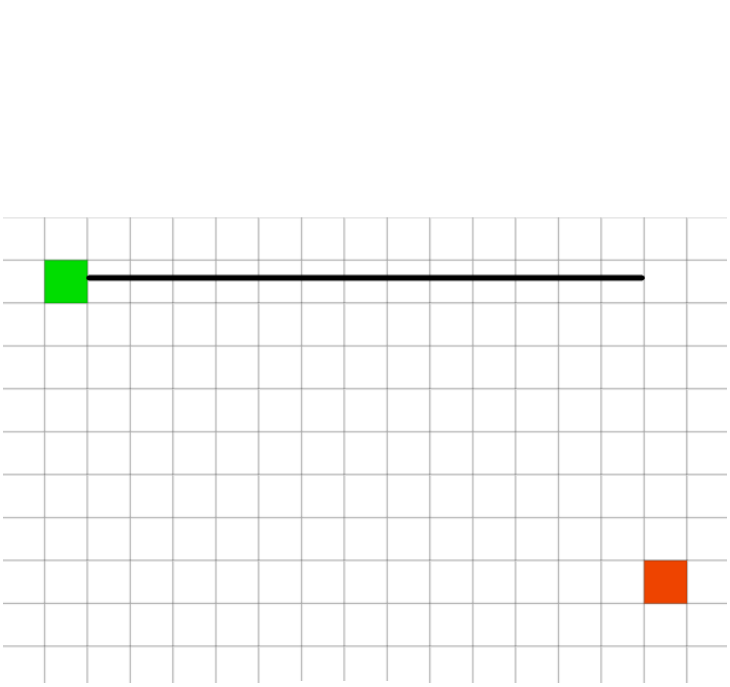
$$D_{Chebyshev} = max(|\Delta x|, |\Delta y|)$$

We can see that $\sqrt{a^2 + b^2} \geq |a|$ and $\sqrt{a^2 + b^2} \geq |b|$ with $a = \Delta x$ and $b = \Delta y$

So $D_{Chebyshev} \leq \sqrt{\Delta x^2 + \Delta y^2}$ < true cost in all states

→ Chebyshev distance is admissible

The Chebyshev distance returns the maximum absolute distance of current cell's coordinates and goal's coordinates respectively.



$\Delta x > \Delta y$

$\Delta x < \Delta y$

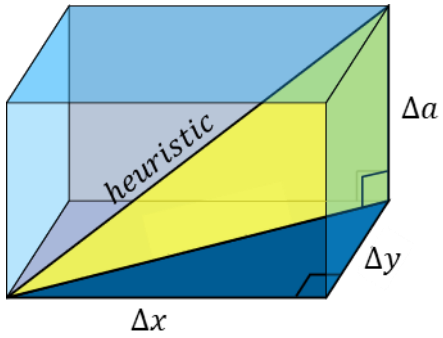
d. Octile distance

$$D_{Octile} = |\Delta x| + |\Delta y| + (\sqrt{2} - 2)min(|\Delta x|, |\Delta y|)$$

$$\begin{cases} (\sqrt{2} - 1)|\Delta x| + |\Delta y| \; with \; |\Delta x| < |\Delta y| \\ (\sqrt{2} - 1)|\Delta y| + |\Delta x| \; with \; |\Delta y| < |\Delta x| \end{cases}$$

e. Euclidean distance 3D

$$D_{Euclid\_3d} = \sqrt{\Delta x^2 + \Delta y^2 + \Delta a^2}$$



f. Diagonal distance 3D

$$D_{Diagonal\_3d} = (D3 - D2)\mathrm{a} + (\mathrm{D2} - \mathrm{D1})\mathrm{c} + \mathrm{D1b}$$

With $a = \min(|\Delta x|, |\Delta y|, |\Delta z|)$, $b = \max(|\Delta x|, |\Delta y|, |\Delta z|)$, $c = |\Delta x| + |\Delta y| + |\Delta z| - a - b$ and $D1 = 1, D2 = \sqrt{2}, D3 = \sqrt{3}$

g. Weighted Manhattan

$$D_{Weighted\_Manhattan} = \frac{|\Delta x|}{|x_1| + |x_2|} + \frac{|\Delta y|}{|y_1| + |y_2|} + \frac{|\Delta z|}{|z_1| + |z_2|}$$

4. Optimization and real-world method

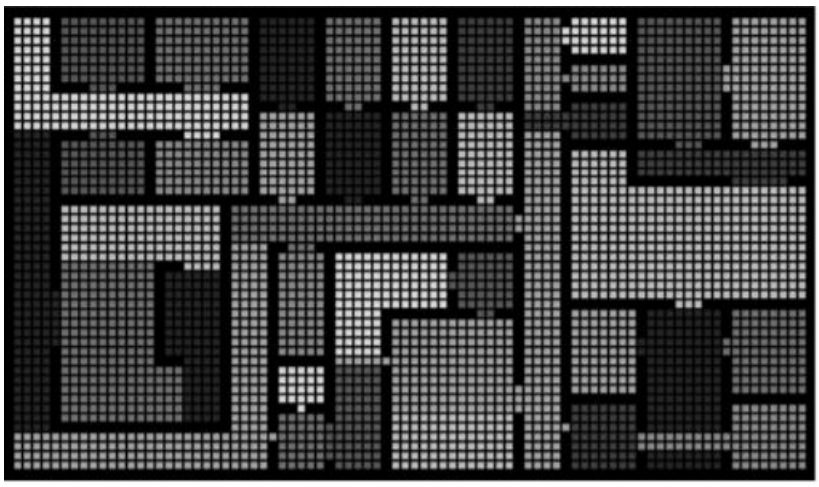
Without making changes to the original A* algorithm, we can reduce the running time by considering the data structure being used. A hash table might be the best choice because it allows constant time storing and searching for node. A priority queue is the best way to maintain an open list and it can be implemented by a binary heap. Using efficient data structure can help speed up A* significantly.

A* extends Dijkstra's algorithm by introducing heuristic approach which improves the computational efficiency. The optimality of A* depends a lot on the heuristic function used. It is ideal to construct a consistent heuristic which will minimize the running time and find best optimal path. In large and complex game maps, the typically used heuristic functions are not efficient as they can result in exploring the entire map to find a path between two distant locations.

In our problem, the constraints of movement which states that we can only move between points only when the height difference is less than or equal to a given parameter m so this constraint can be considered obstacles. We can formalize this problem as a map with obstacles and we need to navigate from start to goal avoiding obstacles.

Typically used heuristics still can be used in our problem. But, in real-time strategy games, we need to construct better heuristics to achieve optimal result. Two effective heuristics for estimating distances are *dead-end* heuristic and *gateway* heuristic which can reduce the exploration and time complexity of A* searching algorithm. Dead-end heuristic avoids areas which lead to a dead-end and gateway heuristic pre-calculates the distances between

entrances/exits of the areas.
Before we apply these two heuristics, the game map needs to be decomposed into several smaller areas, which can be called rooms and corridors. We use the decomposition algorithm to divide the map into zones. The algorithm requires a tile-based map with information for each tile whether it is passable or not and it builds borders as it encounters tiles which satisfy certain conditions.

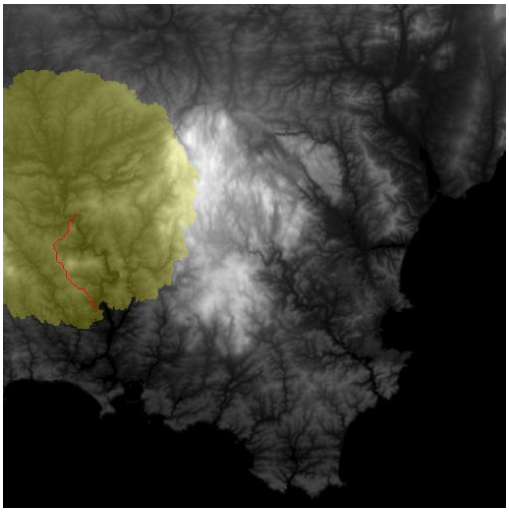


Example of area decomposition

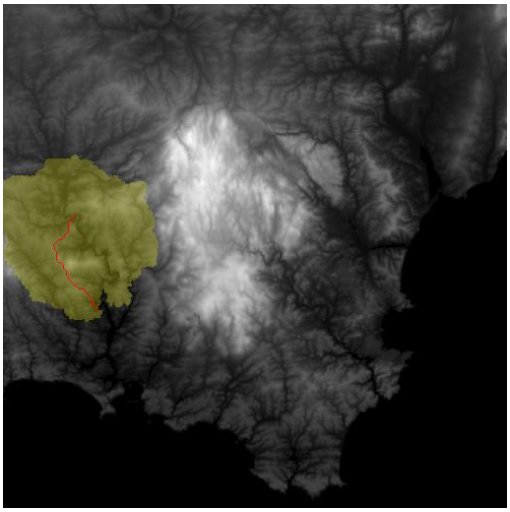
Dead-end heuristic can identify which room leads to a dead-end and therefore there is no need to explore such rooms. Gateway heuristic pre-calculates the distances between entrances/exits of the area. For each gateway we calculate the path distance to all other gateways. Alternatively, we can calculate only the distances between gateways within each room and then use a small search to accumulate the total cost.
Overestimating the heuristic cost a little bit may result in exploring much fewer nodes than non-overestimation heuristic approach. For example, Manhattan distance heuristic in 8-puzzle problem is admissible because we can only move in 4 directions. While in our problem which allows us to move in 8 directions, Manhattan distance heuristic can overestimate the heuristic cost.
As we have mentioned above, one drawback of A* is the huge amount of memory required. An alternative approach to reduce space requirements in A* is to use IDA* (Iterative Deepening A*) – a variant of A* algorithm. In IDA*, at each iteration, the cut-off value is the smallest f-value of any node that exceeded the cut-off on the previous iteration. It avoids the substantial overhead associated with keeping a sorted queue of nodes.

5. Experimental Results

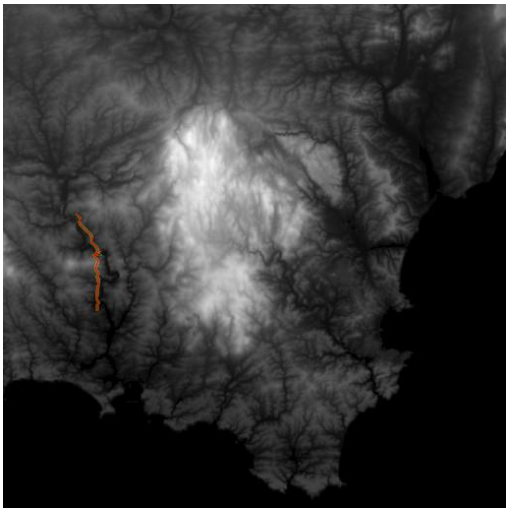
We evaluated all the above heuristics and the ALT pathfinding algorithm with ***start = (74;213), goal = (96;311)*** and ***limit = 10***. We also add UCS to prove A* is faster and Euclidian Squared heuristic to see how inadmissible heuristic can ruin the algorithm. For ALT, we choose 4 corners and 4 side midpoints of the map as landmarks.



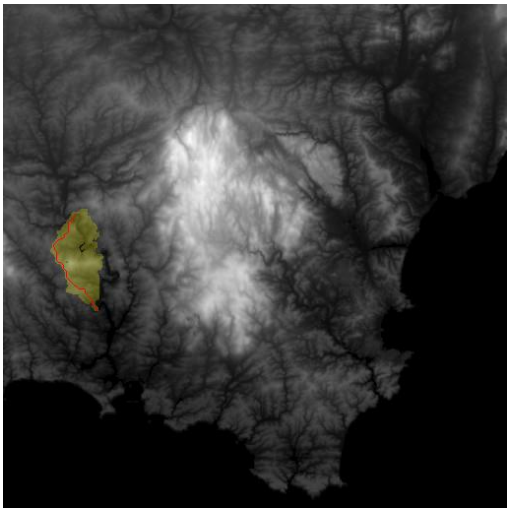
UCS



A* with Euclidean heuristic



ALT with Euclidean Squared heuristic



ALT with 8 landmarks

Demonstrate node expansion (yellow) and path (red)

No.	Heuristic	Time	Examined Nodes	Total Cost
1	UCS	3.1	42400	317.5391052
2	Euclidian	1.45	19531	317.5391052
3	Euclidian 3D	1.29	17310	317.5391052
4	Euclidian Squared	0.008	398	556.8528137
5	Chebyshev	1.61	20522	317.5391052
6	Chebyshev 3D	1.49	19928	317.5391052
7	Octile	1.44	18771	317.5391052
8	Manhattan	1.24	15783	317.5391052
9	Weighted Manhattan	3.2	42125	317.5391052
10	Diagonal 3D	1.29	15963	317.5391052
11	ALT	0.22	3307	317.5391052

6. Reference

A*-based Pathfinding in Modern Computer Games
Adversarial Search
Improved Heuristics for Optimal Pathfinding on Game Maps
Heuristics