Pathfinding using A*

The heart of our AI's thinking process lies in our implementation of the A* Pathfinding algorithm, found in the *tree_search_star.py*. This file contains two classes:

SearchNode:

- An abstraction of the nodes we'll be searching for – Basically boils down to a position on the map. Contains information on the node's position, it's neighbours (nodes that surround it), parent (node that caused us to open/create the node) and cost to our original position and to our target position.

SearchTree:

- The meat of the algorithm, containing the main search for path function and some other utility methods to either compute distances, the node's neighbours and checking if we've reached our goal.

- The actual *search_for_path* function uses a *self.open_nodes* array, that starts containing just the root node, and iterates over it while it's not empty. On every iteration we get the node from the open nodes array that has the shortest calculated distance (i.e shortest **cost to orign + cost to target**). Then we check if that node is our target, if it is we terminate the program and return the path, else we compute the neighbour nodes (ignoring nodes that we've already opened and closed) and add them to the open nodes array. If they were already in the open nodes array or the new computated cost to the neighbour (going through the current parent) is smaller than the one registered on the neighbour node, we update it and then append it. Rinse and repeat

-Note that to prevent computing a path to an impossible to reach location we limit the amount of open nodes we can have by a *self.limit* variable. If we've opened more nodes than allowed we simply return None

Diogo Silva (89348) | Vasco Ramos (88931)

Decision Flow

Our *next_move* function is responsible for picking the next course of action following a well defined decision flow that, in a simplified form, goes through the following steps:

1. Is the exit available and are there no enemies and have we caught all powerups or all over level 10?

YES: Go to the exit

2. Is *self.resting* bigger than 20?

YES: You're stuck in the same position (Check the Dealing with Loops slide)

3. Are there bombs on the map?

YES: Go to the running algorithm

4. Are there powerups on the map?

YES: Go to the powerup getting algorithm

5. Are there enemies on the map?

YES: Is *self.looping* set to bigger than 10?

YES: You're in a loop (Check dealing with Loops slide)

NO: Are you on level 1 or is the enemy reachable?

YES: Go kill the enemy

NO: Go to the wall closest to the enemy and destroy it

6. Are there walls on the map?

YES: Go destroy a wall

Diogo Silva (89348) | Vasco Ramos (88931)

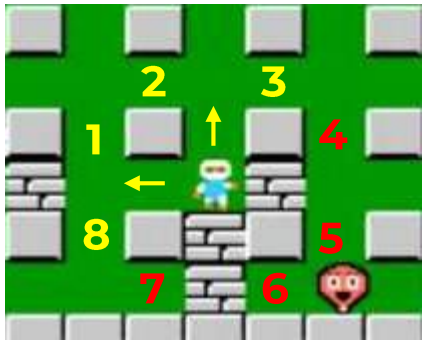
Running from bombs

Our running from bombs algorithm is based around the knowledge that bombs don't explode diagonally, so if we go into a different row and column than the place where we placed the bomb we'll be fine.

We start off by checking if there are any bombs in the map. If so we go into our running algorithm, else we set our control variable *self.running* to 0 (if it was either 1 or 2). In our *run_from_bomb* method we begin by checking our running variable's state. If it's set to **0** it means we're just **starting to run**, so we should pick a starting direction for which to move in. If it's set to **2** then that means that we've reached a **safe position**, so we either detonate the bomb or stay in place and wait for it to explode. Finally, running set **1** means **we've picked a running direction** and are going towards it, whilst if it's set to **3** it means there are **several possible paths** and we should pick one setting running to 1 and returning the key towards the picked direction.

If running is set to 1 we're going in a direction and all we have to do is either try to go to the first free space on the left or right if we're running vertically, or the first free space up or down if we're running horizontally. As soon as we go into one of these spaces, running gets set to 2.

To pick our starting direction we start by checking if the space directly up, left right or down of us is free and if so appending it to a possible directions array. If the length of this array is bigger than 1 then we should check which of those directions have the "first safe spots" (corresponding to the spots shown in the image) available. Note that to check if these spots are available we need to not only check the spot itself but also check if the path to it is free AND if there aren't any enemies in the vicinity. The first path that is proved to be truly safe is set to be the only one in the possible safe directions array. After this we check if we have any possible direction, if so we set our *self.running_direction* to be the first element of the possible_directions array. Else we'll try and see which of our initial runnable directions can be run for a number of steps equal to the radius of the bomb plus one. If there's any direction that meets that criteria we follow it, else we try to run through the direction we came in.

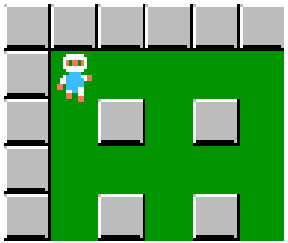


Killing Balloons

Balloms proved to be the toughest enemy for us to kill, hence we came up with a specified algorithm to deal with them (specially on the first 2 levels). This algorithm is called every time all enemies we have are Balloms (i.e, the amount of Balloms is the same amount of enemies in our *self.enemies* array).

The algorithm basically consists of trying to kill a Balloom normally. After doing so we destroy 6 walls before trying to kill another Balloom. We found this to be faster than simply chasing after Ballooms and hoping for the best (since at these early levels we don't have the bomb range necessary to kill them reliably). Repeating this will eventually lead us to a state where there are no more walls bringing us to our algorithms second stage.

When no more walls are available to be destroyed in the map, Bomberman will simply pick the closest corner to it and wait. Eventually (and not that long after) Ballooms will come pouring in making them easy to kill. Only after all Balloms are dead will Bomberman leave the killing floor and move on to it's next objective (either the exit or finding the powerup)



Dealing with loops

Loops may happen due to several reasons, including server desync or during the process of killing an enemy that moves away from the player. As a form of loop prevention we implemented two algorithms:

- **Resting Algorithm**

- This algorithm consists of one of the first conditions checked during our decision flow and boils down to the usage of a *self.resting* variable that counts the amount of times we've stayed in the same position. Useful, for example, in sittuations where we place a bomb, have the detonator and got nowhere to run so we never end up blowing the bomb.

- *self.resting* is incremented by one each time our last position is the same as our current and upon surpassing 20, we reduce it by 5 and check: **1)** Is there a bomb in the map? YES: Detonate it ; **2)** Is the exit available and the level exitable? YES: Go to the exit ; **3)** Are there walls? YES: Blow one up

- **Looping Algorithm**

- For situations in which Bomberman starts running after an enemy in a loop (either not being able to kill him and trying the same thing over and over again or when a server desync occurs and he starts running between the same few spaces) we created this algorithm. It makes use of a simple array that stores the last four visited positions at all time, *self.last_four_pos*, and a counter variable, *self.looping*

- We start off by checking if *self.last_pos* is in *self.last_four_pos*, if it is, and *self.looping* is less than 11, we increment it by 3, if it's not in the array and *self.looping* is bigger than 0, we decrease it by 1. When *self.looping* goes over 10 and there are walls still in the map we simply change our target and try to go over to our closest wall. When that wall gets destroyed we reset our *self.looping* variable to 0 and assume we've broken out of the loop. We of course also have to keep appending our last position to the array and make sure it never has over 4 elements

Dealing with unreachable Enemies

From level 2 onwards we prioritize killing the enemies rather than destroying walls. This can sometimes lead to situations in which enemies are fully surrounded by walls, hence our Pathfinding algorithm won't be able to find a path and will enter an infinite loop.

To deal with this we started by implementing a limit on how many nodes A* can open. If it goes over this limit, it assumes there's no path and returns a None.

All we had to do after this was decide how to deal with this no path response. For this we created a variable, *self.cant_reach_enemy* that is set to 1 when there are enemies, we try to calculate the path to the nearest one and get returned a None. When this variable is set to False, first we verify if there are still destructible walls on the map. If so, we check if our *self.nearest_wall_to_enemy* is unset (i.e its None). The purpose of this last variable is to store the closest wall to the enemy that's accessible by us, in the hopes that by destroying it we'll either be able to reach the enemy or at least get closer to it.

If we haven't set our *self.nearest_wall_to_enemy* yet, we do it by finding the nearest wall to the enemy and checking if it's accessible to us. If it is, good we've found the wall we should go after, if it's not we try to find the nearest wall to that wall, excluding itself, and so on until one eventually is reachable by us..

After we have our nearest wall to enemy we just keep going to it until we destroy it. After we do so we set our *self.cant_reach_enemy* to 0 and proceed with the normal decision flow