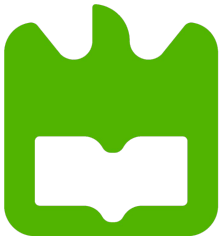


universidade
de aveiro

Inteligência Artificial

Trabalho Prático de Grupo (Bomberman)

André Gual - 88751
Rui Melo - 88889
André Catarino - 89337

Lógica Geral

A lógica de pensamento do Bomberman reside na avaliação de várias cláusulas em cadeia e foi implementada da seguinte forma:

Se existem bombas em campo:
- -fugir da bomba para uma posição segura

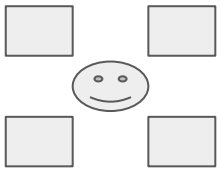
Caso contrário:
- Se existir um powerup no mapa:
  - Se estiver numa posição segura:
    - -Ir buscar o mesmo.
  - Caso contrário:
    - -Partir paredes.
- Caso contrário:
  - Se um inimigo está a uma distância perigosa:
    - -Colocar uma bomba.
  - Caso contrário:
    - Se existirem inimigos:
      - -Matar inimigos ou destruir paredes (dependendo da situação atual)
    - Caso contrário:
      - Se a saída estiver no acessivel:
        - -Ir para a saída
      - Caso contrário:
        - -Destruir paredes

Considerando que existe uma bomba no mapa

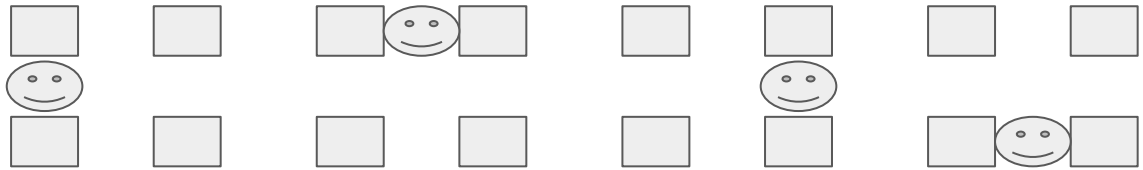
```
# Caso existam bombas no mapa
if exists_bomb(bombs):
    #se ja tiver sido atribuido movimentos para fugir
    if movements_array_to_safety != []:
        key, safe_bomb, movements_array_to_safety = complete_safety_process(movements_array_to_safety, bomberman, safe_bomb)


    else:
        # Se o bomberman nao estiver num posição segura (fugir da bomba)
        if not safe_bomb:···

        # Caso o bomberman esteja numa posição segura (aguardar parado até bomba explodir)
        else:
            key = ""
            if has_detonator(powerups_obtained):
                key="A"
```

Supondo que não se encontra numa posição segura da explosão da bomba, é executado um for com a função bomb_run_v2 na qual o parâmetro security_distance vai diminindo, até a mesma função devolver uma lista com movimentos para a tal posição segura.
De seguida, é retornado como key, o movimento necessário para a primeira posição necessária para alcançar a posição segura e retirar essa posição intermédia da lista (movements_array_to_safety).
A informação da bomba é originalmente criada pela função bomb_info que avalia em que direções os raios da bomba aparecem no mapa, por exemplo: Se a bomba estiver no canto inferior esquerdo do mapa, as explosões serão para cima e para a direita.

Entrando na função bomb_run_v2 com mais detalhe, esta tem como função avaliar as posições próximas ao bomberman em relação à explosão(argumento information gerado por bomb_info) e ao inimigos no mapa. Esta função verifica em que direções a bomba explode, uma vez que, o bomberman só poderá fugir tbm nessas direções. Pegando no exemplo anterior, se a bomba explode para cima e para a direita, o primeiro movimento do bomberman será ou para cima ou para a direita. Com isso em consideração, verifica se o bomberman se encontra “entre” 4 paredes indestrutíveis ou “entre” duas

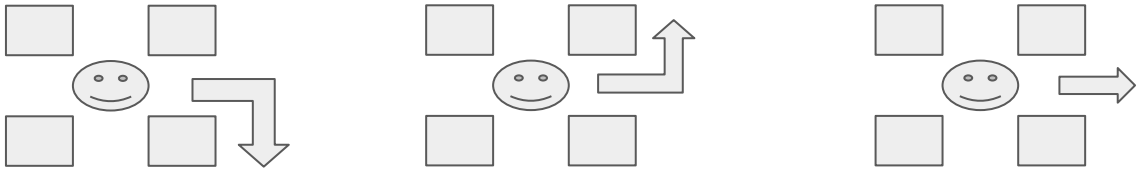


“Entre” 4 paredes indestrutíveis



“Entre” 2 paredes indestrutíveis

Assim, teremos duas situações diferentes, em que teremos 3 soluções genéricas para cada um dos casos (são invertidas/espelhadas conforme a situação)



A função bomb_run_v2 avalia se a posição final é segura da explosão e se o inimigo mais perto dessa posição não se encontra a menos de um valor x, passado como argumento que corresponde ao security_distance, assim como todas as posições intermédias necessárias.
Caso não haja nenhuma opção retorna [ ].
Deste modo, o ciclo for inicialmente falado é corrido outra vez, mas o valor da security distance é decrementado.
Se houver solução, movements_array_to_safety terá um conjunto de posições que, quando existir alguma bomba, terão de ser ocupadas de forma prioritária.

Considerando que não existe uma bomba no mapa

```
# Caso nao existam bombas no mapa
else:
    # Quando a bomba explode estamos safe
    if safe_bomb == True:
        safe_bomb = False

    # Há Algum Powerup no mapa ? (YES - GET THE POWERUP)
    if powerups != []:···

    # Não há PowerUps no mapa
    else:···
```

Caso não existam bombas, a variável booleana safe_bomb retorna a False. De seguida, avalia-se se existe um powerup, uma vez que é um aspeto essencial do jogo para poder avançar, principalmente nos níveis mais avançados.
Se existir (powerups != [ ]), avalia-se se há inimigos e, caso hajam, verifica-se se estão numa posição minimamente longe do powerup para o obtermos, caso contrário, destruímos a parede mais perto.

```
if enemies != []:
    #ha inimigos eo powerup ta acessivel
    near_enemy_from_powerup, enemy_dis_from_powerup = find_near_enemy(enemies, coord)
    if enemy_dis_from_powerup > 5:
        #safe to grab powerup
        key, powerups_obtained, got_powerup = grab_powerup(map_info, state, bomberman, coord, walls, enemies_pos, powerups_obtained, got_powerup)
    else:
        #inimios demasiado perto para grab powerup
        if walls != []:
            wall_to_destroy=closer
            if is_possible_to_destroy_wall(map_info, bomberman, wall_to_destroy, walls, enemies_pos):
                key, tries_to_kill = destroy_wall(map_info, bomberman, wall_to_destroy, tries_to_kill, walls, enemies_pos)

else:
    key, powerups_obtained, got_powerup = grab_powerup(map_info, state, bomberman, coord, walls, enemies_pos, powerups_obtained, got_powerup)
```

Algoritmo de Pesquisa

Antes de se explicar as ações principais correspondentes a destruir paredes e matar inimigos, é necessário apresentar o algoritmo de pesquisa utilizado em grande parte destas ações.

O algoritmo é astar que apenas assume como possível path blocos com valor diferente de 1 no “labirinto” e caso blocks_to_ignore não se mantenha vazio, ignorar esses mesmos blocos quando está a avaliar o caminho. Este blocos a serem ignorados, podem ser paredes destrutíveis e/ou inimigos. Neste astar, está implementado também, um procedimento de segurança (limite) para não continuar a procurar caso não encontre um caminho possível nas primeiras x iterações (x corresponde à largura o mapa mais o seu comprimento) e, deste modo, retorna uma lista vazia.

```
        iter+= 1
        if iter>len(maze)+len(maze[0]):
            return []
```

Destruição de paredes

A função de destruir uma determinada parede é precedida por uma função que verifica se é possível destruir essa mesma parede (o que pode não ser muito eficiente): is_possible_to_destroy_wall
A própria função destroy_wall faz utilização do algoritmo de pesquisa implementado e avisa que quer evitar quaisquer paredes que não seja a própria que quer destruir, assim como os inimigos. Se o bomberman já estiver na posição imediatamente adjacente à parede a ser destruída, é ordenado que largue uma bomba.

```
def destroy_wall(map_info, bomberman, wall_to_destroy, tries_to_kill, walls, enemies_pos):
    key=""
    temp_walls=walls

    if wall_to_destroy in temp_walls:
        temp_walls.remove(wall_to_destroy)

    next_move = astar(map_info, (bomberman[0], bomberman[1]), (wall_to_destroy[0], wall_to_destroy[1]), enemies_pos+temp_walls)

    if next_move != None and len(next_move)>0:
        if len(next_move)>2:
            key = go_to_next_block(next_move[1], bomberman)
        else:
            key = "B"
            tries_to_kill = 0
    return key, tries_to_kill
```

Inimigos - matar

Existem duas “classes” de inimigos:

-Inimigos que se movem independentemente das ações do bomberman;

-Inimigos que se movem conforme os movimentos/posições do bomberman.

Deste modo, criámos dois métodos genéricos para matar os inimigos:

-basic_method_kill_dumb_enemies;

-kill_smarter_enemies.

Em relação ao método mais simples que serve para matar Dolls e Ballooms, a função recebe um conjunto de posições possíveis para se dirigir para os matar (position_to_explode_bomb_Balloom). Se alguma posição tiver inimigos a menos de 2 blocos, não se dirige para essa posição e tenta outra opção por segurança. Assim que conseguir colocar a bomba para os matar, continuará a colocar até os inimigos passarem pela bomba e morrerem.

Inimigos - matar

Em relação ao método para matar inimigos mais inteligentes Oneals, Minvos ou Ovapis, existem 3 situações distintas.

As últimas duas baseiam-se sem verificar se o inimigo está rodeado e só tem uma saida, através da função is_enemy_surrounded_except_by_one_block. Isto apenas serve para largar a bomba com mais distância por segurança, umas vez que o inimigo poderia chocar contra o bomberman.

A primeira condição verifica se o inimigo se encontra num canto do mapa e caso se encontre (o que vai acontecer várias vezes) o bomberman encontra uma posição adjacente a esse canto e calcula, através do algoritmo de pesquisa, o caminho para essa posição adjacente. As várias posições que compõem o caminho são adicionadas a uma lista chamada obligatory_moves.

O objetivo é que o bomberman se aproxime do canto e afugente o inimigo dessa posição para depois o perseguir e matar.

Uma vez compreendida o mecanismo por dentro das ações principais, retornamos à lógica por detrás do bomberman.

Se não houver bombas, nem nenhum powerup disponível e houver inimigos, o bomberman verifica se há algum movimento que é obrigado a fazer (situação dos cantos), caso contrário, verifica se a entidade mais perto é um inimigo inteligente para o matar. De outro modo, destroi as paredes mais próximas, até chegar a um ponto em que irá matar os Ballooms e Dolls com o basic_method_kill_dumb_enemies.

Inimigos - fugir

Para não haver colisão do bomberman com os inimigos, sempre que um inimigo se encontrar a uma certa distância, o bomberman coloca uma bomba e foge.
Esta distância varia de acordo com o inimigo.

```
if enemy_dis < get_safe_dis(near_enemy):
    key = "B"
```

```
def get_safe_dis(near_enemy):
    dis = 0
    if near_enemy['name'] == 'Balloom':
        dis = 3
    if near_enemy['name'] == 'Oneal':
        dis = 1
    if near_enemy['name'] == 'Doll':
        dis = 2
    if near_enemy['name'] == 'Minvo':
        dis = 1
    if near_enemy['name'] == 'Kondoria':
        dis = 1
    if near_enemy['name'] == 'Ovapi':
        dis = 2
    if near_enemy['name'] == 'Pass':
        dis = 2
    return dis
```