Bomberman

Introdução à Inteligência Artificial

Trabalho realizado por:
Pedro Valente (88858)
Tomás Martins (89286)

Tomada de Decisões

O código está dividido em três grandes secções, quando **existem paredes**, quando **não existem paredes** e código que é executado **com ou sem paredes**.

As ações por ordem de prioridade **decrescente** são então:

1. Verifica se existe uma **bomba** no mapa, se sim desloca-se para um **lugar seguro**. Caso não tenha o detonador irá aguardar que a bomba rebente, caso tenha o detonador irá rebentar a bomba assim que chegar a um sítio seguro.
2. Verifica se tem um **inimigo** nas **proximidades**, se sim **coloca** bomba.
3. Verifica se o ***power_up*** está **visível** no mapa, se sim **vai** até ele.
4. Verifica se está **ao lado de uma parede**, se sim **coloca** bomba.
5. Verifica se já não existem **inimigos** no mapa, se o ***power_up*** e se a **saída** se encontram **visíveis**, se isto se verificar **vai para a saída**.
6. Verifica se existe um inimigo que **não seja *SMART.LOW***, se sim tenta perseguir e destruir o **inimigo**, caso contrário, e existindo um inimigo ***SMART.LOW***, destrói as **paredes mais próximas** consecutivamente e assim que não houver paredes irá deslocar-se até uma **posição calculada** pela função *get_corner()* e ficará aí a matar todos os **inimigos que sobram**.
7. Se não existirem **inimigos** e o ***power_up*** e **porta** ainda não estiverem **visíveis**, destrói a **parede mais próxima** consecutivamente.

Algoritmo de pesquisa

O algoritmo de pesquisa escolhido para o desenvolvimento deste projeto foi o **A***, o código para este algoritmo encontra-se no ficheiro ***astar.py***.

Excluindo as funções de fuga da bomba, o A* **calcula todos os outros caminhos necessários**, entre eles, caminho para a parede mais próxima e caminho para inimigos.

Nesta implementação o A* verifica apenas paredes **indestrutíveis**, sendo que caso o algoritmo calcule um caminho que se cruze com inimigos ou paredes destrutíveis coloca uma bomba.

Este algoritmo apresentou vários problemas, sendo que o maior foi o de abrir uma **quantidade excessiva de nós**. Este problema foi remediado colocando um **limite** de nós de 500. No entanto, devido a esta medida, por vezes o algoritmo **não retorna** caminho algum. Para remediar, caso o algoritmo não retorne qualquer caminho, o agente irá movimentar-se numa **posição aleatória** até o A* encontrar um novo caminho.

Este algoritmo foi baseado no código presente no seguinte website:
https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2

Fuga da Bomba

Para **fugir** de uma bomba, são executados alguns passos divididos em funções:

1. Verificar se o agente está posicionado entre duas paredes **indestrutíveis** através da função *is_between_walls()*.
2. O resultado da função anterior indica de que maneira se vai **fugir**, ou seja, que função irá ser utilizada(*bw_is_safe()* e *not_bw_is_safe()*). Ambas as funções funcionam de maneira muito semelhante: Verificam todos os **caminhos possíveis**, desde a posição atual do agente até à **posição** segura mais próxima e adicionam a uma **lista** as *keys* para alcançar essa posição. Estas verificam se o caminho contém paredes destrutíveis, indestrutíveis ou se existem inimigos perto, dado através da função *has_enemies()*.
3. Esta função verifica se **existe** um inimigo numa determinada posição e em todas as suas **adjacentes**.
4. Se as funções *bw_is_safe()* e *not_bw_is_safe()* **não** encontrarem um caminho seguro, irão **chamar** outras funções semelhantes *bw_is_safe_2()* e *not_bw_is_safe_2()* cuja única **diferença** das suas antecessoras é não calcular se existem inimigos.
5. Por último, o agente, através da **lista** de *keys* dada, vai **percorrer** o caminho indicado, e ao chegar ao destino, vai **esperar** que a bomba rebente se ainda não tiver encontrado o *Detonator*, caso contrário, vai **rebentar** a bomba.

Inimigos

Os inimigos foram divididos em **dois** grupos, *Smart enemies* e *Dumb enemies*, é feita a verificação da **existência** dos mesmos no mapa através das funções *has_SmartEnemies()* e *has_DumbEnemies()*. Os *Smart enemies* são constituídos por inimigos que possuam as características *SMART.NORMAL* ou *SMART.HIGH* sendo que os *Dumb enemies* possuem a característica *SMART.LOW*.

Com esta divisão são usadas então **duas soluções**, uma para cada grupo.

No caso dos *Smart enemies* é usada a função *find_SmartEnemies()* que retorna o *Smart enemy* mais **perto** da posição atual do agente. Sendo que este tipo de inimigos possuem **alta prioridade** o agente irá então deslocar-se até à **posição referida pela função anterior** e irá tentar matar o inimigo em questão. **Nem sempre** este **é bem sucedido**, pelo que por vezes, o agente poderá ficar preso num "*loop*" atrás do inimigo, sendo assim a cada *tic* que o agente está a uma **distância menor ou igual** a **cinco** é incrementado um **contador**, se este contador for igual ou superior a **cem** o agente irá **destruir** a **parede mais próxima**. Ao deslocar-se para a parede existe a probabilidade de o inimigo que estava a ser perseguido **fugir na direção contrária**, quebrando assim o "*loop*" e possibilitando o agente a conseguir matar o inimigo.

No caso dos *Dumb enemies* são **destruídas todas as paredes** (destrutíveis) existentes no mapa, sendo depois utilizada a função *get_corner()* para calcular um **canto** no qual o agente possa **esperar** pelos inimigos e matá-los assim. Estes inimigos são **evitados** na **inteira duração do nível**, sendo que se ficarem demasiado próximos o agente **colocará uma bomba e irá fugir da mesma**.

Funções Auxiliares (mais úteis)

calc_distance() : Função que recebendo duas posições calcula a distância de pontos entre as duas.

get_walls() : Função que retorna a posição da parede mais perto do agente.

astar_path() : Função que a partir do algoritmo A* implementado no ficheiro ***astar.py*** retorna a próxima *key*, utilizando uma função auxiliar *walk()*, do percurso calculado previamente.

is_beside_walls() : Função que retorna True se o agente estiver ao lado de uma parede destrutível.

on_same_line() : Verifica se duas posições estão na mesma linha(horizontal ou vertical) e se não existe nenhuma parede indestrutível no meio delas.