RBE470x Project 1 Report

By Josh Keselman, Sam Markwick, and Cam Wian

1. Overall Structure of Our Approach

Our character control code is contained within a file called `aicharacter.py`. Our control code has four basic parts: a "state machine" in `do` method, our A* navigation algorithm, our minimax implementation, and a set of helper functions to locate various objects in the game world.

Our state machine is designed to switch the character between two general states, one without a bomb present on the map (denoted as *no_bomb*) and one with a bomb present (*bomb*). The character starts in the *no_bomb* state, and each of those states has multiple substates. In *no_bomb*, these substates include a general minimax state, to get our next move based on the current game state, as well as determine if we should place a bomb, and states to navigate directly to the goal if no monsters are present or if the character is closer to the goal than any monsters are. In the *bomb* state, the character has a substate to navigate to the goal if no monsters are present or if the character is closer to the goal than any monsters are, and a substate to backtrack previous moves, allowing the character to retreat to safe areas and wait for the bomb to detonate.

We chose to use A* to navigate the game world, as it is an optimal algorithm for discrete pathfinding problems. Despite not being dynamic, it is very effective at finding optimal paths between two points in a grid. This allowed us to use A* not just to find our own path to and distance from the goal, but also to find the distance between the monsters and the goal and the monsters and the character, both helpful for determining the best course of action. Using A* also allowed us to implement a modified priority calculation, that includes not just our heuristic for the goal distance but also an added distance to adjust for nearby monsters.

Our minimax implementation is used to determine the best move for the character to make, based on the current game state. We chose to use minimax, with alpha-beta pruning, due to its relative simplicity to implement and ability to handle large decisions trees through pruning. ...

2. Code Breakdown

This section will dive deeper into specific pieces of our code. We will discuss the structure of our state machine, our A* implementation, our minimax implementation, and our helper functions.

2.1 State Machine

Our state machine, implemented in the `do` function, is designed to ensure that character plays safely and "intelligently" until it is safe to simply spring directly towards the goal. The state machine is generally structured as follows:

```
Check if closer to the goal than monsters
    if so A* straight to goal
    else
```

```
    Check if there is no bomb on the field and for the presence of monsters
        if so check if the closest monster is within 5 A* steps and not at x=0 or
y=0
            if so place bomb
    Check again if bomb is on the field or is being placed on the field
        Check for no monsters
            if so A* straight to goal
        if not alpha beta search for best move
    If bomb was on the field or being placed
        Check if closer than monsters
            Minimax to goal
        if not backtrack to the start and stay till the bomb goes
```

2.2 A* Implementation

Our A* implemention is generally standard, with some small modifications to the priority calculation. The basic structure of our A* algorithm is as follows:

```
def astar(self, world, start, goal):
    path = []
    found = False
    queue = priority queue sorted by lowest cost to move there
    add start to queue with priority 0
    explored = dictionary of explored nodes
    bombs = location of bombs in world
    while queue and we haven't found the goal:
        current element = get next item from queue
        add current element location to explored
        get the cost of the current element
        if current element is the goal:
            found = true
            break
        for each neighbor of the current element:
            if neighbor is not in explored or cost to get to neighbor is better
than existing cost to neighbor:
                monsterCost = 0
                if neighbor is a bomb:
                    monsterCost += 1000 to avoid bombs
                monsters = monsters in world
                for monster in monsters:
                    dist = heuristic distance between neighbor and monster
                    if distance less than or equal to 3:
                        add weighted distance to monsterCost
                add neighbor to queue with priority that includes cost, heuristic,
and monsterCost
    if found is true:
        return a reconstructed path
    else:
        return an empty path
```

This A* implementation also uses three accompanying helper methods. The first, `getNeighbors`, returns the valid 8-neighborhood neighbors of a grid cell, without invalid neighbors caused by grid edges or walls. The second, `reconstructPath`, reconstructs the path created by the A* algorithm using the `explored` dictionary. The third, `heuristic`, calculates the heuristic distance between two points, in this case using Manhattan Distance due to its admissibility and consistency. Our A* implementation was designed to give us paths that were optimal not only for distance to the goal, but also for avoiding enemies. This allowed us to use the A* algorithm to reduce the number of branches to explore in our minimax, as we were naturally avoiding very suboptimal paths by attempting to maintain a safe distance from all monsters naturally.

2.3 Minimax Implementation

Our minimax implementation is fairly standard, so instead of using pseudocode this report will cover specific highlights of our implementation. For starters, to assist in finding the optimal action, we are using a global called `optimalAction` that holds a tuple in the form ((x,y) bomb) as a move for the character. This move is updated in our minimax, and is returned by `abSearch`. This `abSearch` method implements alpha-beta pruning, with a `maxValue` method for character actions and a `minValue` method for monster actions. To generate character actions, we use our `astar` method to find the best path to the goal, then create a list of possible actions based on the optimal path to the goal, a safe move backwards, and an idle, as well as bomb placements for all 3 states if no bomb is present in the world. These actions are then passed into `result`, which is used to determine the state of the world after that action using `SensedWorld.next()`. Finally, utility for any state is calculated using a combination of the time, the character's distance to the goal, the distance away from monsters, and the need to avoid moves that kill the character, such as explosions or monsters.

2.4 Helper Functions

Our four *find* functions all follow a similar structure, shown with the following pseudocode:

```
def findItem(self, world):
    loop through world rows
        loop through world columns
            if row and column contain item
                return (row, column)
    return nothing if item not found
```

This basic structure has slight modifications for each specific item, with each type requiring different return values when not found. The `findMonsters` method uses an array to store the position of monsters, rather than returning directly, as there may be more than one monster present in the world. These functions are also versatile, allowing us to search through either the actual world state or the state of a copied world, which is useful for our minimax implementation.

3. Experimental Evaluation

To test our code for project 1, we ran a series of 10 tests for each variant, with different random seeds for each test. For each test, we recorded whether or not the character successfully reached the goal, and the score of the character when the run terminated. We then averaged these scores to get a sense of how well our code performed. The results of these tests, as well as some brief analysis, are shown below.

3.1 Variant 1

Out of 10 tests, the character successfully reached the goal 10 times. The average score of the character when the run terminated was 4978. This variant should work flawlessly, as it is the simplest of the five variants with no monsters involved.

3.2 Variant 2

Out of 10 tests, the character successfully reached the goal 10 times. The average score of the character when the run terminated was 4983.

3.3 Variant 3

Out of 10 tests, the character successfully reached the goal 10 times. The average score of the character when the run terminated was 4997.

3.4 Variant 4

Out of 10 tests, the character successfully reached the goal 7 times. The average score of the character when the run terminated was 2013.

3.5 Variant 5

Out of 10 tests, the character successfully reached the goal 6 times. The average score of the character when the run terminated was 1068.