



DEETC – Departamento de Engenharia Eletrónica e Telecomunicações e de
Computadores

MEIM - Mestrado Engenharia informática e multimédia

Inteligência Artificial e Sistemas Cognitivos

Trabalho final

Turma:

11D

Trabalho realizado por:

Miguel Távora N°45102

Docente:

Luís Morgado

Data: 21/01/2023

Índice

1. INTRODUÇÃO	1
2. DESENVOLVIMENTO	3
1. REDES NEURONAIAS ARTIFICIAIS	3
1.1 COMPONENTE TEÓRICA	3
1.2 IMPLEMENTAÇÃO	6
1.2.1 PROBLEMA DO XOR.....	6
1.2.2 RESULTADOS OBTIDOS	8
1.2.3 APRENDIZAGEM DOS PADRÕES DE IMAGEM	12
1.2.4 RESOLUÇÃO DE UM PROBLEMA LIVRE	15
2. APRENDIZAGEM POR REFORÇO.....	18
2.1 COMPONENTE TEÓRICA	18
2.2 IMPLEMENTAÇÃO	23
2.2.1 IMPLEMENTAÇÃO DA BIBLIOTECA	24
2.2.2 IMPLEMENTAÇÃO DOS OPERADORES	30
2.2.3 IMPLEMENTAÇÃO DOS CONTROLOS	33
2.2.4 ARQUITETURA GERAL DA SOLUÇÃO E DIAGRAMAS DE SEQUÊNCIA	35
2.2.5 RESULTADOS OBTIDOS	39
3. PROCURA EM ESPAÇOS DE ESTADOS.....	45
3.1 COMPONENTE TEÓRICA	45
3.1.1 PROCURA NÃO INFORMADA	46
3.1.2 PROCURA INFORMADA.....	47
3.1.3 RACIONALIDADE LIMITADA.....	48
3.1.4 ALGORITMO WAVEFRONT.....	49
3.1.5 OTIMIZAÇÃO	50
3.2 IMPLEMENTAÇÃO	52
3.2.1 IMPLEMENTAÇÃO DA PROCURA ESPAÇO DE ESTADOS.....	52
3.2.1.1 IMPLEMENTAÇÃO DA PROCURA A* PONDERADA.....	52
3.2.1.2 IMPLEMENTAÇÃO DAS COMPONENTES DE RESOLUÇÃO DE UM PROBLEMA DE TRAJÉTORIAS	56
3.2.1.3 IMPLEMENTAÇÃO DO PLANO PARA O AGENTE	57
3.2.1.4 IMPLEMENTAÇÃO DO CONTROLO DELIBERATIVO E DAS AÇÕES DO AGENTE	59

3.2.1.5 DIAGRAMAS DE SEQUÊNCIA E ARQUITETURA GERAL DA SOLUÇÃO DA PROCURA A* PONDERADA.....	63
3.2.1.6 RESULTADOS OBTIDOS	65
3.2.2.1 IMPLEMENTAÇÃO DO MÉTODO FRENTE-ONDA	69
3.2.2.2 DIAGRAMAS DE SEQUÊNCIA DO MÉTODO FRENTE-ONDA	72
3.2.2.4 RESULTADOS OBTIDOS	74
3.2.2.3 ARQUITETURA GERAL DA SOLUÇÃO	77
3.2.3.1 IMPLEMENTAÇÃO DA OTIMIZAÇÃO.....	78
3.2.3.2 DIAGRAMAS DE SEQUÊNCIA E ARQUITETURA GERAL DA SOLUÇÃO DA OTIMIZAÇÃO.....	84
3.2.3.3 RESULTADOS OBTIDOS	87
4. TEMA LIVRE	91
4.1 GERAÇÃO DE IMAGENS A PARTIR DE TEXTO.....	91
3. CONCLUSÕES	97
4. BIBLIOGRAFIA.....	99

Índice ilustrações e tabelas

Figura 1 - exemplo de uma rede neuronal	4
Figura 2 - exemplo de função de ativação.....	4
Figura 3 - gráfica convergência da descida de gradiente.....	5
Figura 4 - regiões de decisão do XOR.....	7
Figura 5 - estrutura da rede utilizada	8
Figura 6 - primeira região de decisão	9
Figura 7 – terceira região de decisão	9
Figura 8 - segunda região de decisão.....	9
Figura 9 - padrões de imagem	12
Figura 10 - constituição da rede	13
Figura 11 - padrões de teste	13
Figura 12 - exemplo de bota	15
Figura 13 - exemplo de t-shirt	15
Figura 14 - constituição da rede	15
Figura 15 - exemplos de outras classes do <i>dataset</i>	16
Figura 16 - constituição da rede multiclasse	17
Figura 17 - matriz de confusão resultante	17
Figura 18 - iteração do agente no ambiente.....	19
Figura 19 - diferença entre algoritmo SARSA e Q-Learning.....	21
Figura 20 – diagrama de classes da seleção de ação	24
Figura 21 - diagrama de atividades da seleção de ação	25
Figura 22 – diagrama de classes da memória de aprendizagem.....	26
Figura 23 - diagrama de classes da implementação do algoritmo aprendizagem.....	26
Figura 24 - diagrama de classes do algoritmo Q-Learning com memória episódica	27
Figura 25 - diagrama de classes do algoritmo Dyna-Q	28
Figura 26 - diagrama de classes do mecanismo de aprendizagem por reforço.....	29
Figura 27 - diagrama de classes da percepção e da ação mover	30
Figura 28 - custo de movimento do agente.....	31
Figura 29 - diagrama de classes do agente	32
Figura 30 - diagrama de classe do controlo do agente.....	33
Figura 31 - diagrama de estado da interface gráfica.....	34
Figura 32 - constituição dos <i>packages</i> do programa	35
Figura 33 - diagrama de sequência da execução do programa	36
Figura 34 - diagrama de sequência da seleção de ação	37
Figura 35 - diagrama de sequência da aprendizagem.....	38
Figura 36 - resultado do Q-Learning a explorar antes de chegar ao objetivo.....	39
Figura 37 - resultado após o agente ter encontrado 11 vezes o objetivo	40
Figura 38 - resultado da procura com Q-Learning com memória episódica após o agente encontrar 1 vez o objetivo.....	41
Figura 39 - resultado da procura com Q-Learning com memória episódica após o agente encontrar 5 vezes o objetivo.....	42
Figura 40 – resultado da procura Dyna-Q após o agente encontrar 1 vez o objetivo	43
Figura 41 - resultado da procura Dyna-Q após o agente encontrar múltiplas vezes o objetivo	44
Figura 42 - Algoritmos de procura	48
Figura 43 – diagrama de classes do mecanismo da procura em espaços de estados	53
Figura 44 - memoria de procura com prioridade	54
Figura 45 - algoritmos de melhor primeiro	55
Figura 46 - componentes necessárias para resolver o problema.....	56
Figura 47 - exemplo de implementação das componentes	57
Figura 48 – diagrama de classes do plano da procura em espaço de estados	57
Figura 49 - diagrama de classes do controlo deliberativo	59
Figura 50 – diagrama de classes da deteção e ação do agente.....	60
Figura 51 - classe da interface gráfica	61
Figura 52 - ficheiro para correr o programa	62

Figura 53 - diagrama de sequência da execução do programa	63
Figura 54 - método execute() do DelibControl.....	64
Figura 55 - método resolve() da WeightedAStar.....	65
Figura 56 - teste para a procura A* ponderada com peso igual a 0	66
Figura 57 - resultado obtido para a procura A* ponderada com peso igual a 5	67
Figura 58 - teste para múltiplos alvos com peso 5.....	68
Figura 59 - continuação da figura anterior	68
Figura 60 - mecanismo de procura do método frente-onda.....	69
Figura 61 - plano da frente-onda	70
Figura 62 - execução do programa da frente-onda.....	71
Figura 63 - diagrama de sequências do controlo deliberativo do método frente-onda	72
Figura 64 - diagrama de sequência do algoritmo frente-onda	73
Figura 65 - resultado obtido da aplicação do método frente-onda para o ambiente 2	74
Figura 66 - resultado do algoritmo frente-onda para mais do que um objetivo.....	75
Figura 67 - resultado após o agente chegar ao primeiro objetivo	76
Figura 68 - diagrama geral da solução.....	77
Figura 69 - diagrama de classes do algoritmo Hill-Climbing estocástico	78
Figura 70 - diagrama de classes do algoritmo Simulated Annealing	79
Figura 71 - componentes do problema das N-Rainhas	80
Figura 72 - componentes do problema do caixeiro-viajante	81
Figura 73 - gráficos de exibição dos resultados	82
Figura 74 -execução do problema das N-Rainhas	83
Figura 75 - execução do problema do caixeiro-viajante.....	83
Figura 76 - diagrama de sequências do algoritmo Hill-Climbing estocástico	84
Figura 77 - diagrama de sequências do algoritmo Hill-Climbing estocástico	85
Figura 78 - diagrama de sequências do algoritmo Simulated Annealing	86
Figura 79 - arquitetura geral da solução	87
Figura 80 - resultado do caixeiro-viajante com o algoritmo Simulated Annealing.....	88
Figura 81 - resultado gráfico da execução do resultado do algoritmo Simulated Annealing com custo 0	89
Figura 82 - blocos de construção do gerador e discriminador.....	92
Figura 83 - funcionamento entre o gerador e o discriminador	93
Figura 84 - imagem gerada com a palavra "Lisboa"	94
Figura 85 - imagem gerada com a palavra "Portugal".....	94
Figura 86 – retrato de Edmond de Belamy.....	95

Tabela 1 - operação lógica XOR	6
Tabela 2 - tabela com o número de iterações até convergir (momento = 0).....	9
Tabela 3 - tabela com o número de iterações até convergir (momento = 0.5).....	10
Tabela 4 - tabela com o número de iterações até convergir (momento = 1).....	10
Tabela 5 - tabela com o número de iterações (momento = 1, dados baralhados)	11
Tabela 6 - tabela com o número de iterações (momento = 1, codificação binária)	11
Tabela 7 - resultados das iterações para diferentes parâmetros	14
Tabela 8 - resultado das iterações com diversas funções de ativação	14
Tabela 9 - custos resultantes de 10 execuções do problema do caixeiro-viajante	88
Tabela 10 - custos resultantes de 10 execuções do problema das N-Rainhas	90

1. Introdução

Primeiramente para se entender o que é inteligência artificial é importante definir o que é a inteligência. A definição da palavra inteligência tem sido alvo de diversos debates, sendo definida de diversas maneiras. Uma das definições foi como habilidades de nível elevado como raciocínio abstrato, representação mental, resolução de problemas e tomada de decisão. Outras definições para inteligência é a habilidade de aprender, conhecimento emocional, criatividade e adaptação para responder eficazmente às exigências do ambiente.

A inteligência divide-se em:

- **Cognição:** conhecido como o processo de conhecer. Em termos gerais é a capacidade de realizar a ação adequada dadas as condições do ambiente.
- **Racionalidade:** capacidade de realizar a ação certa dado o conhecimento que possui, nomeadamente sobre o ambiente, percepções e ações. Isto resulta na capacidade de conseguir o melhor resultado possível perante os objetivos que se pretende atingir. Para avaliar as ações corretas são utilizadas funções de utilidade.

Para se entender o que foi desenvolvido no trabalho final primeiro é necessário entender o que é a inteligência artificial. Este tema é altamente debatido e as respostas variam entre os especialistas. Uma resposta curta e sucinta é o ramo que estuda a síntese e análise de agentes computacionais que agem intelligentemente. Uma outra definição é a capacidade de imitar as capacidades de resolução de problemas e de tomada de decisão da mente humana para executar tarefas que requerem inteligência humana.

A partir da inteligência artificial é possível desenvolver sistemas cognitivos. Um sistema cognitivo é um sistema capaz de utilizar a informação do ambiente ao seu redor e tomar decisões de forma autónoma. Por vezes é feita a analogia da cognição humana para replicar em sistemas artificiais através da compreensão da forma como os humanos atuam perante situações complexas. Os sistemas cognitivos são os sistemas que possuem maior flexibilidade e autonomia.

A inteligência artificial possui três paradigmas principais, que são:

- **Conexionista:** a inteligência é uma propriedade resultante de interação de um número elevado de unidades elementares de processamento. Ex: redes neurais.
- **Simbólico:** inteligência é resultante da ação de processos computacionais sobre estruturas simbólicas. Este paradigma acredita que um sistema símbolos físico tem os meios necessários e suficientes para a atividade inteligente em geral.
- **Comportamento:** a inteligência resulta da dinâmica comportamental individual e conjunta de múltiplos sistemas e diferentes escalas organização. Onde surge o termo de inteligência artificial restrita, onde a inteligência existe somente para realizar uma tarefa específica. A inteligência artificial geral é caracterizada por ser possível realizar diversas tarefas ao mesmo tempo.

Este projeto será repartido em três objetivos que serão desenvolvidos em capítulos distintos. Os objetivos são nomeadamente:

- Objetivo 1: Resolver problemas com recurso a redes neurais artificiais.
- Objetivo 2: Desenvolvimento de uma biblioteca de aprendizagem por reforço com visualização gráfica dos resultados.
- Objetivo 3: Desenvolvimento de uma biblioteca de raciocínio automático para planeamento e outra biblioteca para algoritmos de otimização dos problemas. Em ambas as bibliotecas será desenvolvido uma forma de uma visualização gráfica dos resultados.

2. Desenvolvimento

1. Redes neurais artificiais

1.1 Componente teórica

As redes neurais são um subconjunto de aprendizagem automática e são elas o coração dos algoritmos da aprendizagem profunda. Foi atribuído este nome pelo facto da sua estrutura ser baseada no cérebro humano, nomeadamente na forma como os neurónios comunicam entre eles. Cada neurónio artificial conecta-se a outro e tem associado um peso e um limiar, caso o resultado de um neurónio for superior ao limiar ele envia uma mensagem para a próxima camada senão não é enviada nenhuma informação.

Cada rede neuronal é sempre constituída por um conjunto de camadas nomeadamente:

- Camada de entrada
- Uma ou mais camadas escondidas
- Camada de saída

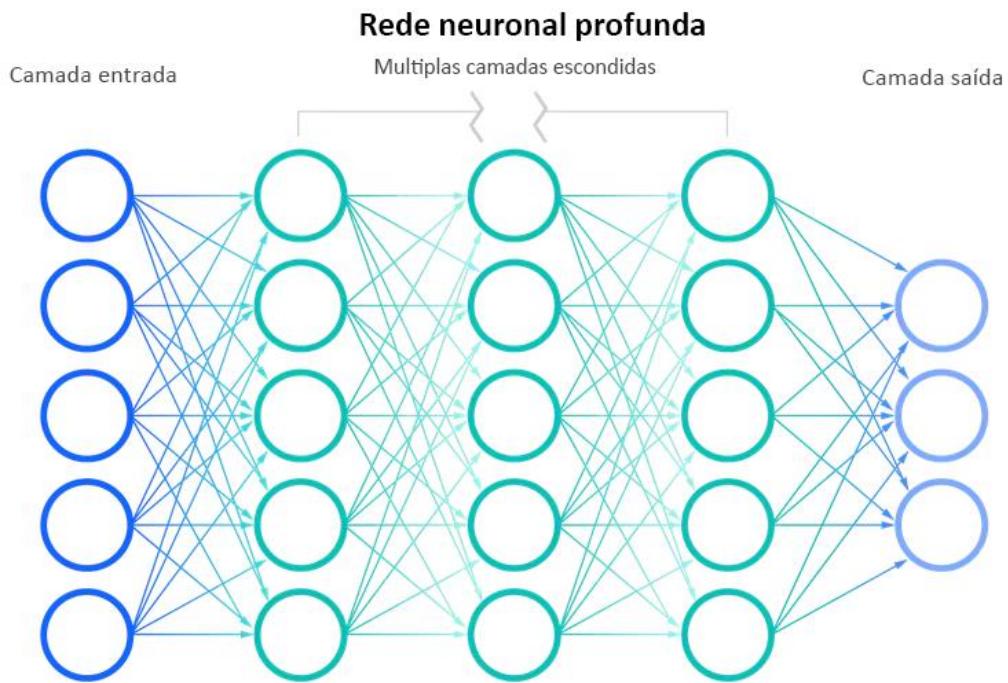


Figura 1 - exemplo de uma rede neuronal

Quando a camada de entrada está determinada os pesos são atribuídos na altura do treino da rede. Estes pesos ajudam a determinar a importância de cada variável para o resultado, onde existem variáveis que contribuem mais do que outras para determinar o resultado final. Todas as entradas são multiplicadas pelos respetivos pesos e depois somados obtendo um valor numérico. Depois este resultado é passado por uma função de ativação, quando o resultado é maior que o limiar da função de ativação ele ativa o neurónio e envia informação para a próxima camada. O propósito da função de ativação é para adicionar não linearidade à rede neuronal. Este tipo de arquitetura de passar dados de uma camada para a próxima é definida como uma rede *feedforward*.



Figura 2 - exemplo de função de ativação

Para avaliar a precisão é necessário ter uma função de perda. Isto geralmente é referido como o erro quadrático médio. O objetivo final é minimizar a função de perda para garantir o treino correto para qualquer observação. Enquanto o modelo ajusta os seus pesos, ele utiliza a função de perda para chegar a um ponto de convergência. O processo em que o algoritmo ajusta os seus pesos é conhecido como descida de gradiente, deixando o modelo determinar a direção para reduzir os erros. Na descida de gradiente o montante adicionado aos pesos para convergirem para o erro mínimo é designado taxa de aprendizagem. Este termo pode ser difícil de escolher, visto que demasiado grande o algoritmo pode ultrapassar o mínimo e demasiado pequeno e o treino pode demorar muito tempo. O objetivo na descida de gradiente é tentar ter o mínimo possível e não ficar por um mínimo local. Em cada exemplo de treino os parâmetros do modelo são ajustados gradualmente para convergir para um mínimo.

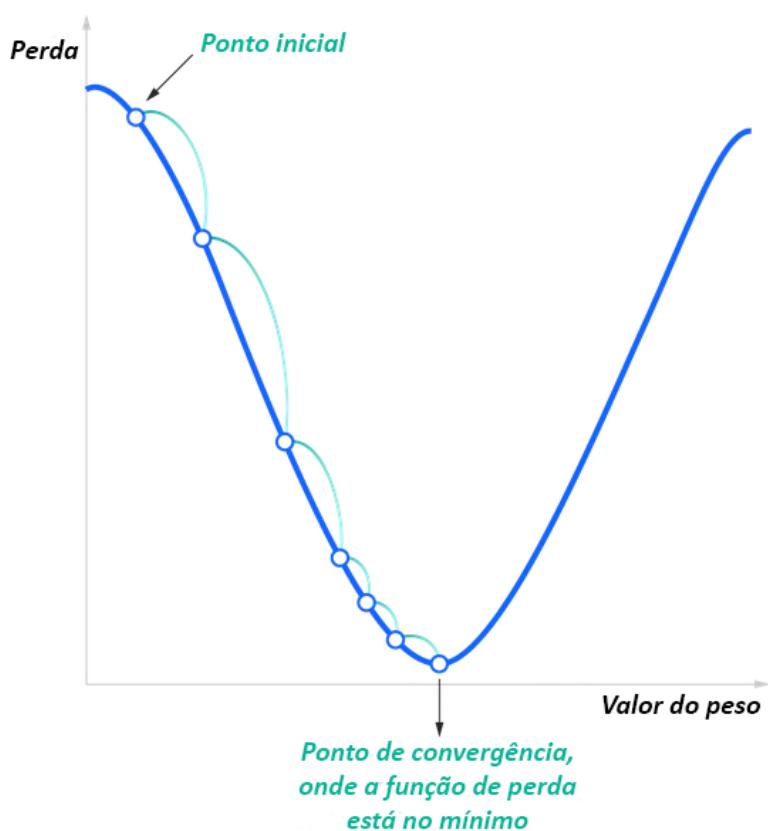


Figura 3 - gráfica convergência da descida de gradiente

Para acelerar o processo de otimização do cálculo do menor valor de erro pode ser utilizado o termo de momento. A descida de gradiente sem momento depende somente do gradiente atual para alterar os pesos. O momento substitui o gradiente atual por um valor m , que é um agregado de gradientes. Este agregado é o movimento médio exponencial dos gradientes atuais e passados.

1.2 Implementação

1.2.1 Problema do XOR

Para resolver o problema da aprendizagem da função lógica XOR foi necessário primeiramente definir o conjunto de treino dos dados. Os dados serão nomeadamente as diferentes entradas de uma função lógica, nomeadamente: 00, 01, 10 e 11. Como as redes neurais são do tipo de aprendizagem supervisionada precisam das classes correspondentes para cada dado. Desta maneira o resultado será falso caso as entradas possuam o mesmo valor e verdadeiro caso sejam diferentes. A tabela de verdade da operação lógica XOR é a que se segue:

Operação lógica XOR		
Entradas		Saída
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 1 - operação lógica XOR

Para verificar se a rede de facto conseguiu aprender a função lógica do XOR foi criado um conjunto de teste igual ao conjunto de treino, mas com os dados baralhados. Desta forma é possível verificar qual a precisão de classificação da rede neuronal, somente quando o *score* (resultado numérico da quantidade de acertos sobre os resultados totais) for de 1 é que se considera que a rede aprendeu.

Para que a rede neuronal consiga solucionar este problema é necessário pelo menos dois separadores lineares de regiões, querendo isto dizer que a rede precisa de pelo menos dois neurónios na camada escondida para resolver o problema. Um exemplo de separação das regiões é o que se segue:

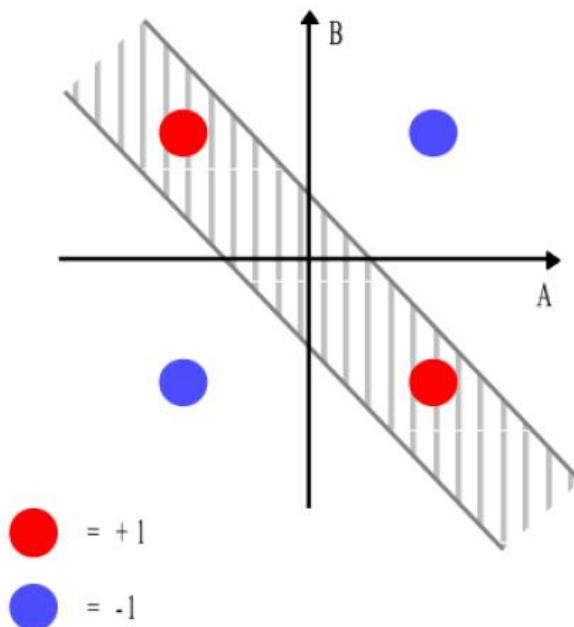


Figura 4 - regiões de decisão do XOR

Para implementar a rede neuronal foi utilizada a biblioteca sklearn e foi utilizado o classificador MLPClassifier que utiliza um perceptrão multicamada. Neste classificador foram utilizados os seguintes parâmetros:

- solver: “sgd” utiliza uma descida de gradiente estocástica.
- activation: “relu” que é função linear retificada retorna $f(x) = \max(0, x)$.
- max_iter: 10 000 que é o número máximo de iterações que um algoritmo realiza.
- tol: $1 * e^{-6}$ valor que quando o score não melhora pelo menos esse valor termina a execução.
- hidden_layer_sizes: (8,) quer isto dizer que a rede neuronal possui uma única camada escondida constituída por 8 neurónios. Foi escolhido este valor para conseguir que a rede convergisse rapidamente para uma solução.

Além destes parâmetros que foram mantidos constantes em todos os testes, serão feitos múltiplos testes com diferentes valores de taxa de aprendizagem e utilização de um valor de momento.

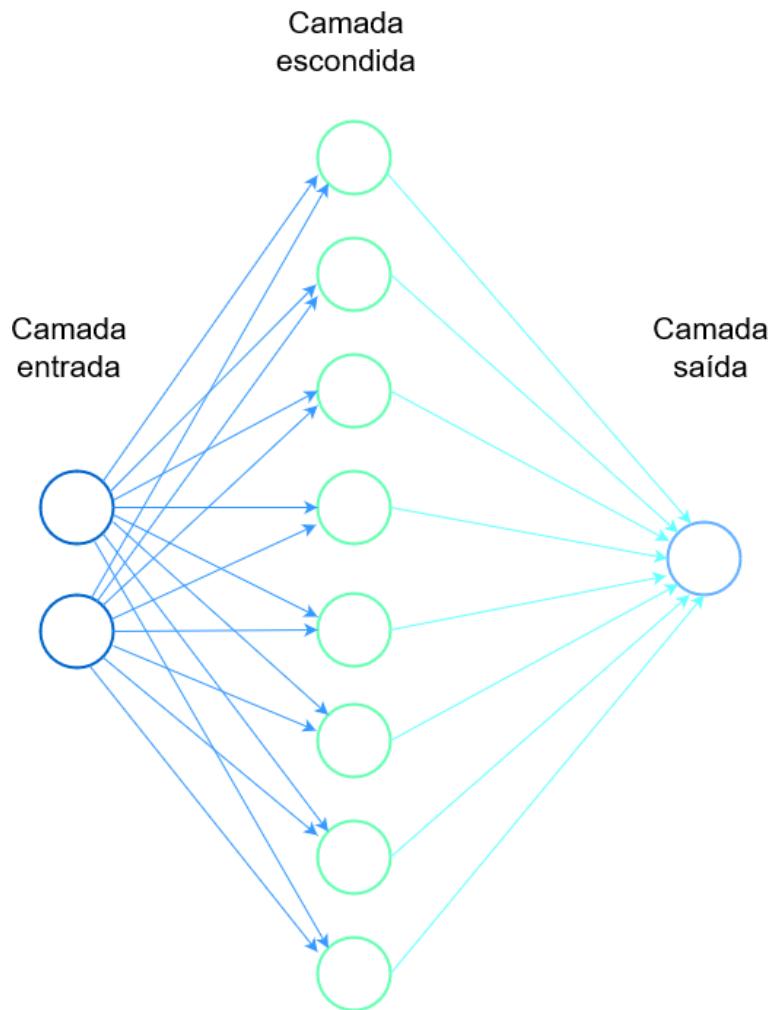


Figura 5 - estrutura da rede utilizada

1.2.2 Resultados obtidos

Para se concluir que a rede aprendeu o problema do XOR foi feito uma condição onde somente quando a função de perda é inferior a 0.1 é que é feita a classificação dos dados de teste. As regiões de decisão obtidas foram as que se segue:



Figura 6 - primeira região de decisão

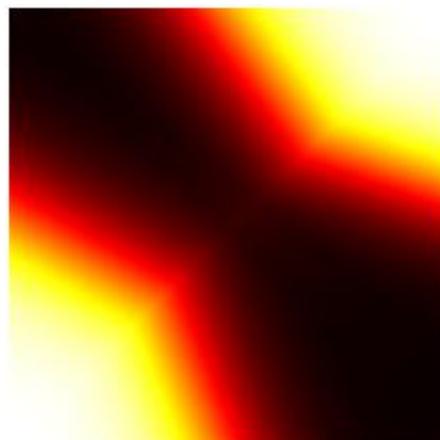


Figura 8 - segunda região de decisão

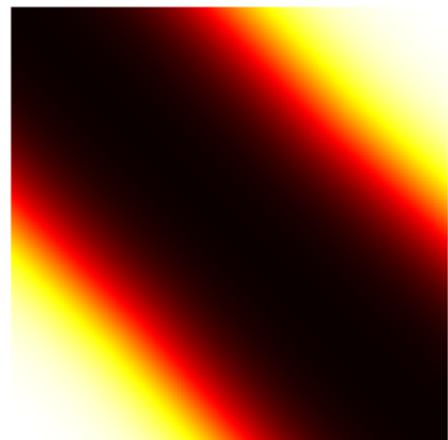


Figura 7 – terceira região de decisão

Nos testes inicialmente foram utilizados os dados em formato bipolar. O formato bipolar possui os dados numa estrutura que só podem assumir os valores de -1 ou 1. Este formato é uma alternativa do formato binário que só assume os valores de 0 ou 1. Os atributos que serão alterados no classificador serão: `learning_rate_init`, `momentum`. Os resultados foram os que se segue:

Execução	r = 0,05	r = 0,25	r = 0,5	r = 1	r = 2
1	3147	1096	1675	501	807
2	3088	1554	1015	1173	343
3	3473	1118	750	634	342
4	3740	1230	807	597	348
5	2853	1181	763	527	799
6	2751	1319	1680	505	823
7	3024	1157	862	580	358
8	3331	1111	774	503	806
9	3412	2391	776	504	339
10	3726	1176	778	574	353
Média:	3254,5	1333,3	988	609,8	531,8

Tabela 2 - tabela com o número de iterações até convergir (momento = 0)

Execução	r = 0.05	r = 0.25	r = 0.5	r = 1	r = 2
1	1901	758	1135	378	563
2	3953	761	687	366	233
3	2021	945	493	922	553
4	1995	714	647	337	220
5	1888	736	1464	386	213
6	1996	747	505	380	215
7	2013	786	530	371	220
8	3829	764	505	813	553
9	1890	752	507	403	220
10	2070	888	571	337	216
Média:	2355,6	785,1	704,4	469,3	320,6

Tabela 3 - tabela com o número de iterações até convergir (momento = 0.5)

Execução	r = 0.05	r = 0.25	r = 0.5	r = 1	r = 2
1	54	29	26	21	18
2	60	49	25	21	18
3	58	41	25	26	19
4	79	45	29	32	19
5	82	56	36	23	23
6	64	32	37	20	20
7	65	39	33	33	18
8	55	45	26	22	20
9	68	48	26	20	21
10	56	48	29	22	18
Média:	64,1	43,2	29,2	24	19,4

Tabela 4 - tabela com o número de iterações até convergir (momento = 1)

Perante os resultados é possível observar que de facto coincide com a teoria. Os algoritmos de descida de gradiente que precisam de menos iterações são os com a taxa de aprendizagem de maior (valor 2) e com o valor de momento mais alto (valor 1). Todos os testes anteriores foram feitos com os dados de treino organizados, se seguida será comparado os resultados organizados com não organizados, alterando o argumento `shuffle`. Os resultados foram os que se segue:

Execução	$r = 0.05$	$r = 0.25$	$r = 0.5$	$r = 1$	$r = 2$
1	70	39	26	21	17
2	52	32	24	22	21
3	53	36	29	28	19
4	76	31	25	22	22
5	84	28	27	21	25
6	62	37	27	20	20
7	57	49	31	23	19
8	83	35	34	21	23
9	55	40	27	29	23
10	50	31	28	21	23
Média:	64,2	35,8	27,8	22,8	21,2

Tabela 5 - tabela com o número de iterações (momento = 1, dados baralhados)

Perante os resultados é possível observar que de facto não existe uma diferença muito relevante entre a tabela apresentada com os dados baralhos e não baralhados com os mesmos parâmetros de taxa de aprendizagem e valor do momento. A descida de gradiente que precisa de menos iterações com os dados baralhados em média é 21,2 enquanto sem os dados baralhados é 19,4. Quer isto dizer que em geral os dados não baralhados produzem ligeiramente melhores resultados.

Por fim será testado os dados com codificação binária em vez de bipolar. Todos os outros testes foram realizados com codificação bipolar, por isso nesta etapa será testado com os dados em formato binário.

Execução	$r = 0.05$	$r = 0.25$	$r = 0.5$	$r = 1$	$r = 2$
1	100	46	34	30	26
2	98	46	41	31	26
3	98	38	35	26	26
4	84	40	32	28	23
5	79	42	45	36	24
6	108	58	52	25	24
7	70	60	42	29	32
8	153	41	30	35	25
9	80	55	52	30	28
10	93	59	60	25	26
Média:	91,8	42,4	37,4	30,2	25

Tabela 6 - tabela com o número de iterações (momento = 1, codificação binária)

Pelos resultados é possível observar que de facto a codificação bipolar possuí melhores resultados que a codificação binária. A menor quantidade de iterações em média com codificação bipolar foi 19,4 enquanto a codificação binária possui 25. Isto deve-se ao facto de que é mais fácil dividir valores entre -1 e 1 do que entre 0 e 1.

1.2.3 Aprendizagem dos padrões de imagem

Nesta etapa é pretendido criar uma rede neuronal capaz de aprender um padrão de imagens. As imagens são constituídas por 4x4, isto é, por quatro linhas e quatro colunas. No total a imagem possuí 16 quadrados, estes quadrados podem estar ou não preenchidos. Os padrões são os que se seguem:

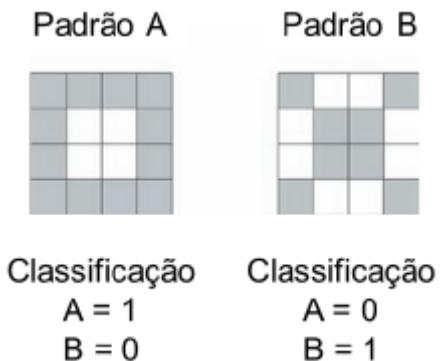


Figura 9 - padrões de imagem

O problema possuí duas classes como saída da rede neuronal, sendo elas o valor A e o valor B. Neste problema só existem duas regiões de decisão, por isso a rede pode utilizar somente um neurónio na camada escondida. Contudo neste problema foi utilizado duas camadas na camada escondida com 8 neurónios cada uma delas. Foi utilizada esta metodologia para acelerar o processo de aprendizagem da rede. O esquema da rede resultante foi o seguinte:

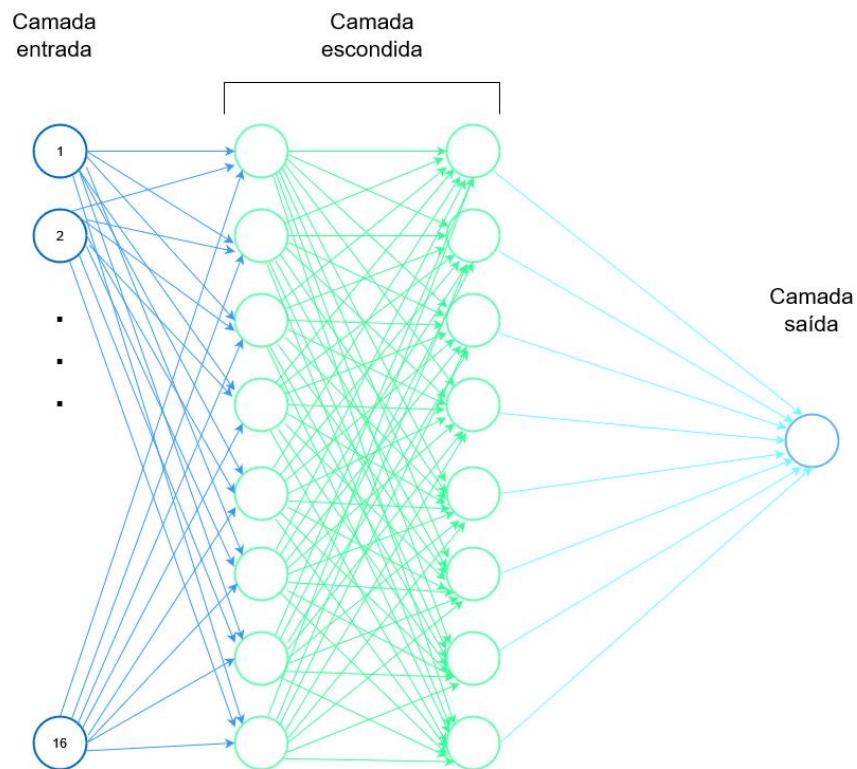


Figura 10 - constituição da rede

Para criar os dados de treino foram utilizados 10 padrões de matrizes de 4 por 4 com diferentes combinações, onde cada uma delas está mais próxima da combinação do padrão A ou do padrão B. Somente quando a rede neuronal conseguir ter uma taxa de acerto de 100% é assumido que a rede conseguiu aprender os padrões de imagens. Para testar a aprendizagem foram utilizados os seguintes padrões:

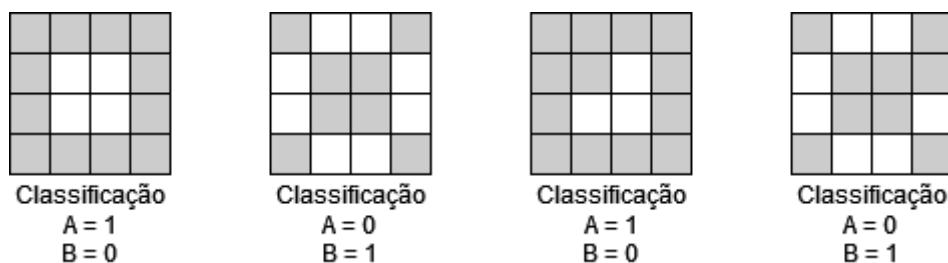


Figura 11 - padrões de teste

Para testar a rede neuronal com diversos parâmetros primeiramente foi alterada a taxa de aprendizagem. Destes testes foram obtidos os seguintes resultados:

Taxa de aprendizagem	Momento	Número iterações
0.1	1	351
0.01	1	400
0.001	1	836

Tabela 7 - resultados das iterações para diferentes parâmetros

Como se pode observar na tabela anterior de facto quanto maior for a taxa de aprendizagem menos iterações é necessário para conseguir que a rede neuronal consiga aprender o problema em questão. De seguida foi testado o mesmo problema com diversas funções de ativação para a rede neuronal. Nestes testes foi utilizada sempre uma taxa de aprendizagem de 0.001 e um valor de momento de 1. Os resultados foram os que se segue:

Função de ativação							
relu		identity		logistic		tanh	
Nº iterações	Perda	Nº iterações	Perda	Nº iterações	Perda	Nº iterações	Perda
435	0.00019	460	0.0002	1196	0.0041	607	0.0013
526	0.00022	401	0.00019	1280	0.004	593	0.0014
709	0.00156	413	0.00015	1322	0.005	618	0.0016
770	0.0023	406	0.00022	1205	0.0034	655	0.0019
799	0.00304	402	0.00016	2903	0.0063	671	0.0022
Média	647,8		416,4		1581,2		628,8

Tabela 8 - resultado das iterações com diversas funções de ativação

Perante os resultados é possível observar que de facto a função de ativação que precisa de uma menor quantidade de iterações é a função **identity**.

1.2.4 Resolução de um problema livre

Neste problema é pretendido resolver um problema de escolha livre. Para a resolução deste problema foi utilizado o *dataset* do fashion Mnist. Este *dataset* é constituído por 10 classes com imagens a preto e branco com tamanho de 28x28 pixéis. Foi escolhido resolver dois problemas associados a este *dataset*, um problema binário e um problema multi-classe. Para o problema binário foi escolhido que a rede neuronal deve conseguir classificar se uma peça de roupa é uma t-shirt ou um top ou se é uma bota. Exemplos de imagens de cada classe binária é a que se segue:

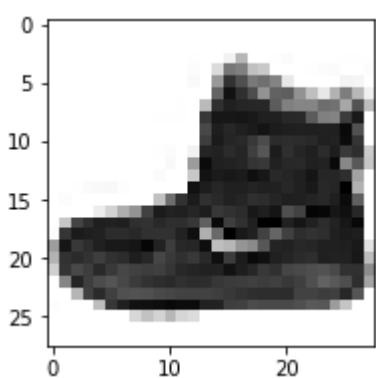


Figura 12 - exemplo de bota

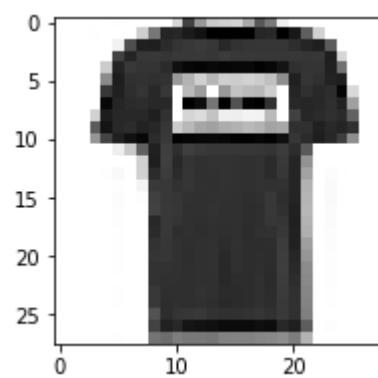


Figura 13 - exemplo de t-shirt

Para resolver este problema foi utilizado a biblioteca TensorFlow, primeiramente foi obtido só as botas e as t-shirts e de seguida foram baralhos os dados. A rede utilizada possuía a seguinte constituição:

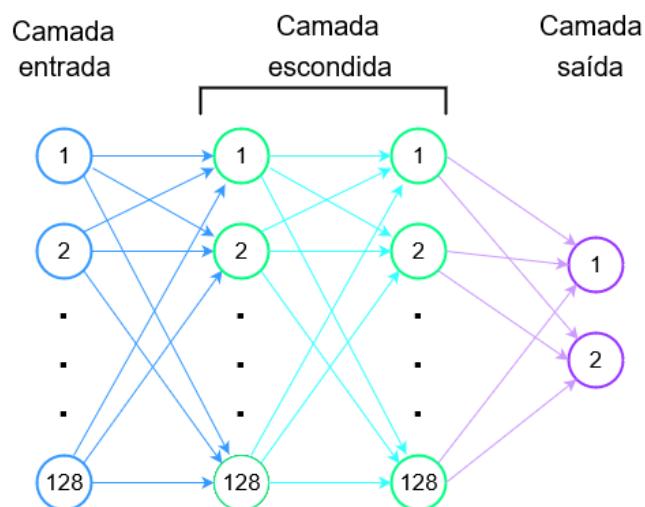


Figura 14 - constituição da rede

A partir desta rede com 5 *epoch* (número de vezes que o algoritmo vai percorrer todo o *dataset* de treino) foi possível obter uma taxa de acertos 99,9%. O resultado da matriz de confusão foi o que se segue:

$$\begin{bmatrix} 1000 & 0 \\ 2 & 998 \end{bmatrix}$$

Para a classificação multiclasse foram utilizadas todas as 10 classes de todo o *dataset* do fashion Mnist. Outros exemplos da roupa do *dataset* é por exemplo vestidos e camisolas como se observa na seguinte figura:

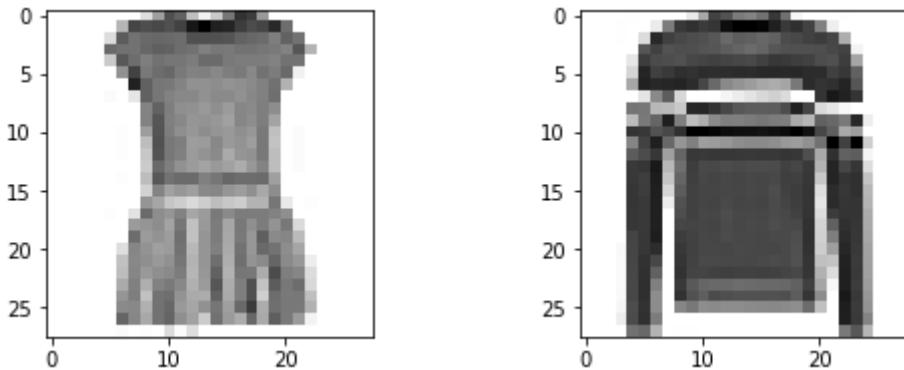


Figura 15 - exemplos de outras classes do dataset

Para solucionar este problema foi utilizada a mesma constituição da rede para o problema binário e foi alterado somente a camada de saída. Como neste problema existem 10 classes possíveis de ser atribuídas a camada de saída foi postas com 10 tensores de saída. Por isso a rede passou a ter a seguinte constituição:

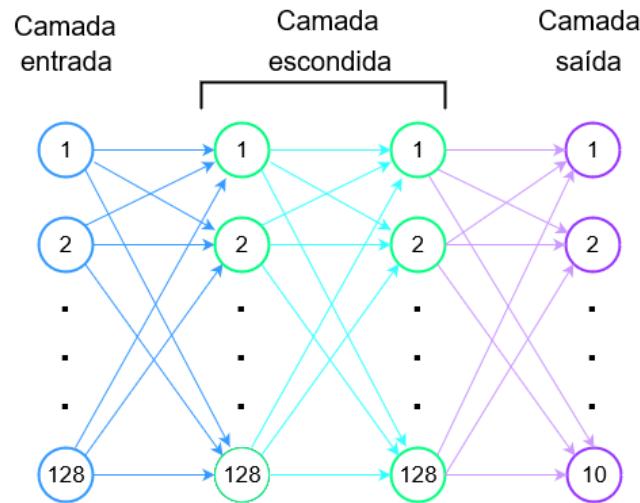


Figura 16 - constituição da rede multiclassse

Perante esta rede com 5 epoch foi possível de obter uma taxa de acertos 83.21%. A matriz de confusão resultante foi a que se segue:

```
[[710  4   6 110   0   1 156   0  13   0]
 [ 0 961  0  29   1   0   8   0   1   0]
 [ 5   1 732  15  67   0 176   0   4   0]
 [ 7  13   3 905  22   0  46   1   3   0]
 [ 1   1 170  52 591   0 181   0   4   0]
 [ 0   0   0   0   0 930   2  39   3  26]
 [116  4  90  68  44   2 659   0  17   0]
 [ 0   0   0   0   0  25   0 951   1  23]
 [ 0   0   1 12   2   3  25   3 954   0]
 [ 0   0   0   0   0  13   0  58   1 928]]
```

Figura 17 - matriz de confusão resultante

2. Aprendizagem por reforço

2.1 Componente teórica

A aprendizagem por reforço é uma aprendizagem de comportamentos. Para este tipo de aprendizagem ele aplica tentativas e erros para chegar a uma solução para o problema. Para que o computador consiga realizar o que o programador pretende o agente ganha recompensas e penalizações por ações que realiza, onde o objetivo é maximizar a recompensa total. O programador não dá qualquer sugestão em como resolver o problema, somente a função de recompensas e penalizações. Depois o modelo deve trabalhar para maximizar a recompensa.

Isto é realizado através do seguinte procedimento:

- O agente interage com o ambiente e toma decisões. Para o treino o agente é providenciado com informação sobre o ambiente e as escolhas. A interação é feita por meio de pares estado – ação, onde o estado representa onde o agente se encontra num determinado instante temporal e a ação é o que move o agente para um novo estado.
- O agente é providenciado com recompensas ou penalizações, designado reforço, baseadas em quão bem as ações tomadas pelo agente contribuíram para o objetivo desejado. Ao conjunto de pares estado-ação e as devidas recompensas e penalizações obtidos pelo agente é designado a política, a política ótima é o conjunto de pares estado-ação que obtém a recompensa cumulativa mais alta.

O esquema que se segue representa uma iteração do agente com aprendizagem por reforço no ambiente. O s representa um estado, o S representa o conjunto total de estados, a representa uma ação, A representa o conjunto de todas as ações e r representa o reforço (recompensa) para a ação tomada.



Figura 18 - iteração do agente no ambiente

Inicialmente o agente com aprendizagem por reforço não sabe nada e por isso explora o mundo tomando decisões aleatoriamente e observa o seu reforço. Quando recolhe informação suficiente começa a aproveitar o conhecido e começa a realizar as ações que sabe que produzem a melhor recompensa. Contudo existe um problema que é decidir quando é que o agente aprendeu o suficiente para começar a aplicar o que aprendeu. Por isso existem dois paradigmas:

- Exploração: o agente escolhe uma ação que permite explorar o mundo para melhorar a aprendizagem, resultando na obtenção de experiência.
- Aproveitamento: o agente escolhe uma ação que leva à melhor recompensa conhecida de acordo com a sua aprendizagem (ação gananciosa). Este paradigma está limitado às ações que conhece serem favoráveis.
- Para ser possível convergir para a política ótima não se pode só explorar, nem se pode só aproveitar. Deve-se progressivamente reduzir a exploração e começar a aproveitar mais o conhecimento.

Para conseguir uma estratégia que consiga aplicar a exploração e aproveitamento de uma forma sistemática existe a estratégia ϵ – *greedy*. Esta estratégia explora com uma determinada probabilidade que vai sendo sistematicamente reduzida conforme o agente ganha mais conhecimento. Por isso consegue realizar o balanceamento entre exploração e aproveitamento. Este método segue a seguinte fórmula:

$$a_t = \begin{cases} a_t^* \text{ com probabilidade } 1 - \varepsilon \\ \text{ação aleatório com probabilidade } \varepsilon \end{cases}$$

Para a aprendizagem por reforço o processo de aprendizagem é feito por meio de regras entre estímulo e resposta e pela valorização entre o par estado-ação $Q(s, a)$. A aprendizagem é feita por diferença temporal, ou seja, a atualização de uma estimativa de valor estado – ação é feita com base na mudança do valor em instantes temporais distintos. Este paradigma segue as seguintes fórmulas:

- $Q(s, a) \rightarrow Q(s, a) + \alpha[R - Q(s, a)]$
- $R = r + \gamma Q(s', a')$
- r é o reforço atual
- $r + \gamma Q(s', a')$ estimativa atual do reforço
- $Q(s, a) \rightarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$

Por cada par estado-ação, o resultado do reforço é sempre o valor do reforço num determinado estado mais a diferença entre o novo estado descontado no tempo e o estado presente. A diferença temporal é importante pois permite ao agente tomar decisões de ser preferível receber agora uma recompensa menor, ou de não a receber agora e receber mais tarde uma recompensa maior.

Existem duas formas principais de aprendizagem. Existe o algoritmo Q-Learning e SARSA. O algoritmo SARSA utiliza o controlo de aprendizagem *On-Policy* enquanto o algoritmo Q-Learning utiliza *Off-Policy*. Estes controlos consistem em:

- *On-Policy*: utiliza a mesma política para a seleção da ação e aprendizagem através de propagação de valor, onde explora todas as ações (*e-greedy* e *e-soft*).
- *Off-Policy*: existe uma separação entre ação e aprendizagem através de políticas específicas, onde para a aprendizagem utiliza política *greedy* e para seleção de ação utiliza política *e-greedy*. Isto resulta na otimização da função de valor $Q(s, a)$.



Figura 19 - diferença entre algoritmo SARSA e Q-Learning

A aprendizagem por reforço possui algumas limitações, nomeadamente a complexidade dos espaços de estados e o tempo que pode demorar o modelo a convergir. As soluções podem partir como ter uma memória das experiências realizadas, utilizar modelos do mundo, arquiteturas híbridas entre outros.

Uma das maneiras de acelerar o processo de aprendizagem é através da memória de experiência. Onde as experiências do agente são memorizadas numa memória de experiência. Num instante t , a experiência do agente é definida por $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$. Esta memória de experiência real é utilizada para simular a experiência real, através da aprendizagem de valores e políticas de ação. Contudo é necessário ter em conta a dimensão da memória de experiência, onde esta pode começar a crescer muito rapidamente e torna o processamento muito lento. Por isso é necessário remover as experiências mais antigas ou remover por critério de menor relevância. A seleção de amostras pode ser aleatória ou por critério de relevância.

Uma outra maneira de aprendizagem é através de modelos do mundo. O modelo do mundo é uma representação das características relevantes do domínio do problema, nomeadamente a estrutura (estado), dinâmica (transição de estado) e o valor (recompensa). Isto permite a aprendizagem indireta através da simulação de experiência a partir de modelos internos, onde a interação de comportamentos é feita onde não é possível realizar diretamente no ambiente.

2.2 Implementação

Nesta fase é pretendido implementar uma biblioteca de aprendizagem por reforço. Na aprendizagem por reforço é pretendido que sejam implementados os seguintes métodos de aprendizagem:

- Q-Learning
- Q-Learning com memória episódica
- Dyna-Q

Após feita a implementação da biblioteca também é pretendido criar um agente capaz de navegar num espaço de dimensões discreta de forma a testar o funcionamento da biblioteca desenvolvida. O espaço é composto por obstáculos e um alvo. O objetivo é o agente conseguir aprender a recolher o alvo tendo por base os métodos de aprendizagem por reforço. Para ser perceptível o comportamento do agente será desenvolvido uma interface gráfica do comportamento do agente.

Para desenvolver a biblioteca de aprendizagem por reforço assim como os operadores e a interface gráfica foi escolhida a linguagem de programação Python. Inicialmente será explicada a biblioteca desenvolvida, de seguida os operadores e por fim a interface gráfica. O termo operador é referente ao conjunto de atributos constituintes de o que é um estado, uma ação e conseguir alterar o ambiente de acordo com a ação.

2.2.1 Implementação da biblioteca

Para a implementação da biblioteca por reforço, foram criadas as seguintes classes:

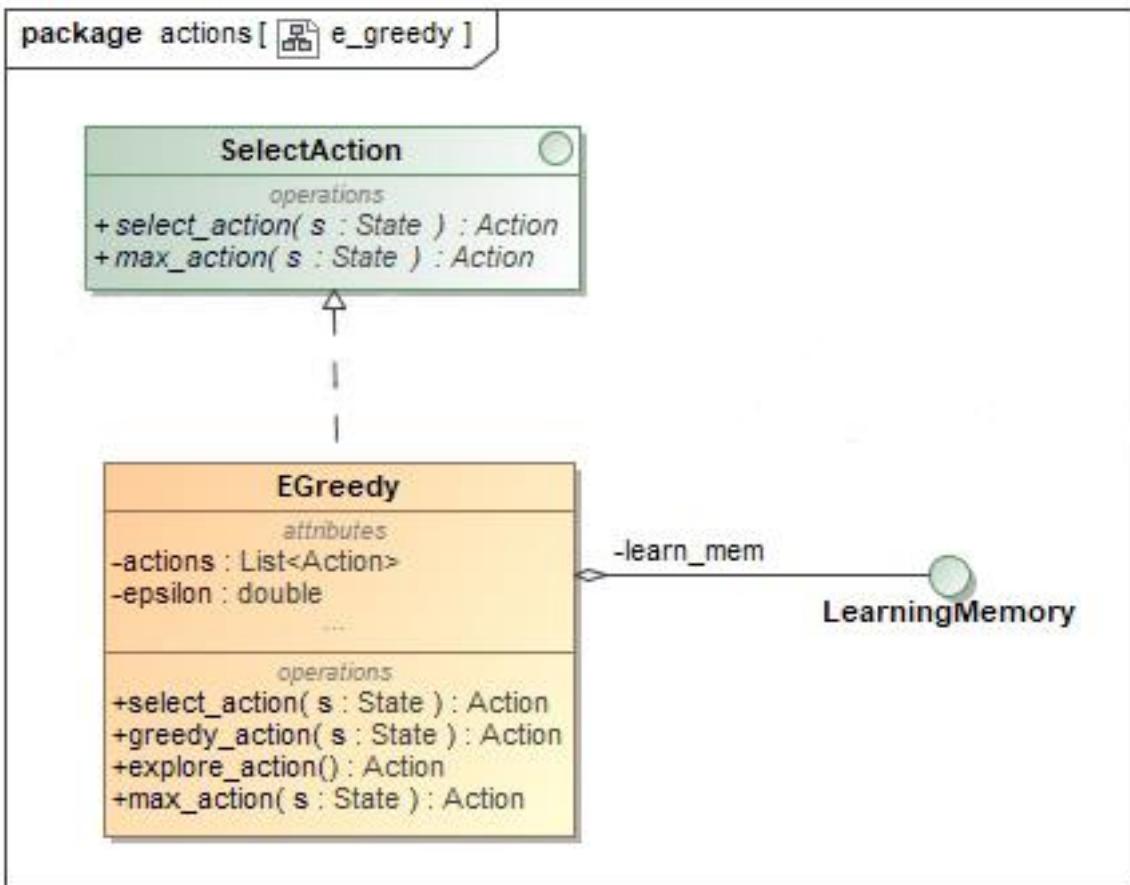


Figura 20 – diagrama de classes da seleção de ação

A seguinte classe **EGreedy** implementa a seleção de uma ação pelo agente, através do método **select_action()**. Esta escolha tem uma probabilidade de escolher uma ação que produz o reforço com valor mais alto ou escolher uma ação aleatória. Esta classe também precisa de uma referência para a **LearningMemory**, pois será a partir dela que conseguirá saber as ações que produzem um reforço mais elevado. O diagrama de atividades desta classe é o que segue:

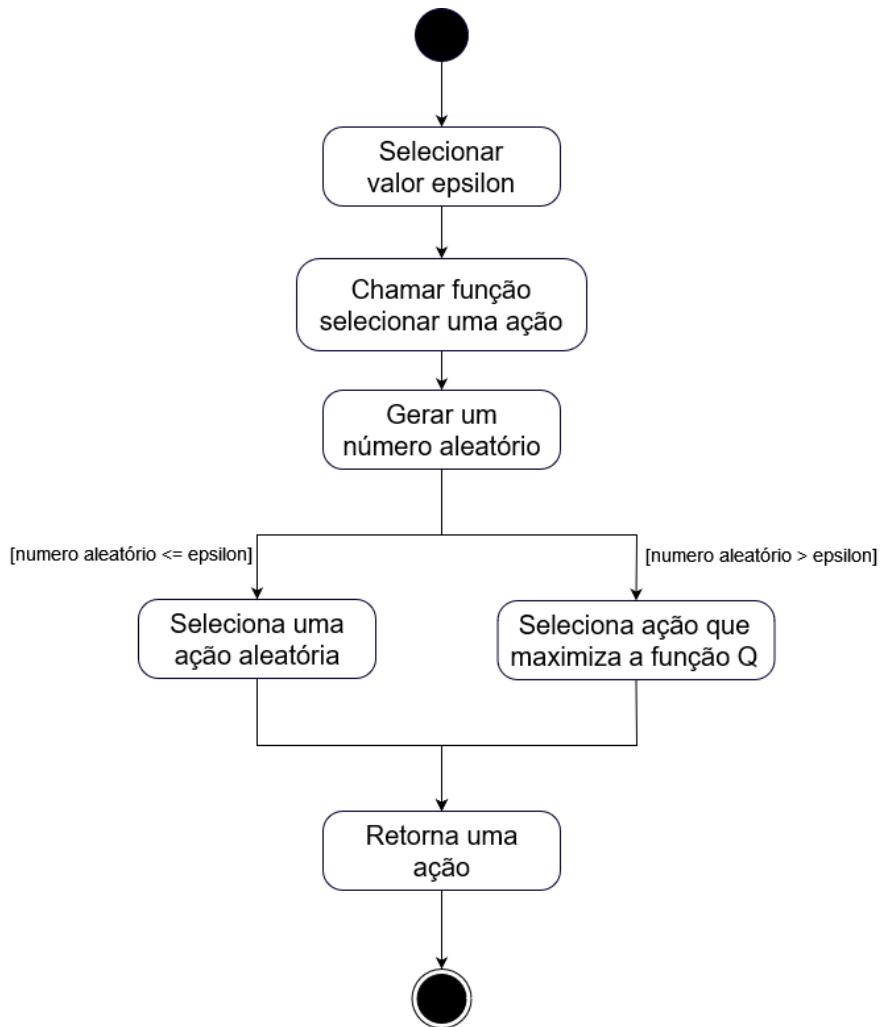


Figura 21 - diagrama de atividades da seleção de ação

Para guardar os resultados do reforço obtido pelos algoritmos de aprendizagem por reforço de acordo com um par estado-ação é utilizada uma memória. Esta memória também é conhecida como política, pois ela é que dita que ações são boas e que ações são más. A classe representativa deste comportamento é `SpreadMemory`. Sempre que o agente passa novamente num par estado-ação e toma uma ação conhecida o valor do reforço obtido é atualizado. O diagrama de classes representativo desta memória é o seguinte:

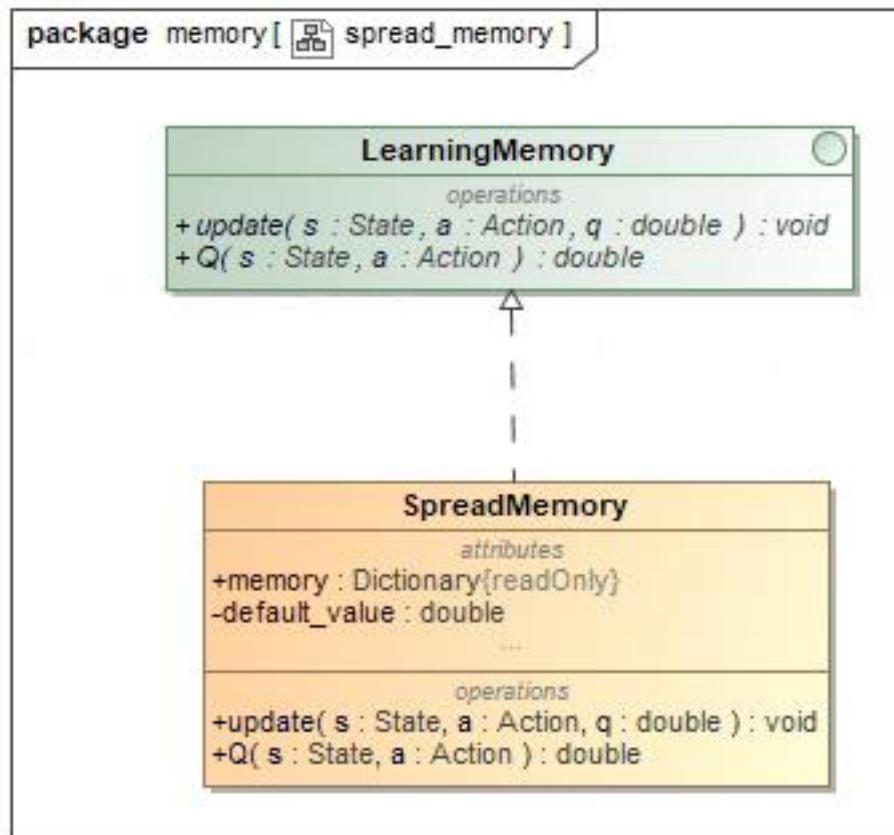


Figura 22 – diagrama de classes da memória de aprendizagem

A classe responsável pela implementação dos algoritmos de aprendizagem é a classe SARSA e QLearning. Estas classes estendem de uma classe abstrata designada ReinfLearning. O diagrama é o que se segue:

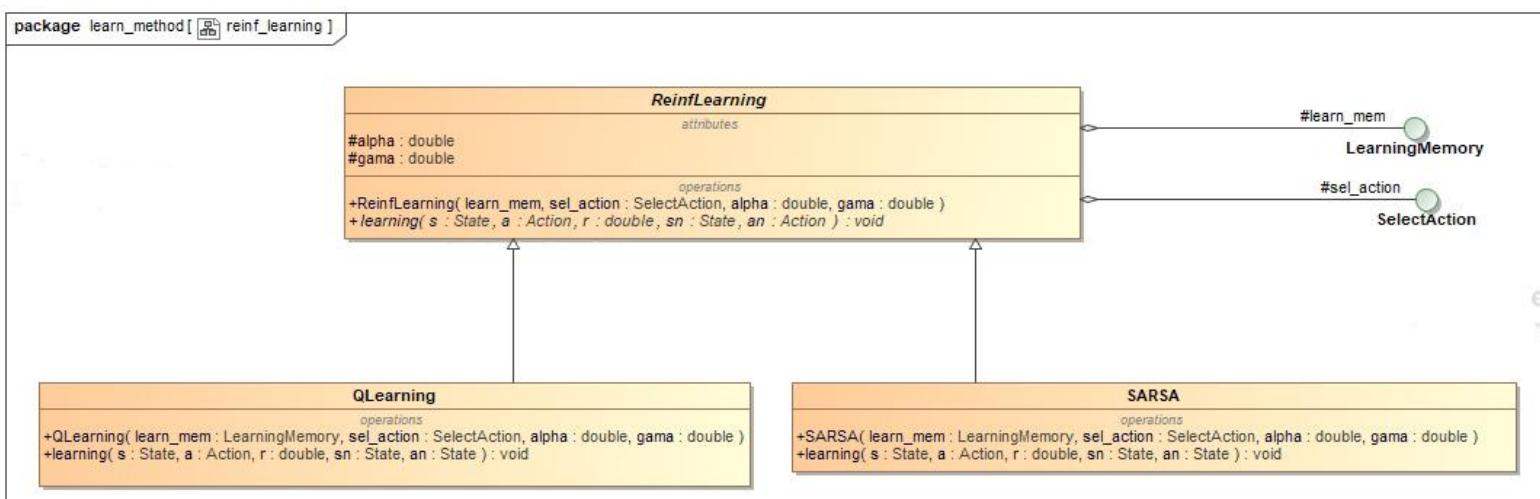


Figura 23 - diagrama de classes da implementação do algoritmo aprendizagem

A classe `ReinfLearning` possui uma referência para a classe que implemente `SelectAction` e `LearningMemory`, que representa a memória e a seleção de ação referidas anteriormente. Cada classe que estende de `ReinfLearning` tem de implementar o método `learning()`, que corresponde respetivamente ao algoritmo de aprendizagem. Além destes algoritmos existe também o algoritmo com memória episódica que segue o seguinte diagrama de classe:

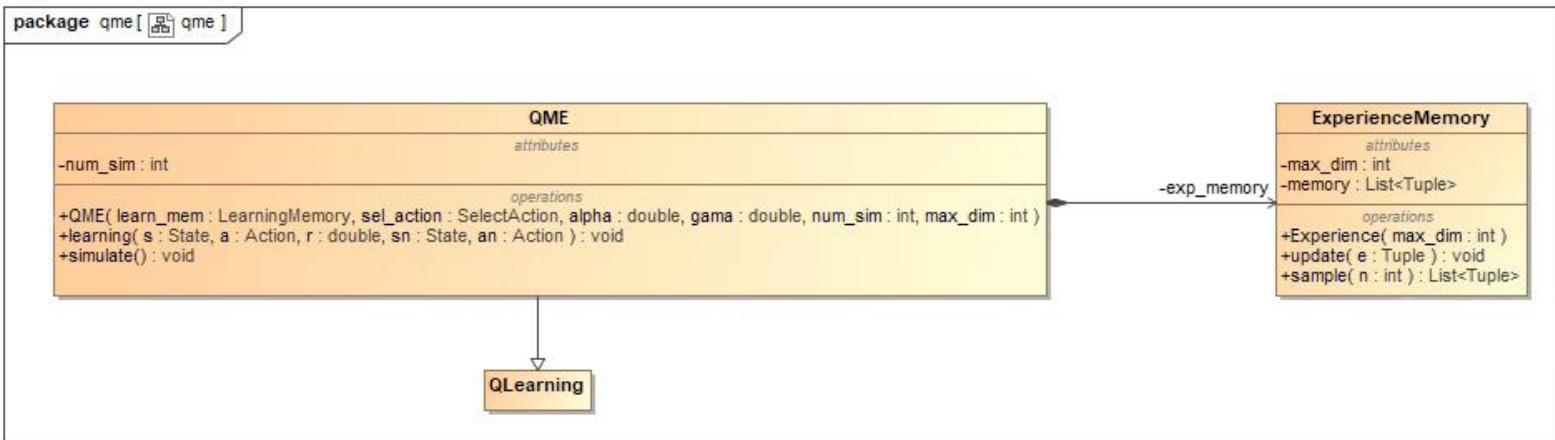


Figura 24 - diagrama de classes do algoritmo Q-Learning com memória episódica

Este algoritmo é representado pela classe QME e esta classe utiliza uma memória de experiência para simular a experiência real. Além deste algoritmo existe mais um algoritmo que é o algoritmo Dyna-Q que é dado pelo seguinte diagrama:

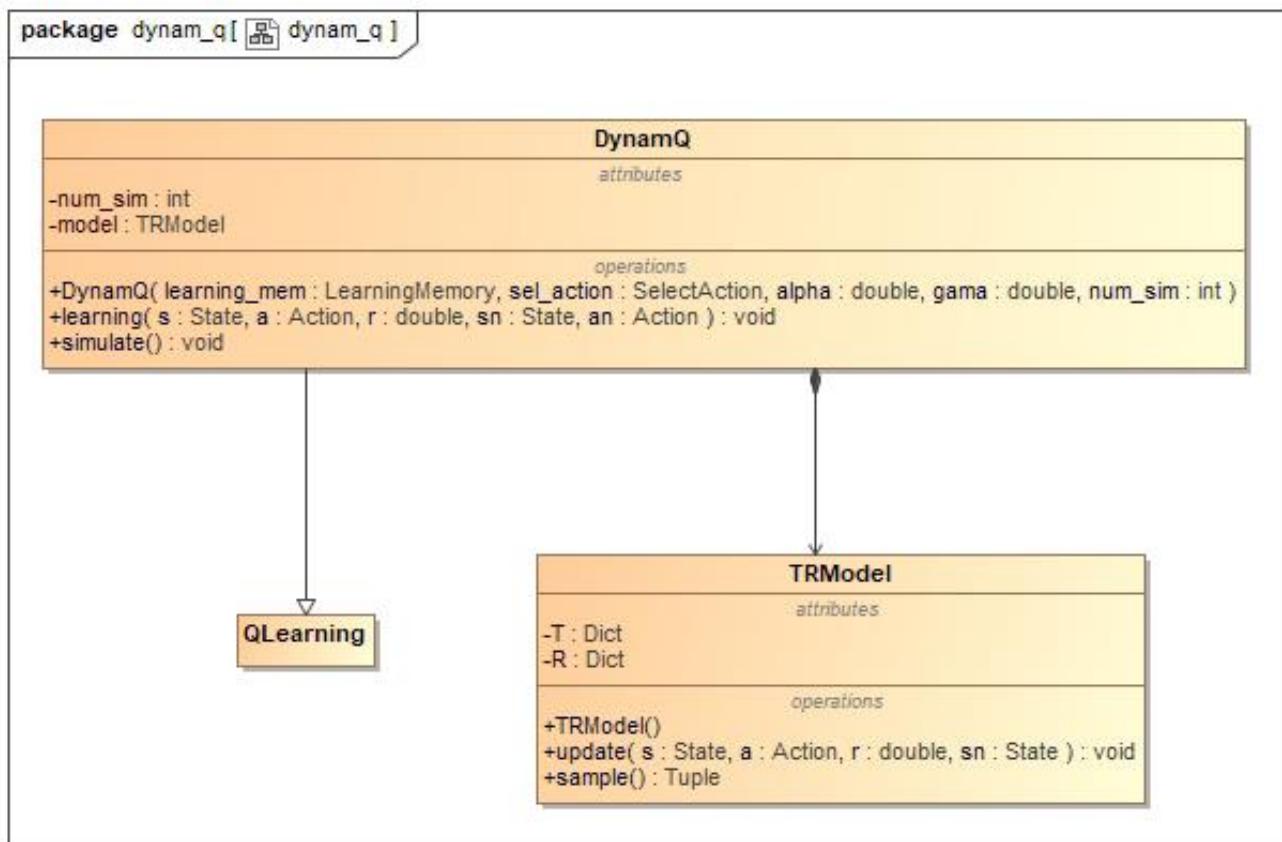


Figura 25 - diagrama de classes do algoritmo Dyna-Q

O algoritmo é implementado na classe **DynamQ**. Este algoritmo utiliza modelos do mundo, designado pela classe **TRModel**, para obter características relevantes do domínio do problema. Estes modelos permitem a aprendizagem indireta através da simulação de experiência a partir de modelos internos.

```
package reinforcement_learning [  reinf_engine ]
```

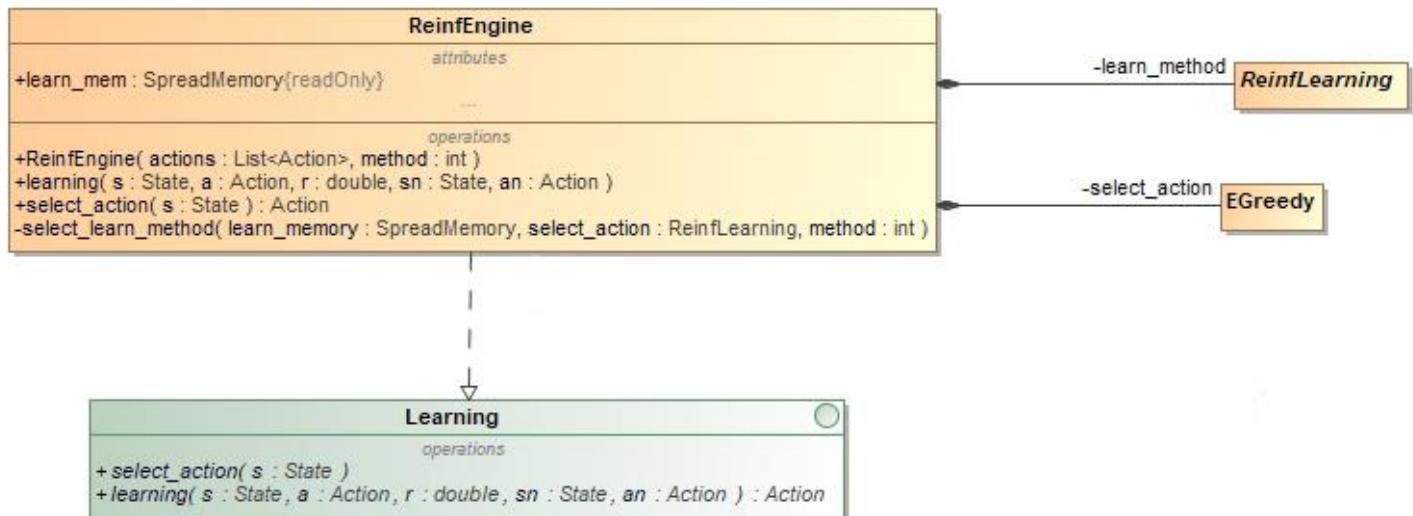


Figura 26 - diagrama de classes do mecanismo de aprendizagem por reforço

A classe `ReinfEngine` é a classe responsável por encapsular todos os conceitos referidos anteriormente. Será esta classe que instancia as diferentes componentes e também será a classe visível para o exterior da biblioteca de aprendizagem por reforço. Esta classe para funcionar precisa de uma lista de ações definida por quem vai utilizar a biblioteca e por um valor inteiro que dita que algoritmo será utilizado. Os valores são:

0. SARSA
1. Q-Learning
2. QME
3. Dyna-Q

2.2.2 Implementação dos operadores

Os operadores são os responsáveis por tudo o que diz respeito ao estado do mundo, desde observar o estado do mundo à posição do agente. Os operadores devem enviar essa informação à biblioteca e também de mover o agente no mundo de acordo com as ações escolhidas pela biblioteca. Os operadores também definem o que é um estado, uma ação e o objetivo.

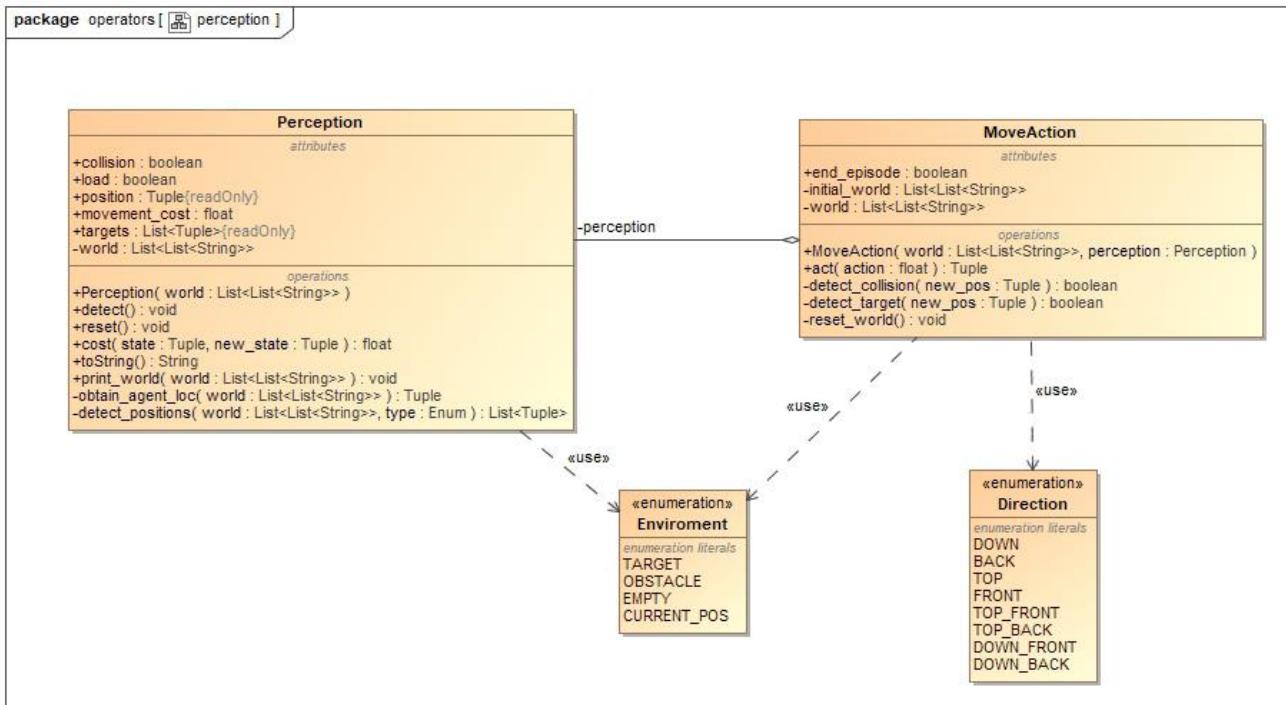


Figura 27 - diagrama de classes da percepção e da ação mover

A classe **Perception** é responsável por obter a posição do agente, do(s) objetivo(s) e o que é um estado. Um estado é nomeadamente um tuplo constituído por:

- (posição em x, posição em y)

Esta também é a classe que o agente utiliza para saber as informações de se o agente atingiu um objetivo, colidiu e qual foi o custo do movimento. Para saber este tipo de informação existe um ficheiro em texto com o mapa do agente, onde os valores estão representados através do enumerador **Enviroment**. O enumerador facilita a conversão de o que cada caracter do ficheiro de texto representa através do seu valor. O enumerador possui os seguintes valores:

- TARGET: objetivo
- OBSTACLE: obstáculo
- EMPTY: posição vazia
- CURRENT_POS: posição corrente do agente

A classe `MoveAction` é a classe responsável por mover o agente no ambiente de acordo com a ação selecionada. Esta classe é que atribui à classe `Perception` se colidiu, atingiu o objetivo e qual foi o custo de movimento. A classe também permite dar *reset* ao mundo quando o agente atinge o objetivo. O custo de movimento é obtido através do método `cost()` da classe `Perception`, dependendo do movimento feito pelo agente. Os diferentes movimentos do agente são ditados pelo enumerador `Direction`, onde possuí movimento para uma vizinhança de 8 e o custo segue o teorema de Pitágoras. Como se observa no seguinte diagrama:

$\sqrt{2}$	1	$\sqrt{2}$
1		1
$\sqrt{2}$	1	$\sqrt{2}$

Figura 28 - custo de movimento do agente

Por isso observa-se que o agente consegue movimentar-se na horizontal, vertical e nas diagonais. O custo das diagonais é um valor com maior custo, pois o movimento é feito em duas posições simultaneamente.

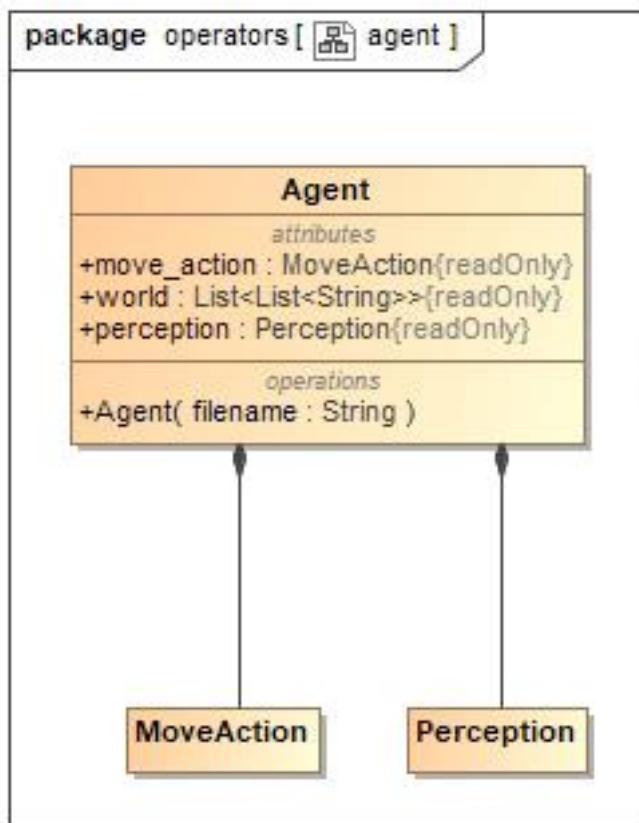


Figura 29 - diagrama de classes do agente

A classe que encapsula a percepção e ação de mover é a classe **Agent**, esta classe permite a leitura destas classes. Também é esta classe que faz a leitura dos ficheiros e converte para uma lista de **Strings** para poder serem alterados.

2.2.3 Implementação dos controlos

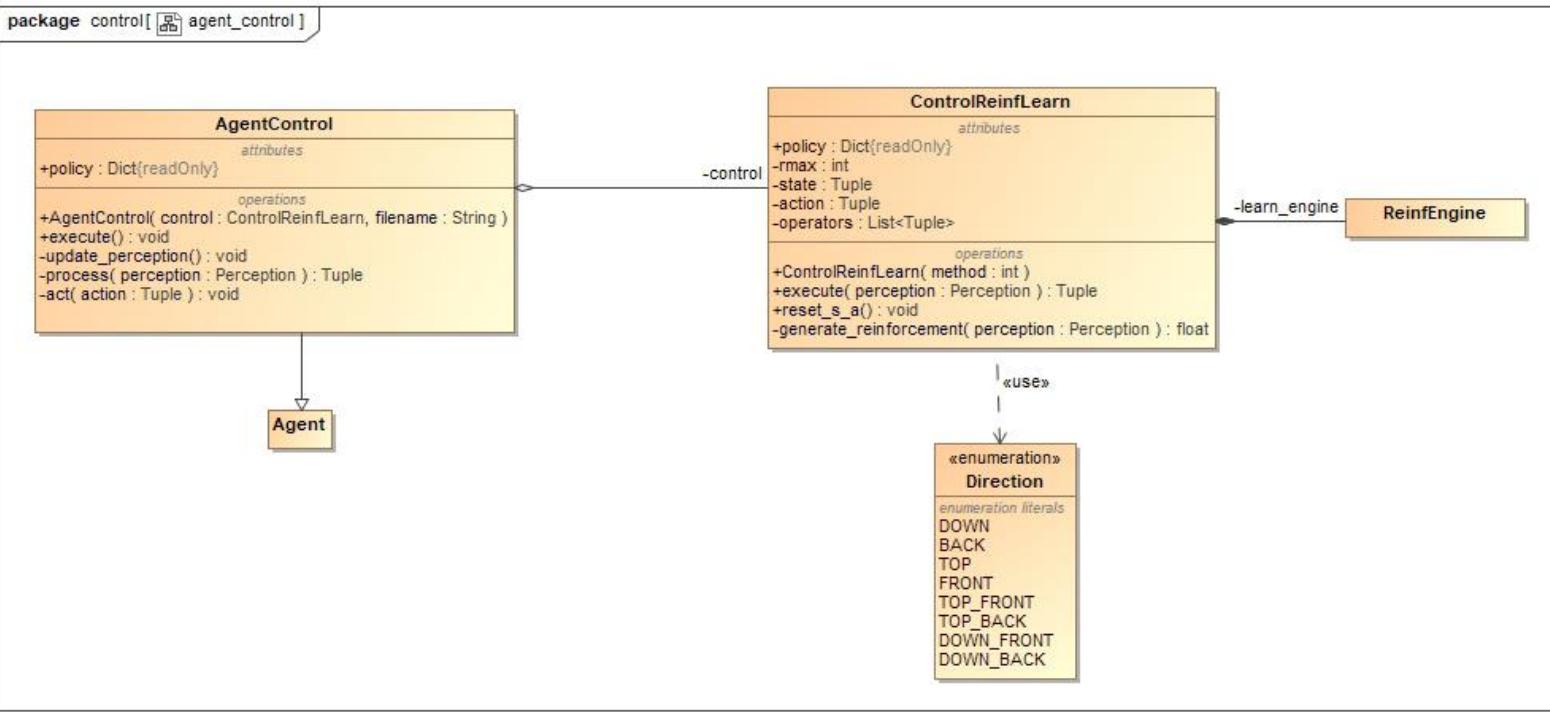


Figura 30 - diagrama de classe do controlo do agente

A classe responsável por definir as diferentes ações que o agente pode realizar é a **ControlReinfLearning**. As ações correspondem ao valor do enumerador **Direction**, que corresponde à vizinhança de 8 do agente referida na classe **MoveAction**. Esta classe é que utiliza a classe exposta pela biblioteca de aprendizagem por reforço. O método que executa a aprendizagem por reforço é o **execute()**, ele permite também gerar o reforço em cada episódio e fazer *reset* nas variáveis de colisão, custo movimento, entre outros no final de cada episódio.

A classe que junta todas as funcionalidades numa única é a classe **AgentControl**. Caso fosse pretendido utilizar o agente em modo consola bastaria somente imprimir o mundo na consola e a política obtida. Contudo foi criada uma interface gráfica para facilitar a visualização do comportamento do agente e da política.

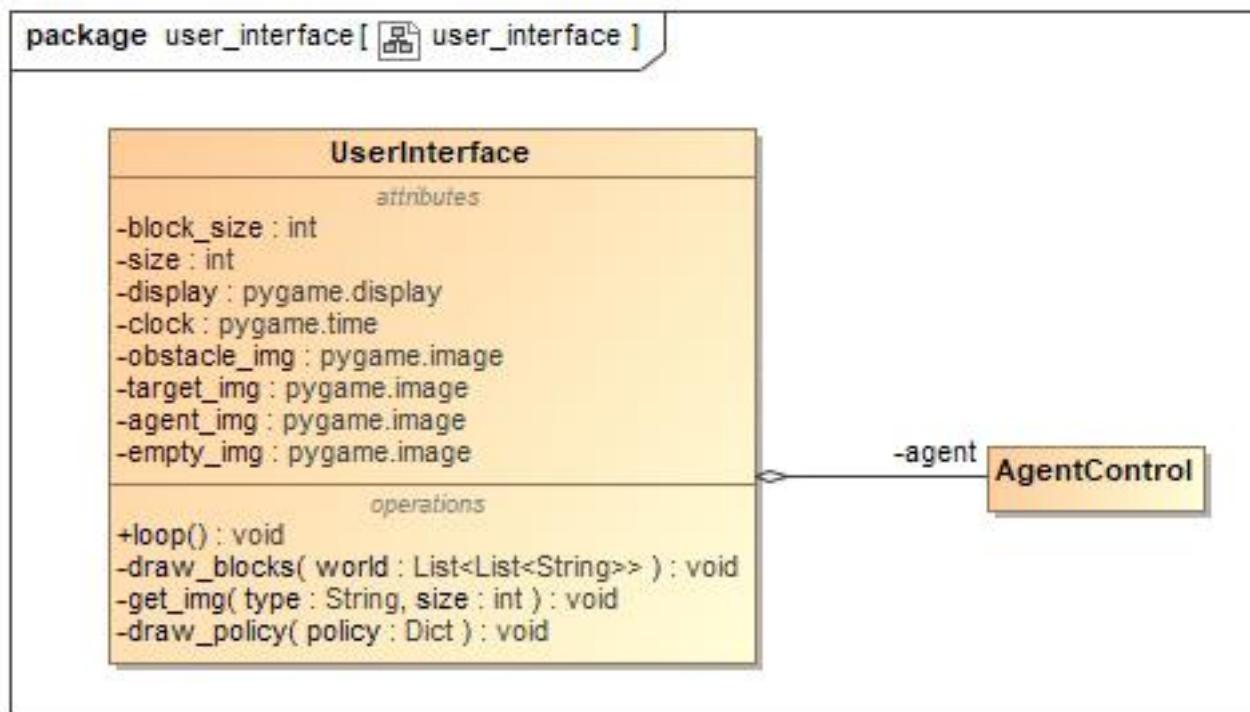


Figura 31 - diagrama de estado da interface gráfica

A interface gráfica essencialmente utiliza a classe `AgentControl` para executar todo o movimento e aprendizagem do agente. A classe `UserInterface` é responsável por desenhar o ambiente e a política obtida pelo agente.

2.2.4 Arquitetura geral da solução e diagramas de sequência

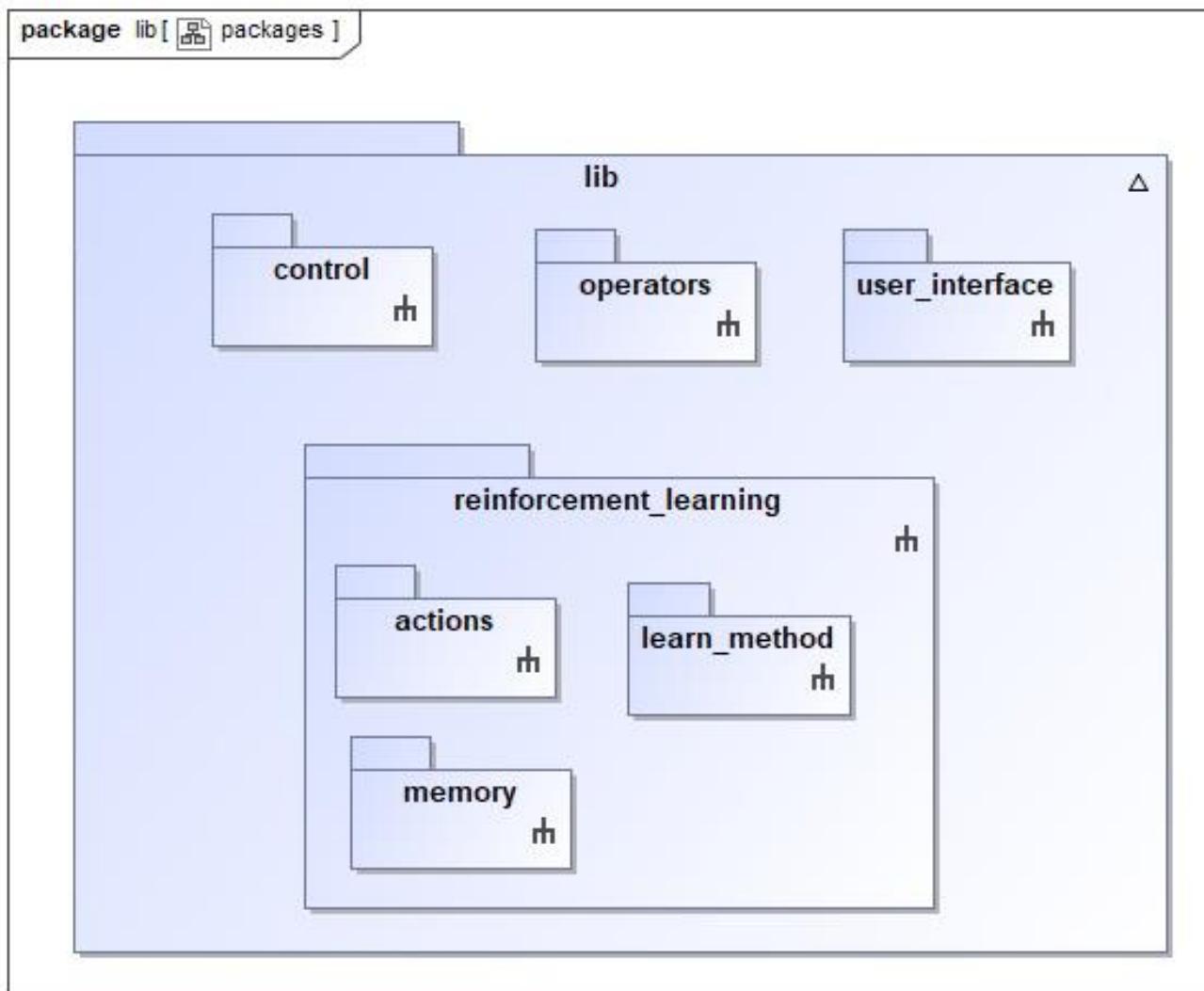


Figura 32 - constituição dos *packages* do programa

Toda a implementação do trabalho está feita dentro da pasta lib. Esta pasta contém a biblioteca de aprendizagem por reforço que se encontra na pasta reinforcement_learning. Os operadores é onde se encontra as componentes referentes ao ambiente e ao agente. O controlo é responsável por encapsular a biblioteca de aprendizagem por reforço com os operadores. A pasta user_interface é onde está a classe da interface gráfica exibida pelo agente e as imagens utilizadas para construir a interface.

A interação das classes segue os diagramas de sequência que se segue:

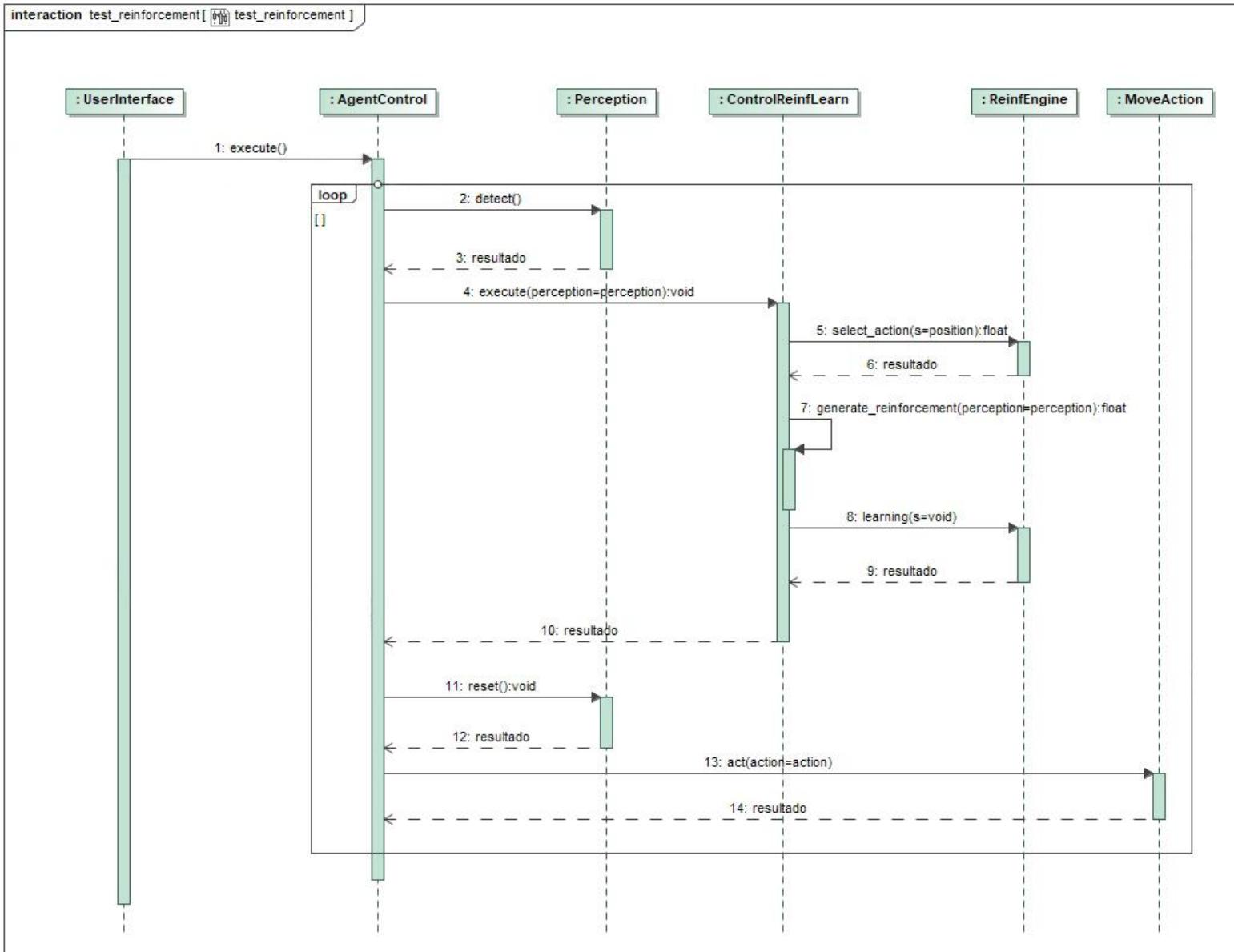


Figura 33 - diagrama de sequência da execução do programa

Para facilitar a interpretação, a interação foi dividida em três diagrama diferentes. No diagrama seguinte é explicado a interação realizada quando é chamado o método `select-action()` na classe `ReinfEngine`. Depois desse diagrama está descrito quando é chamado o método `learning()` da classe `ReinfEngine` para o caso da classe `DynamQ`.

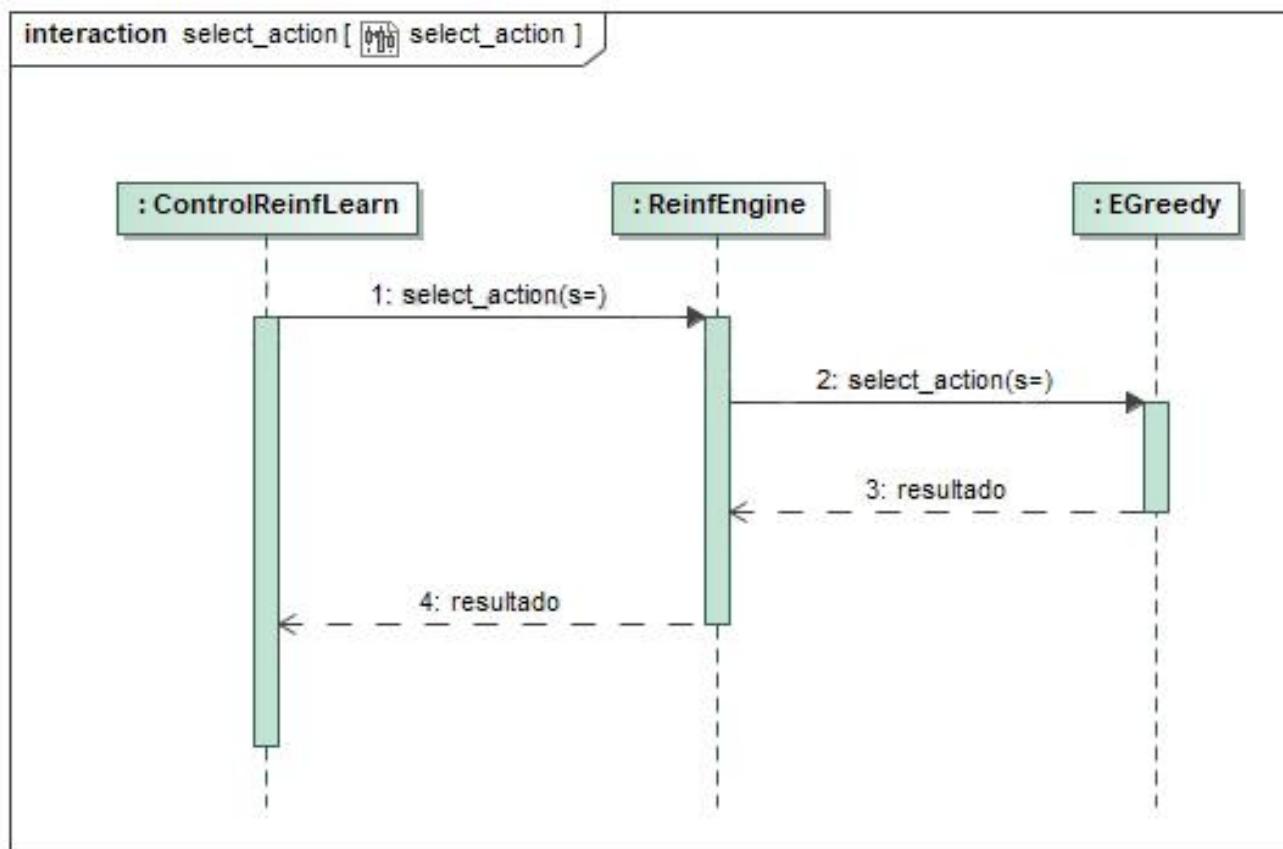


Figura 34 - diagrama de sequência da seleção de ação

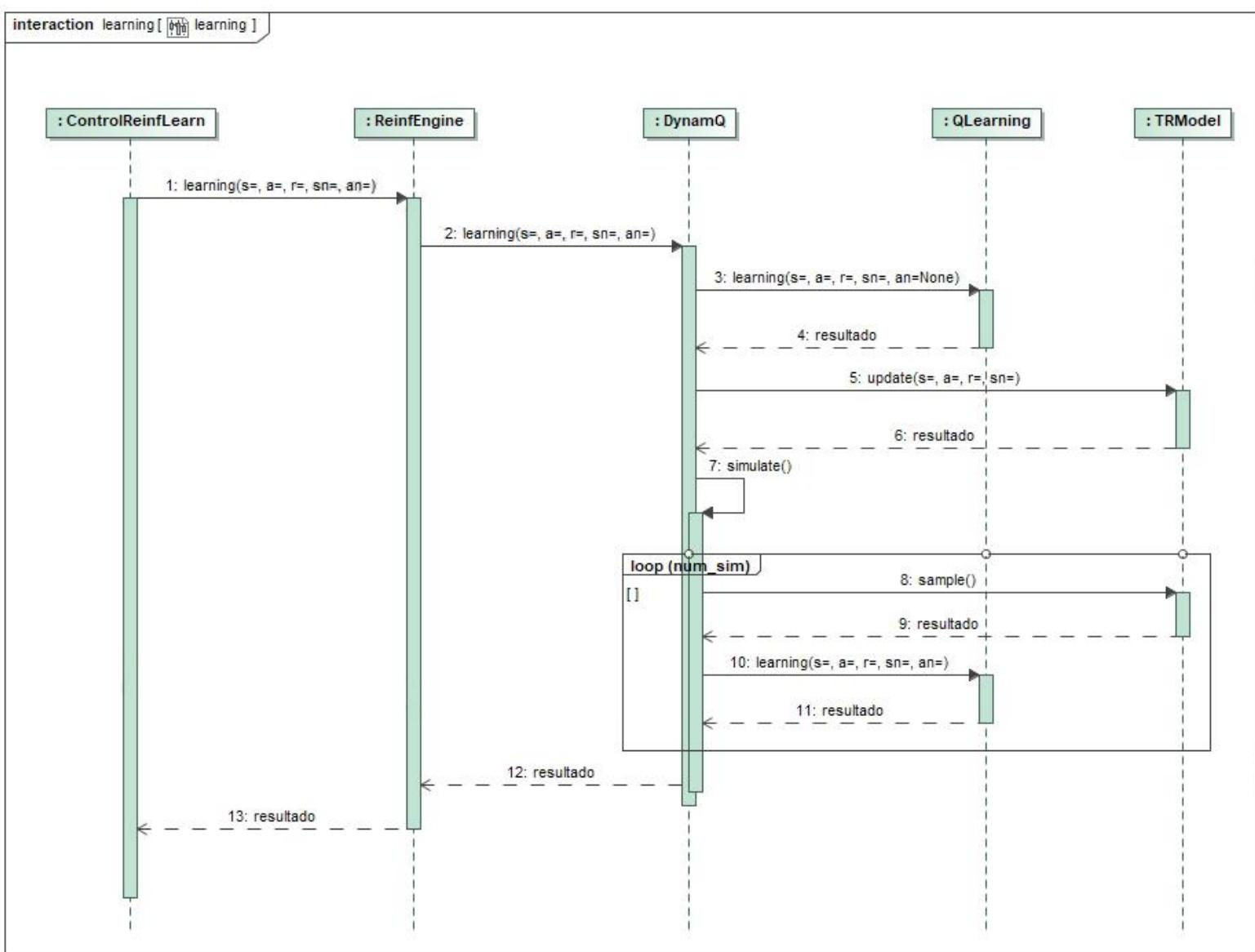


Figura 35 - diagrama de sequência da aprendizagem

2.2.5 Resultados obtidos

Em termos dos resultados obtidos, foi testado todos os algoritmos implementados nos dois ambientes disponibilizados. A ver pelos resultados obtidos, nomeadamente o caminho que o agente percorre ao fim de algum tempo de explorar o ambiente e encontrar o objetivo, este começa de facto a melhorar o caminho. Esta melhoria é cada vez notória por cada vez que o agente encontra o objetivo. Nas demonstrações será utilizado somente o ambiente dois, por conter mais estados e ser mais perceptível a política, contudo o programa funciona em ambos os ambientes.

Primeiramente será mostrado o funcionamento do algoritmo Q-Learning, que é mostrado na seguinte figura:

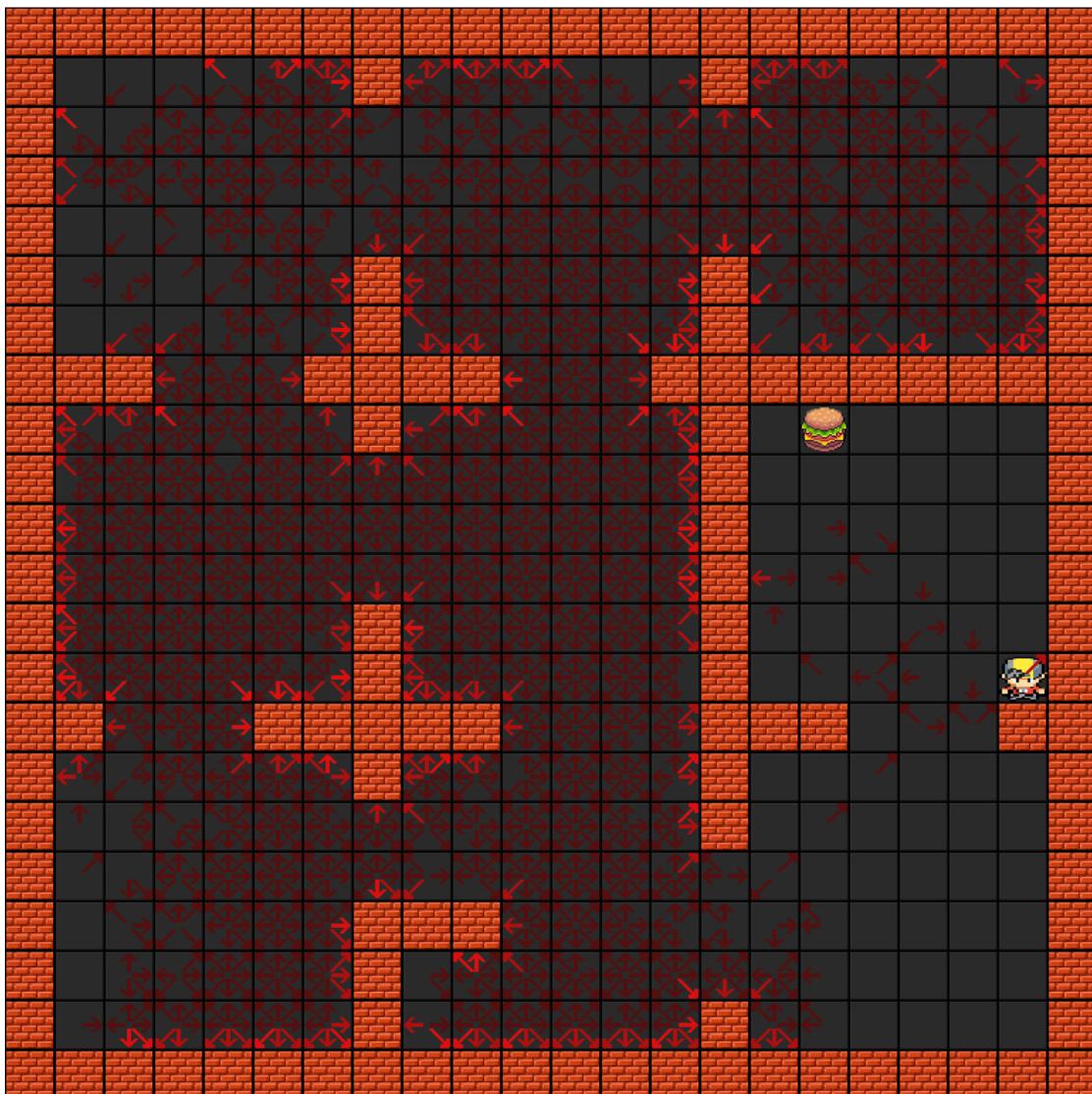


Figura 36 - resultado do Q-Learning a explorar antes de chegar ao objetivo

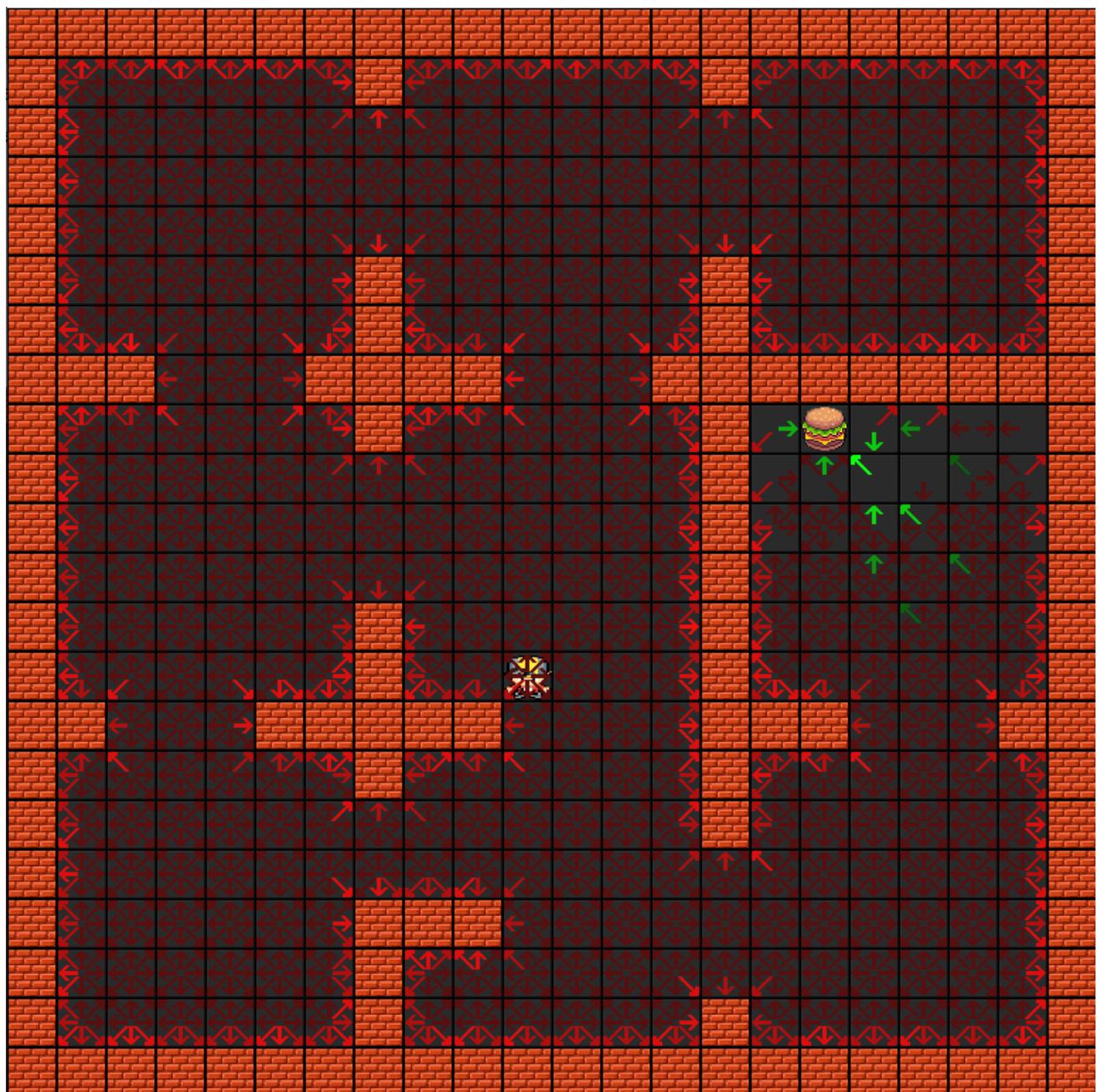


Figura 37 - resultado após o agente ter encontrado 11 vezes o objetivo

Perante o comportamento do agente e pela política representa através de setas com cor vermelho e verde é possível denotar que de facto o agente está a melhorar o seu conhecimento. As setas de cor vermelho-claro, representam descontos muito grandes no reforço que é nomeadamente quando o agente colide com um obstáculo. As setas verdes representam o reforço positivo por ter chegado ao objetivo, como se observa existem 11 setas verdes que corresponde à propagação de valor positivo do algoritmo Q-Learning. Conforme o agente chega mais vezes ao objetivo ele melhora o caminho realizado. Desta maneira acredita-se que de facto a implementação esteja correta.

A figura seguinte representa o resultado do algoritmo Q-Learning com memória episódica. O resultado foi o seguinte:

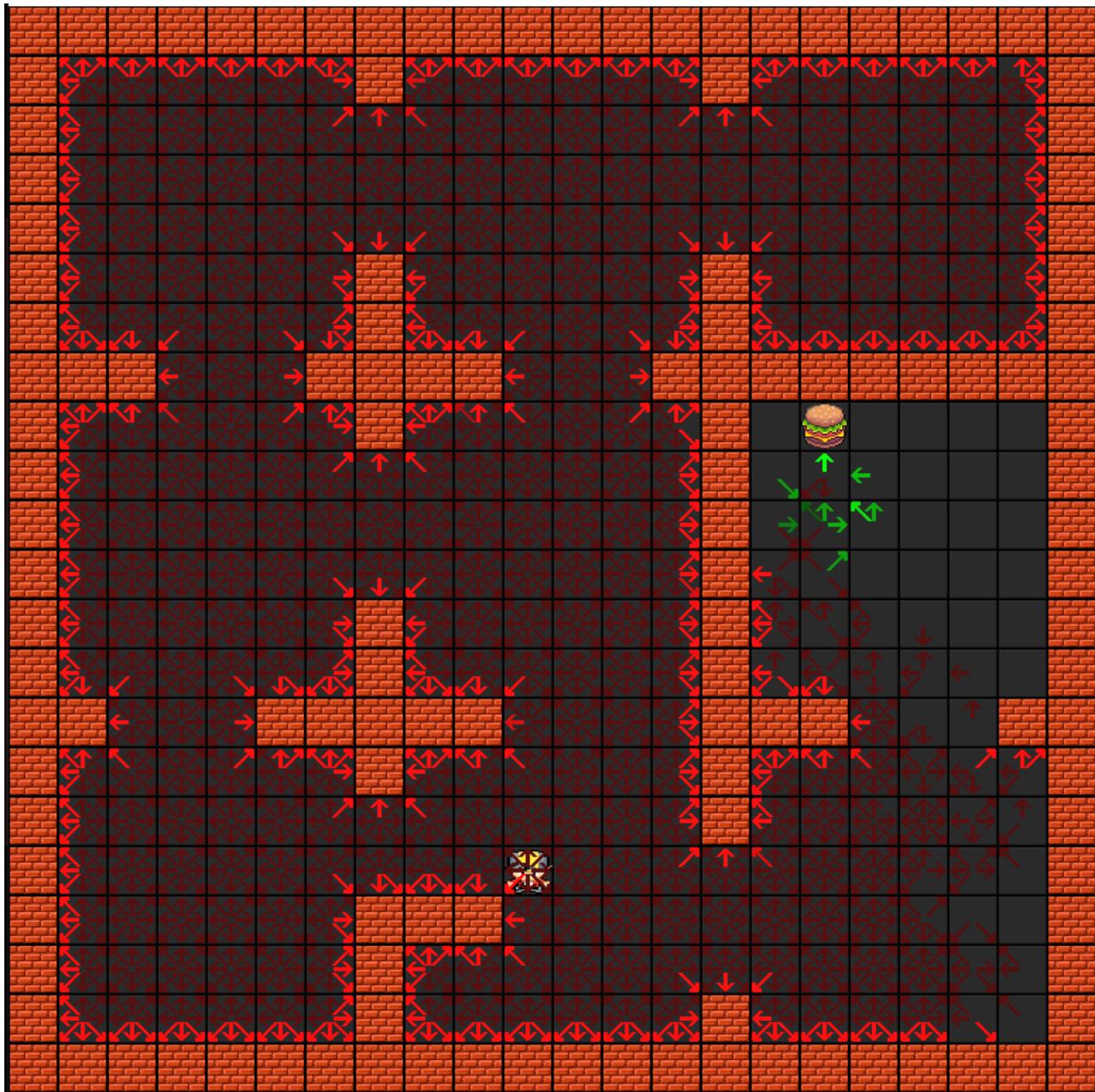


Figura 38 - resultado da procura com Q-Learning com memória episódica após o agente encontrar 1 vez o objetivo

O resultado obtido foi respetivo aquando do agente encontrou uma vez o objetivo e possui o número de simulações igual a 5. Este resultado parece correto, pois os cinco estados antes do objetivo passaram todos a ter um reforço positivo após o agente ter conseguido chegar ao objetivo.

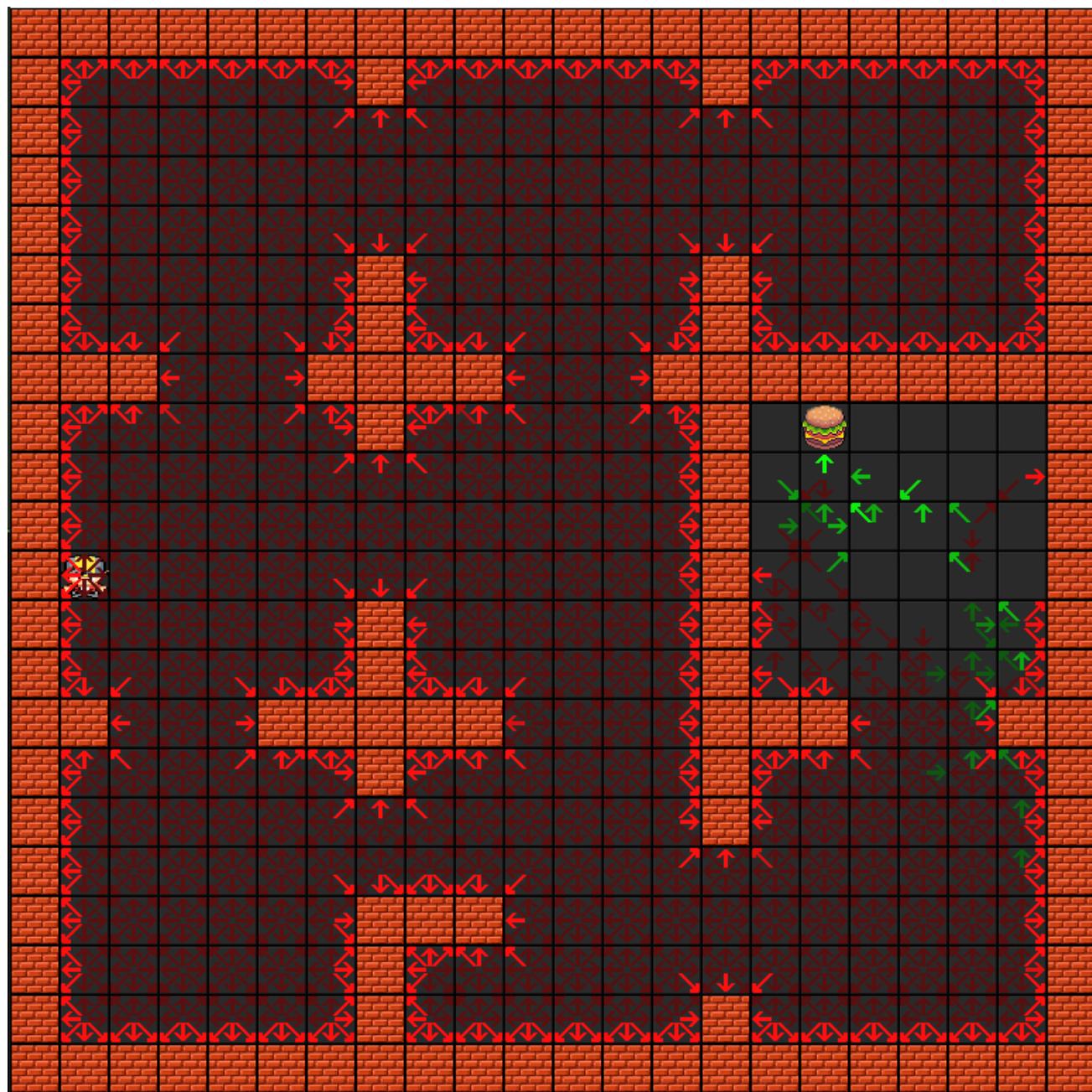


Figura 39 - resultado da procura com Q-Learning com memória episódica após o agente encontrar 5 vezes o objetivo

As figuras seguintes são respetivas ao algoritmo Dyna-Q. Os resultados obtidos foram os seguintes:

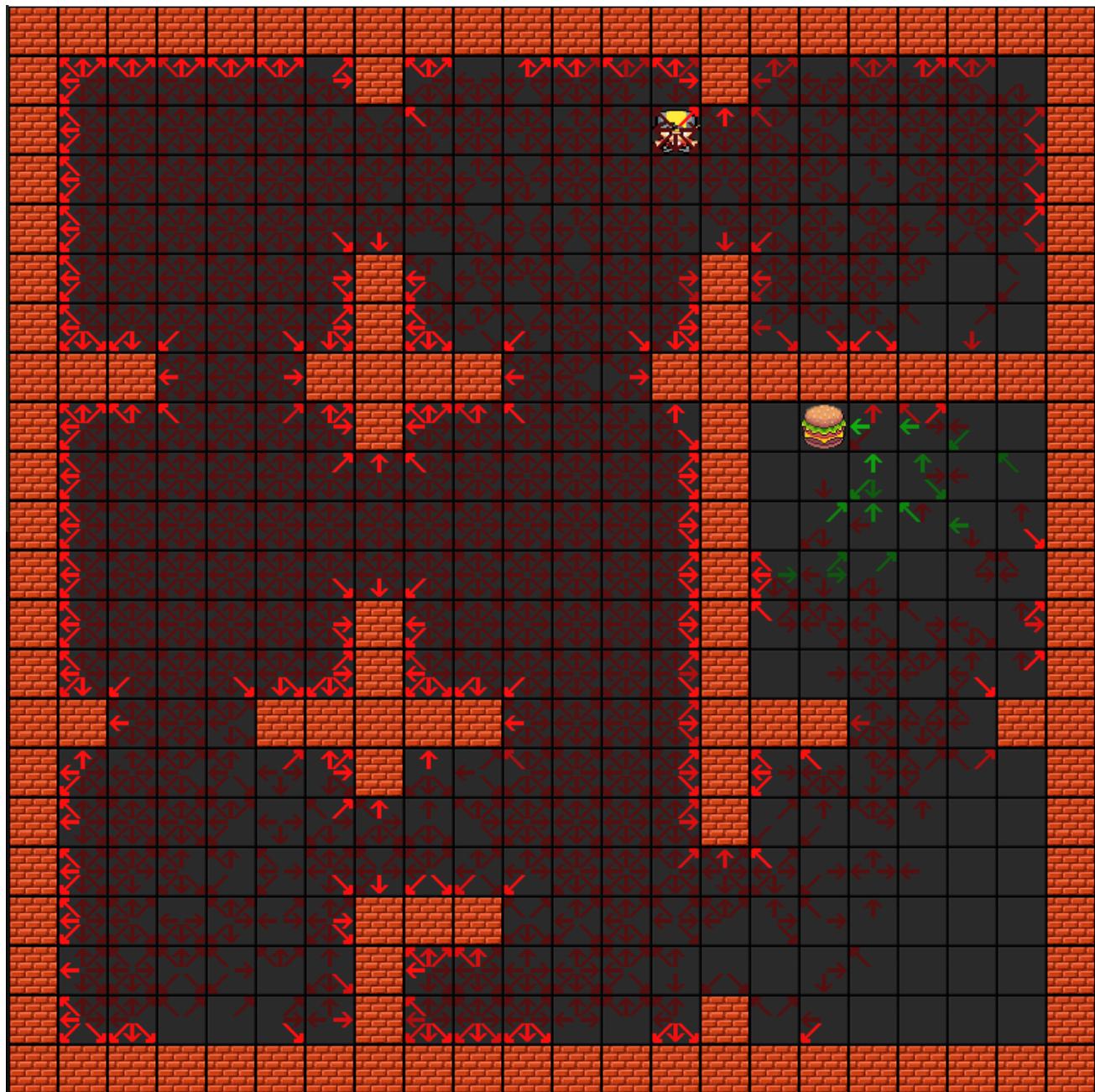


Figura 40 – resultado da procura Dyna-Q após o agente encontrar 1 vez o objetivo

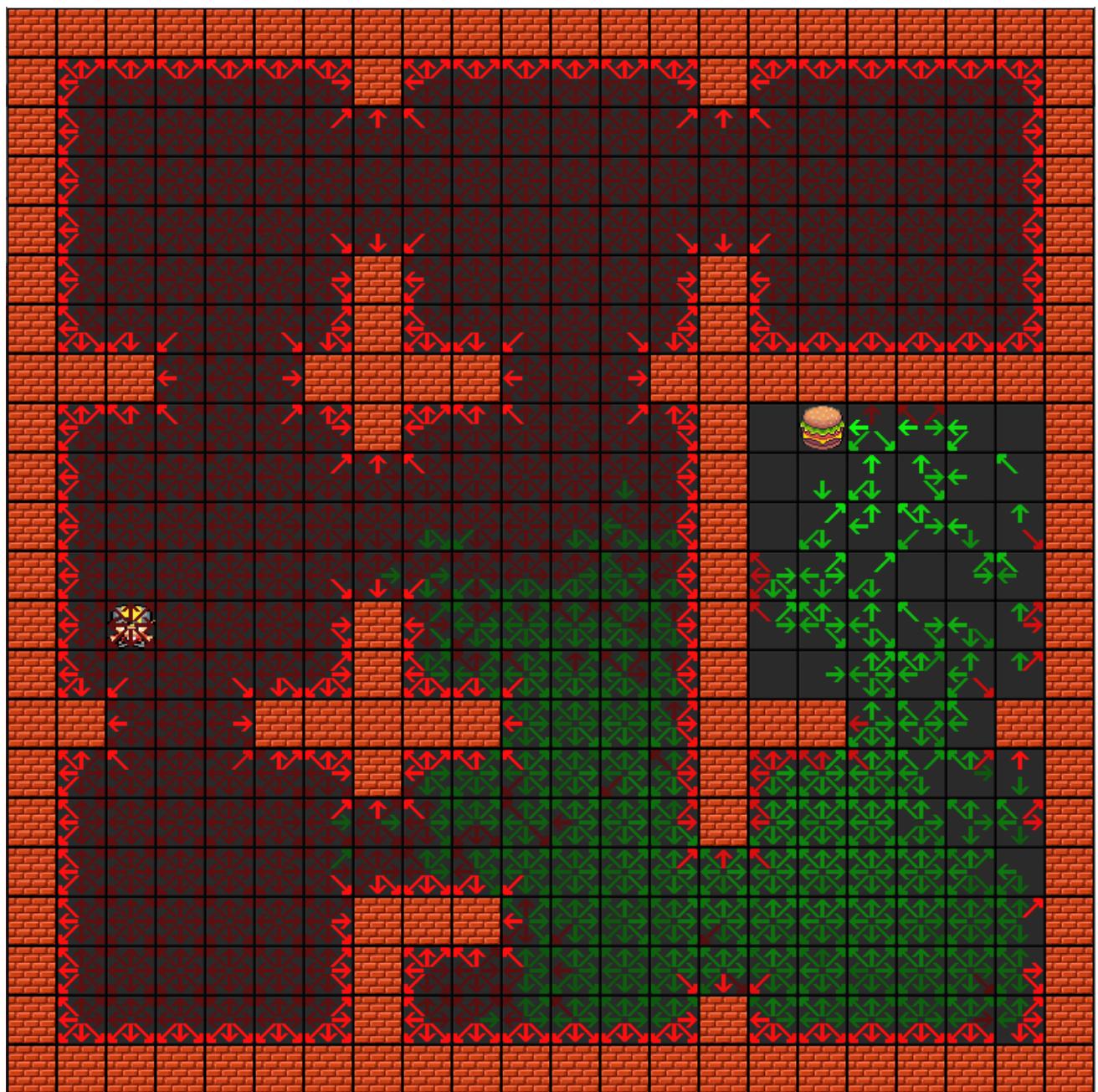


Figura 41 - resultado da procura Dyna-Q após o agente encontrar múltiplas vezes o objetivo

Perante o resultado observado na figura quando o agente encontra múltiplas vezes o objetivo é possível denotar uma grande melhoria comparativamente aquando só encontra uma vez o objetivo. Sempre que o agente encontra uma vez o reforço dos estados que deram origem à solução começam sucessivamente a passar a verde. Desta forma também admite-se que este algoritmo foi conseguido implementar com sucesso.

3. Procura em espaços de estados

3.1 Componente teórica

A procura em espaços de estados são problemas modelados como espaços de estados, isto é, um conjunto de estados em que o problema pode estar. O conjunto de estados forma uma gráfico e dois estados estão ligados se existir uma operação que possa ser aplicada para transformar o primeiro estado no segundo. A procura em espaços de estados, ao contrário dos métodos tradicionais de informática, não guardados todos os estados em memória. Não é possível guardar todos os estados, porque a maioria dos problemas é demasiado complexo e com demasiados estados para serem todos guardados em memória. Em vez disso, os nós são gerados à medida que são explorados e tipicamente descartados posteriormente. Uma solução consiste num caminho de estado inicial até um estado objetivo. Desta forma surge o conceito de raciocínio automático.

Na informática o raciocínio automático é a capacidade de um sistema computacional resolver um problema automaticamente tendo como base conhecimento sobre o respetivo domínio do problema. No final é produzido uma solução a partir das diversas alternativas possíveis. O processo cognitivo pode ser: representação de conhecimento, exploração de alternativas e controlo de processamento.

A resolução automática de problemas pode ser de quatro tipos. O primeiro é do tipo decisão sequencial, que representa a capacidade de saber o que fazer e o resultado são políticas de ação. O segundo é o planeamento que representa a capacidade de saber como fazer e os resultados são planos. O terceiro é a otimização que é a capacidade de saber qual é a melhor configuração de parâmetros perante um conjunto de restrições e o resultado é uma configuração de parâmetros. Este tipo de resolução será explorado mais para a frente. Por fim é a classificação que é saber o que é. Este tipo é referente à aprendizagem automática e o resultado são conceitos e relações.

Os elementos base da procura em espaços de estados são: estado, transição, problema e solução. O estado define uma situação, a transição define uma transformação no estado para outro estado. No problema consta o estado inicial, o estado final, os operadores e a função de avaliação do custo. Por fim a solução representa o percurso no espaço de estados. Os aspectos a ter em conta na escolha de um método de procura são:

- Completo: método de procura que garante que caso exista uma solução, está será encontrada.
- Ótimo: método de procura que garante que existindo várias soluções, a solução encontrada é sempre a melhor.
- Complexidade temporal: tempo necessário para encontrar uma solução.
- Complexidade espacial: memória necessária para encontrar uma solução.

3.1.1 Procura não informada

Para realizar a procura é necessária uma metodologia para procurar pelo conjunto de estados. Desta forma surgiu primeiramente os métodos de procura não informada. Esta estratégia não tira proveito do conhecimento do domínio do problema para ordenar os estados por mais ou menos promissor. Neste tipo de procura estão inseridas as seguintes procuras:

- Procura em profundidade: procura que tem como critério de exploração o nó com maior profundidade. Quer isto dizer que ele explora o primeiro nó gerado. Existem variantes a esta procura como a procura em profundidade limitada e iterativa. A procura iterativa define uma profundidade e não procura além do limiar definido, passando ao primeiro nível de exploração e explorando o segundo nó do primeiro nível. A procura limitada não passa do limiar e termina caso chegue.
- Procura em largura: procura que utilizada como critério a menor profundidade. Desta forma ela procura no primeiro nó e volta ao nível anterior explorar o segundo nó e assim sucessivamente.
- Procura de custo uniforme: procura que possui como critério de exploração o custo da solução. Esta procura explora sempre primeiro os nós com custo menor. Para ser

definido esta procura é necessário definir previamente ou definir uma função que atribua os custos aos estados.

Contudo nestas procuras existem problemas nomeadamente os estados repetidos na árvore de procura. Isto pode ser devido a múltiplas transições que resultam no mesmo estado. Isto pode resultar em expandir estados já anteriormente analisados, havendo desperdício de tempo e memória. No pior dos casos pode resultar em o grafo do espaço de estados criar ciclos. Desta maneira foram criadas duas memórias de nós processados. A primeira memória é designada abertos, que representa os nós gerados, mas não expandidos. Desta forma não é adicionado novamente os estados que já foram gerados em memória. A segunda memória é designada fechados, que representa os nós já explorados. Esta memória difere da primeira na medida que qualquer nó nesta memória não será explorado novamente, enquanto na primeira memória os nós ainda podem explorados.

3.1.2 Procura informada

A procura informada é uma procura que tira partido de uma avaliação do estado. Esta avaliação é feita a partir do conhecimento do domínio do problema. Para isso utilizada uma função para avaliação de cada nó gerado e avalia o seu custo e utilidade. A avaliação é feita por quanto menor for o valor do custo da função mais promissor é o nó. Nesta procura a fronteira de exploração é ordenada por ordem crescente da função de avaliação. Neste contexto surge a função heurística.

A função heurística representa uma estimativa do custo do percurso desde o nó n até ao nó objetivo. Esta função reflete conhecimento acerca do domínio do problema e é independente do percurso, depende apenas do estado a que está associado e ao objetivo.

Desta forma as variantes de procura principais são:

- Procura custo uniforme: quando a função de avaliação é somente o custo do nó corrente.
- Procura sôfrega: utilização somente da função heurística para avaliação do custo. Não tem em conta o custo do percurso explorado e produz soluções sub-ótimas, mas muito expeditas.
- Procura A*: utilização do custo do percurso explorado e da função heurística. Desta maneira existe a minimização do custo global e produz soluções ótimas.

A procura A* é método de procura de eficiência ótima. Não existe nenhum outro algoritmo que expandirá menos nós sendo completo e ótimo. Contudo não resolve o problema da complexidade combinatória, onde o número de nós expandidos dentro do contorno do nó objetivo ainda é uma função exponencial da dimensão do percurso até ao objetivo. A função heurística diminui o contorno de procura, mas não o suficiente.

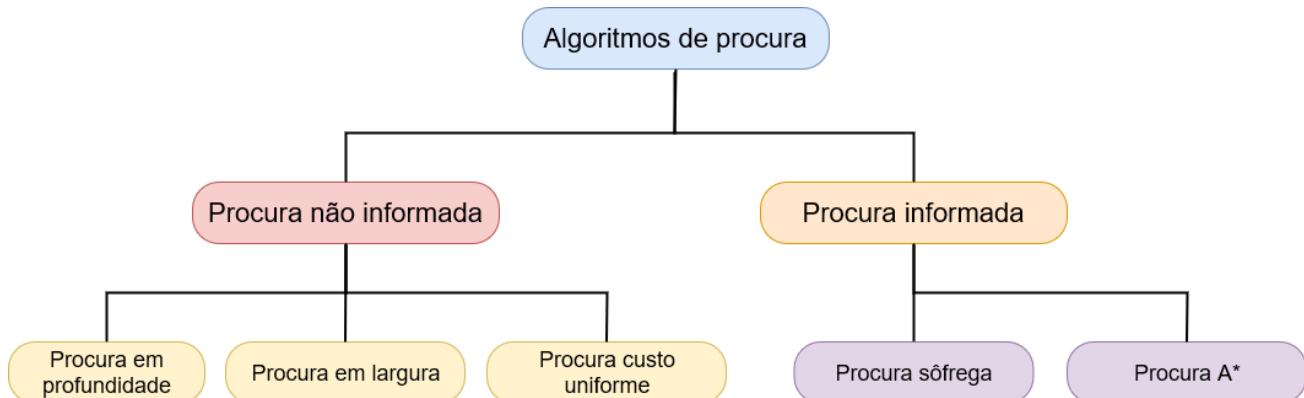


Figura 42 - Algoritmos de procura

3.1.3 Racionalidade limitada

Um sistema é racional caso consiga realizar a ação certa dado o conhecimento que possui. Desta forma é a capacidade de conseguir agir e obter o melhor resultado possível perante os objetivos que se pretende atingir. Contudo existe um preço que é a complexidade computacional. Desta forma surgiu o conceito de rationalidade limitada que tem as características de informação incompleta e recursos limitados. O resultado são soluções

satisfatórias de acordo com os níveis de referência sub-ótimos e o processo de decisão pára quando o nível de referência é satisfeito.

Neste contexto surge o algoritmo procura A* ponderada. Este algoritmo é do tipo ε -ótimo. Este algoritmo é dado pela fórmula:

$$f(\mu) = g(\mu) + (1 + \varepsilon) \cdot h(\mu)$$

Desta forma o valor que muda na equação é o valor de ε . Sempre que este valor é 0 então é procura A* e caso o valor seja 1 ele utiliza 2 vezes mais a heurística do que o valor do custo do nó. Existem algumas alternativas a esta equação como alterar também o valor do custo do nó com o valor de ε , contudo essas equações não respeitam a seguinte inequação:

$$f(u) \leq (1 + \varepsilon) \cdot \delta(s, T)$$

3.1.4 Algoritmo Wavefront

Uma abordagem diferente é o algoritmo Wavefront. Este algoritmo insere-se na procura informada. Será dado mais enfase a este algoritmo pois será implementado. Este algoritmo utiliza uma abordagem baseada em campos de valor. Os campos de valor são uma simulação do modelo do mundo com base em representações internas. Os campos de valor possuem um único máximo global. Este algoritmo pode utilizar duas formas de planeamento do modelo de navegação que são: procura local feita através de Hill-Climbing e estratégia global onde utiliza uma política comportamental. Para a implementação deste algoritmo será utilizada a estratégia global. Este algoritmo, contudo, possui alguns problemas. Estes problemas são nomeadamente a utilização extensiva de memória, necessários operadores bidirecionais e é aplicável apenas a regiões limitadas do espaço de estados.

3.1.5 Otimização

A otimização consiste na seleção da melhor opção a partir de um conjunto de opções disponíveis de acordo com um critério de avaliação, podendo ele ser maximizar ou minimizar algum valor. Existem várias formas da avaliação do desempenho da otimização nomeadamente: por performance (ganho), por perda (perda) e por adequação. Desta maneira o resultado é uma configuração de parâmetros, que é um estado, de modo a maximizar uma função de valor ou de adequação.

Desta maneira surgiu o algoritmo Hill-Climbing um algoritmo de otimização que pertence à família das pesquisas locais. Este algoritmo começa com uma solução arbitrária para o problema, depois tenta encontrar uma solução melhor fazendo mudanças incrementais na solução. Caso seja encontrada uma solução melhor é substituída e vai procurando sucessivamente até não encontrar nenhuma solução melhor. Este algoritmo possui as características de não ser completo e não ser ótimo.

Do algoritmo Hill-Climbing surgiram algumas variantes nomeadamente o Hill-Climbing estocástico, Hill-Climbing estocástico com único sucessor e Hill-Climbing com reinício aleatório. Destes algoritmos todos só será implementado o Hill-Climbing estocástico, por isso só será detalhado este algoritmo. As propriedades diferentes do algoritmo Hill-Climbing clássico para o estocástico é que ele faz uma escolha aleatória entre sucessores que aumentam o valor de estado. Este algoritmo, contudo, possui geralmente uma convergência mais lenta que o Hill-Climbing e pode encontrar melhores soluções dependendo da topologia do espaço de estados. Este algoritmo também não é completo.

Um outro algoritmo de otimização é o Simulated Annealing. Este algoritmo possui uma adaptação dinâmica dos parâmetros do método de procura. Este método cria uma amostragem dos estados e possui um argumento que é a temperatura. A temperatura possui uma analogia feita à temperatura do metal, com o aquecimento surge mais flexibilidade e com o arrefecimento é criado restrição.

Os elementos principais da resolução de problemas de otimização são os que se segue:

- Estado: representa uma configuração de resolução do problema. Neste elemento é feito o encapsulamento da configuração de estado.
- Operador: representa uma transformação de estado. Aplicado a um estado resultando num estado sucessor ou vários estados sucessores. Deve produzir transformações locais de estado, refletindo características do domínio do problema.
- Problema: este guarda as funções de avaliação de estado, qual é o estado inicial e quais os operadores de transformação de estado. Quer isto dizer que representa informação de contexto do problema.
- Resultado: este elemento guarda o número de iterações para obter uma solução. Este elemento também define qual é a qualidade da solução.

3.2 Implementação

3.2.1 Implementação da procura espaço de estados

Nesta fase é pretendido implementar a biblioteca de procura em espaços de estados utilizando os algoritmos:

- Procura A* ponderada (Weighted A*)
- Planeamento automático com método frente-onda (Wavefront)

Da mesma maneira que a biblioteca da aprendizagem por reforço também é pretendido criar um agente capaz de navegar num espaço de dimensões discrete de forma a testar o funcionamento da biblioteca desenvolvida. O espaço é composto por obstáculos e um alvo. O objetivo é a biblioteca criar um caminho a partir da biblioteca desenvolvida para o agente utilizar. Para desenvolver a biblioteca de procura em espaços de estados também foi escolhida a linguagem de programação Python.

3.2.1.1 Implementação da procura A* ponderada

Inicialmente será explicada a biblioteca e posteriormente a implementação do algoritmo da procura A* ponderada e posteriormente o método frente-onda. Nesta fase também será explicado como foi implementado o sistema para a navegação do agente num ambiente discreto.

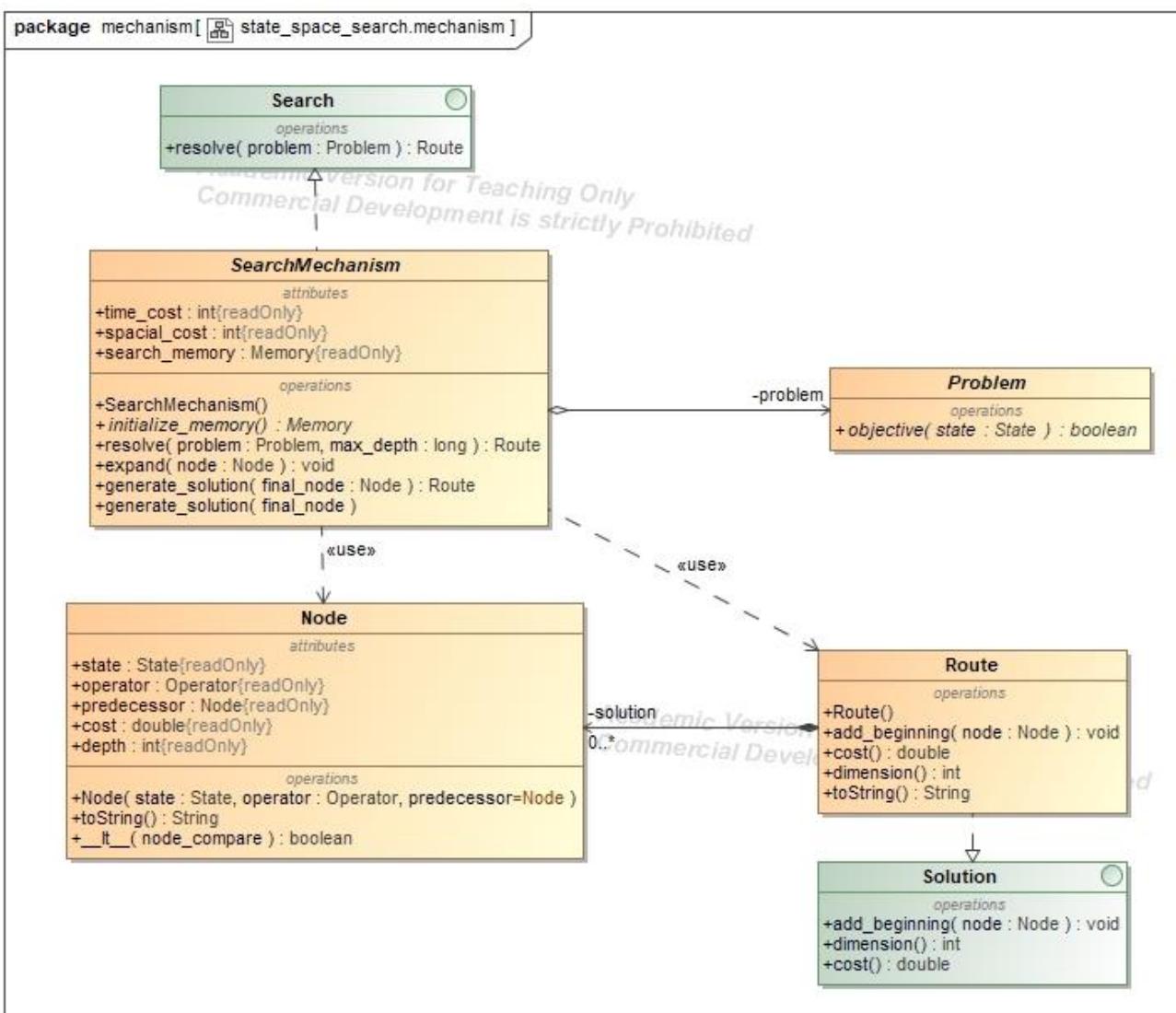


Figura 43 – diagrama de classes do mecanismo da procura em espaços de estados

Inicialmente é necessário existir uma classe que seja capaz de navegar nos nós até encontrar o nó correspondente à solução. Esta classe é designada **SearchMechanism**, esta classe é abstrata pois diversos algoritmos irão utilizá-la para resolver os problemas. O método abstrato neste caso é o **initialize_memory()**, pois os diferentes algoritmos utilizam memórias distintas. Esta classe implementa a interface **Search**, onde obriga a implementação do método **resolve(problem)**, que é o método responsável por realizar a procura no espaço de estados. Esta classe recebe um **Problem** que é responsável por ditar qual é o estado inicial, os operadores e o estado objetivo. A classe de procura utiliza a classe **Node** que representa um nó na árvore de procura que possui atributos úteis para a procura como o custo, profundidade entre outros. Ela também precisa do método **__lt__()**, que diz como é que instâncias da mesma classe podem ser comparadas, isto será utilizado posteriormente na memória de prioridade. Por

fim a classe de procura utiliza a classe **Route**, que simboliza o caminho feito desde o nó inicial até à solução. Isto é feito através da adição sucessiva do nó explorado no atributo **solution**. Por fim na classe de procura para obter o caminho desde o início até ao fim é utilizado o método **generate_solution(final_node)**, que retorna uma instância de **Route** que será posteriormente utilizada pelo agente para realizar o caminho. A classe **Route** implementa **Solution** que é a interface que dita como é que uma solução deve ser construída.

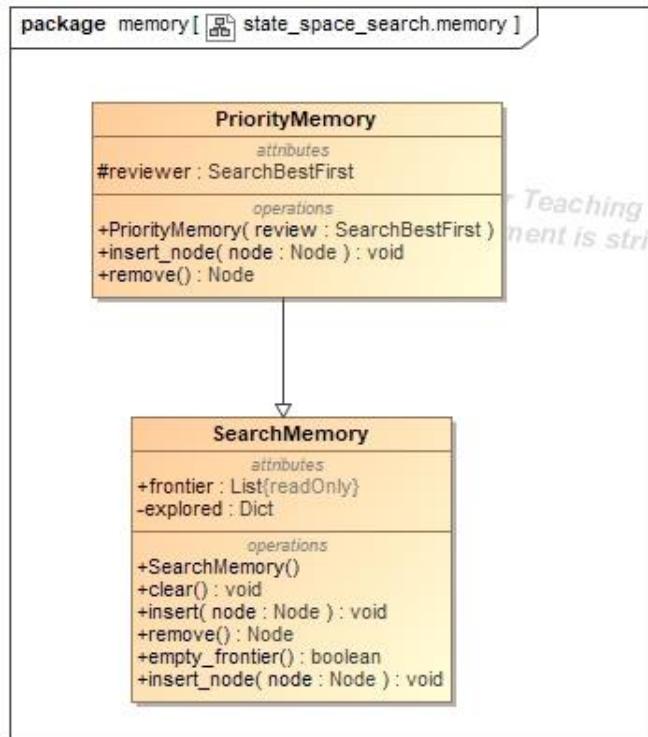


Figura 44 - memoria de procura com prioridade

Para ser possível guardar quais foram os nós já abertos e quais foram já explorados foi criada a memória de procura (**SearchMemory**). Contudo esta memória não ordena por prioridade, nomeadamente por custo inferior. Desta forma foi criada a classe **PriorityMemory**, utilizando o `heappush` e o `heappop` em Python cria uma lista ordenada por prioridade.

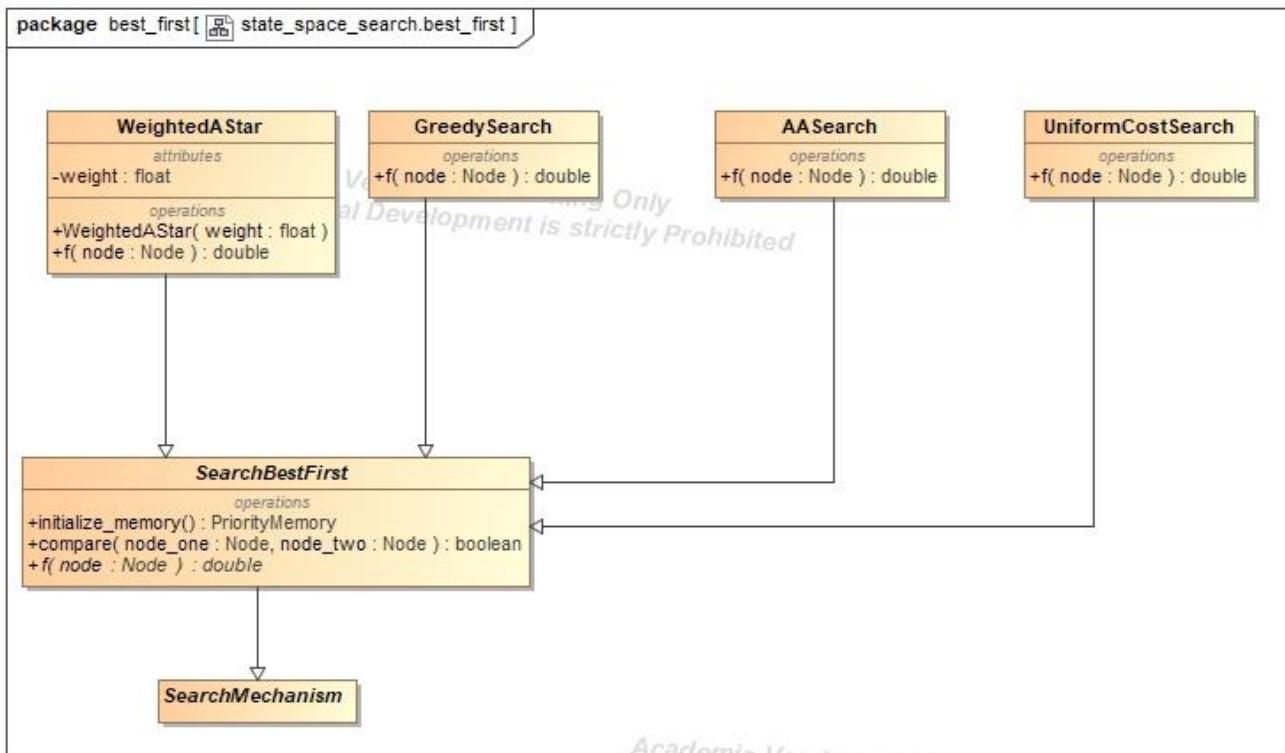


Figura 45 - algoritmos de melhor primeiro

A procura em espaços de estados utiliza algoritmos que determinam qual é o valor que deve ser associado a cada estado, esta função é designada por $f()$. Desta forma é possível guiar a procura de acordo com o critério do algoritmo. Desta maneira surge a classe **SearchBestFirst**, que estende de **SearchMechanism** e inicializa a memoria de prioridade com uma instância de **PriorityMemory**. A procura melhor primeiro utiliza sempre a memoria de prioridade, pois tem em conta o custo de cada nó gerado. Existem diversas classes que estendem de **SearchBestFirst**, contudo a única importante é a **WeightedAStar**. As outras classes foram utilizadas somente para testar. A classe da procura A* ponderada recebe um valor de peso como argumento que dita qual será o peso da heurística na avaliação da solução, quanto maior for esse valor maior será a importância da função heurística.

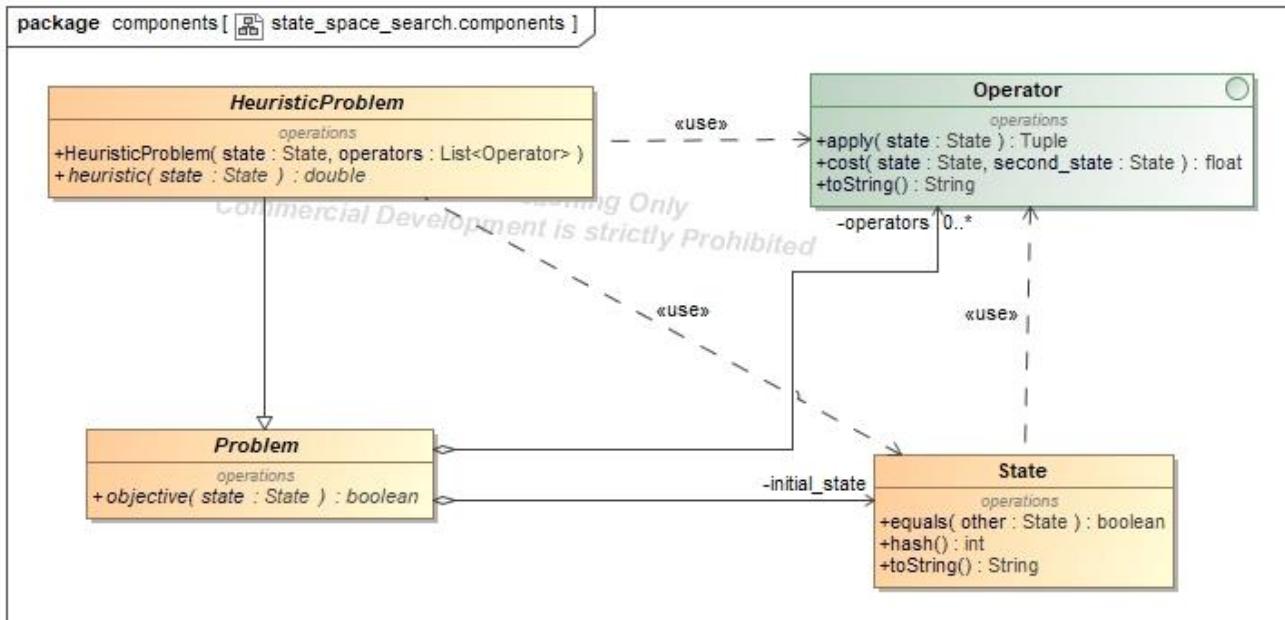


Figura 46 - componentes necessárias para resolver o problema

As classes apresentadas nos componentes, são as classes que é preciso definir quando é pretendido resolver um problema de procura em espaços de estados. As classes têm de implementar a interface `Operator` e estender da classe `Problem` ou `HeuristicProblem`, caso seja uma procura que utilize a função heurística.

3.2.1.2 Implementação das componentes de resolução de um problema de trajetórias

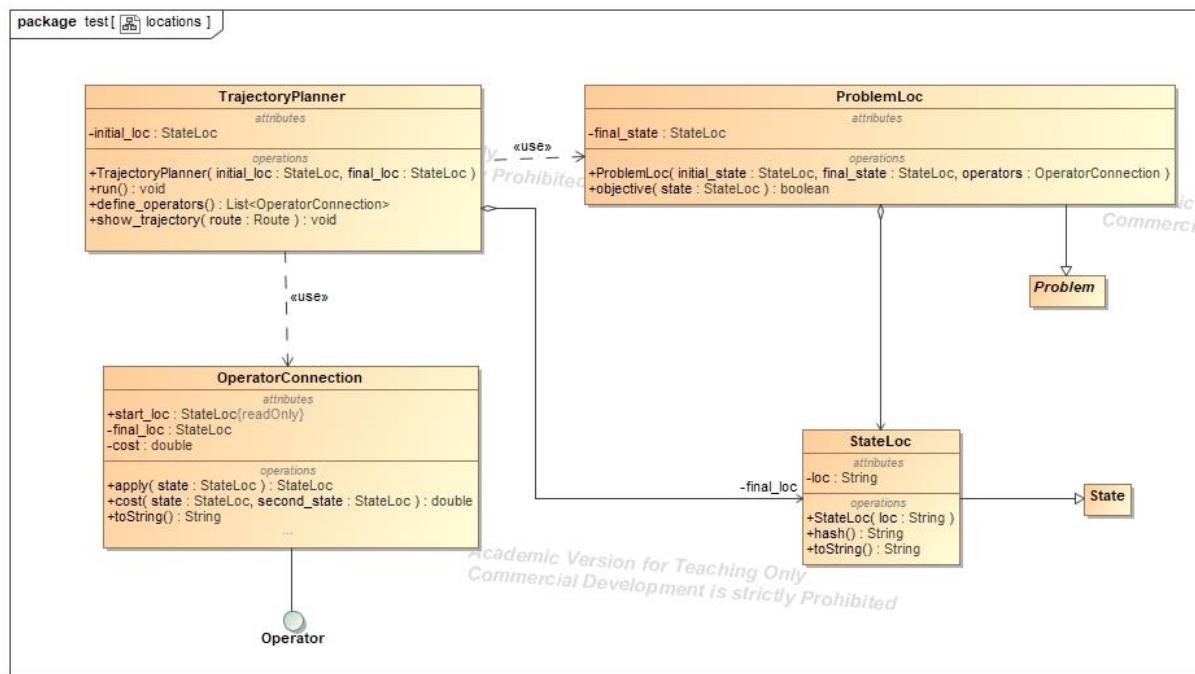


Figura 47 - exemplo de implementação das componentes

O diagrama de classes anterior foi utilizado somente para testar o funcionamento da biblioteca desenvolvida. Onde foi criado uma tabela com localizações e custos e o objetivo é ir de uma localização para o final obtendo o menor custo possível. Este esquema representa a necessidade de implementar as componentes referidas anteriormente.

3.2.1.3 Implementação do plano para o agente

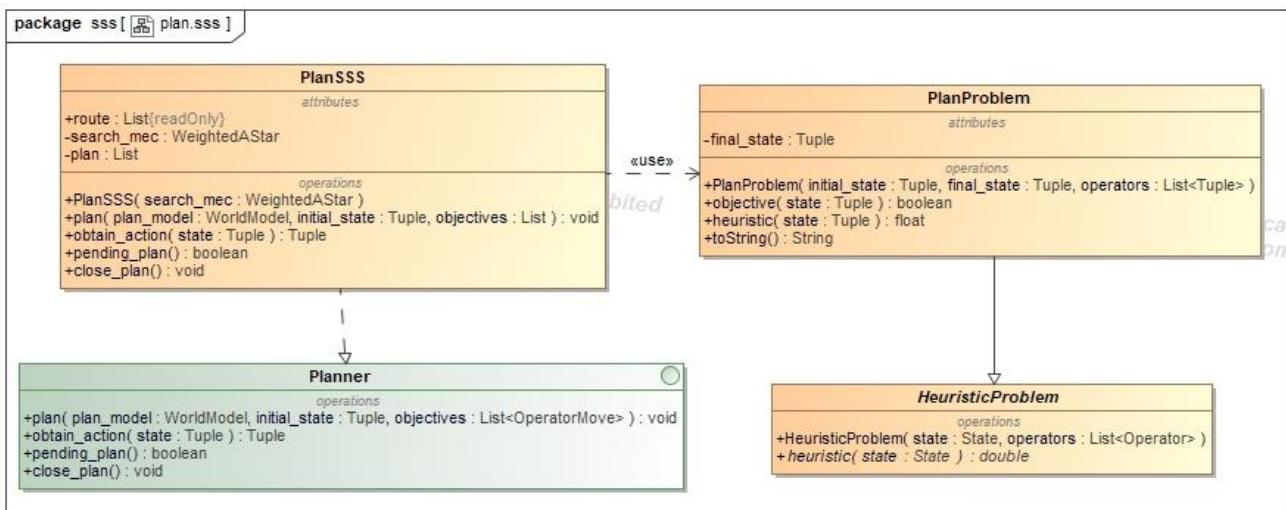


Figura 48 – diagrama de classes do plano da procura em espaço de estados

Para ser possível utilizar a biblioteca tendo por base os conceitos de operador, problema e estado é necessário implementar as classes que utilizem essas mesmas componentes. Por isso é necessário definir como é que estas funcionam. No diagrama anterior é onde será criado o plano que o agente utilizará para se mover no ambiente discreto. A classe `PlanProblem` é a classe que define o problema heurístico, desta maneira ela define como é calculada a heurística, que neste caso é a distância de Manhattan. A classe `PlanSSS` é a classe que irá construir o plano, ela implementa a interface `Planner` de forma a construir o código de forma modular. Desta forma todas as classes que constroem um plano têm de estender de `Planner`.

A classe `PlanSSS` recebe no construtor o mecanismo de procura, que neste caso é a procura A* ponderada. Definindo a classe `PlanProblem` dentro do método `plan()`, ela define o problema e envia este problema para o método de procura. Por sua vez o método de procura devolve o caminho que deve ser realizado. Este caminho depois é guardado para ser utilizado pela seguinte classe.

3.2.1.4 Implementação do controlo deliberativo e das ações do agente

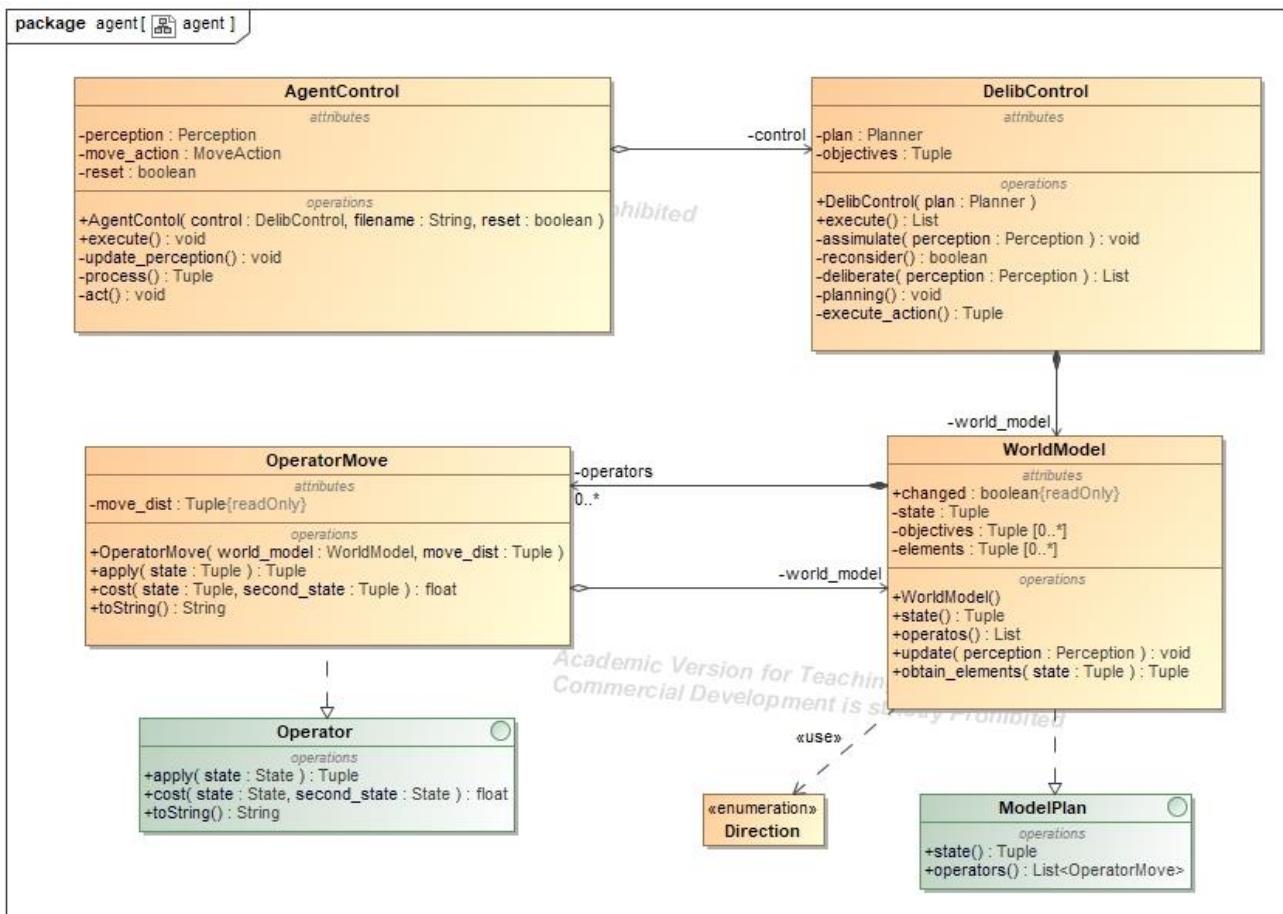


Figura 49 - diagrama de classes do controlo deliberativo

A classe responsável por conter o plano é a classe **DelibControl**, esta classe é responsável por controlar as ações de acordo com o ambiente e o plano obtido. Quer isto dizer que esta classe decide se deve ser utilizado o plano obtido ou se deve ser gerado um novo. Isto é feito pelo método **execute()**. Para que esta classe consiga saber o estado do mundo e determinar que ação deve tomar ela utiliza a classe **WorldModel**. Esta classe diz qual é o estado corrente, quais são os operadores e atualiza os estados de acordo com uma percepção. Esta classe implementa a interface **ModelPlan**, que diz como é que um modelo de planeamento deve ser construído. Por fim o modelo do mundo cria uma lista com todos os operadores de movimento (**OperatorMove**), que é a classe que representa o operador neste problema. Esta classe implementa o **Operator** e necessita de uma referência para o modelo do mundo para saber qual é o elemento a que corresponde um determinado estado, assim poder aplicar o operador. A última classe no diagrama é a **AgentControl**, esta classe é a classe que faz executar o

agente de acordo com o controlo deliberativo. Isto é feito no método `execute()`. Para isto ser possível é necessário a classe possuir as classes `Perception`, que é a responsável por obter toda a informação do mapa, e a classe `MoveAction`, que faz a afetação do movimento do agente no mapa. Esta classe primeiramente obtém a informação da percepção, depois envia essa informação para o controlo deliberativo e este retorna uma ação. Por fim é enviada a ação obtida para a ação de movimento para movimentar o agente no mapa.

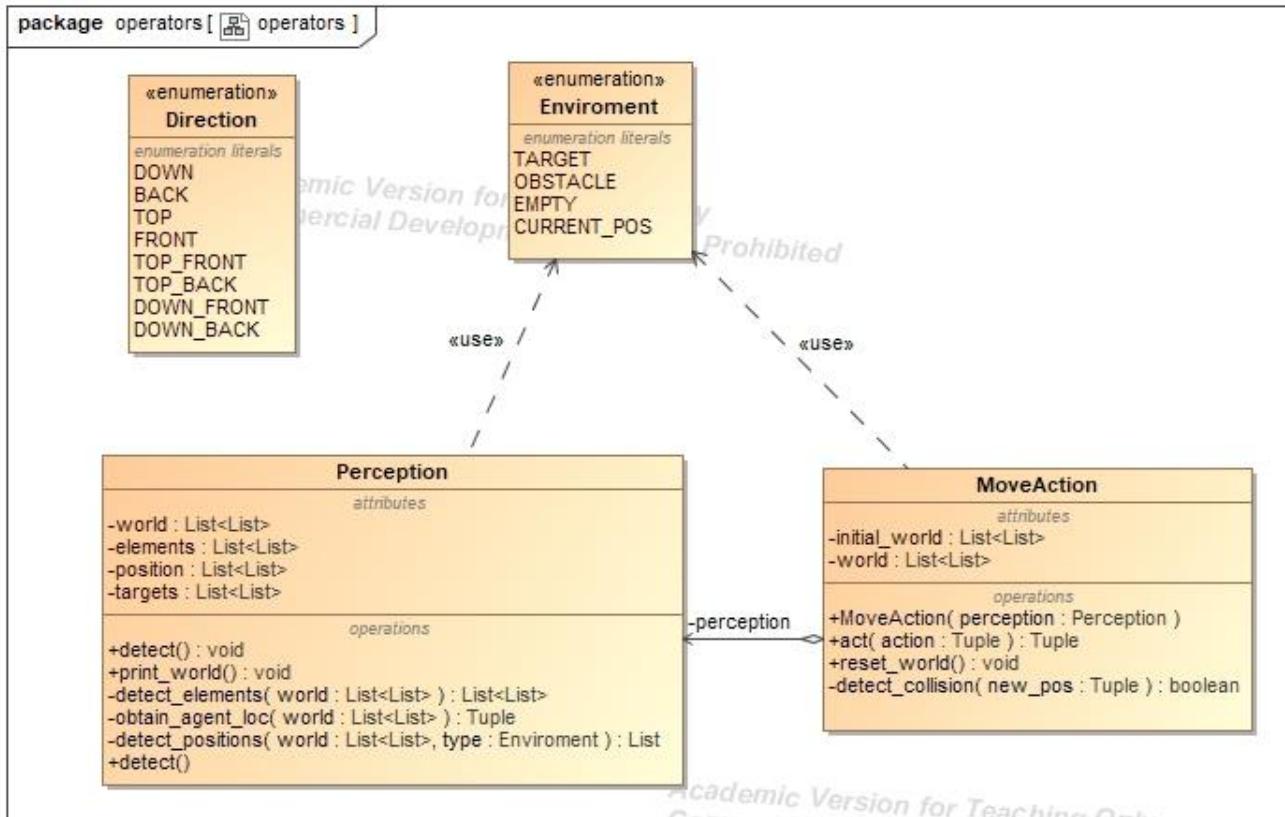


Figura 50 – diagrama de classes da deteção e ação do agente

Como foi referido anteriormente a classe `Perception` é a classe responsável por obter toda a informação do mundo. Isto é possível porque o mapa é representado pelo atributo `world`. Para que o `AgentControl` consiga obter os valores do mapa ele torna os atributos `elements`, `position` e `targets` disponíveis somente para leitura (propriedades). A classe `MoveAction` é a responsável por a receber uma ação e mover o agente no mapa de acordo com a ação, sendo isto feito no método `act()`. O enumerador `Enviroment` é igual à classe no reforço e define os diferentes valores do mapa e o que eles representam. Nomeadamente qual é o valor de um obstáculo, de uma posição vazia, do agente e do objetivo. Por fim o enumerador `Direction`

representa quais são as direções para os quais o agente se pode mover, neste caso é para toda a vizinhança de 8 da posição corrente.

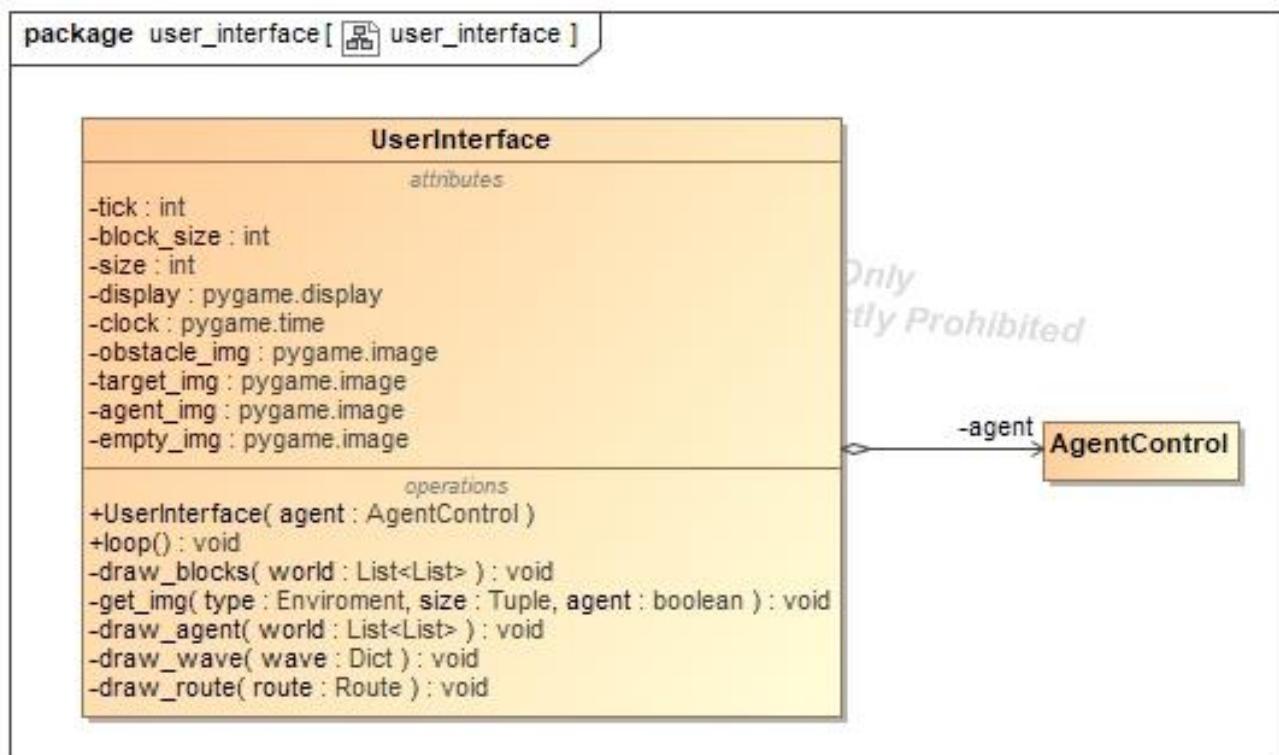


Figura 51 - classe da interface gráfica

Por fim para a visualização gráfica do comportamento do agente no mundo foi utilizada a biblioteca pygame. Para encapsular a parte gráfica foi criada a classe **UserInterface**, nesta classe é construída toda a apresentação desde a apresentação do mapa até ao caminho que será realizado. Para que esta classe consiga saber o estado do mapa e qual o caminho que deve desenhar ela utiliza uma referência da classe **AgentControl**.

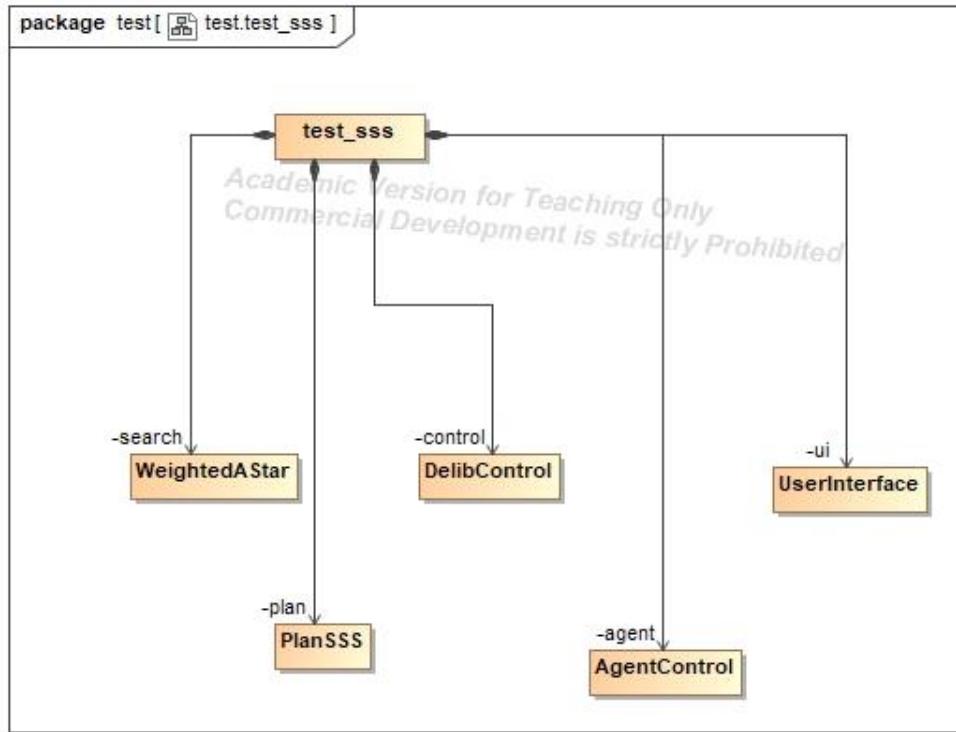


Figura 52 - ficheiro para correr o programa

Por fim para correr o programa é utilizado o ficheiro **test_sss**. Na mesma diretoria que este ficheiro está um ficheiro **batch**, que deve ser utilizado para correr o programa. Desta maneira é possível executar o programa, contudo este ficheiro possuí o caminho local do computador onde foi testado, para funcionar é necessário alterar o caminho no ficheiro **batch**. Um diagrama de sequência que mostra todo o funcionamento do sistema é o seguinte:

3.2.1.5 Diagramas de sequência e arquitetura geral da solução da procura A* ponderada

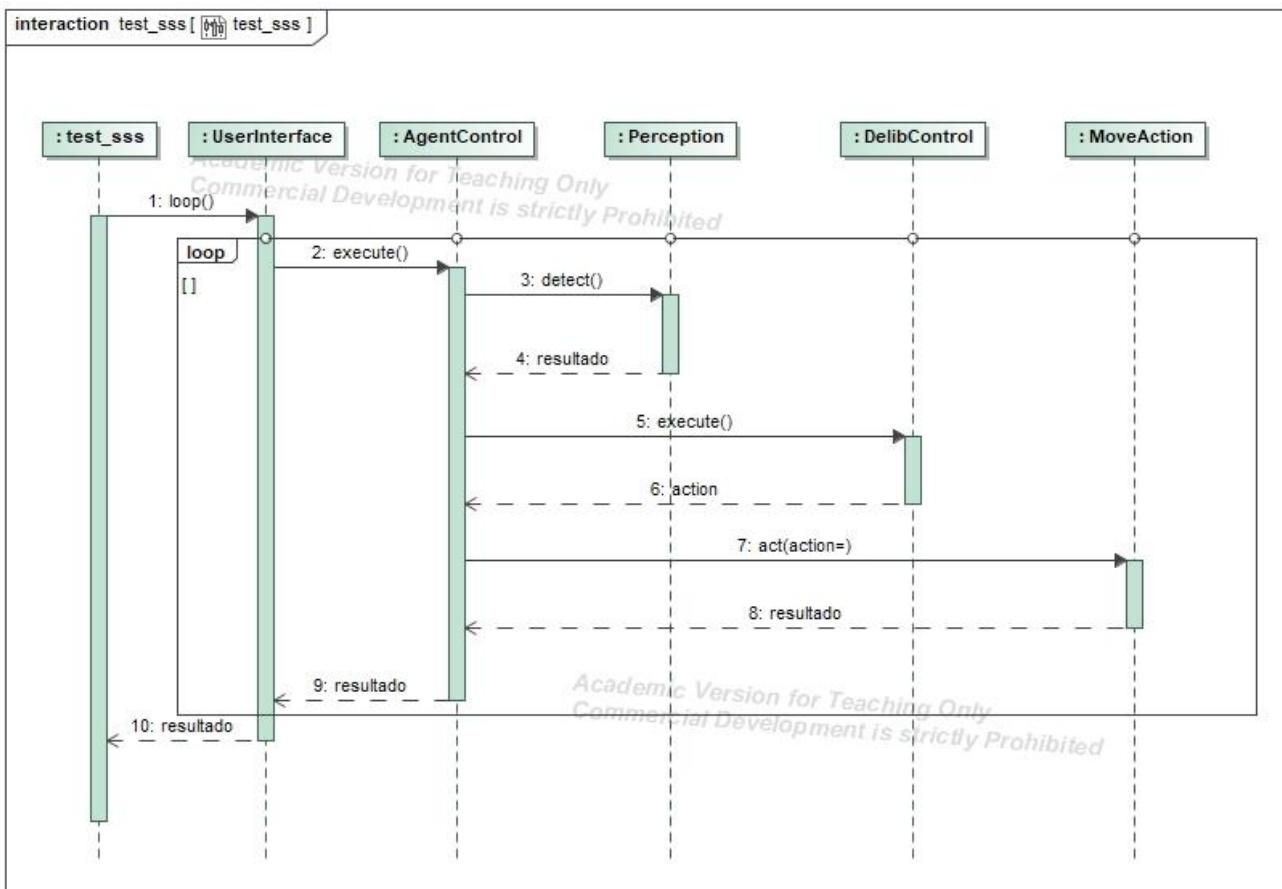


Figura 53 - diagrama de sequência da execução do programa

O diagrama da página seguinte representa o que acontece dentro do método `execute()` do `DelibControl`.

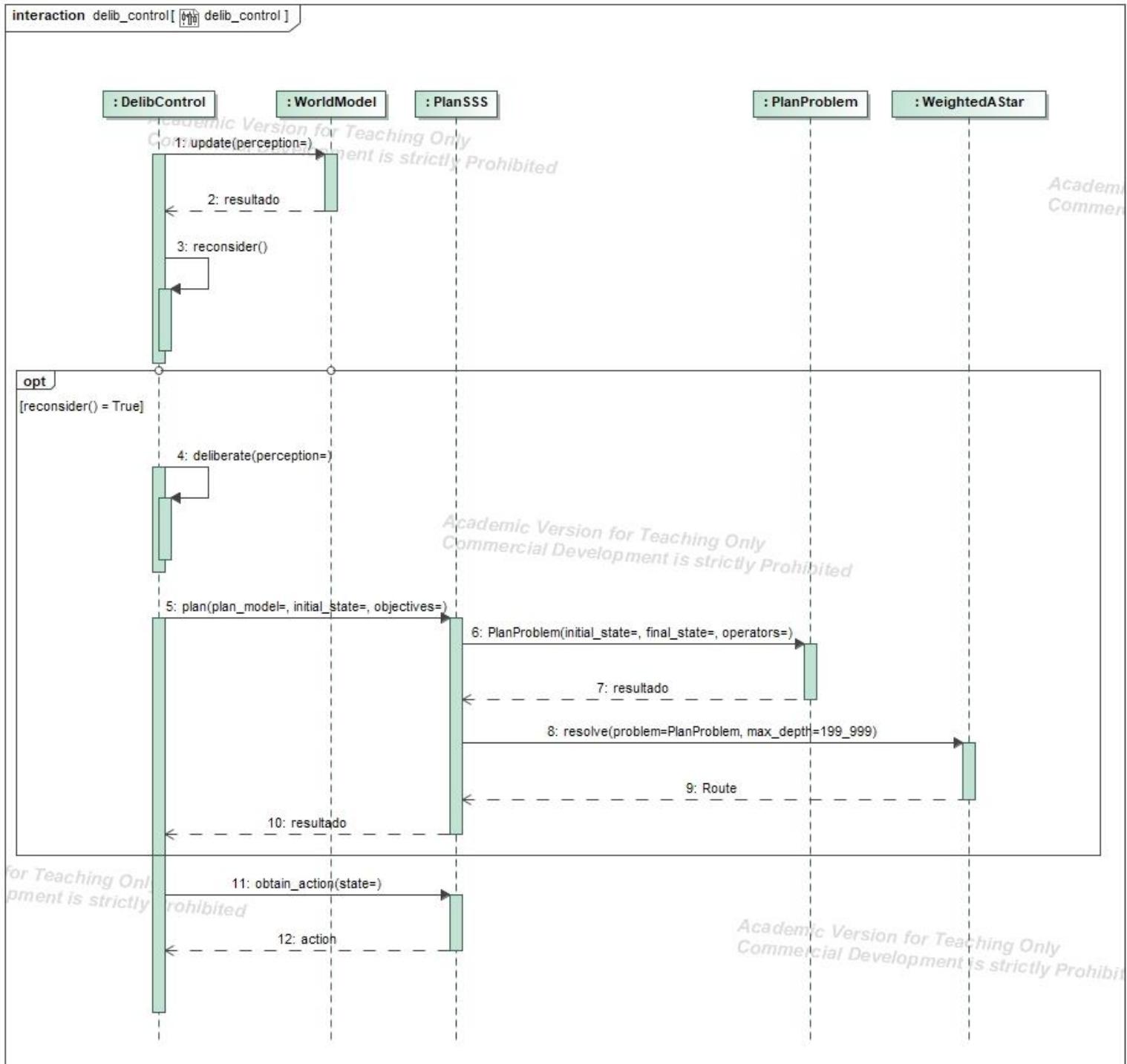


Figura 54 - método execute() do DelibControl

No diagrama da página seguinte mostra o que acontece dentro do método `resolve()` da classe `WeightedAStar`.

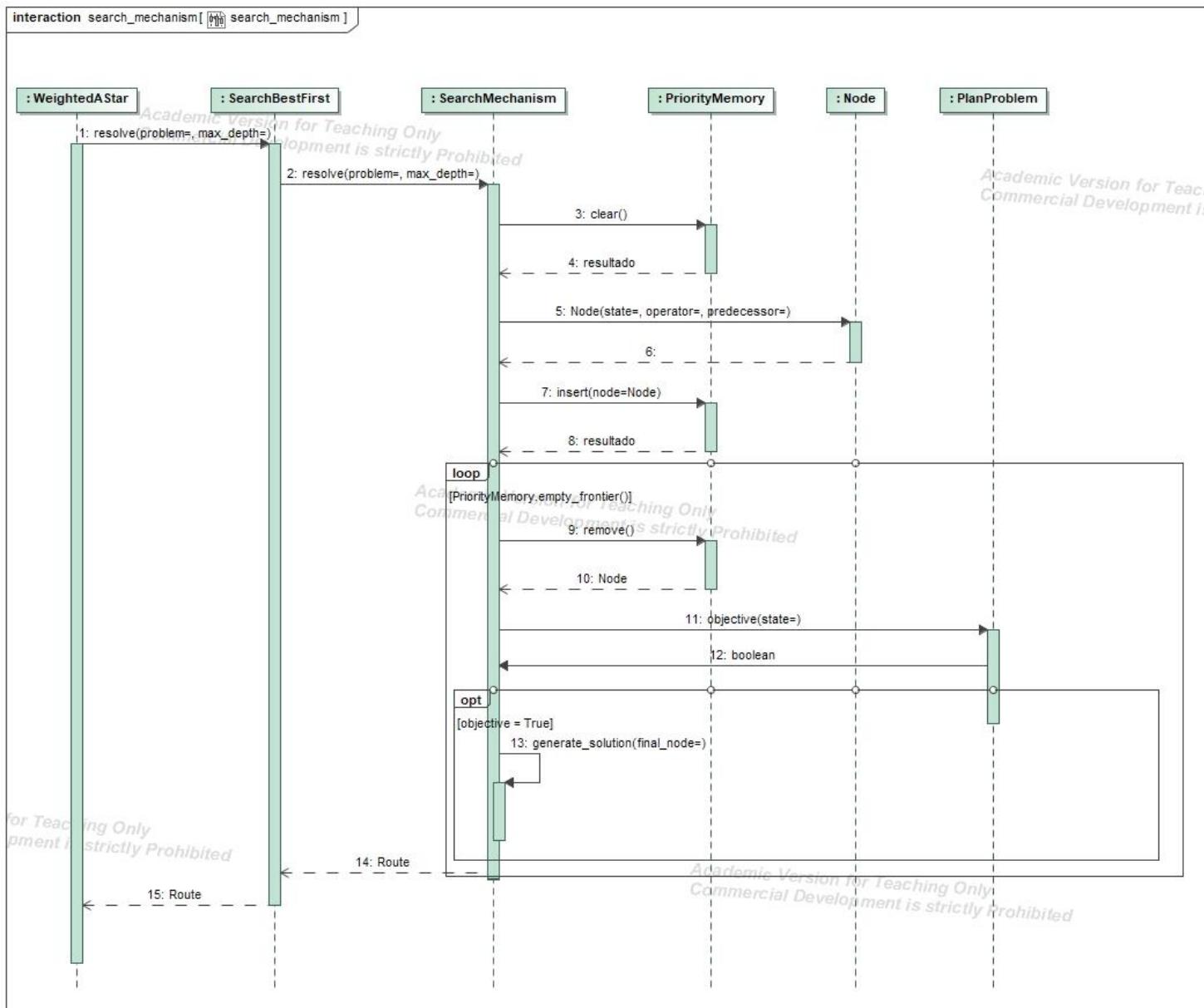


Figura 55 - método resolve() da WeightedAStar

3.2.1.6 Resultados obtidos

Os testes realizados para a procura A* ponderada serão feitos somente no ambiente 2. Nestes será testado variar o valor do peso no algoritmo para observar os diferentes resultados. Também serão testados os resultados tendo mais do que um objetivo no ambiente.

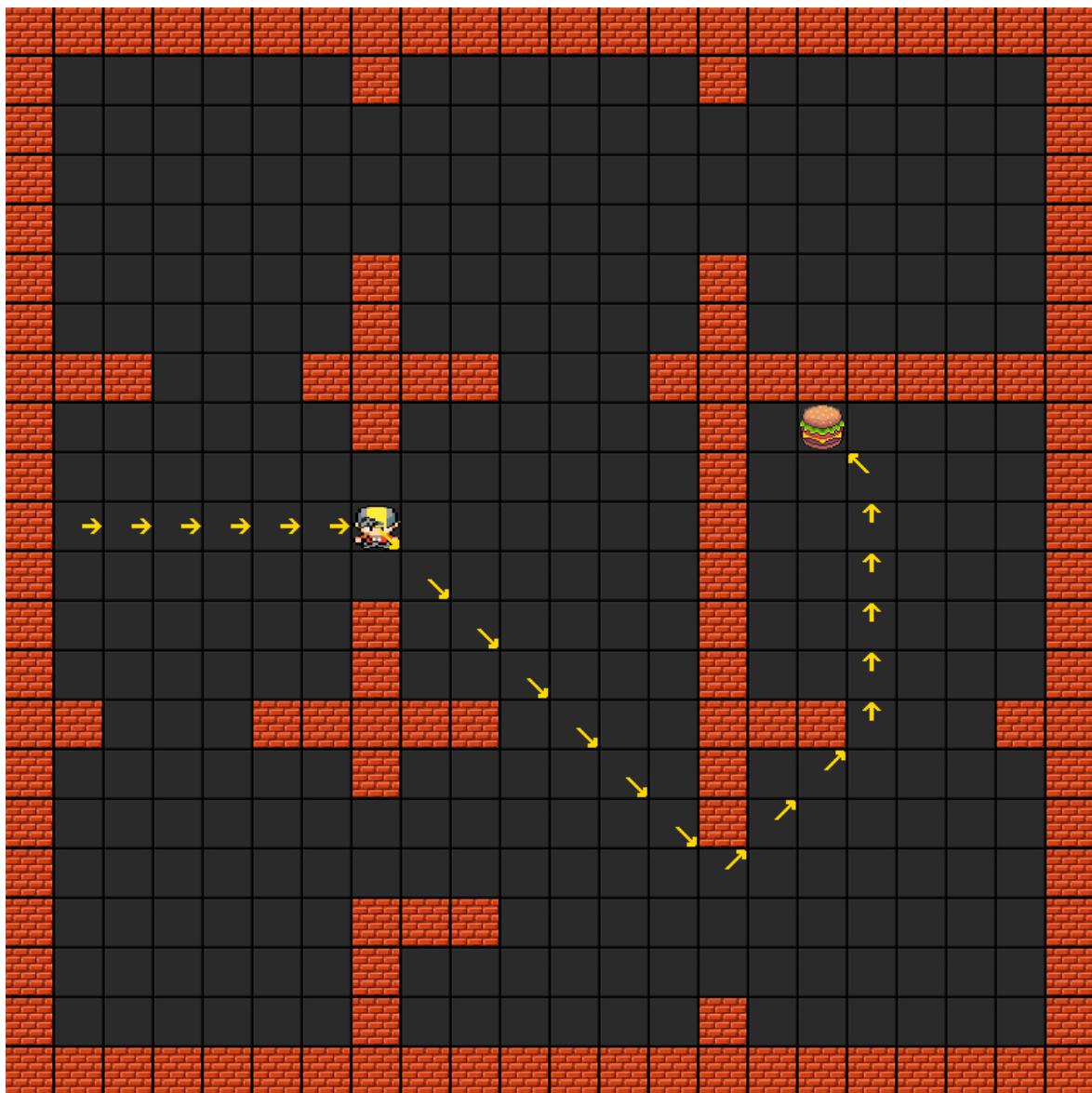


Figura 56 - teste para a procura A* ponderada com peso igual a 0

A procura A* ponderada com o peso a 0 fica a procura A* clássica, sendo uma procura ótima e completa. Este comportamento observa-se de facto, por isso acredita-se que esteja correto.

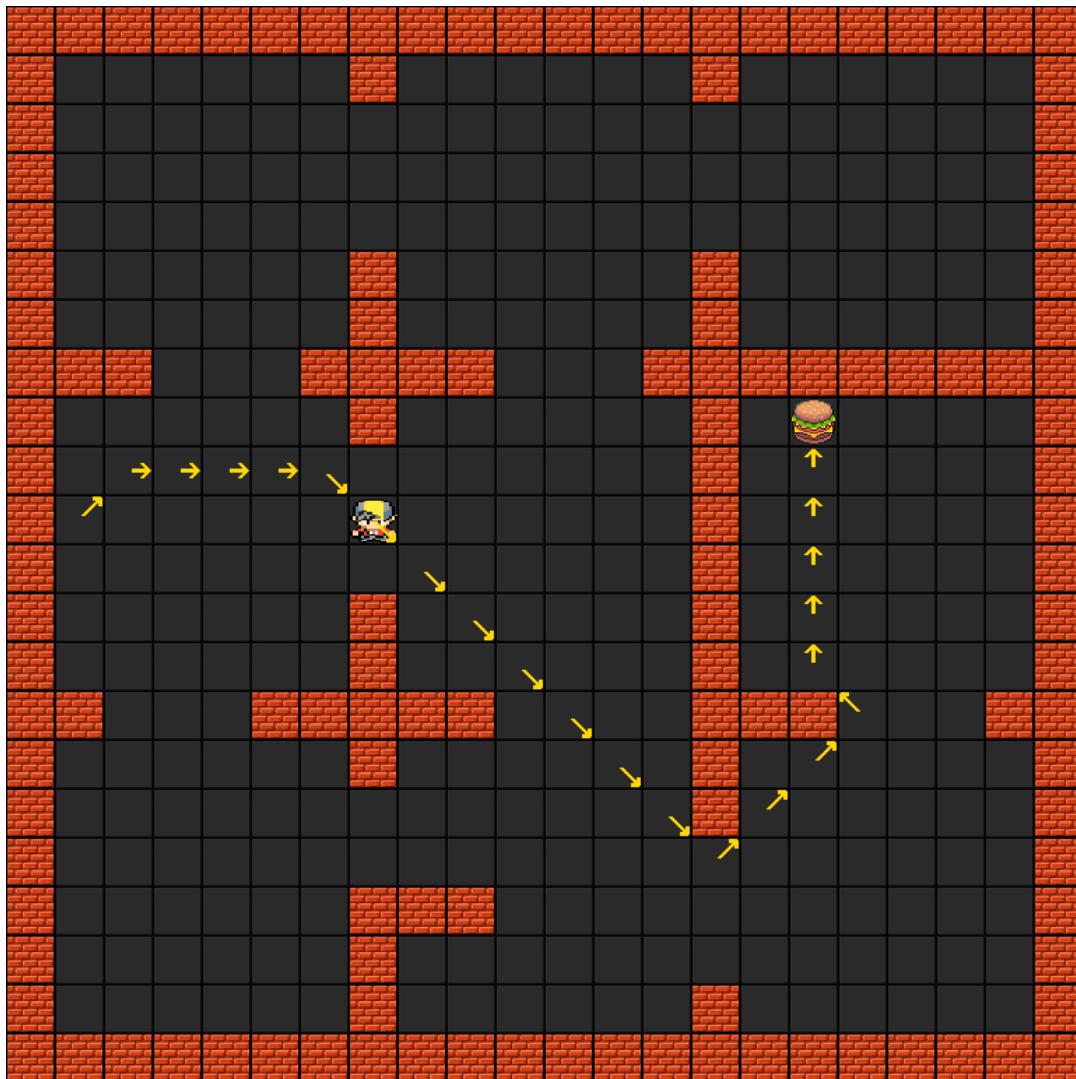


Figura 57 - resultado obtido para a procura A* ponderada com peso igual a 5

O resultado para a procura com peso igual a 5 é visível o quanto a procura é levada para o lado ganancioso da solução. Isto resultou num caminho sub-ótimo devido à componente heurística pesar muito mais do que a função do custo do nó na avaliação do caminho. Desta maneira a implementação cumpre o pretendido no que diz respeito ao algoritmo.

Para testar o funcionamento para múltiplos objetivos foi criado um novo mapa igual ao segundo, mas com mais alvos no qual se obteve o seguinte resultado:

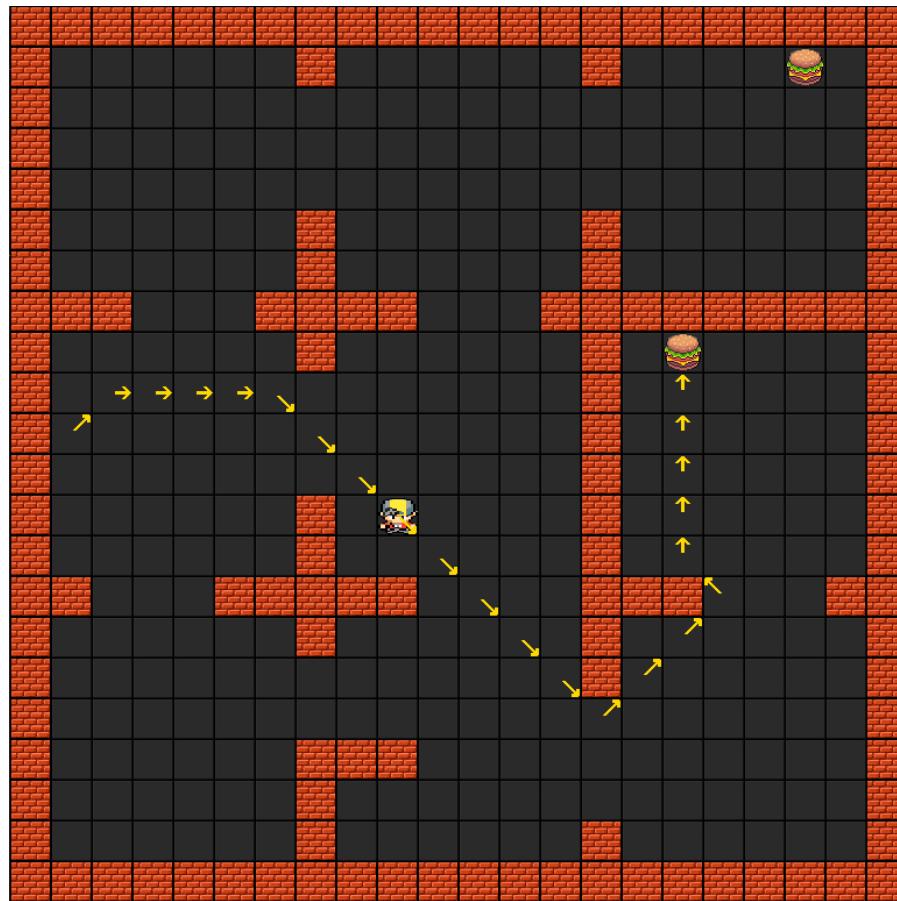


Figura 58 - teste para múltiplos alvos com peso 5

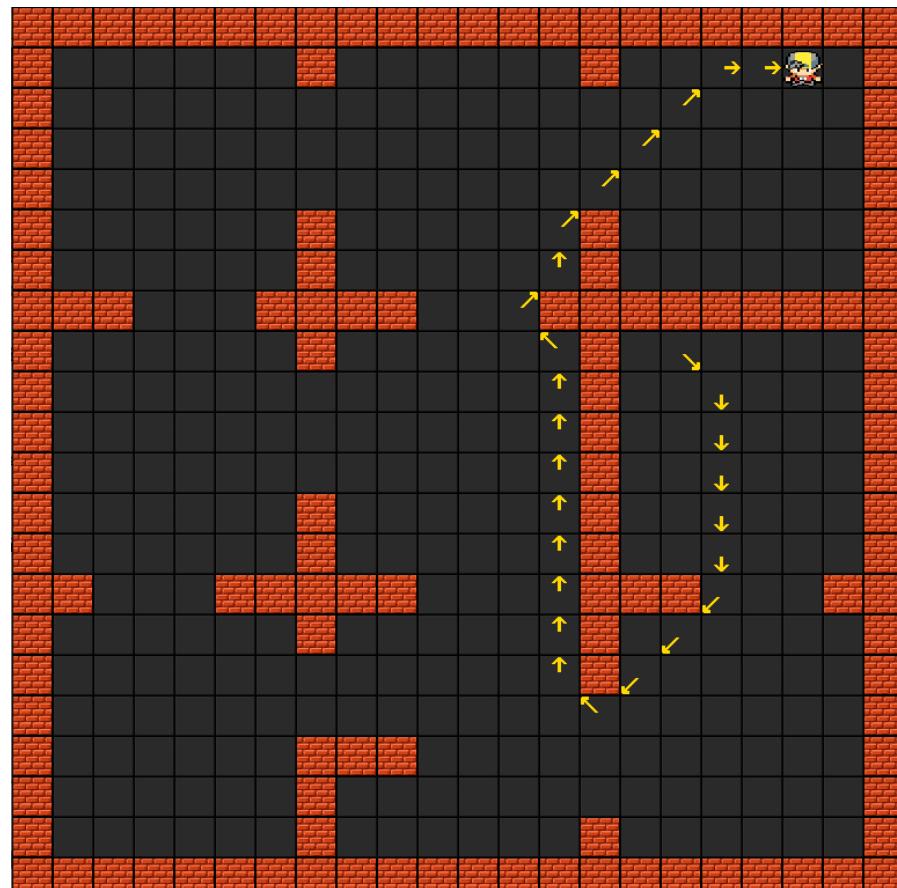


Figura 59 - continuação da figura anterior

3.2.2.1 Implementação do método frente-onda

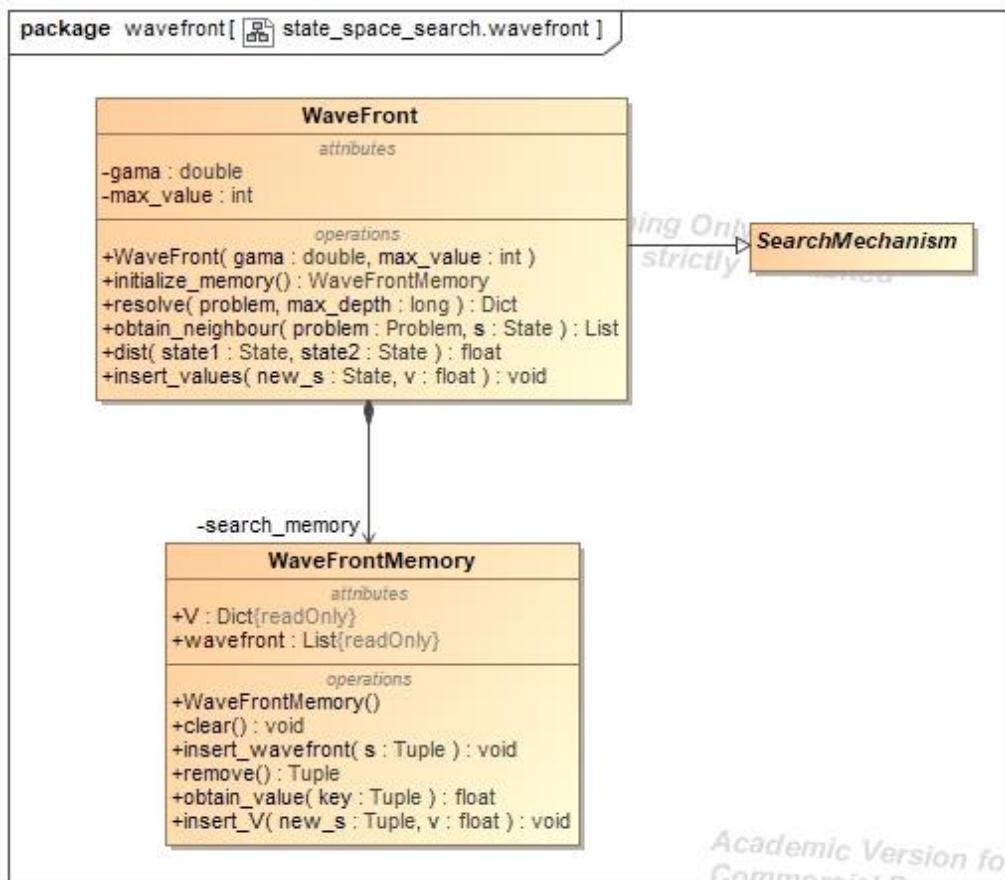


Figura 60 - mecanismo de procura do método frente-onda

O mecanismo de procura do método frente de onda é feito na classe **WaveFront**. Esta classe estende da classe **SearchMechanism** e faz *override* do método **resolve()**. Foi necessária esta abordagem pois o funcionamento do algoritmo de frente-onda não obedece à forma como está implementado na classe **SearchMechanism**. O método de frente-onda utiliza uma memória especial que para cada estado é associado um valor, quanto melhor o estado maior o valor. Este comportamento é feito através da classe **WaveFrontMemory**. O retorno do algoritmo da frente-onda é um dicionário da memória, que para cada estado fica associado um valor de qualidade de estado.

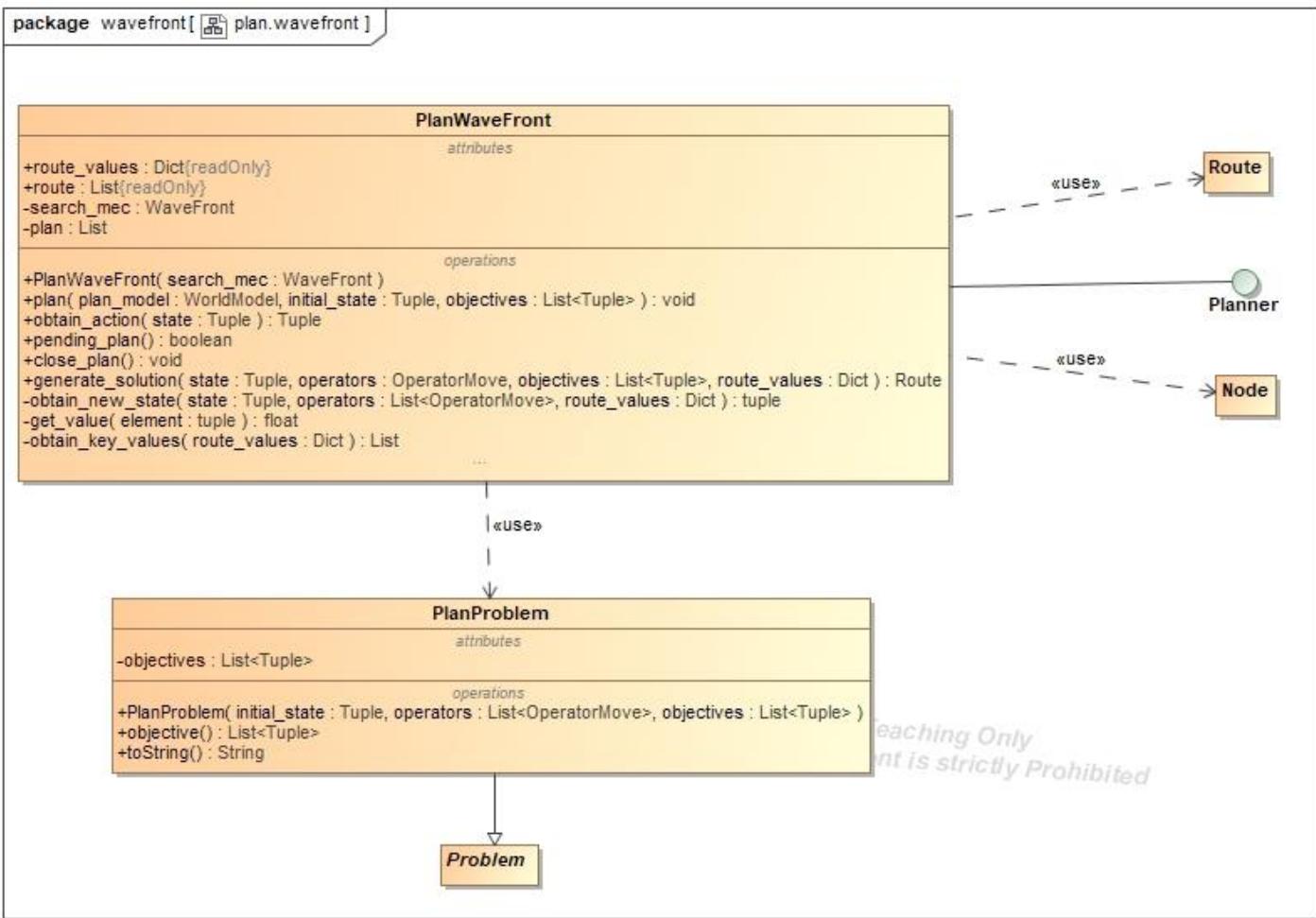


Figura 61 - plano da frente-onda

Da mesma maneira que a procura A* ponderada, o algoritmo de frente-onda também utiliza um planeador. Para isso primeiro foi criado um plano do problema, nomeadamente a classe **PlanProblem**. Apesar de ter o mesmo que a classe utilizada nas procuras melhores primeiro esta classe não é a mesma classe. Desta forma esta classe possui a função de definir quais são os objetivos. A classe responsável por gerar um plano de execução do agente é a classe **PlanWaveFront**. Esta classe utiliza o mecanismo de procura, nomeadamente a classe **WaveFront** referida anteriormente. A partir do dicionário retornado no mecanismo de procura esta classe cria uma solução através da classe **Route**, para construir o caminho completo ele utiliza a classe **Node** para definir cada passo da solução. No final é guardado o resultado no atributo **route**. Os restantes métodos desta classe funcionam de forma igual à classe **PlanSSS**, permitindo obter a ação, saber se existe um plano e terminar o plano corrente.

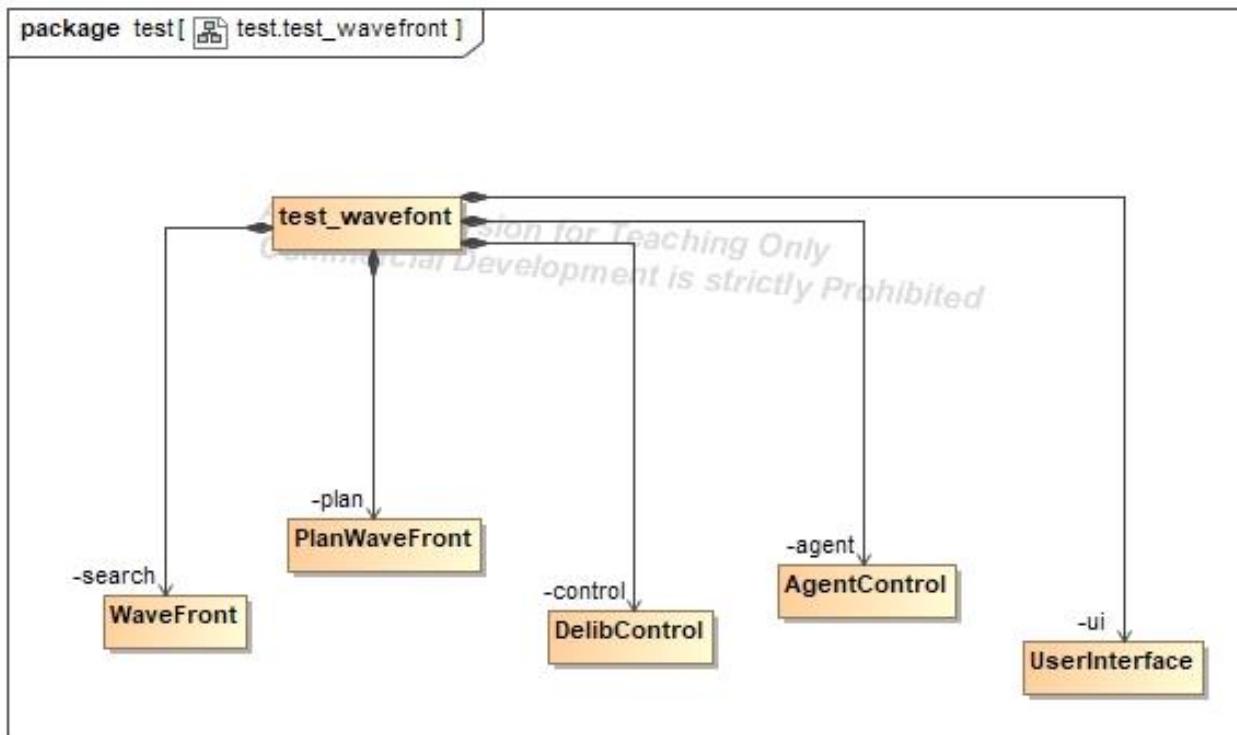


Figura 62 - execução do programa da frente-onda

No diagrama seguinte é mostrado onde é corrido o programa para testar o método de frente-onda. Isto é feito na classe `test_wavefront` e é também utilizado um ficheiro `batch` para correr o programa. Em relação às restantes classes estas são iguais às utilizadas na procura A* ponderada. O diagrama de sequência que representa o funcionamento do método frente-onda possui muitas semelhanças com a procura A* ponderada, pois utiliza muitas classes iguais. Desta maneira será apenas representado o digrama de sequência da parte diferente da procura A* ponderada. Desta maneira obteve-se os seguintes diagramas:

3.2.2.2 Diagramas de sequência do método frente-onda

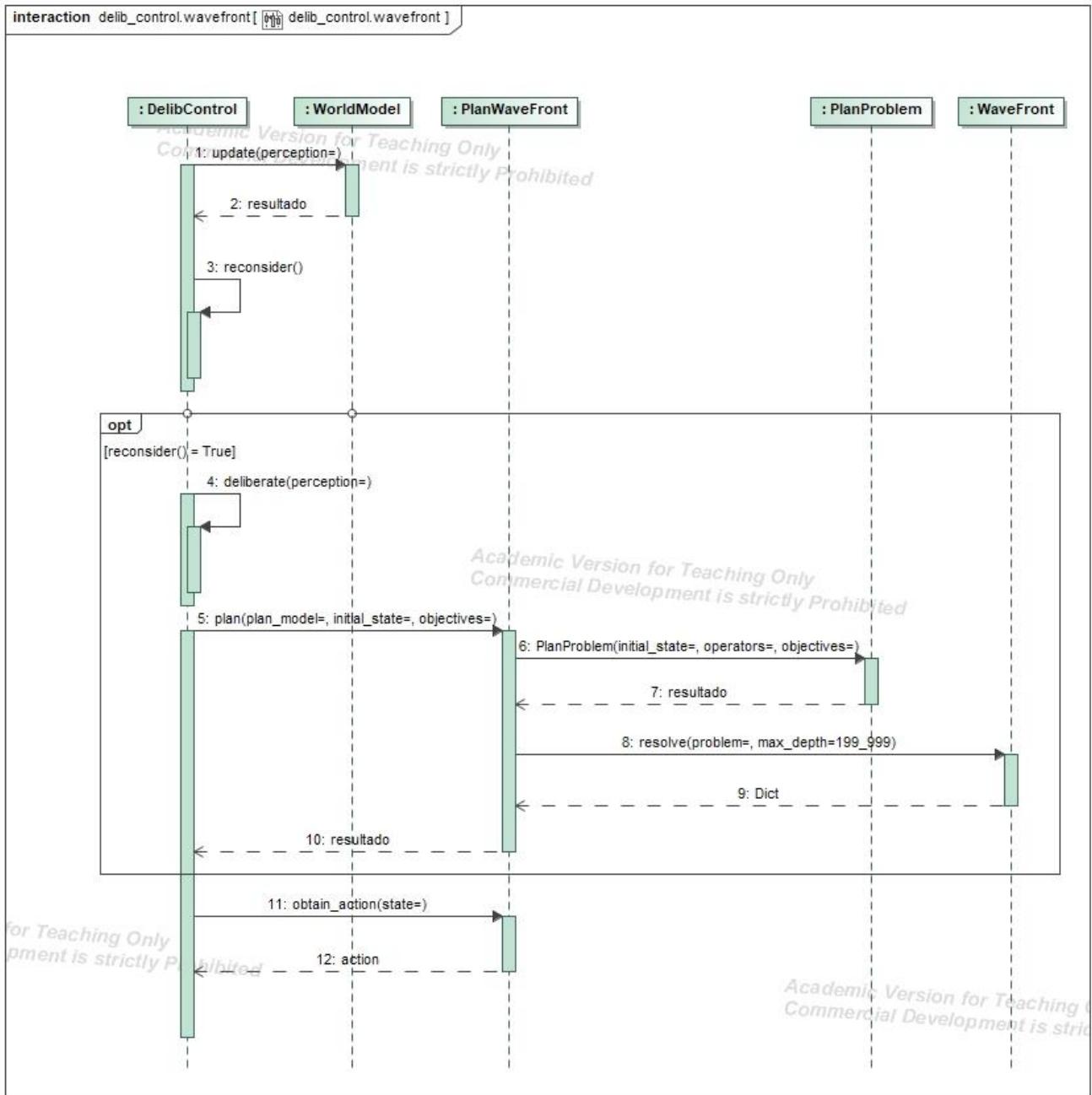


Figura 63 - diagrama de sequências do controlo deliberativo do método frente-onda

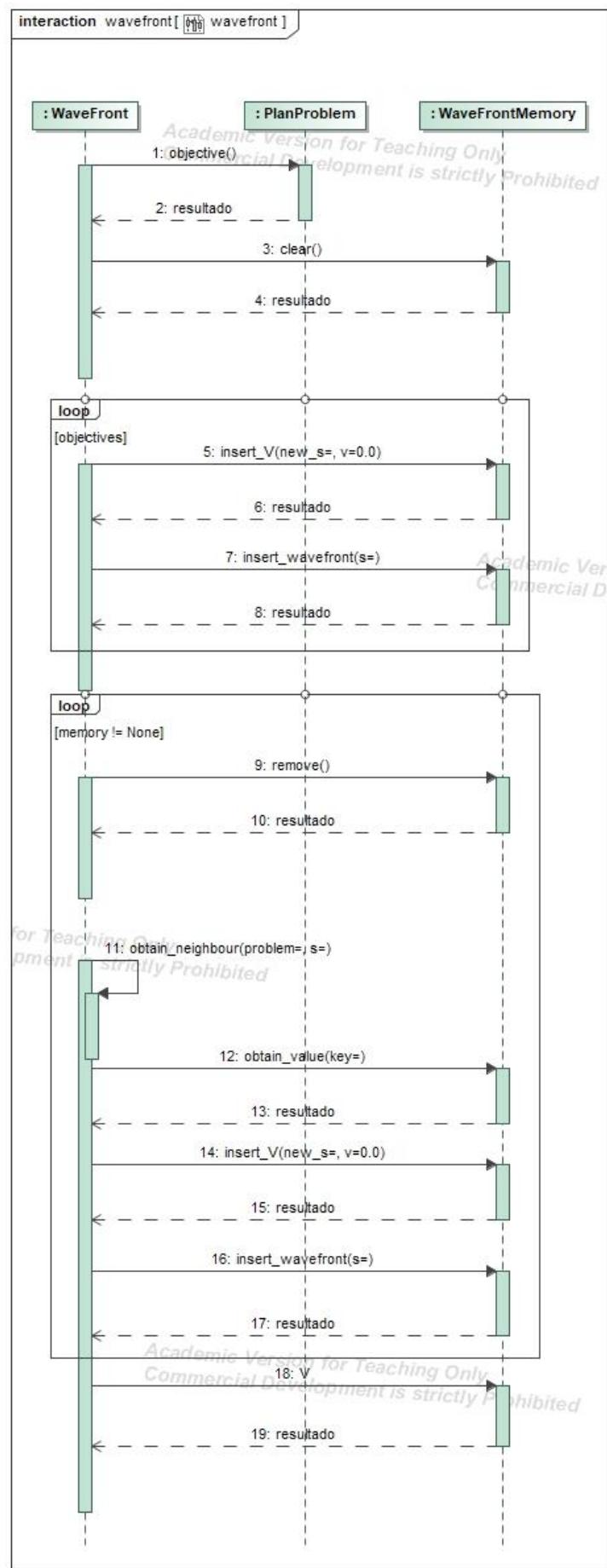


Figura 64 - diagrama de sequência do algoritmo frente-onda

3.2.2.4 Resultados obtidos

Em termos dos ambientes, estes seguem a mesma lógica que a procura A* ponderada, sendo mostrado somente o ambiente 2 e foi criado um terceiro com mais objetivos. Desta forma será testado a generalidade do funcionamento do algoritmo não somente para um objetivo, mas para um número ilimitado de objetivos. Além desta funcionalidade também foi adicionado um atributo booleano ao controlo do agente, quando o valor é verdade ele recomeça a simulação do início.

O resultado obtido para o ambiente 2 foi o seguinte:

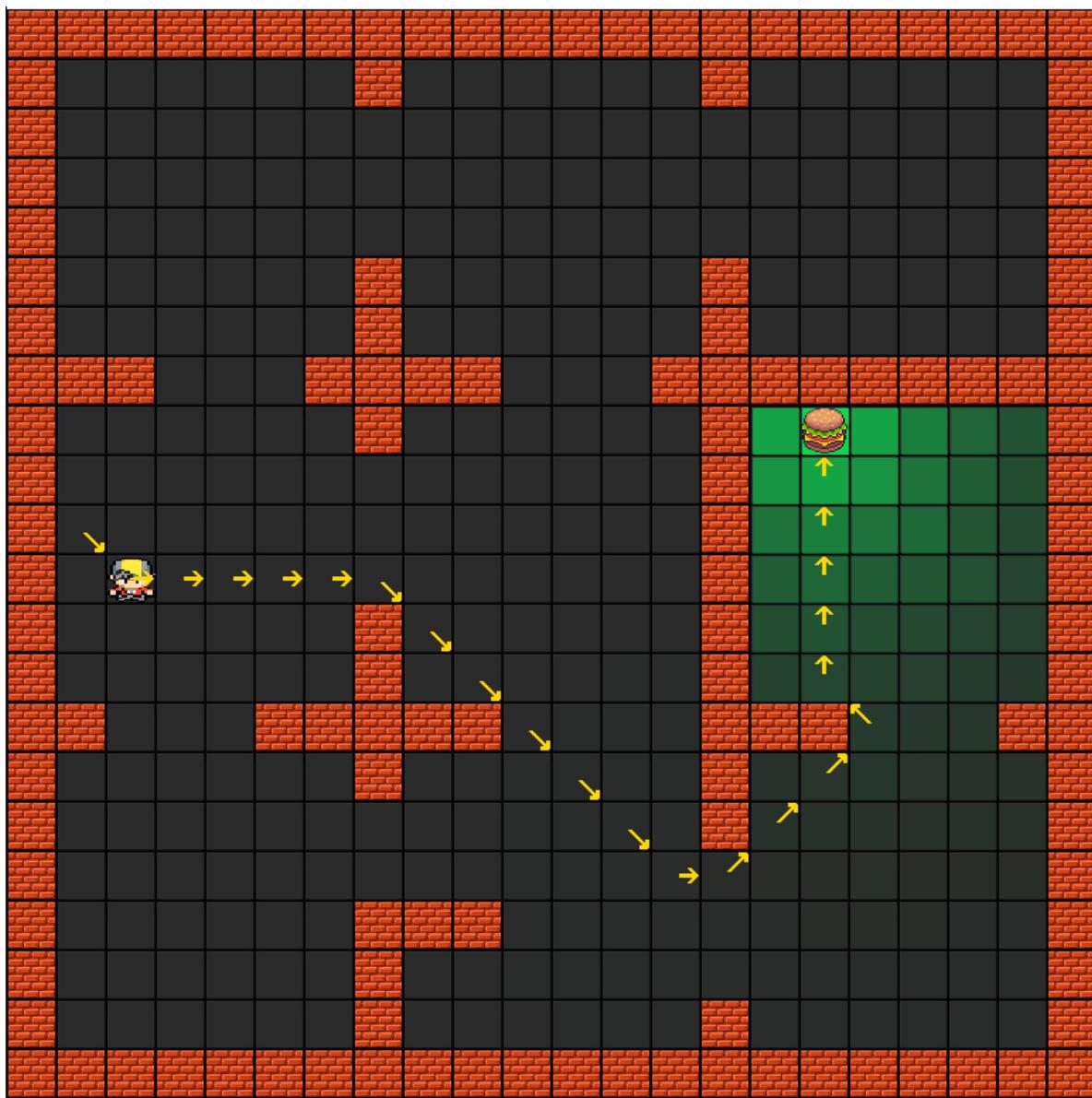


Figura 65 - resultado obtido da aplicação do método frente-onda para o ambiente 2

Na figura anterior observa-se duas componentes fundamentais, que são nomeadamente as setas amarelas que representam o caminho que será realizado pelo agente. A segunda componentes são os quadrados de cor verde que representa o valor associado a cada estado. Desta maneira verifica-se que parece correto o funcionamento do algoritmo de frente-onda.

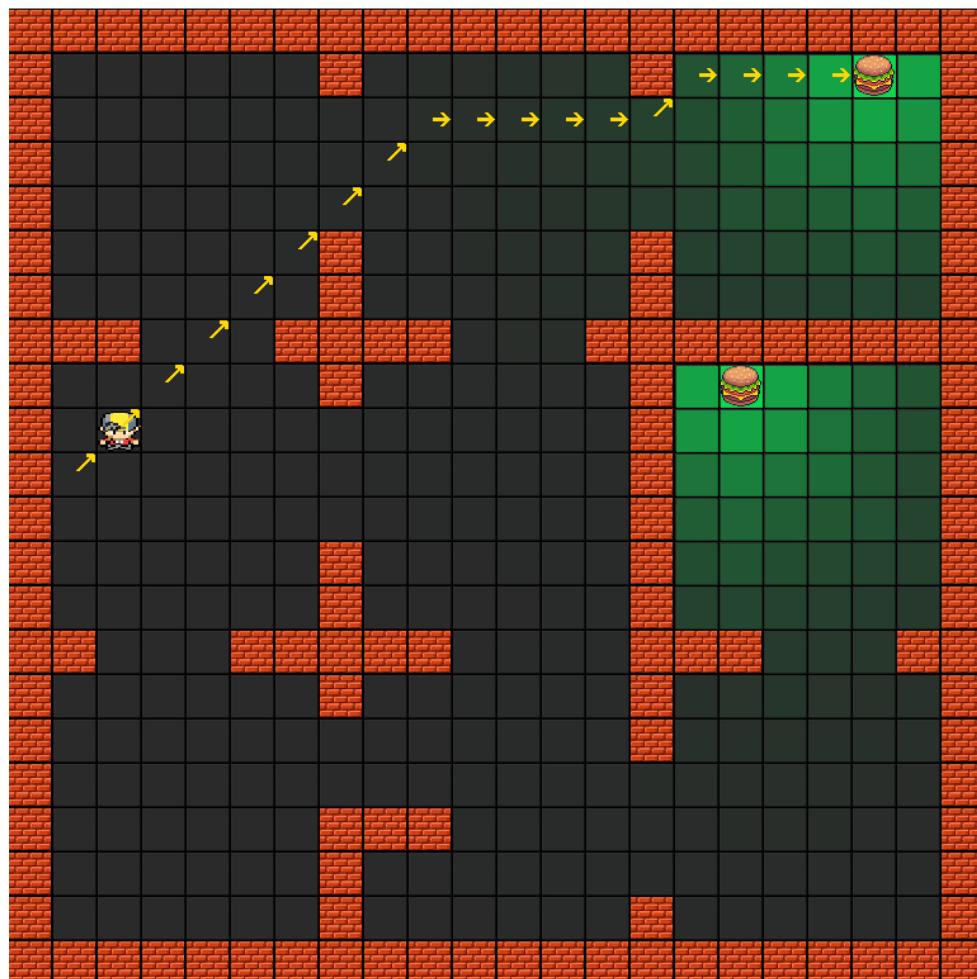


Figura 66 - resultado do algoritmo frente-onda para mais do que um objetivo

Este foi o resultado obtido quando o mapa possui mais do que um objetivo. Da mesma maneira que a figura anterior parece fazer o caminho ótimo e os estados com maior valor estão mais próximos do objetivo. Assim que o agente chega ao primeiro objetivo é obtida a seguinte figura:

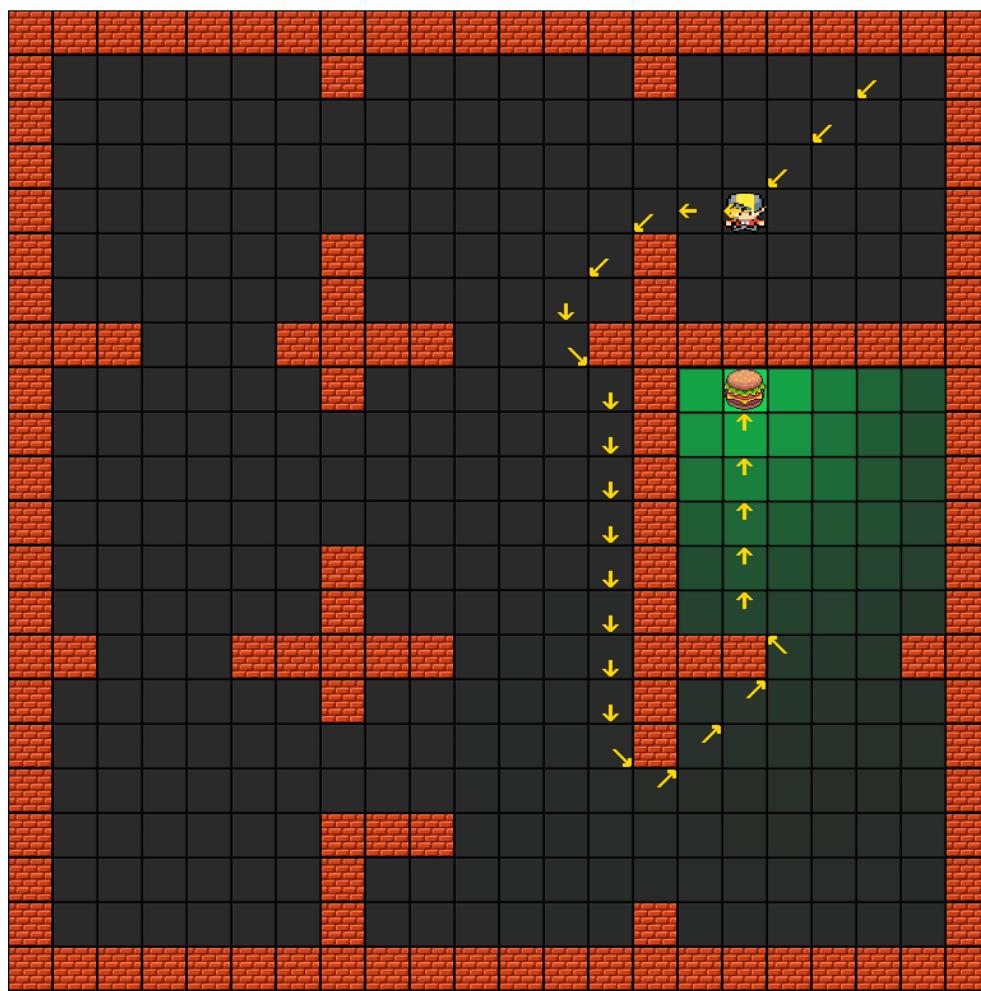


Figura 67 - resultado após o agente chegar ao primeiro objetivo

3.2.2.3 Arquitetura geral da solução

Por fim será mostrado a arquitetura geral da solução. Neste caso será mostrado como estão distribuídas as pastas do trabalho e como é que cada pasta interage com as restantes. O resultado final foi o seguinte:

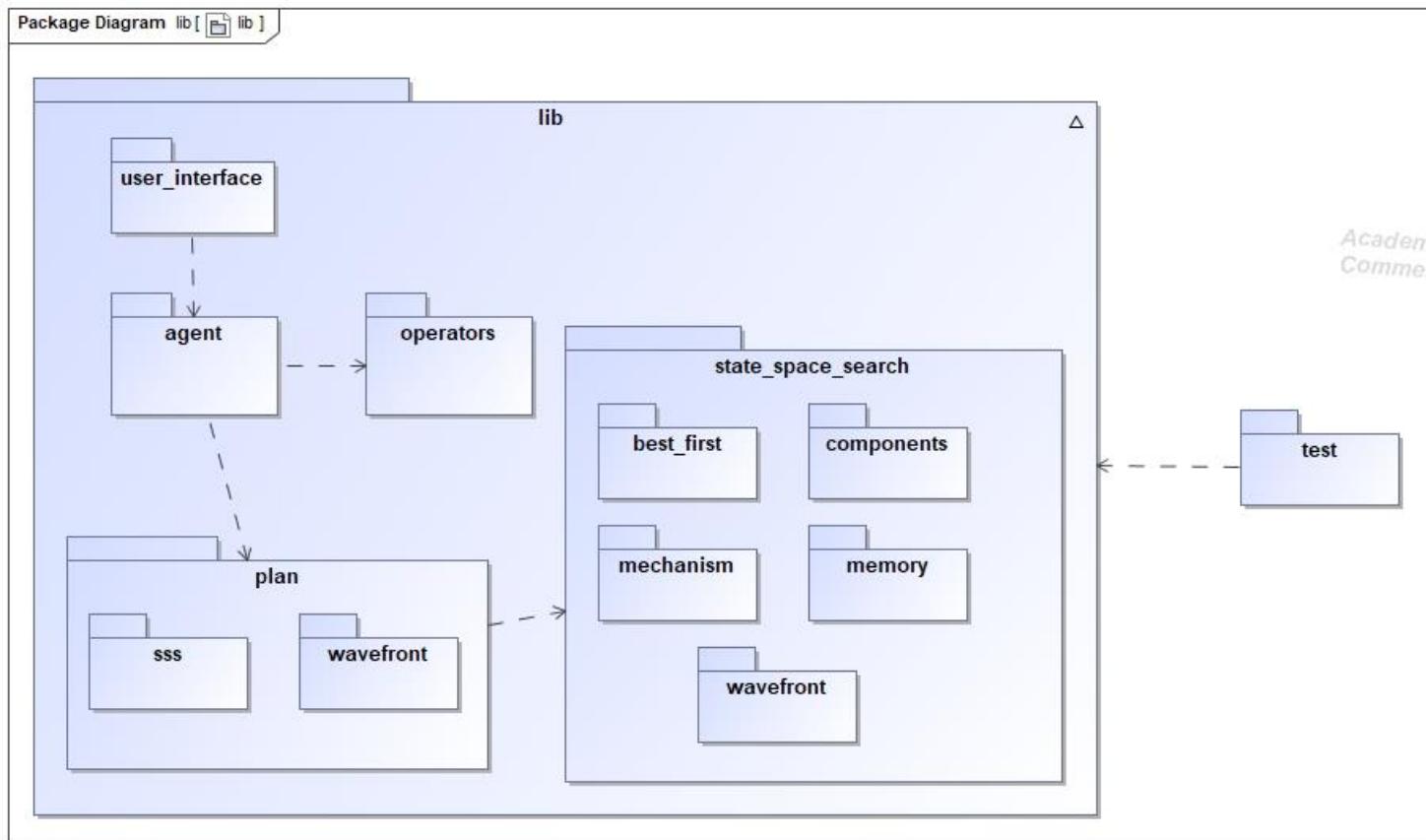


Figura 68 - diagrama geral da solução

3.2.3.1 Implementação da otimização

A biblioteca de otimização foi construída separadamente da biblioteca de procura em espaços de estados. Foi decidido assim, pois não seria fácil de integrar os dois aumentando muito a complexidade de cada um dos projetos. Os métodos escolhidos para implementar na biblioteca de otimização foram o *Hill-Climbing* estocástico e o *Simulated Annealing*. Apesar de não ter sido pedido também foi implementado o *Hill-Climbing* clássico. Para testar o funcionamento da biblioteca será necessário esta biblioteca resolver certos problemas. Os problemas são nomeadamente: o caixeiro-viajante e o problema das N-Rainhas.

Primeiramente será explicado como foi implementada a biblioteca e posteriormente como foi implementada cada componente do problema das N-Rainhas e do caixeiro-viajante.

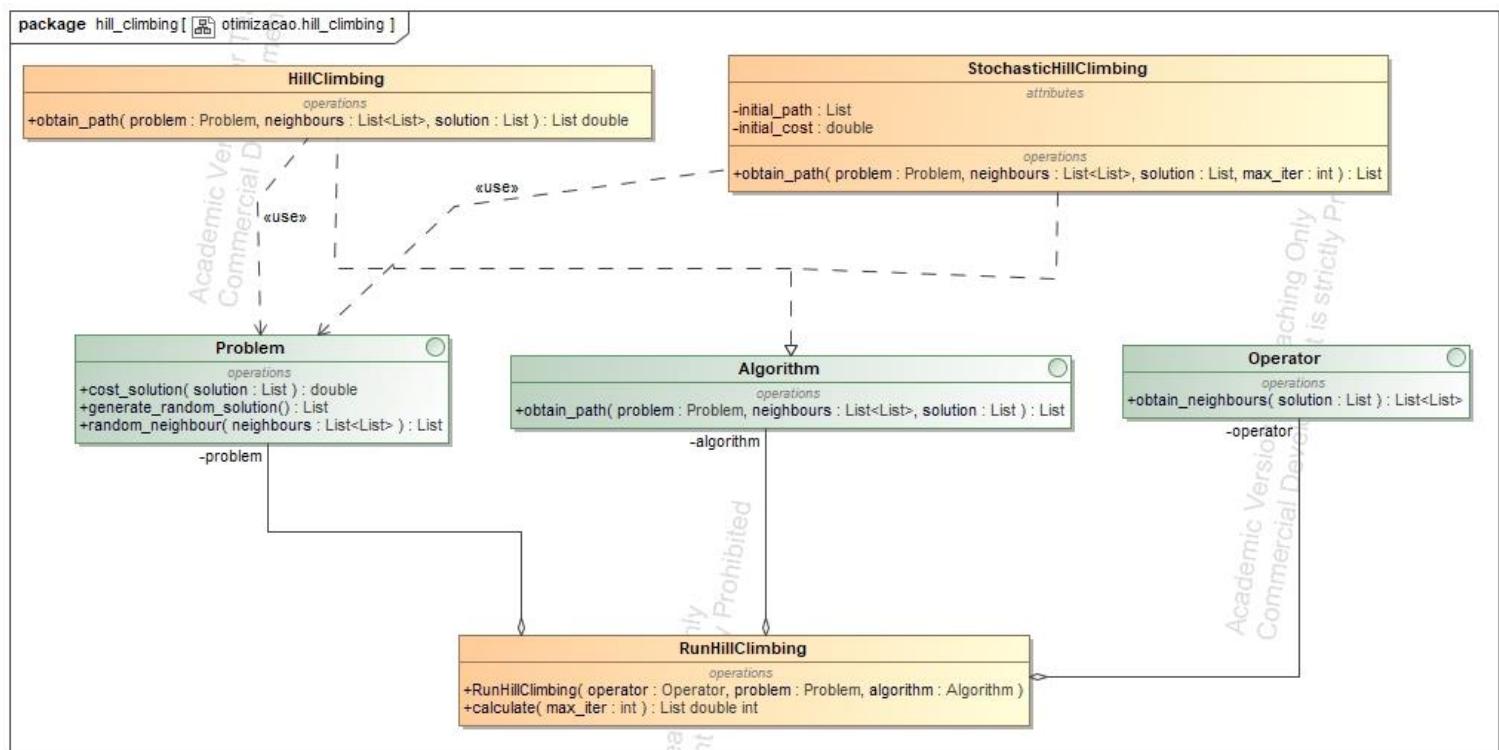


Figura 69 - diagrama de classes do algoritmo Hill-Climbing estocástico

No diagrama de classes anterior é representado o funcionamento do algoritmo do *Hill-Climbing* estocástico e também do *Hill-Climbing*. A classe que realiza o algoritmo do *Hill-Climbing* estocástico é a classe **StochasticHillClimbing**. Esta classe utiliza um ciclo do

tamanho definido pelo número máximo de iterações e sempre que encontra uma solução com um custo inferior à solução corrente ele retorna essa solução. Isto é feito no método `obtain_path()`, método esse que é obrigado a ser implementado devido à classe implementar a interface `Algorithm`. Da mesma maneira o algoritmo *Hill-Climbing* clássico é feito na classe `HillClimbing`. Este método é feito de tal maneira que percorre todos os vizinhos existentes para uma determinada solução e obtém a solução com menor custo de todos os vizinhos. Da mesma maneira este algoritmo é feito no método `obtain_path()`. Isto porque esta classe também implementa a interface `Algorithm`. A classe responsável por executar todo o funcionamento do algoritmo do *Hill-Climbing* é a classe `RunHillClimbing`. Esta classe primeiramente gera uma solução aleatória, obtém o seu custo e os seus vizinhos. Para obter uma solução aleatória e o custo a classe utiliza a classe que implementar a interface `Problem`. Para a classe obter os vizinhos utiliza a classe que implementar a interface `Operator`. Para obter de todos os vizinhos o melhor é utilizado uma classe que implementa a interface `Algorithm`. Quer isto dizer que será ou a classe `HillClimbing` ou `StochasticHillClimbing`. Por fim a classe `RunHillClimbing` utiliza um ciclo para estar sistematicamente a obter os vizinhos e qual é a melhor solução de entre os vizinhos. Este ciclo termina quando não é encontrado nenhum vizinho com menor custo que a solução corrente.

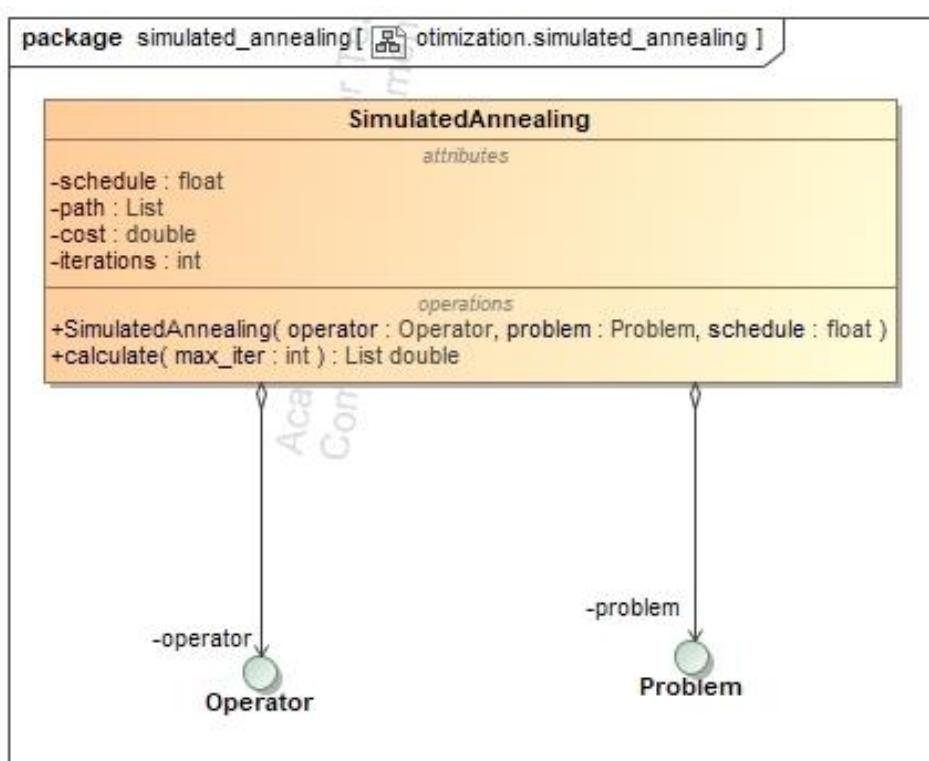


Figura 70 - diagrama de classes do algoritmo Simulated Annealing

No diagrama de classes mostra o funcionamento do algoritmo *Simulated Annealing*. Este algoritmo não possui o mesmo comportamento que o *Hill-Climbing* e por isso foi necessário criar uma classe nova. Esta classe é designada **SimulatedAnnealing**. Este algoritmo funciona de maneira que vai obtendo os vizinhos e o custo, sempre que obtém um custo inferior ele define como o caminho corrente. Esta classe possui um atributo que é a temperatura definido como *schedule*. Conforme ele vai correndo o algoritmo ele vai aumentando as restrições diminuindo a temperatura.

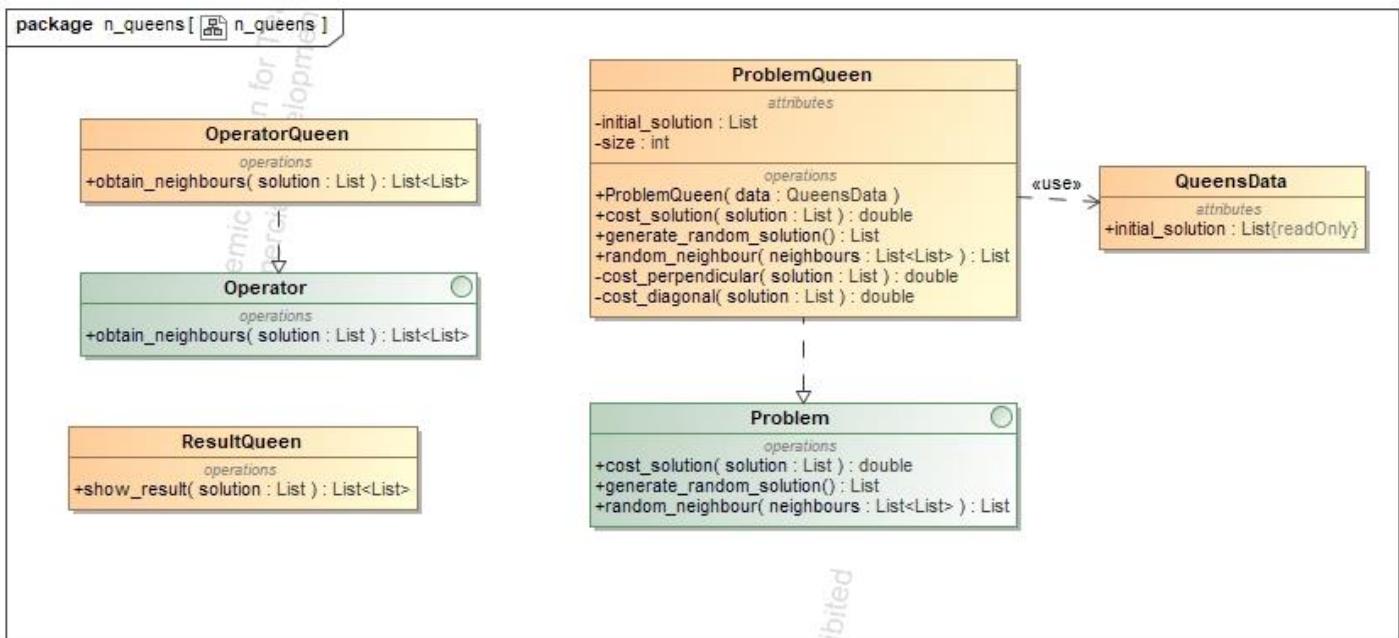


Figura 71 - componentes do problema das N-Rainhas

Para resolver o problema das N-Rainhas foram criadas quatro classes distintas. A primeira classe é a **QueensData**. Esta classe define apenas um estado aleatório para a posição das diferentes rainhas. A classe **OperatorQueen** é um operador e por isso implementa a interface **Operator**. Esta classe retorna um conjunto de configurações de vizinhos que representam possíveis soluções para o problema das N-Rainhas. A classe **ProblemQueen** é um problema e também por isso implementa a interface **Problem**. Esta classe possui o método para obter o custo de uma solução, obter uma solução aleatória e selecionar uma solução aleatória dentro dos vizinhos obtidos. Para obter o custo foi necessário verificar nas diagonais e nas perpendiculares do tabuleiro. Por fim a classe **ResultQueen** é responsável por pegar no resultado gerado pelo algoritmo e transformar num tabuleiro para poder ser posteriormente

apresentado graficamente.

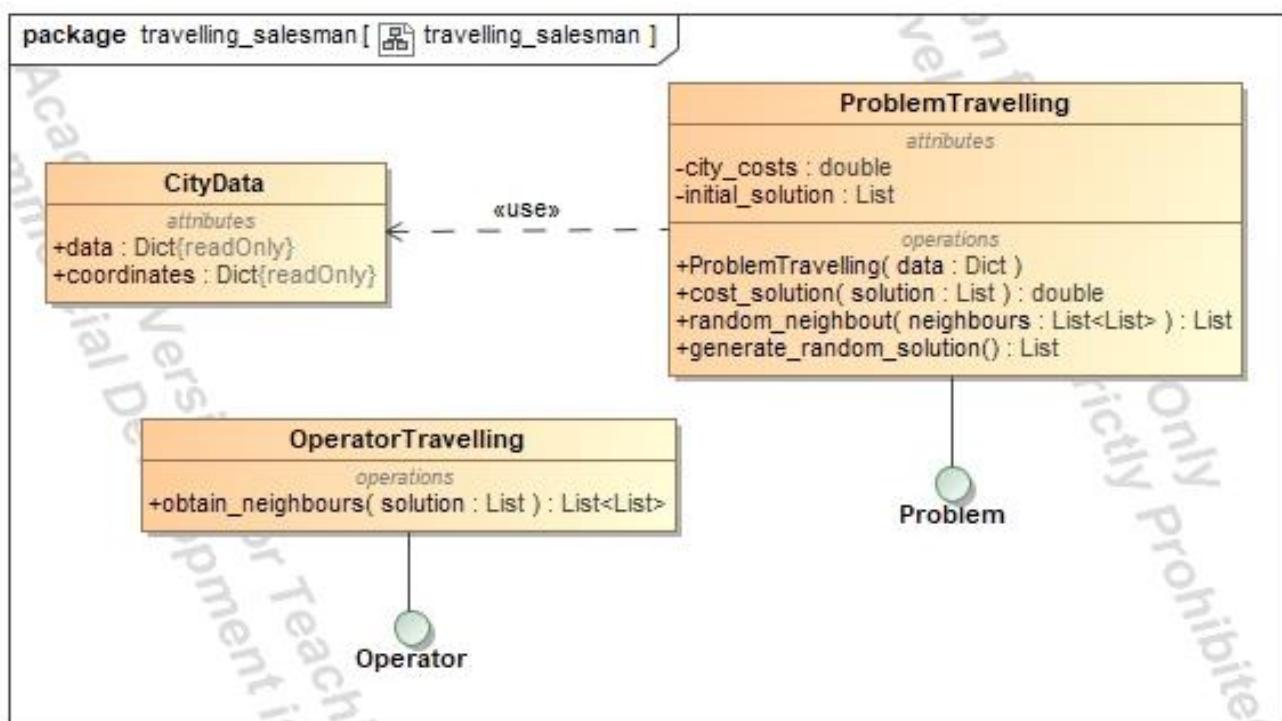


Figura 72 - componentes do problema do caixeiro-viajante

Para resolver o problema do caixeiro-viajante foram criadas três classes. A classe **CityData** é a classe que contém os dados referentes às cidades existentes, ao custo de viajar de uma cidade para outra e as coordenadas geográficas utilizadas posteriormente para desenhar o gráfico. A classe **OperatorTravelling** implementa a interface **Operator** e é responsável por obter todos os vizinhos de uma determinada solução. A classe **ProblemTravelling** é a classe que representa o problema em questão.

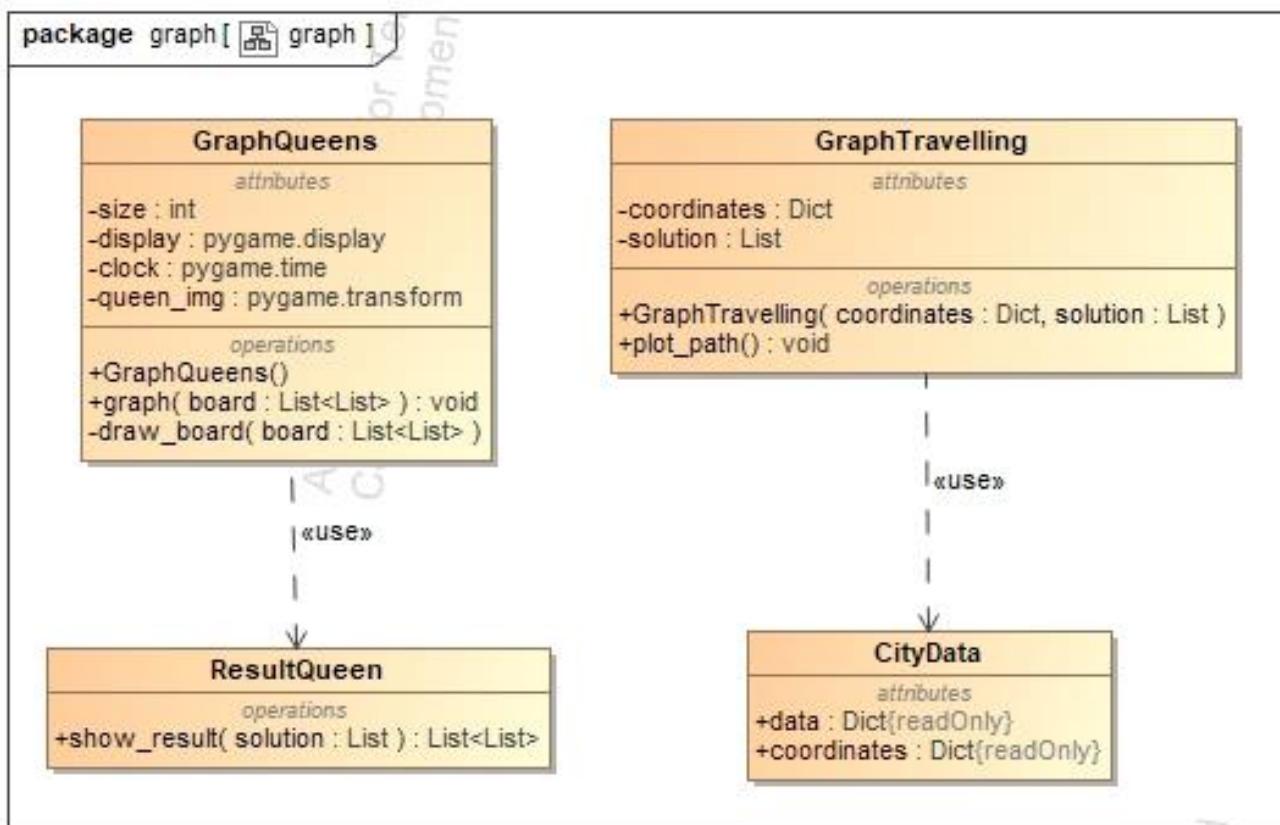


Figura 73 - gráficos de exibição dos resultados

No diagrama anterior é mostrado as classes responsáveis por criar um gráfico representativo da solução obtida. A classe **GraphQueens** utiliza a biblioteca `pygame` para criar um tabuleiro de xadrez para representar a solução obtida. Neste caso o método responsável por desenhar o gráfico é o método `graph()`, onde recebe como argumento a solução em forma de tabuleiro da classe **ResultQueen**. A classe **GraphTravelling** utiliza a biblioteca `matplotlib` para criar um gráfico de coordenadas geográfica para representar o caminho final obtido pelos algoritmos. Isto é feito pelo método `plot_path()`.

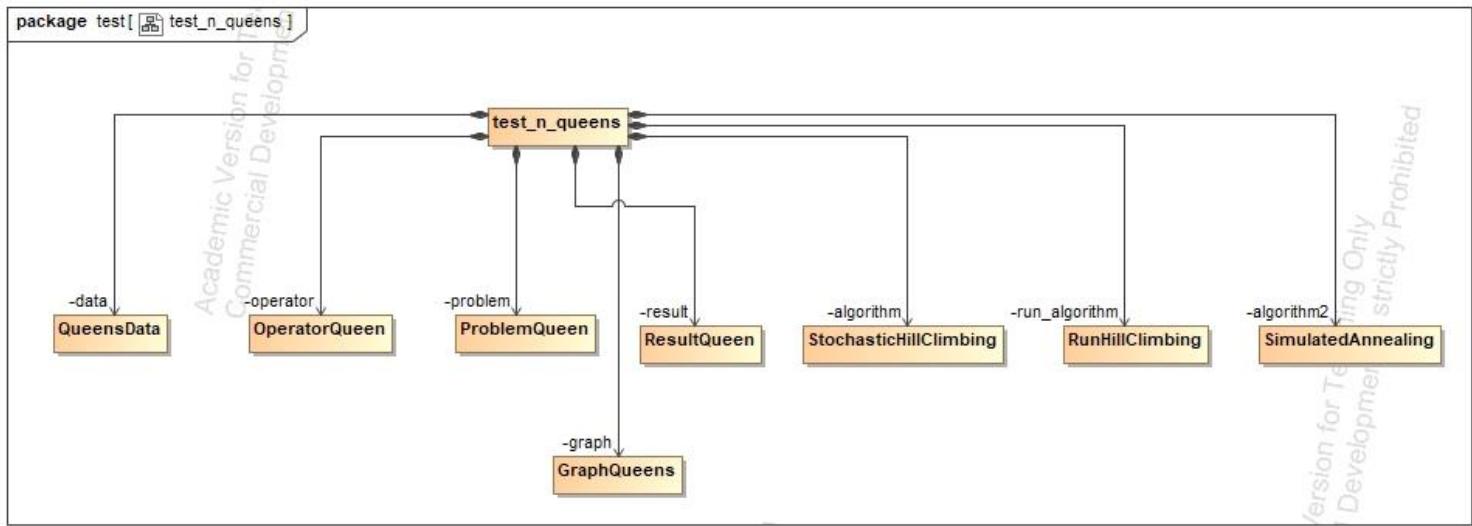


Figura 74 -execução do problema das N-Rainhas

No diagrama seguinte é mostrado como é executado o problema nas N-Rainhas. Neste programa são corridos tanto o algoritmo do Hill-Climbing estocástico como o algoritmo Simulated Annealing. Para estes algoritmos são utilizadas as componentes do problema das N-Rainhas e o gráfico correspondente. Da mesma maneira que a procura em espaço de estados é utilizado um ficheiro **batch** para executar o programa.

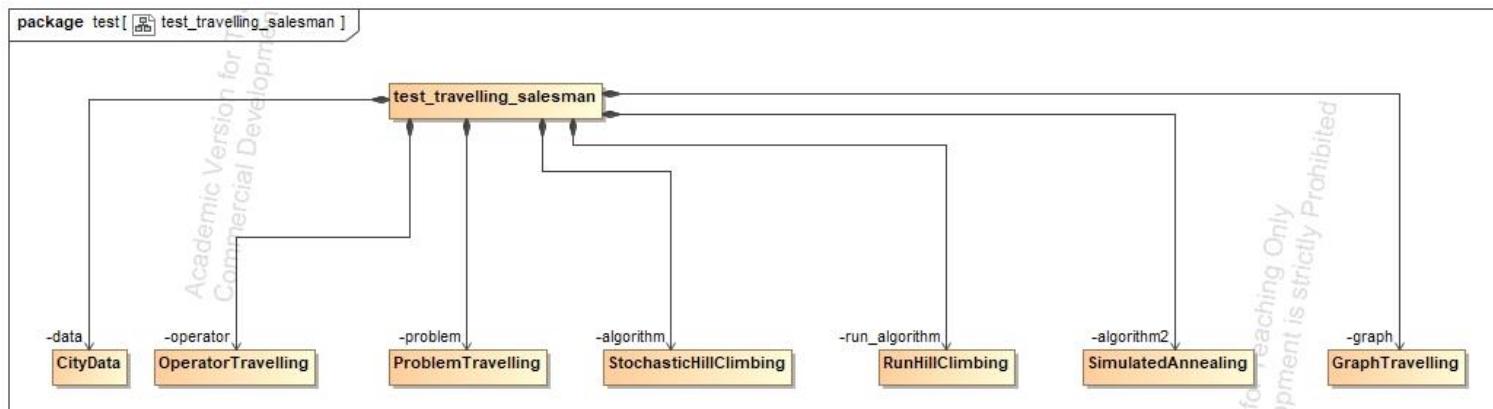


Figura 75 - execução do problema do caixeiro-viajante

A execução do programa do caixeiro-viajante possui as mesmas propriedades que o problema das N-Rainhas. A diferença está nas classes dos componentes e do gráfico.

3.2.3.2 Diagramas de sequência e arquitetura geral da solução da otimização

De seguida será mostrado o diagrama de sequências do problema das N-Rainhas para o algoritmo Hill-Climbing estocástico. Só será mostrado para o problema das N-Rainhas pois o diagrama é igual mudando só para as instâncias do problema do caixeiro-viajante.

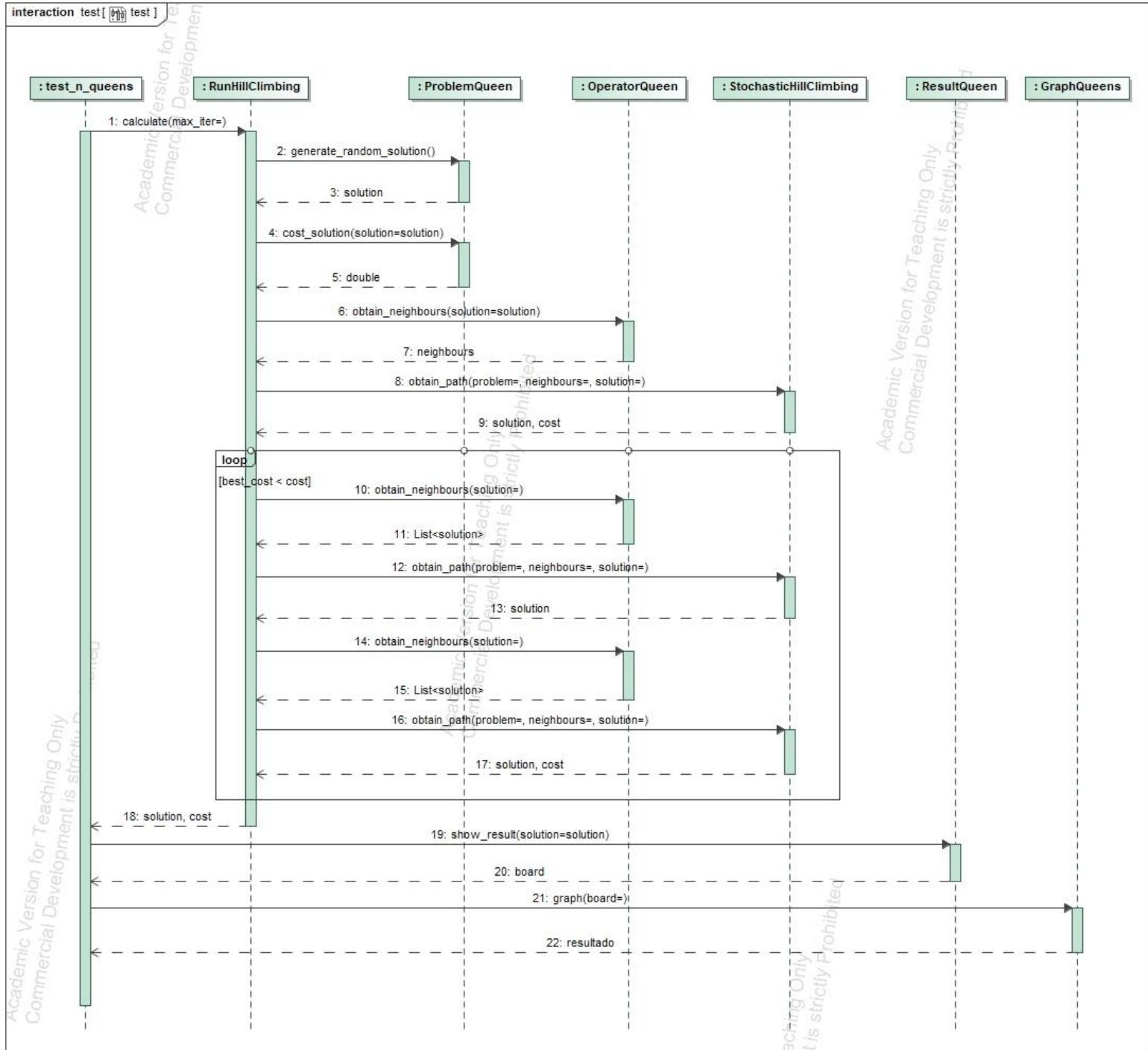


Figura 76 - diagrama de sequências do algoritmo Hill-Climbing estocástico

Dentro do método `obtain_path()` do `StochasticHillClimbing` surge o seguinte diagrama de sequências:

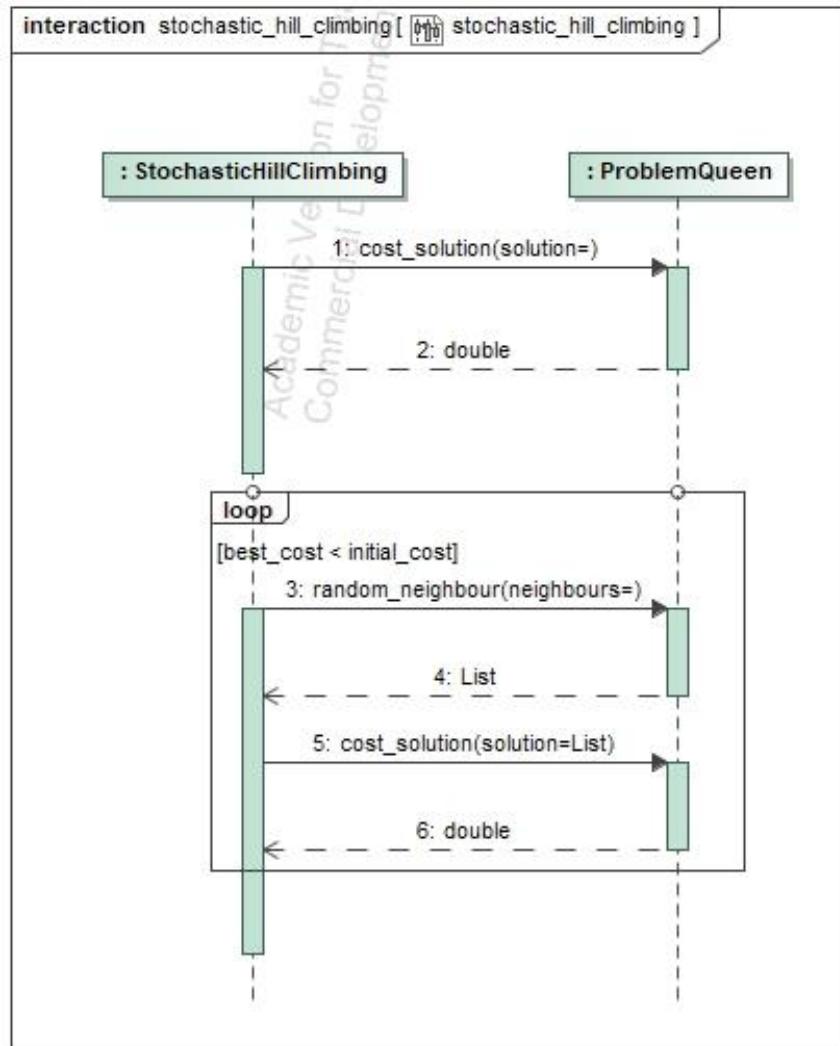


Figura 77 - diagrama de sequências do algoritmo Hill-Climbing estocástico

O algoritmo Simulated Annealing segue o seguinte diagrama de sequências:

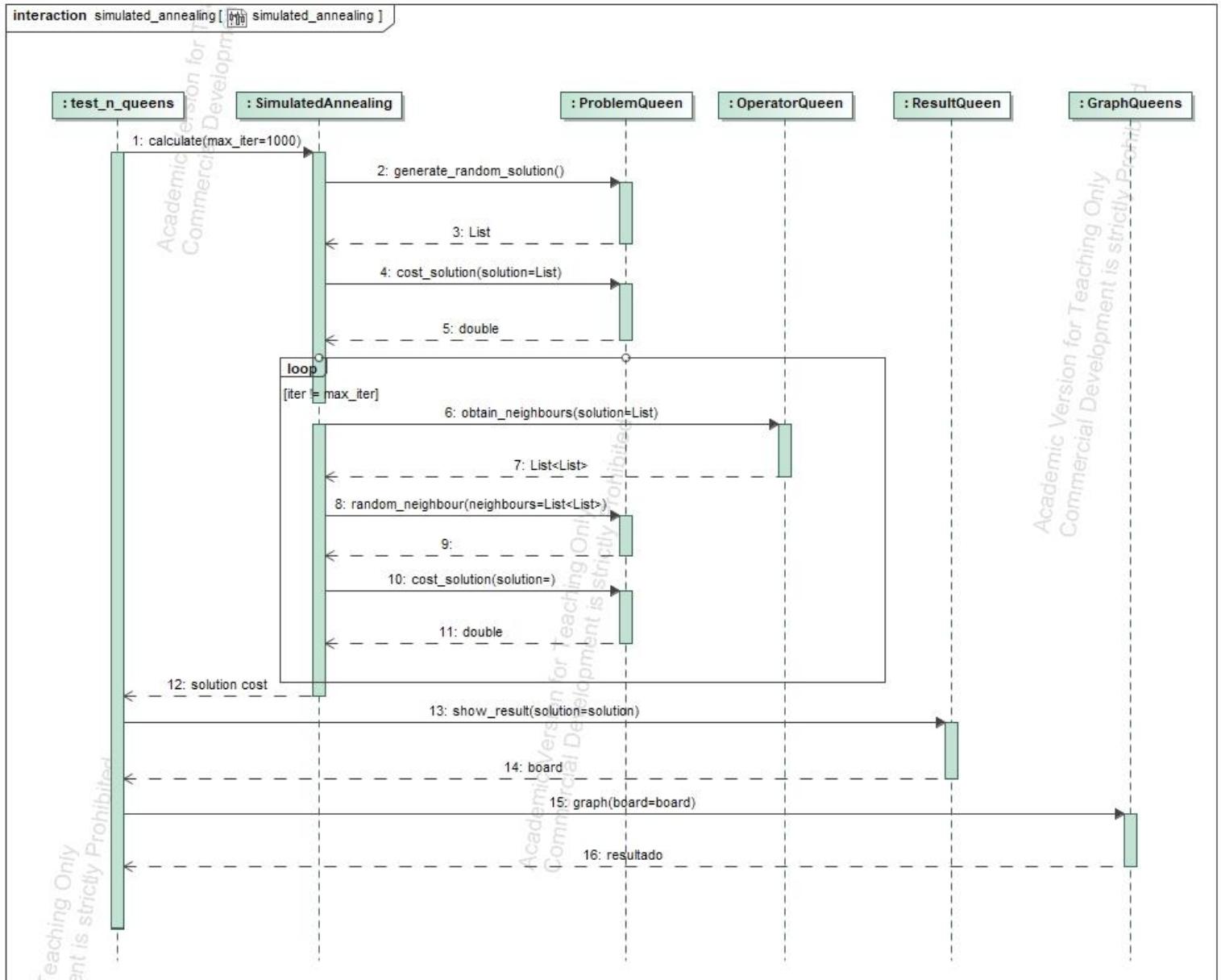


Figura 78 - diagrama de sequências do algoritmo Simulated Annealing

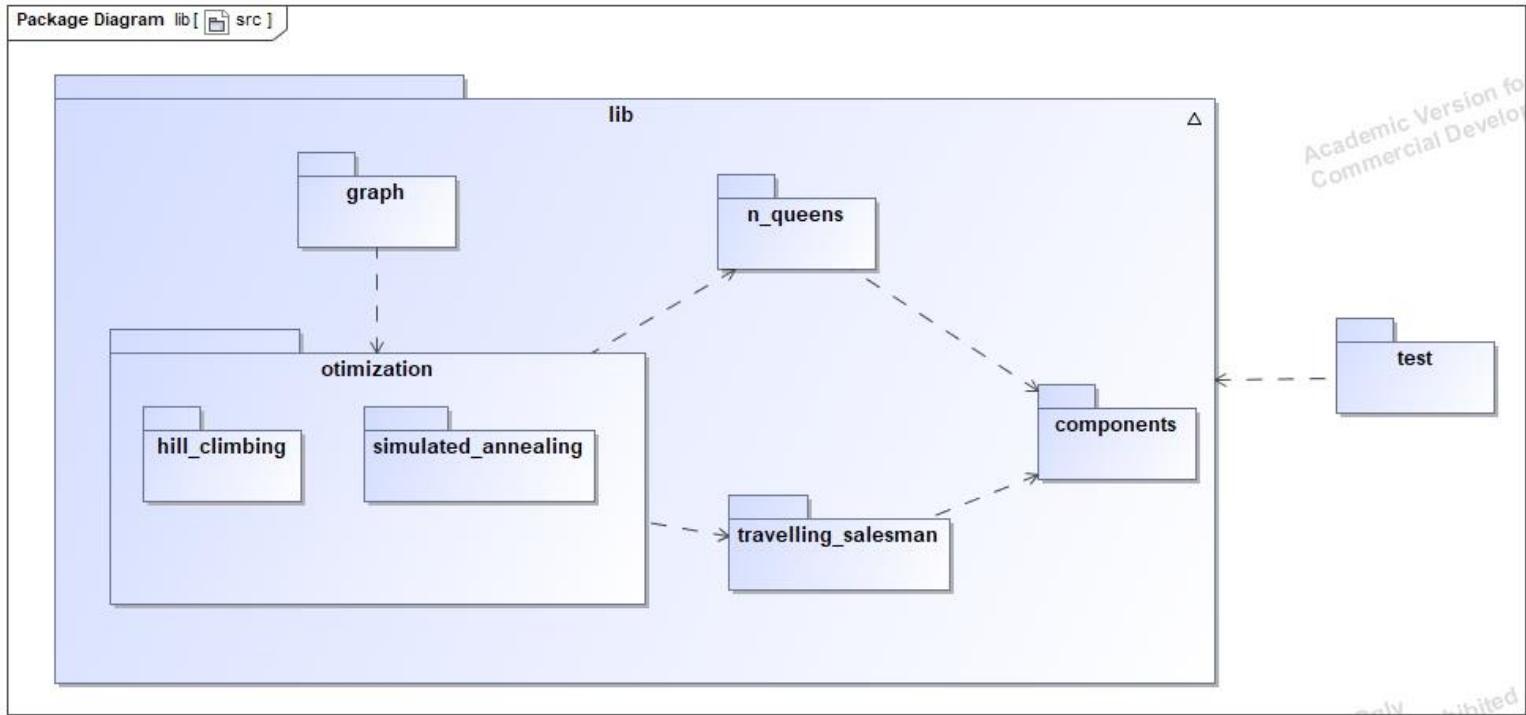


Figura 79 - arquitetura geral da solução

A arquitetura geral da solução da biblioteca e dos problemas segue o diagrama apresentado anteriormente. Esta arquitetura basicamente possui todas as componentes referidas anteriormente dentro da pasta lib. Esta possui as bibliotecas de otimização, as componentes dos dois problemas e os respectivos gráficos. A pasta test utiliza todos os componentes dentro de lib.

3.2.3.3 Resultados obtidos

Em relação aos resultados obtidos estes são medidos pelo custo da solução obtidos. Nesta etapa será exibido um exemplo da demonstração gráfica. Serão também apresentados gráficos respetivos com mais do que uma execução do programa para obter diversos custos e verificar o funcionamento do algoritmo.

O resultado obtido com o algoritmo Simulated Annealing para o problema do caixeiro-viajante graficamente foi a seguinte figura:

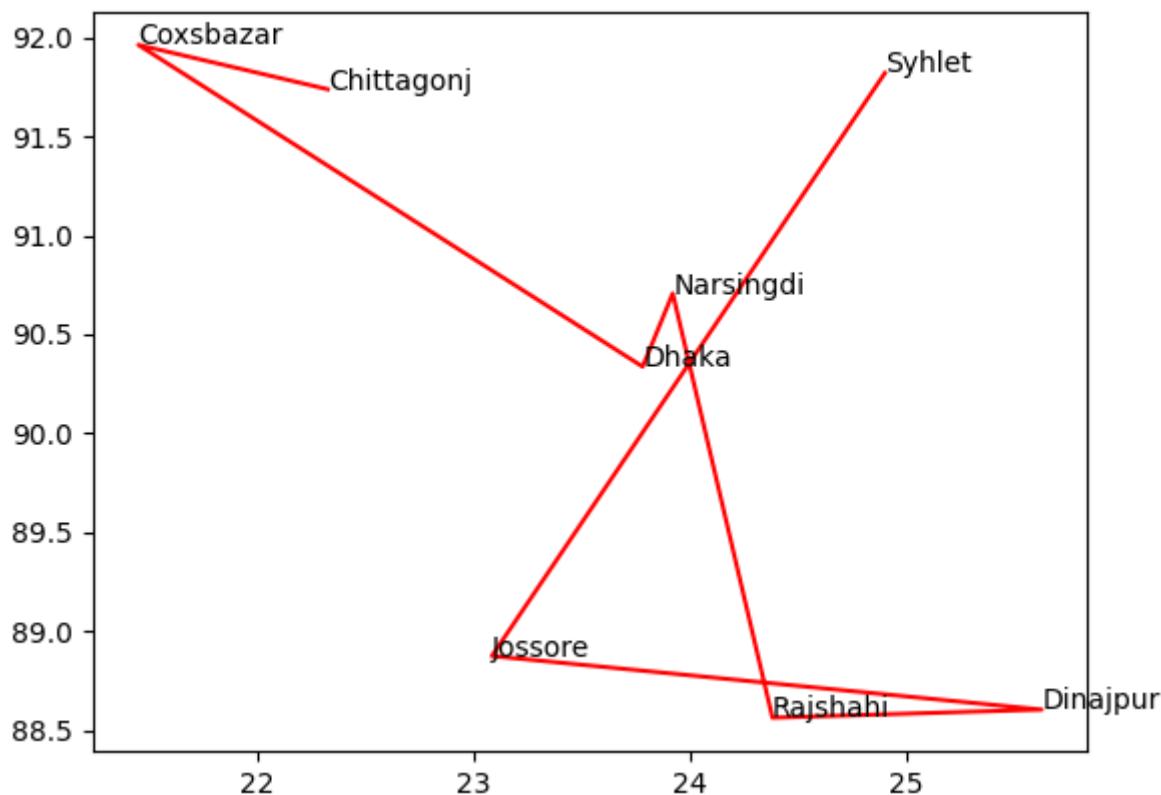


Figura 80 - resultado do caixearo-viajante com o algoritmo Simulated Annealing

O resultado para dez execuções do algoritmo Simulated Annealing e do Hill-Climbing estocástico foi a seguinte tabela:

	Simulated Annealing	Hill-Climbing estocástico
Custo	1973	1973
	1973	2004
	1973	1973
	2006	2004
	1973	2004
	1973	2004
	1973	1973
	1973	2006
	2004	1973
	1973	2004

Tabela 9 - custos resultantes de 10 execuções do problema do caixearo-viajante

Perante a tabela observa-se que de facto os valores variam sempre entre 1973 e 2006. Para o problema das N-Rainhas vai ser mostrado um exemplo gráfico do resultado e também uma tabela com diversos valores obtidos em diferentes execuções.

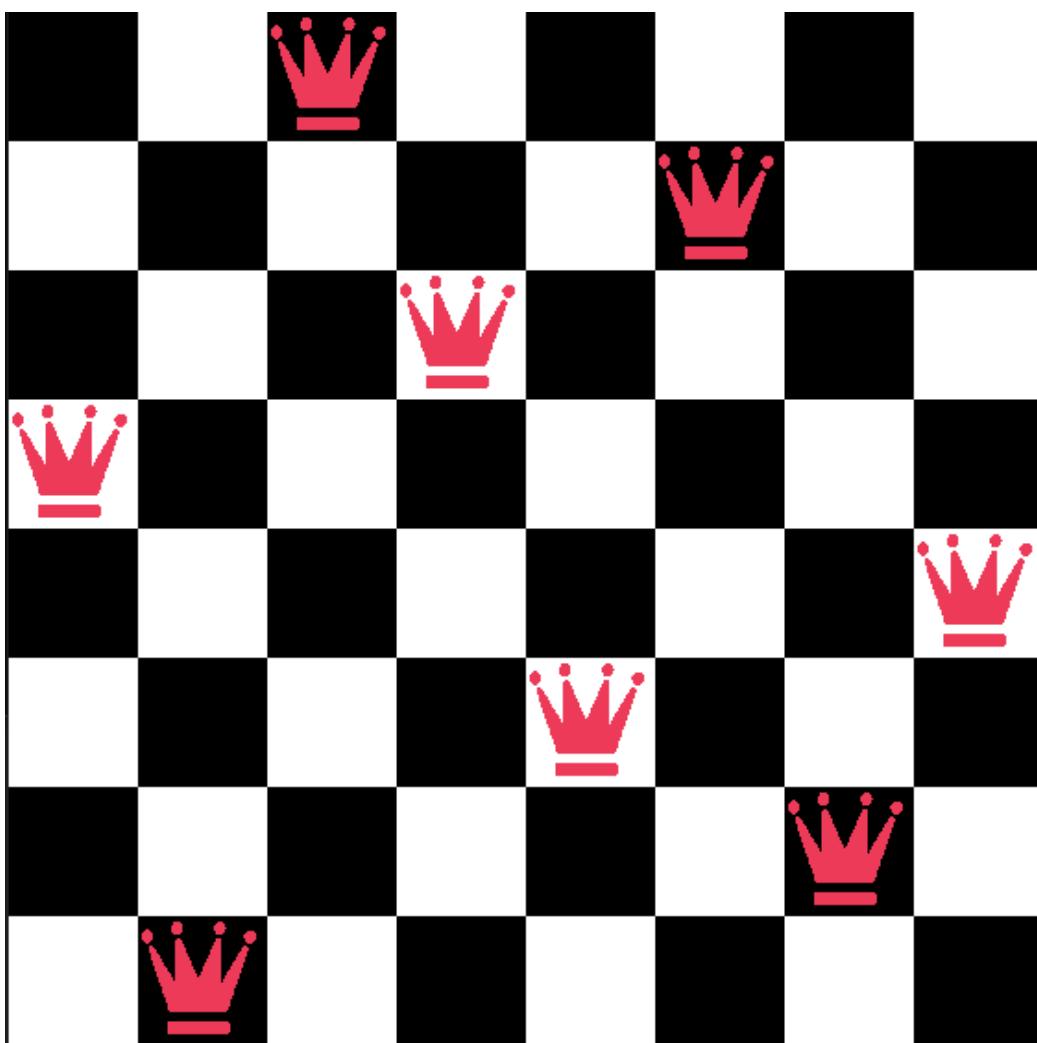


Figura 81 - resultado gráfico da execução do resultado do algoritmo Simulated Annealing com custo 0

O resultado em forma de tabela de diversas execuções dos dois algoritmos foi a seguinte tabela:

	Simulated Annealing	Hill-Climbing estocástico
Custo	0	2
	0	1
	0	0
	1	1
	0	2
	0	2
	0	2
	1	1
	0	1

Tabela 10 - custos resultantes de 10 execuções do problema das N-Rainhas

4. Tema livre

4.1 Geração de imagens a partir de texto

Durante décadas que os especialistas de inteligência artificial utilizam a aprendizagem automática para treinar computadores a compreender o mundo. Geralmente são utilizadas redes neurais para determinar se existem objetos numa imagem. Contudo nesta fase é a de aprender a criar imagens. Antigamente uma das dificuldades da geração de imagens credíveis era a necessidade de um grande conjunto de dados. Contudo essa barreira já foi ultrapassada com o poder computacional atual e a enorme quantidade de dados recolhidos.

Para criar imagens a máquina utiliza duas redes neurais. A primeira rede neuronal é utilizada para criar a imagem com base no texto introduzido pelo utilizador. A segunda rede neuronal analisa a imagem gerada com imagens de referência. Esta segunda rede compara a foto gerada com o que aprendeu com as imagens de referência e atribui uma pontuação para determinar a precisão da imagem gerada. Esta pontuação é devolvida e caso este valor seja muito baixo volta a gerar uma nova imagem para obter uma melhor pontuação.

Um método de geração de imagens é a transferência de estilo neural (NST). A NST utiliza um conjunto de algoritmos para recriar uma imagem existente para um novo estilo. Um exemplo seria ter uma foto da Marilyn Monroe e uma imagem do estilo de "The Weeping Woman" de Pablo Picasso o resultado seria uma imagem da Marilyn Monroe no estilo de mulher chorão. Este tipo de metodologia utiliza imagens pré-existentes para funcionar, funcionando como um filtro semelhante aos filtros utilizados na aplicação Instagram. Para gerar verdadeiramente uma imagem através de texto é necessário utilizar uma arquitetura do tipo Generative Adversarial Network (GAN).

As GANs são atualmente o tipo de rede mais semelhante aos artistas humanos em inteligência artificial. Neste tipo de modelo as redes possuem dois papéis, nomeadamente; o crítico (discriminador) e o gerador. O gerador possui fica encarregado de gerar uma imagem. Esta

rede aprende sobre estilo e conteúdo, podendo misturar e interpolar estilos. O crítico aprende arte a partir de imagens já existentes. A base de dados utilizada pelo crítico é utilizada para aprender e conseguir definir se a imagem gerada pelo gerador se parece ou não com arte criada por humanos. O crítico tenta detetar a imagem falsa e em cada ronda o gerador aprende e melhora a forma de enganar o crítico para acreditar que a imagem gerada é arte real. Desta maneira o discriminador é um classificador binário. Caso a imagem engane o crítico pode ser utilizada como resultado final. Da mesma maneira que o gerador o discriminador é penalizado por classificar incorretamente dados reais e falsos. A penalização é feita por retropropagação para pesar a perda na rede neuronal. Querendo isto dizer que são duas redes adversárias, melhorando os dois simultaneamente.

As arquiteturas GAN possuem os seguintes componentes:

- vetor de entrada ruidoso;
- rede de geradores, que transformam os dados de entrada em dados aleatórios;
- rede de discriminadores, para classificar os dados gerados;
- penalização em forma de reforço, quando o gerador não consegue enganar com sucesso o discriminador;

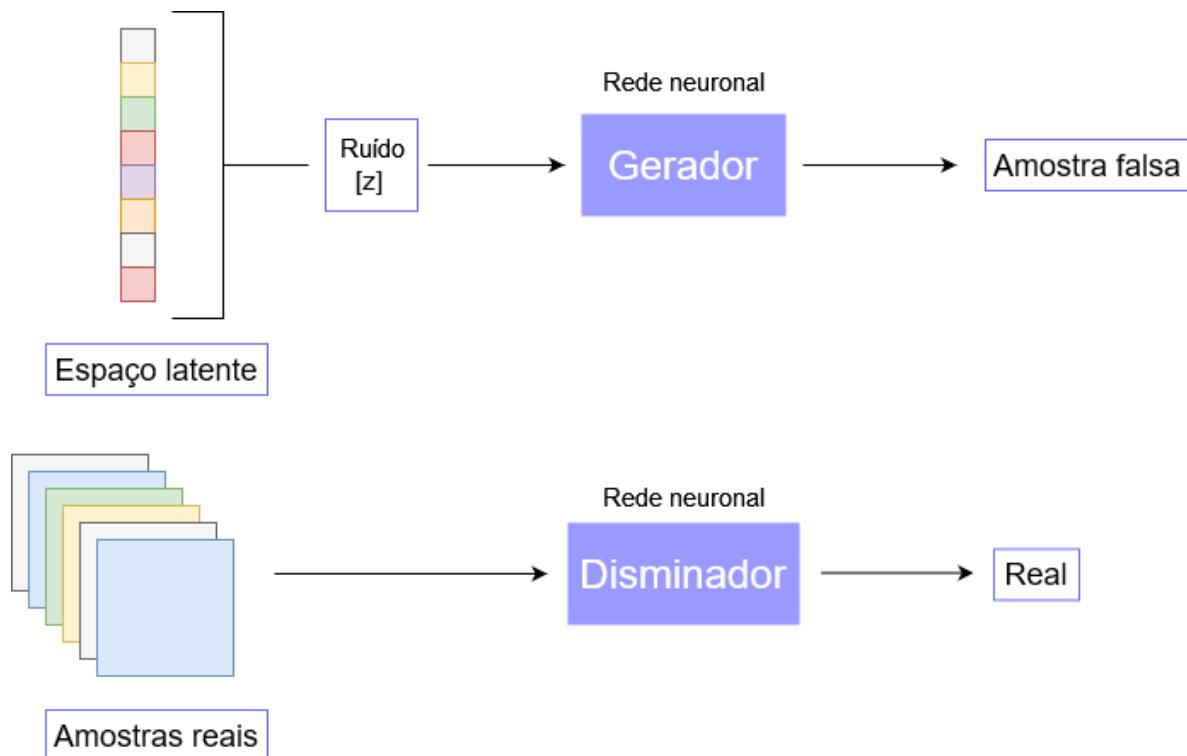


Figura 82 - blocos de construção do gerador e discriminador

Quando o discriminador devolve uma pontuação, esta pode ser utilizada para alterar os pesos do gerador. Os dados de testes do discriminador vêm de duas fontes, nomeadamente:

- imagens reais, sendo fotografias ou pinturas da vida real a serem utilizadas como os dados positivos;
- imagens falsas feitas pelo gerador são utilizadas como dados negativos;

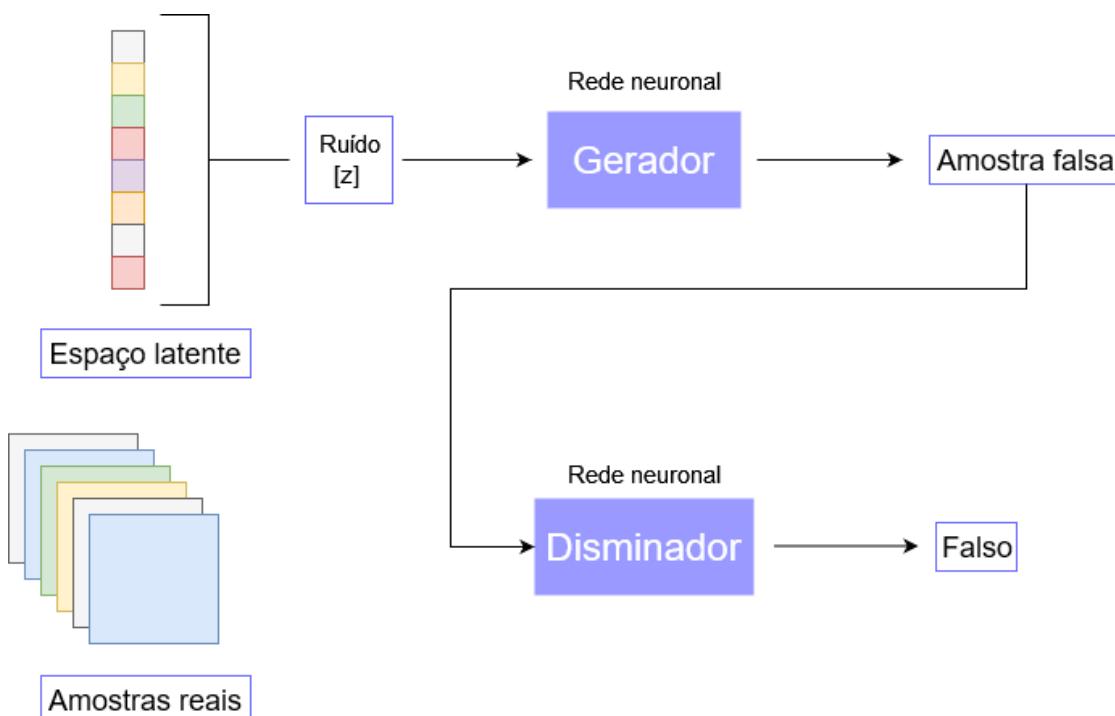


Figura 83 - funcionamento entre o gerador e o discriminador

Para criar uma rede do tipo GAN é necessário seguir os seguintes passos:

- definir o problema
- escolher qual será a arquitetura do tipo GAN que se pretende
- obter os dados reais para serem utilizados no discriminador
- utilizar o gerador para criar dados falsos
- treinar o discriminador utilizando os dados verdadeiros e falsos
- treinar o gerador utilizando a saída do discriminador

Quando é gerada uma imagem final, possivelmente foram geradas muitas outras imagens que não foram mostradas. Estas imagens são imagens que não conseguiram uma pontuação suficientemente alta para que o gerador as devolvesse. Contudo estes algoritmos possuem algumas limitações, porque a máquina está a treinar-se a si própria. Os dados utilizados nem sempre são exatos e o facto de o computador não compreender as imagens da mesma maneira que os humanos. Uma outra limitação é o facto de o computador não conseguir distinguir o que deve ou não deve criar.

Uma limitação desta tecnologia é a criatividade. Algumas imagens geradas nem sempre possuem a melhor qualidade. Isto porque a tecnologia ainda está a evoluir e a ser aperfeiçoada. Esta tecnologia pode ser utilizada para criar imagens para modelos 3D, publicidade, imagens para blogues, fazer arte entre outros. Foram realizados alguns testes na biblioteca `text2image` da pixray feito pelo link [3]. Os resultados obtidos foram os seguintes:



Figura 84 - imagem gerada com a palavra "Lisboa"



Figura 85 - imagem gerada com a palavra "Portugal"

Esta tecnologia tem a vantagem para os artistas de poder trabalhar mais rapidamente porque a máquina irá gerar o trabalho mais rapidamente. Desta forma poupa-se tempo e dinheiro e começa-se com uma ideia em vez de uma página em branco. Também possui a vantagem de poder inspirar os artistas para direções que não tinha sido possível sem a geração de imagens. Por exemplo, ideias para ícones podem escrever o texto e obter novas ideias. Contudo isto também provoca que menos pessoas sejam precisas para estes trabalhos. Sendo preciso menos designers, artistas e ilustradores.

Atualmente existe um mercado para obras de arte originadas por inteligência artificial. Tendo sido vendida uma obra de arte por 8000 dólares, que resultou da colaboração de um humano com uma máquina numa exposição feita pela Google. Uma outra peça de arte totalmente gerada por inteligência artificial do retrato de Edmond de Belamy foi leiloado por 610 000 dólares.



Figura 86 – retrato de Edmond de Belamy

3. Conclusões

Em suma, com a realização deste projeto foi possível adquirir os seguintes conhecimentos:

- Aprender de forma geral o que é uma rede neuronal e as suas capacidades para resolver determinados problemas.
- Aprendizagem de como é constituída uma rede neuronal, nomeadamente as camadas e a funcionalidade de cada camada.
- Utilização de bibliotecas de aprendizagem automática profunda para resolução de problemas.
- Aprender o que é aprendizagem por reforço e as capacidades deste algoritmo para resolver problemas não tendo qualquer conhecimento de como resolver o problema.
- Qual é a necessidade de definir a função das recompensas e as penalizações para a aprendizagem por reforço.
- Quais são os diferentes algoritmos de propagação do valor da política comportamental.
- Necessidade da definição das componentes de percepção e ação para o agente saber o ambiente que o rodeia e conseguir mover-se nele.
- De que maneira a política influencia o comportamento do agente por reforço.
- Necessidade de desenvolvimento da arquitetura do código de forma a explicar o código graficamente e controlar a entropia inerente do código.
- Aprender o que é a procura em espaço de estados e de que forma estes operam para resolver o problema da melhor forma possível, tendo em conta a solução ser ótima, completa, complexidade temporal e espacial.
- O que é uma função heurística e como esta é aplicada num problema de procura em espaço de estados.
- Entendimento de o que é a procura A* ponderada e de que forma o peso pode alterar a qualidade da solução.
- Entendimento de o que é o método frente-onda, e os custos em termos de memória

associados a este método.

- Entendimento de o que é um problema para otimização.
- Necessidades inerentes a um problema de otimização, nomeadamente o conceito de operador, problema, estado e solução.
- Algoritmos existentes para resolver problemas de otimização.
- Entendimento de como é feita a geração de imagens a partir de texto.
- Qual é a arquitetura da rede feita para a geração de imagens a partir de texto.

Em suma, neste trabalho foi possível concluir todas as etapas pretendidas para o desenvolvimento do trabalho. Estas etapas são desde a implementação do código até à criação da arquitetura do sistema. Em termos dos resultados estes parecem estar todos corretos, concluindo que o trabalho foi possível concluir com sucesso.

4. Bibliografia

- [1] Textos de apoio, Inteligência Artificial e Sistemas cognitivos, Luís Morgado, 2022
- [2] https://en.wikipedia.org/wiki/Software_engineering
- [3] <https://replicate.com/pixray/text2image>
- [4] <https://towardsdatascience.com/the-emerging-world-of-ai-generated-images-48228c697ee9>
- [5] <https://bernardmarr.com/artificial-intelligence-can-now-generate-amazing-images-what-does-this-mean-for-humans/>
- [6] <https://neptune.ai/blog/6-gan-architectures>