Algoritmo A* no ambiente Agent0

Bruno **B**rum (20182444@uac.pt); **D**iogo **C**arreiro (20182453@uac.pt);

Tiago **F**ilipe (20182443@uac.pt

Resumo

O objetivo deste trabalho era conseguir replicar o algoritmo A*, algoritmo de procura informado que corre escolhendo os nodes com menor custo, adicionando a funcionalidade de ajuste tendo em conta obstáculos. O ambiente apresentado ao correr é explicativo, sendo apresentado todo o "pensamento" do algoritmo por prints. Após várias fases de experimentação(testes) e revisão concluímos que o algoritmo como está funciona perfeitamente, tendo sucesso nos últimos testes com perfeita precisão, conseguindo chegar ao objetivo pelo menor caminho possível.

Introdução

O objetivo do trabalho é implementar o algoritmo de procura A*, com sucesso sobre o programa previamente fornecido pelo professor (Agent0). Para tal, tivemos que adicionar uma nova forma de decisão de nodes baseada numa heurística, que neste caso é a distância do node ao objetivo. Colocamos no github a nossa versão Agent0-Minotaur-A-Star com base no código do repositório: Agent0_minotauro(retirado em 18/11 às 19:30).

Descrição do Algoritmo ou Algoritmos e outro contexto de aplicação

O algoritmo A* implementado escolhe sempre um node seguinte cujo custo seja inferior aos restantes, chegando ao destino(Goal) da forma mais "eficiente", ou seja, a com custo menor possível. Para isto, o algoritmo funciona através da escolha de custo menor, sendo este sempre a soma do seu PathCost + Heurística.

```
pathCost = parent_node.getPathCost() + self.getPatchCost(state)
```

Fig 1. O PathCost é a soma do custo total que foi chegar até aquele node (custo do node "pai" mais o próprio).

Para obter a Heurística decidimos usar a distância (neste caso manhattan como o agente não consegue movimentar-se na diagonal) do node ao node objetivo(GoalNode)

```
def dist(x1, y1, x2, y2):
    # Irá devolver distância entre 2 pontos, sendo utilizado para verificar distância entre node atual e goal
    result = abs(x2 - x1)+ abs(y2 - y1)
    return result
```

Fig 2. Função dist().

```
class Node:
    def __init__(self,state,parent,action,path_cost,heuristica):
        self.state = state
        self.parent = parent
        self.action =action
        self.path_cost = path_cost
        # Heurística é a distância do node até ao goal, calculada na função "dist()" criada
        self.heuristica = heuristica
        # O custo do algoritmo A* será então a soma do custo para ir para a node mais o da heurística selecionada
        self.cost = self.path_cost + self.heuristica
```

Fig 3. Custo total é a soma destes outros dois custos, inserido na classe Node.

No nosso caso, tínhamos o objetivo extra de colocar o algoritmo a funcionar com obstáculos visíveis e invisíveis, sendo os visíveis obstáculos que o agente recebe e conhece no início, antes de pesquisar e os invisíveis obstáculos que o agente só chega a conhecer quando “bate” nestes, sendo que neste caso este “aprende” e volta a efetuar o “pensamento” do melhor caminho até ao Goal, tendo aprendido sobre a existência do obstáculo invisível e atuando com conhecimento deste, neste novo pensamento.

Explicação do software em runtime:

Primeiro que tudo iniciamos com a função run() do Agente, dando como parâmetro uma lista vazia, sendo esta a lista de obstáculos invisíveis.

[Imagem 1. do Anexo1]

Na primeira vez que corre a position é colocada como None, visto que a posição onde se encontra é a posição inicial. Usaremos esta position no caso do agente “bater” num obstáculo invisível e seja necessário correr o run() de novo, sendo neste caso necessário saber a posição que se encontra o agente.

[Imagem 2. do Anexo1]

Limpamos os nodes das fronteiras e dos visitados com uma função clear() criada.

[Imagem 3. do Anexo1]

Como comentado na seguinte imagem, o “state” do agente será a posição nova, caso não seja a primeira vez que “o run() pensa”.

Verificarmos a lista de obstáculos invisíveis vinda dos parâmetros da função(não vazia), adicionamos este nodes à lista de obstáculos visíveis, visto que já é um obstáculo conhecido(aprendeu-se ao “bater” neste). Para isso, igualamos a 1 na sua devida posição na lista de obstáculos(número 1 equivale a obstáculo e 0 como não obstáculo).

Inicializamos os obstáculos com o valor dos obstáculos totais(todos visíveis atualmente), por uma função simples que criamos no Agente. Precisamos desta criação para não serem esquecidos os novos obstáculos invisíveis encontrados.

[Imagem 4. do Anexo1]

De seguida, iniciamos o loop de while not end, para o agente ir procurando os caminhos na lista de nodes “fronteira”(não incluindo obstáculos), e selecionando o node com menor custo para colocar na lista de visitados e expandir este.

É verificado se o node selecionado para se visitar é o Goal, se sim, temos o melhor caminho encontrado e iremos mostrar este(prints) e sair do loop(end=True) após colocar o caminho(found) igual ao node selecionado(que irá conter a path total quando percorrermos os seus nodes “pais”).

[Imagem 5. do Anexo1]

De seguida, atualizamos o estado do agente para o estado da node que selecionamos porque queremos movimentarmo-nos para este. Para esse efeito, removemos da fronteira o nó expandido e, após verificar que a posição do node a expandir não é um obstáculo, inserimos na lista dos visitados este.

Se for inserido nos visitados o algoritmo irá tentar “expandir” para as quatro direções. Criamos lista auxiliar “list_visited” para guardar todos os estados das nodes visitadas para confirmar com os nodes a expandir não são repetições(não voltar para trás). É testado outra vez se estes nodes a “expandir” têm obstáculos e se não então são inseridos na lista de fronteira.

[Imagem 6. do Anexo1]

Segue-se no software uma opção de mostrar todos os caminhos encontrados e são mostrados sempre os nós fronteira e visited com o seu custo total.

Se conseguirmos chegar ao fim corremos o self.exe com o caminho(found) e se não mostramos com um print informativo que não foi possível chegar ao destino(caso de obstrução total).

[Imagem 7. do Anexo1]

Se encontrado um caminho então percorremos o caminho ao contrário, deste on Goal até ao “root”(posição inicial) através da visualização dos “parents” de cada node.

São percorridos percorrido os steps(caminho criado) e arranjando o conjunto de ações que irão permitir percorrer este caminho. As ações a fazer ficam guardadas em “actions” e são mostradas com um print.

[Imagem 8. do Anexo1]

Finalmente, corremos todas as ações e se for uma de movimento aumentamos o valor de “i”, o que irá servir para confirmar a posição atual na lista de steps. Executamos a ação e depois fazemos uma verificação com uma outra função(checkstuff()).

Na função verificamos com “actual_pos2” a posição atual do agente depois da ação. Fazemos uma verificação de, se a posição antes da ação for igual à posterior da ação, se a posição depois do movimento for diferente da posição que deveria ter(“step”) e se a ação tivesse sido de movimento(forward) consideramos que bateu num obstáculo invisível.

Se for o caso, inserimos a posição(step) na lista de obstáculos e retornamos este obstáculo numa lista, senão retornamos False. Caso seja retornada a lista com um obstáculo invisível e corre-se o run() com este obstáculo(irá ser inserido na lista de obstáculos visíveis) e a posição do agente atual no labirinto.

Caso não, as ações continuam a ser percorridas, atualizando a posição do agente.

[Imagem 9. do Anexo1]

Experiências Realizadas ou Exemplos de Aplicação

Experiência 1 - Custos

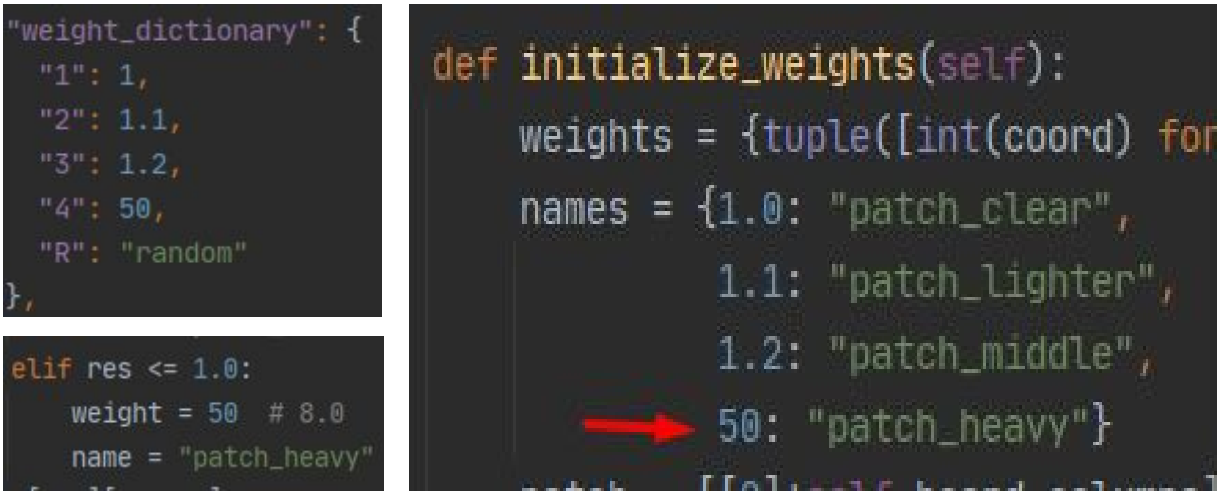

Fig A: Mudança de Custos

Mudando a patch_heavy para 50 o agente irá tentar o melhor que consegue para o evitar

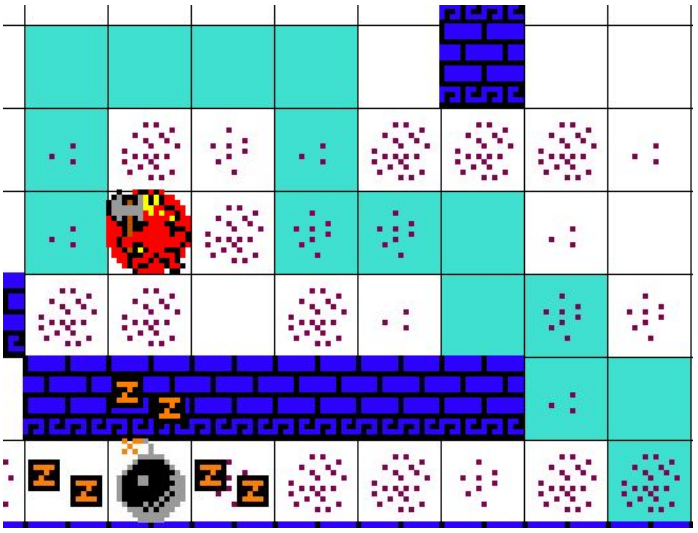

Fig 4: Custos Variados

Conseguimos ver que o agente evitou ao máximo os custos pesados(vários pontos)

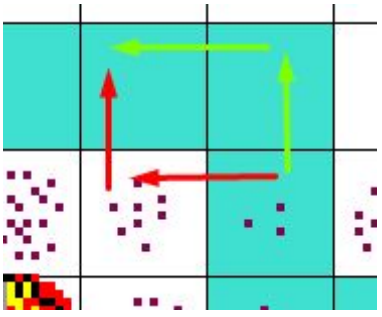

Fig 4.1: Seleção com escolha

Com a visualização da Figura 4.1 conseguimos confirmar que o agente escolhe o caminho de setas verde por ter um custo inferior(sem pontos = 1 e com esses pontos = 1.2).

Experiência 2 - Obstáculos Invisíveis

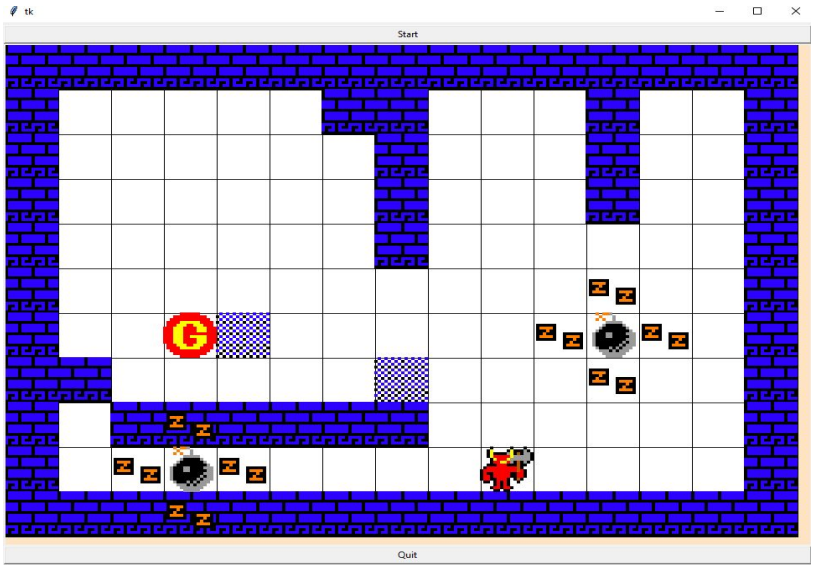

Fig 5: Percorrer com encontro de obstáculos invisíveis.

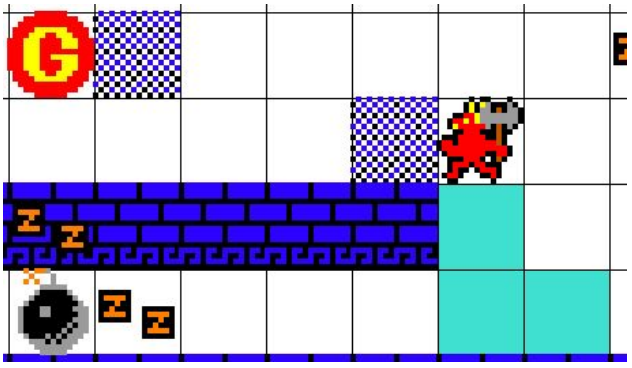

Fig 5.1: Primeira solução e primeira colisão.

Ele procura e encontra caminho (Final Path [[(9, 9), 0], [(8, 9), 1], [(8, 8), 2], [(8, 7), 3], [(7, 7), 4], [(6, 7), 5], [(5, 7), 6], [(4, 7), 7], [(4, 6), 8], [(3, 6), 9]]) , mas bate neste primeiro obstáculo e começa a pensar novamente.

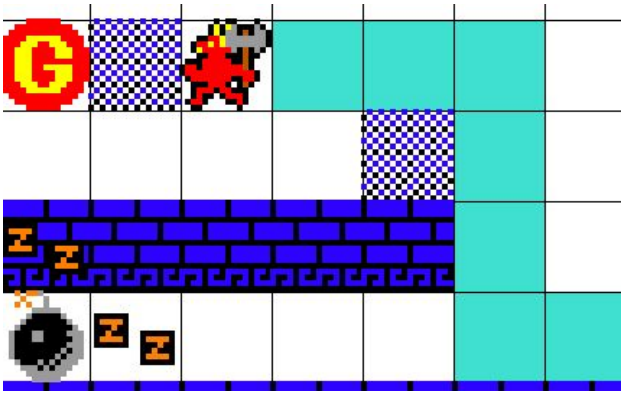

Fig 5.2: Segunda colisão e novo pensameto.

Após isto, chega a outra conclusão: Final Path [(8, 7), 0], [(8, 6), 1], [(7, 6), 2], [(6, 6), 3], [(5, 6), 4], [(4, 6), 5], [(3, 6), 6]] ; Irá bater no obstáculo e pensar de novo, tendo em mente este.

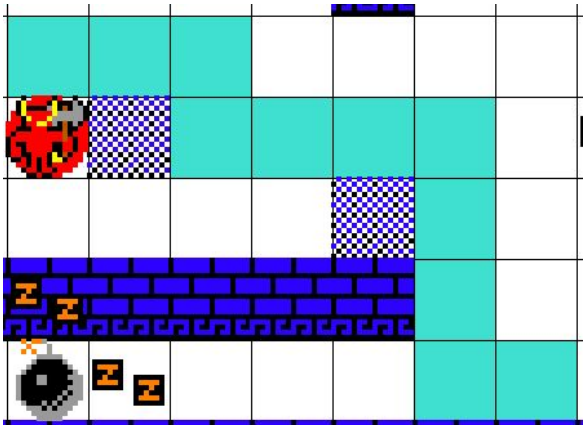

Fig 5.3: Solução final.

Finalmente, encontra uma path Final Path Final Path [[(5, 6), 0], [(5, 5), 1], [(4, 5), 2], [(3, 5), 3], [(3, 6), 4]]

Experiência 3 - Ambos casos

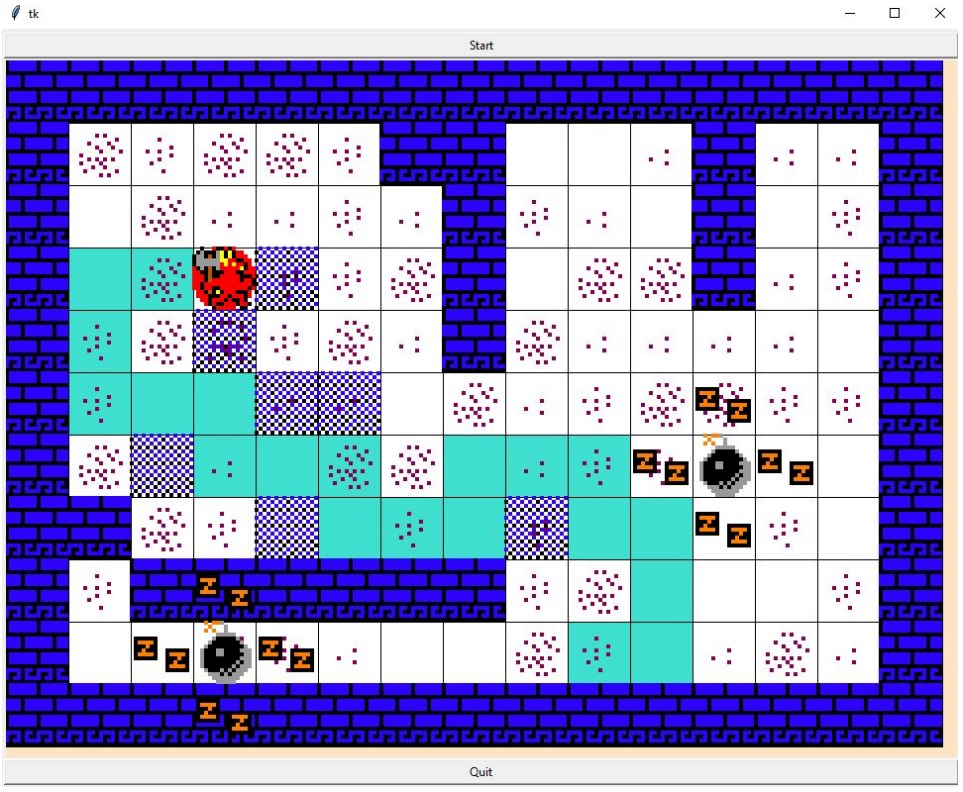

Fig 6: Caminho complexo tendo em conta custos(super elevado em patch_heavy) e obstáculos, visíveis e invisíveis

Discussão e Conclusão

Consideramos que os resultados obtidos foram excelentes para os objetivos que escolhemos atingir. A simulação feita utiliza o espaço que achamos necessário (mínimo possível) e não demora mais tempo do que o preciso, só sendo “desperdiçado” tempo ao serem mostradas informações, por exemplo a lista de nós fronteir e visitados, o que achamos serem benéficos o suficiente em relação ao pequeno aumento de tempo que causa (no caso de mostrar todas possíveis soluções deixamos em comentário porque eleva bastante o tempo, mas é uma opção).

Bibliografia

Sebastian Lague(16/12/2014). A* Pathfinding (E01: algorithm explanation). Consultado em 16 de Nov. 2020. Disponível em https://www.youtube.com/watch?v=-L-WgKMFuhE

Thaddeus Abiy, Hannah Pang, Beakal Tiliksew,Karleigh Moore;Jimin Khim(?/?/?).A* Search. Consultado em 16 de Nov. 2020. Disponível em https://brilliant.org/wiki/a-star-search/

José Cascalho.(2020).Pesquisa Heurística e outras formas de pesquisa(Mo2b-2-2020-2021.pdf). Universidade Dos Açores.