Mar-Z AI Design
(A Star & Barricade AI)

COMP4981 Linux Game

AI Team

Fred Yang A00777792

Overview

In video games, Artificial Intelligence (AI) is used to simulate human-like intelligence and intelligent behaviors primarily in non-player characters like NPCs and minions.

Generic Mode

In the project, the behavior of a minion can be classified as 3 generic modes: Idle, Move, and Attack.

Idle – Not able to find an available cell or be in the stop-move-condition.

Move – Able to find an available cell to move on. Movement towards towers, a minion can evaluate its surroundings in real time and perform its actions such as searching for directions, pathfinding at a crossway, and pathfinding at the end of a path.

Attack –The mode of a zombie will be switched to Attack if a tower, turret, barricade or player is in its attack range.

A* Pathfinding

A-star (A*) is probably the most commonly used algorithm in games pathfinding and graph traversal. With the A-star search algorithm it's possible to find the shortest path from one node to another on a tile map, especially in scenarios when there're chasms, rocks or other obstacles on the way.

A* uses a best-first search and finds a least-cost path from a given initial node to one goal node by maintaining a pair of lists: open list (fringe) and closed list. open list (fringe) is a list of all locations immediately adjacent to areas that have already been explored and evaluated, while closed list is a list of all locations which have been explored and evaluated by the algorithm. For every iteration, A* always picks the neighbor with a least actual cost to proceed to.

Basic Concept

The classic cost representation in A^* algorithm is

f(n) = g(n) + h(n)

g(n) represents the total cost from the starting node to the current node.

h(n) is the estimated cost from the current node to the destination. A heuristic function is used to create this estimate on how far away it will take to reach the goal state.

f(n) is the sum of **g(n)** and **h(n)**. This is the cost of the current estimated shortest path.

Here are a few more concepts need to be considered:

- **Node:** It can be a tile (cell) in a tile-based environment or a point along a path. An A* path contains at least 2 nodes: a starting node and an ending node. A* will calculate all the nodes to get from the start to the end.
- **Parent Node:** When a node is examined, its neighbors (directly connected nodes) are assigned that node as their parent.
- **Open list (fringe):** a list of all locations immediately adjacent to areas that have already been explored and evaluated (the closed list).
- **Closed list:** a list of all locations which have been explored and evaluated by the algorithm.
- **Cost:** Cost is the distance travelled. In pathfinding, nodes with a lower cost are preferably used over nodes with a higher cost.
- **Heuristic:** The heuristic function tells A* an estimate of the minimum cost from a particular node to the destination. It's important to choose a good heuristic function in searching for a path.
- **Euclidean Distance:** The shortest distance from a particular node to the goal. If your objects can move at any angle instead of grid directions, then you should probably use a straight line distance.
- **Manhattan Distance**: The standard heuristic for a square grid is the Manhattan distance. This is an approximation of the distance between two points based on adding the horizontal distance and vertical distances rather than computing the exact difference. That would look like this:

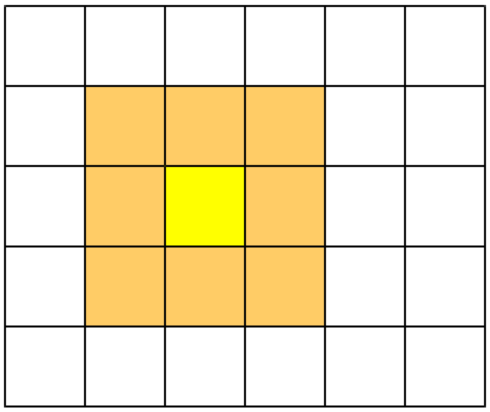


Figure 1. The current location is the yellow square, it is now part of the closed list. The orange squares surrounding around the yellow are the fringe, these are the possible options which the algorithm can experiment with.

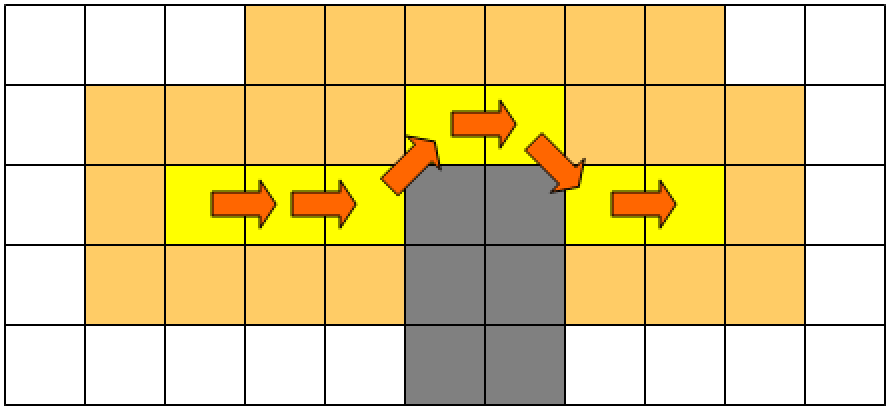


Figure 2. As the path progresses, the closed and fringe lists grow. Note that this path cuts corners. If the gray area represents an obstacle, like a wall, this path might be invalid since it passes unhindered through the wall.

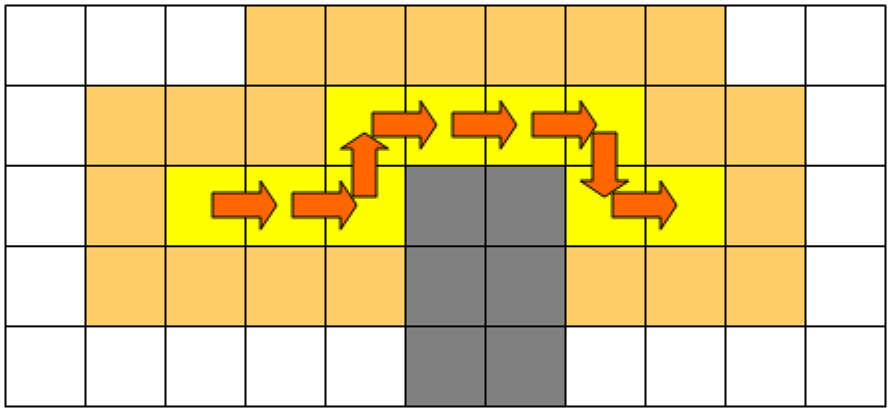


Figure 3. When cornering rules are imposed, the path will be better suited to avoiding obstacles.

How A* Actually Works

Now that you understand the basic concepts, before we start coding, let's just take a look at how A* actually works.

1). Provide a starting node (or square) and an ending node (or square).

2). Add the starting node to the open list.

3). Start the search loop.

> a). Look for the lowest f(n) cost on the open list– we refer to this as the current node. When we first run the loop this will obviously be our starting node.
>
> b). If the current node is the same as the destination then we are done and we can move on to step 4 to build the path.
>
> c). For each of the 8 neighbor nodes adjacent to the current node:

- If the node is not traversable or if it’s on the closed list, skip it. Otherwise do the following.
- If it isn’t on the open list, add it to the open list. Calculate the cost of that node and set the current node as the parent.
- If it’s already on the open list, see if this path to that node has a lower G cost. If so, switch the parent of the node to the current node, and recalculate the G and F scores of the square.

d) Quit the loop when:

- The target node is added onto the closed list, which means the path has been found. Or
- Fail to find the target node, and the open list is empty. In this case, there is no path.

4). Construct the path. Working backwards from the target node, go from each node to its parent until it reaches the starting node.

Why A* Be Used?

As mention above, A-star (A*) is probably the most commonly used algorithm in games pathfinding and graph traversal. A* can help find the shortest path from one node to another on the map, especially in scenarios when there’re chasms, rocks or other obstacles on the way.

As shown in figure 4, the zombie can easily find the most efficient way to get to the player using A* algorithm. Other algorithms, such as Monte Carlo algo, or Manhattan algorithm, are not possible to find the shortest path on the map shown in figure 4.

Although A* time complexity is O(logN), which is longer than Monte Carlo / Manhattan algorithm, testing of endless zombie waves on several work stations did NOT see any delays.

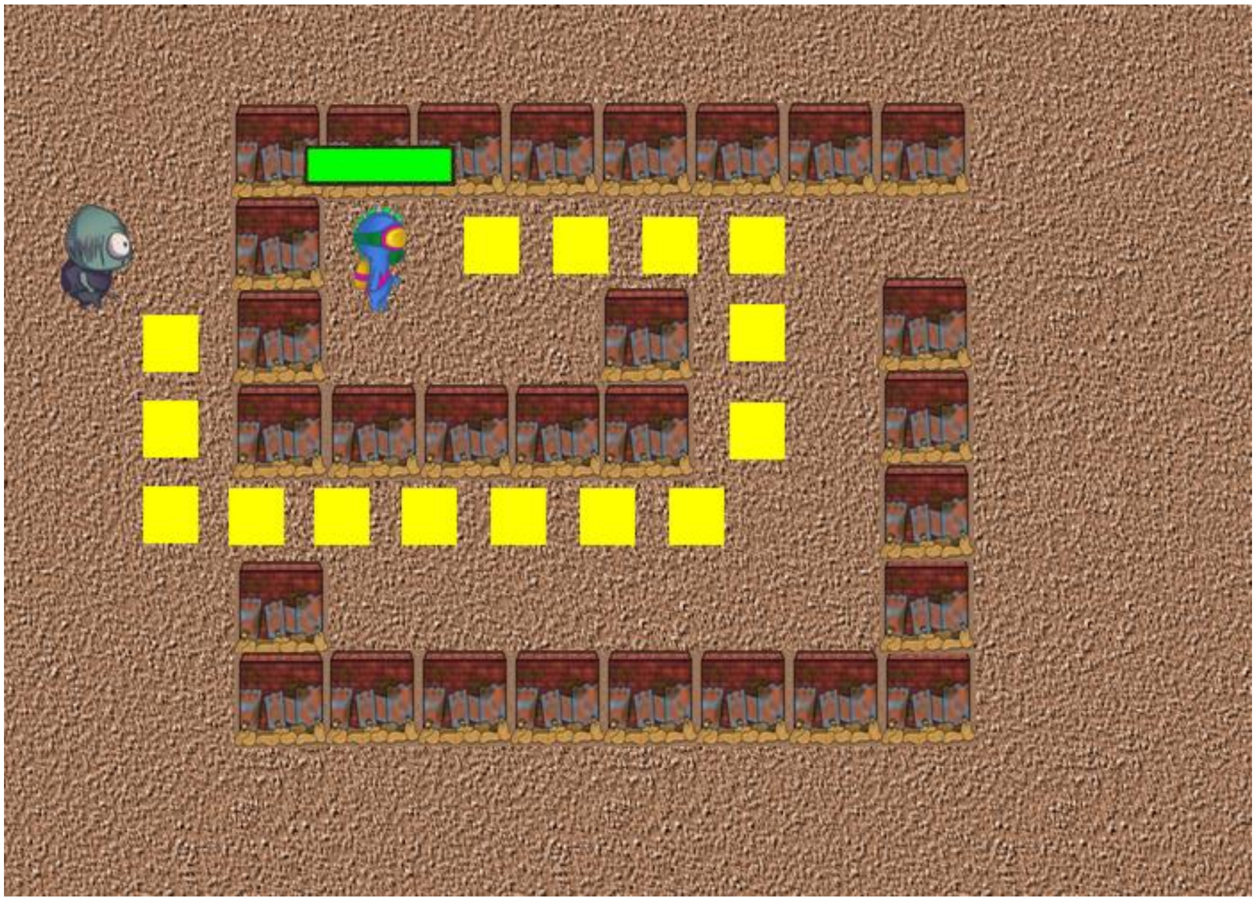


Figure 4 The zombie finds the shortest path to the player using A* algorithm

A* Time Complexity

A* time complexity is, O(logN),

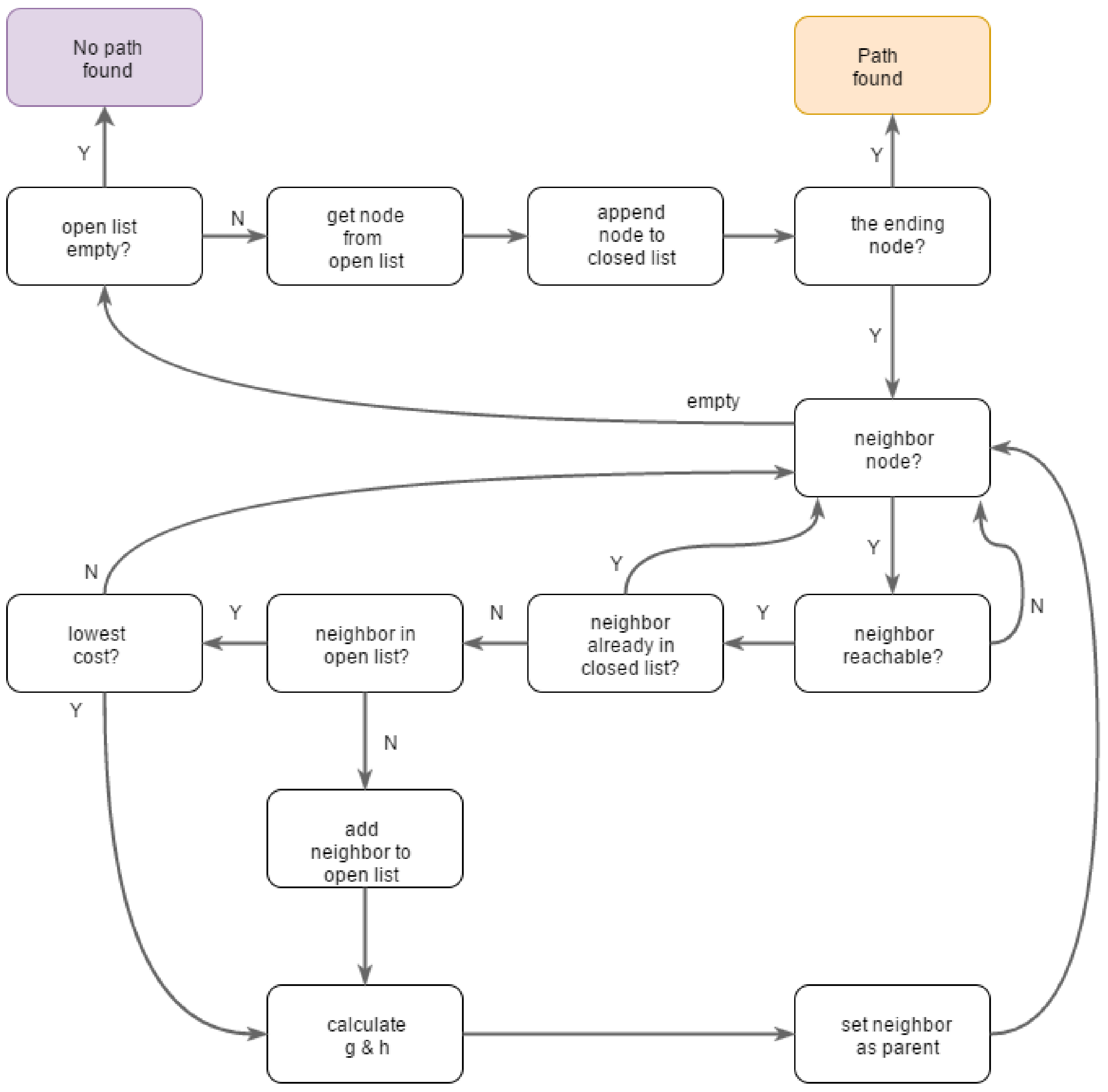
here N = max{numOfRowFromBase, numOfColumnFromBase}

3 <= N <= 25 for Mar-Z game.

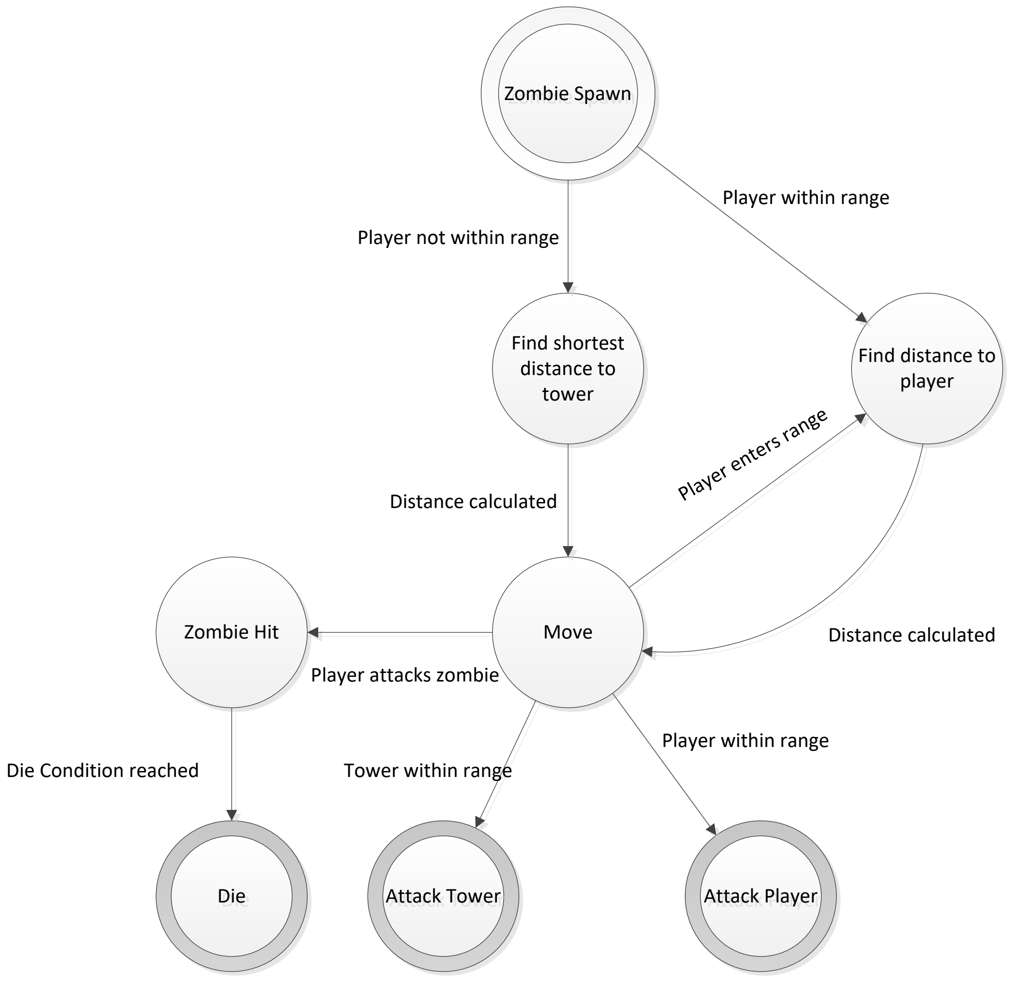
Monte Carlo / Manhattan time complexity is, O(1)

State Flow Diagrams

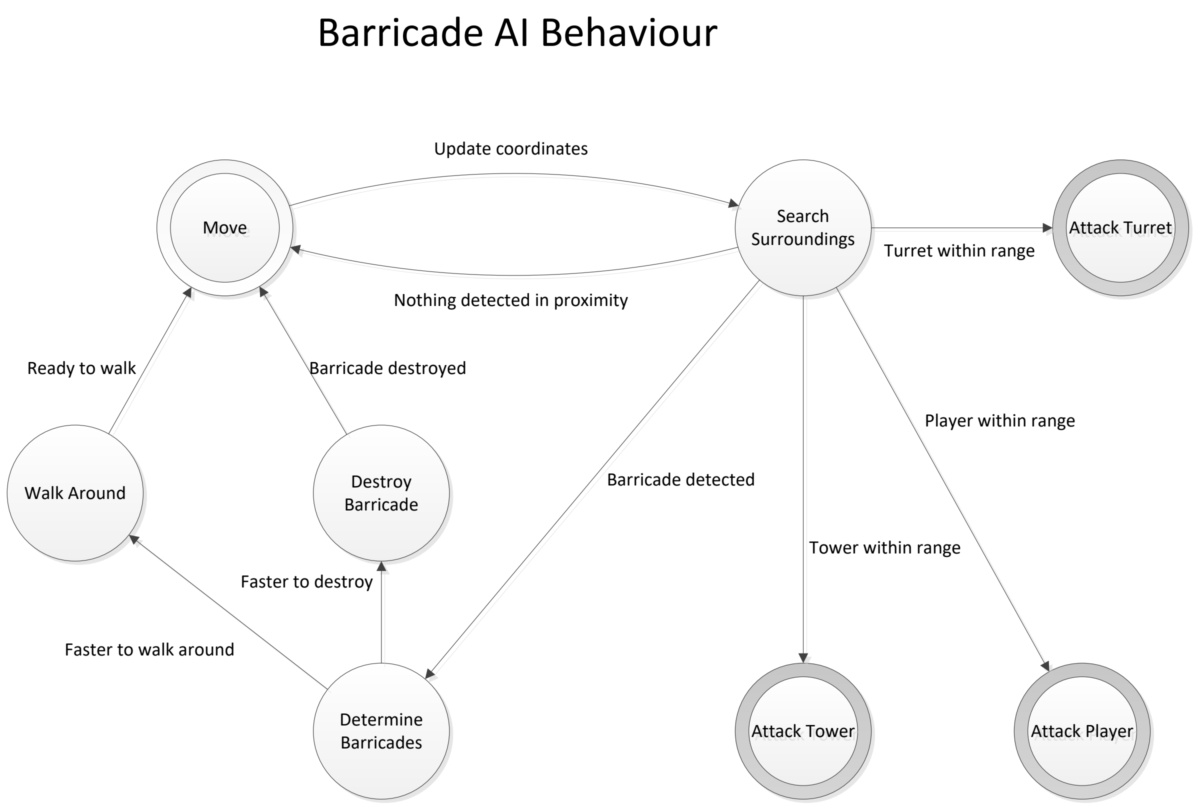
A* Pathfinding STD



Zombie AI STD



Barricade AI STD



Pseudocode

A* Pathfinding

```
Initialize the open list
Initialize the closed list
Put the starting node on the open list

While the open list is not empty
  Find the node with the least cost on the open list, call it Q
  Pop Q off the open list
  Generate Q's 8 successors and set their parents to Q
  For each successor
        If successor is the goal, stop the search
        Calculate total cost between successor and Q
        Calculate estimated cost from goal to successor
        Set current cost = total cost + estimated cost

    If a node with the same position as successor is in the OPEN list
      which has a lower current cost than successor, skip this successor
    if a node with the same position as successor is in the CLOSED list
      which has a lower current cost than successor, skip this successor
    otherwise, add the node to the open list
  push Q on the closed list
end
```

Zombie AI

Idle

```
Pick a random (SEEDED) spot on the border of the map
Initialize the zombie's coordinates
Initialize any needed attributes (eg. Health, speed, armour, etc)
If a player is within "X" range, find path to player
Otherwise find path to tower
Place zombie on the spot
MOVE[STATE]
```

Move

```
Use current position as starting node
If player is "nearby" and tower is not "nearby", switch target to player
        If target is a player and tower is nearby, switch target to tower
                Traverse the path from the starting node to ending node (A*)
                While the zombie is not at the target
                        Set mode as MOVE
                        If the zombie is attacked, go to HIT[STATE]
                        If another potential target is nearby, switch target
                        Update minion’s coordinates, move on to next node
                Target is in range, go to ATTACK[STATE]
```

Attack

```
If a tower/player is in range
        Set mode as ATTACK
        Update minion’s attributes
        Update tower’s attributes
        Update server that an attack has occurred
```

Hit

```
Update minion’s attributes
If HP reaches zero
        Go to DIE[STATE]
        Update server on what has happened
```

Die

```
Cancel any ongoing operation
Remove from map
Free resource and memory associated with minion object
Update server that a zombie has died
```

Barricade AI

Search surroundings

```
Retrieve zombie’s coordinates
Retrieve zombie’s directions (one of 8 directions)
Retrieve objects nearby and put onto a barricades list
If a player is within the range
    Find path to the player and attack him
Otherwise if a tower is within the range and no barricades
    Go ahead and attack it
Otherwise if a barricade is within the range
    Go to DETERMINE BARRICADES [state] { barricades }
Otherwise
        Go back to MOVE [state]
```

Determine barricades

```
If a new barricade added
        Check if there exists a faster path to tower if walk around it
        (the cost of walk-around path must be less than the specified value, eg. < 130%)
        If so, go WALK AROUND [state]
Otherwise go DESTROY BARRICADES [state]
Otherwise the barricade has already been evaluated
```

```
    Go back to MOVE [state]
```

Destroy barricades

```
While the barricade isn't destroyed
    Attack the barricade
    Update barricade's attributes (decrement HP)
    If the barricade has been destroyed
        Go back to MOVE [STATE]
```

Walk around

```
If the cost of destroying a barricade is larger than the threshold
    Generate a path to walk around the barricade
    Go back to MOVE [state]
Otherwise
    Destroy the barricade
```

Listings

All files of Zombie /Barricade AI reside in **creeps** folder.

GameMap.h – WYSIWYG game map for A* algo. The game map is loaded from map.csv file later.

Node.h – This header file is used for the A* algorithm in navigating the map.

Zombie.h – This header file includes the macro definitions and function prototypes for Zombie class.

Zombie.cpp – this class wraps A* / Monte Carlo algorithms and other basic methods for Zombie.