Relatório do Projeto 2 IA – Grupo 42

Redes Bayesianas

A parte 1 do projeto 2 de IA tinha como objetivo implementar os cálculos probabilísticos de uma rede Bayesiana. A nossa implementação obteve os resultados esperados em todos os testes que nos foram fornecidos, pelo que assumimos que está correta, mesmo não tendo sido testada exaustivamente.

Métodos implementados:

Na classe **Node** recebemos uma lista com os pais desse nó e as probabilidades desse nó ser verdadeiro, tendo em conta se os pais são verdadeiros ou falsos.
No método **computeProb** pretende-se computar a probabilidade desse nó tendo em conta uma certa evidência, que nos diz se cada um dos nós da rede é verdadeiro ou falso. Tendo em conta a evidência que nos é dada e os pais do nó, escolhemos a probabilidade correspondente a partir das probabilidades que nos são dadas ao inicializar o nó. A implementação foi feita de forma a que independentemente do número de pais, conseguimos sempre aceder à probabilidade correspondente. Não tem desvantagens nem limitações.
No método **computeJointProb** tendo em conta a independência entre os nós e a regra da cadeia das probabilidades, podemos calcular a probabilidade de uma evidência multiplicando as probabilidades de cada nó condicionado aos seus pais baseadas nessa evidência, ou seja, multiplicando entre si os computeProb's de cada nó para essa evidência. Não tem desvantagens, nem limitações.
No método **computePostprob** pretende-se computar a probabilidade de um certo nó ser verdadeiro, tendo em conta uma certa evidência que não contém os valores boleanos de todos os nós. Ex: considerando nós A, B, C, D calcular P(A|B,C). Para resolver o problema, teremos que somar todas as probabilidades para combinações possíveis de valores desconhecidos e conhecidos com A positivo e, dividir pela soma de todas as probabilidades para combinações possíveis de valores desconhecidos e conhecidos.

Análise das Complexidades:

- computeProb: O(n) -> sendo n o número de pais
- computeJointProb: O(mn) -> sendo m o número de nós e n o número de pais do nó com o maior número de pais
- computePostProb: O(m*(2**j)) -> sendo m o número de nós e j o número de nós cujo valor booleano na evidência pedida é desconhecido

Tendo em conta o problema dado, em que os testes serão realizados no máximo com 10 nós, a nossa solução seria capaz de calcular qualquer caso rapidamente. No pior caso, para o computePostProb, que tem a maior complexidade na nossa implementação, teríamos m=10 e j=9, pelo que O(10*(2**9))=O(5120), o que não justifica tentar encontrar soluções com complexidade não exponencial. No entanto, principalmente tendo em conta a complexidade do método computePostProb, a nossa implementação não seria eficiente em casos com redes Bayesianas com um maior número de nós.

Aprendizagem Por Reforço

A parte 2 do projeto 2 de IA tinha como objetivo implementar um robot que, ao interagir com o mundo, descobria a trajetória ótima, isto é, a trajetória que lhe desse a maior recompensa total. Para isto implementámos o algoritmo de Q-Learning, em que um robot pode interagir com um dado ambiente de dois modos diferentes: 'exploration' ou 'exploitation'. No modo 'exploration' o robot deve descobrir as 'regras' do ambiente, randomizando as suas ações para maximizar a descoberta de diferentes possíveis interações com o ambiente e as respetivas consequências (recompensas). No modo 'exploitation' o robot deve agir de forma a que obtenha a maior recompensa total possível, tendo em conta a informação que tem. A nossa implementação obteve os resultados esperados em todos os testes que nos foram fornecidos, pelo que assumimos que está correta, mesmo não tendo sido testada exaustivamente.

Métodos implementados:

Na classe **finiteMDP** recebemos um ambiente (estados, ações possíveis e recompensas). Nesta classe implementámos duas funções.
A função **traces2Q** completa a matriz Q para um ambiente, percorrendo uma lista de ações efetuadas e atualizando a matriz Q a cada ação. Isto é, para uma ação (estado inicial ei, ação a, estado seguinte es, recompensa r), atualiza a matriz Q de acordo com a fórmula:
Q(ei,a) = Q(ei, a) + alfa *(r + gama*max(Q(es)) – Q(ei,a))
Quando as diferenças totais na matriz após uma atualização dada uma ação forem menores que um valor (margem de erro permitida) definido pelo programador, a função termina e devolve a matriz Q mais recente. Esta matriz Q é então considerada uma aproximação do ambiente.
A função **policy** deve apenas retornar a ação a ser realizada por um robot, dado um estado e uma política. Existem duas opções: para a política 'exploration', o robot deve realizar uma ação randomizada, pelo que a função devolve um número aleatório entre os possíveis índices das ações para o estado atual do robot; para a política 'exploitation' a função devolve o índice da ação com a maior recompensa para o estado atual.
No 'mainRL.py', na função runPolicy alterámos o número de amostras para 10000 no caso da política 'exploration', de forma a que o robot tenha amostras suficientes para gerar uma aproximação dentro do previsto.

Análise das Complexidades:

- policy: O(nA) -> sendo nA o número de ações por estado
- traces2Q: O(indefinida) -> a complexidade da função traces2Q depende da margem de erro escolhida pelo programador, isto é, quanto menor a margem de erro, mais tempo este demorará a convergir para esta margem de erro. No nosso programa escolhemos '0.001'. A convergência para a margem de erro pode variar com o ambiente, o alfa escolhido e o número de amostras.

Sendo que notámos uma grande diferença no tempo de execução do exercício 1, decidimos mostrar as diferenças obtidas numa tabela:

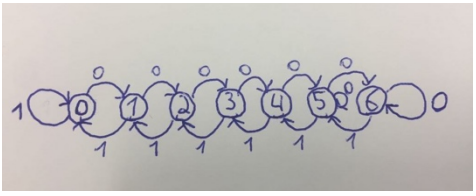
Variação do tempo do exercício 1 para diferentes alfas com gama constante:

Alfa	0.01	0.1	0.3	0.5	0.7	0.9
Tempo de execução (s)	2.28	0.51	0.35	0.29	0.29	0.28

Nota-se então que o alfa tem um grande impacto no tempo de convergência do exercício 1, sendo que pelos resultados obtidos deve ser escolhido um valor >0.5 para alfa, pois o tempo de execução do

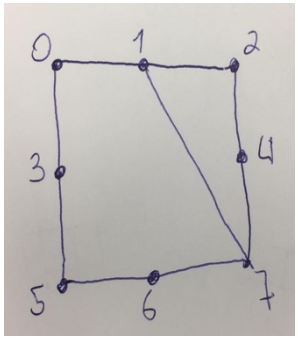
programa estabiliza a partir de alfa = 0.5. Deve-se também ter cuidado, pois para alfas elevados estamos a sobrevalorizar o impacto da ação mais recente, o que em certos casos pode levar a erros na aproximação de Q.
O exercício 2 teve um comportamento semelhante tendo em conta a variação dos valores de alfa.

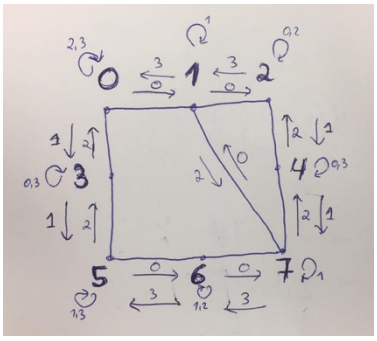
Descrição do ambiente 1:



O ambiente no exercício 1 é o mostrado na imagem ao lado.
Nos 6 estados é possível ir para o estado anterior com a ação '1' ou seg->uinte com a ação '0', isto é, no estado 3 é possível ir para o estado
2 com a ação '1' e para o estado 4 com a ação '0'. É de notar que nos dois estados extremo, 0 e 6, as ações '0' e '1' respetivamente, levam-nos a si próprios. No estado 5, a ação '0' é não-determinística: pode levar para o estado 5 ou 6, sendo que é mais provável ir para o estado 6 (p6=0.9 e p5=0.1).
Neste ambiente o robot recebe uma recompensa para qualquer ação realizada no estado 0 ou 6. Não recebe uma recompensa em qualquer outro caso.
Quando o robot recebe uma recompensa, volta sempre ao estado inicial, o estado 3. Este estado inicial é definido num dos argumentos da função runPolicy.
A política ótima é uma mistura da política de 'exploration' com a política de 'exploitation' de forma a que o robot efetue as ações que levam à maior recompensa possível. Para que o robot efetue a trajetória ótima terá que conhecer o ambiente, pelo que primeiro deve ser utilizada a política de 'exploration' com um número suficientemente grande de amostras. Neste ambiente, o robot deve-se movimentar sempre para o estado 0 a partir do estado inicial 3, sendo a sequência de ações o loop(3->2->1->0). Apesar de o estado 6 também dar a mesma recompensa que o estado 0, o robot ao passar no estado 5 pode não ir logo para o estado 6 com probabilidade de 0.1.

Descrição do ambiente 2:





O ambiente no exercício 2 é o mostrado nas imagens ao lado. Todas as ações possíveis são determinísticas, ou seja, para cada ação existe apenas um estado seguinte tendo em conta o estado inicial. A forma geral do ambiente pode ser descrita por um quadrado com um túnel entre os estados 1 e 7.
Neste ambiente o robot é penalizado por todas as ações exceto aquelas cujo estado inicial é o 7, isto é, tem uma penalização de -1 por todas as ações exceto aquelas no estado 7, que têm uma 'penalização' de 0.
Sendo assim, a política ótima neste ambiente é chegar ao estado 7 no menor número de ações possível e repetir continuamente a ação 1 para permanecer neste estado. Desta forma o robot apenas é penalizado até chegar ao estado 7.
Um exemplo de trajetória ótima, começando no estado 0 é: 0->1->7, com uma penalização de -2. Pode haver mais que uma trajetória ótima, como por exemplo se o estado inicial for o estado 2: 2->4->7 ou 2->1->7, ambos com penalização de -2.