Projecto de Inteligência Artificial

Grupo 7
86466 – Madalena Pedreira
86496 – Pedro Custódio

O relatório que se segue está dividido em duas partes: a primeira, relativa à secção de *Redes Bayesianas* e, a segunda, ao agente de aprendizagem *Q-learning*.

Parte 1: Redes Bayesianas

Correndo o exercício no mainBN.py os resultados coincidiram com o output esperado. Este é resultado da implementação de 2 classes e de um conjunto de várias funções, que serão brevemente discutidas.

A primeira, na classe *Node*, (a função *computeProb*) recebe uma lista de probabilidades para um determinado nó e um conjunto de evidências. Dados os parâmetros, a função devolve para esse mesmo nó um *array* de probabilidades com dois elementos [<*prob. nó ser False*> , <*prob. nó ser True*>]. Esta forma de cálculo não tem em conta o nó que está a ser tratado, na medida em que as evidências dadas têm informação sobre nós que não são necessários (nós que não têm ligação direta ao nó que está a ser tratado) e como tal são desnecessárias de passar como um todo em cada uma das funções. Uma solução deste tipo seria incomportável para uma rede maior e é, de qualquer das formas, um gasto de memória. Seria mais prudente dar apenas os dados de variáveis não independentes em cada nó. Quanto à sua complexidade temporal[1], tem apenas de percorrer o vetor de pais uma vez, tendo este um tamanho máximo de dois[2] pelo que é constante (**O(2)**).

A classe *BN*, por sua vez, guarda informação duplicada, uma vez que encerra uma representação do grafo com os diferentes pais dos elementos, informação essa que poderiam ser consultados na estrutura de nós que também vão encapsular. Poder-se-ia, portanto, dispensar o seu atributo *gra* mantendo os nós.

Quanto à função *computeJointProb,* da classe *BN*, a soma dos resultados da mesma aplicada a todos os nós do grafo concorda com a propriedade probabilística de que a soma do universo das probabilidades de cada elemento de um conjunto é igual a 1. A sua complexidade é linear e depende do número de nós no grafo (**O(n)**, n- nós no grafo).

Já a função computePostProb, que dada uma evidência extrai a probabilidade esperada num determinado nó, também opera positivamente. Com a informação disponível nos nós, necessita de interpretar a lista de evidencias e fazer o levantamento das várias possibilidades de probabilidades caso os nós desconhecidos de input ou o próprio nó para o qual se pretende fazer o cálculo nó seja *True* ou *False*. Esta função usa como auxiliar a função *computeJointProb* para a combinação de evidência gerada, de maneira que a sua complexidade temporal vai ser **O(n) + O(i * e)** , i – combinações parâmetros desconhecidos; e – evidências.

Face aos exemplos dados, podemos concluir que apesar do output ser correto, para redes de maior dimensão ter-se-iam de pensar em estratégias mais condensadas de representação de evidências e da própria rede. Para além disso, como nós independentes, por definição, não têm probabilidades condicionadas de variáveis do grafo, poder-se-iam poupar quaisquer cálculos de probabilidade para nós que não tivessem pais, melhorando a eficiência do algoritmo.

Parte 2 – Aprendizagem com Agente Q-Learning

O agente *Q-learning* vai, sem modelo, interagir num dado ambiente e procura chegar através de um conjunto de funções a uma política ótima. A política ótima é guardada numa matriz Q atualizada um n número de vezes pela função *runPolicy*. Foram usadas para o estudo um n de 10000 iterações. Quantas mais vezes forem feitas iterações mais próximo convergirá o agente de valores de política mais corretos.

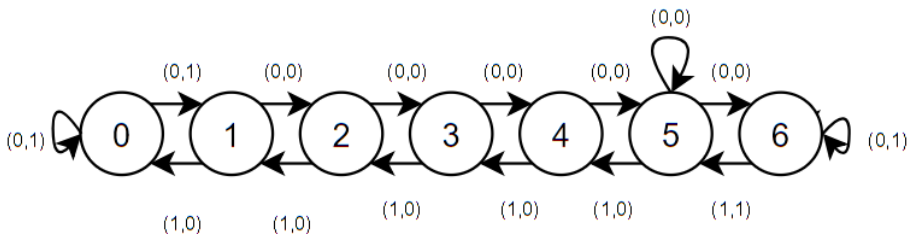
Seguem-se as representações dos ambientes fornecidos para o estudo com uma máquina de estados. Em cada transição está um par de valores que representam (<*ação*>, <*recompensa*>), tal que *ação* é a ação que o agente tem de escolher para transitar para o estado e *recompensa*, o valor que ganha ao efetuar essa transição.

Sublinha-se a particularidade do primeiro ambiente não ser determinístico, o que significa que ao efetuar uma escolha, o agente tem associada uma probabilidade de realmente a executar (*array Pl* dá estes valores). Este facto reflete-se na forma como o

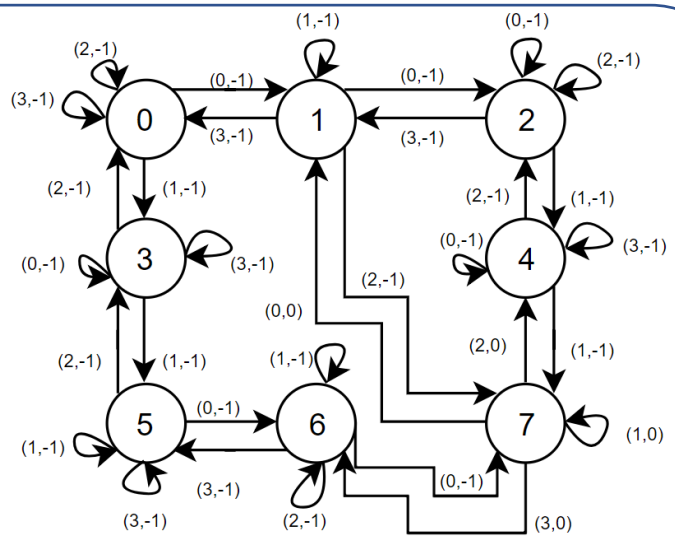
[1] será sempre feita uma abordagem de pior caso na análise de complexidades
[2] cada nó só tem 2 pais, no máximo - propriedades das redes Bayesianas

agente vai aprender a política, já que este apesar de escolher uma ação pode chegar a um estado diferente do esperado e, portanto, dificultar a associação de uma consequência de ação a um determinado estado de recompensa (maior medidor de motivação de aprendizagem para o agente), já que é tida em conta a ação escolhida antes ainda de ser executada.

Ambiente 1



Ambiente 2

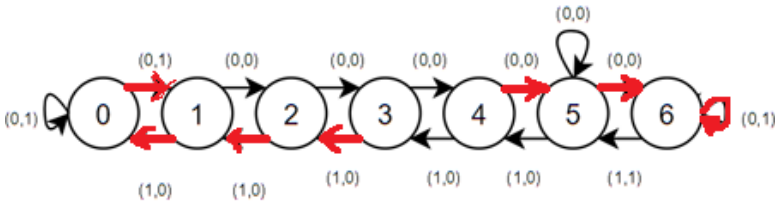


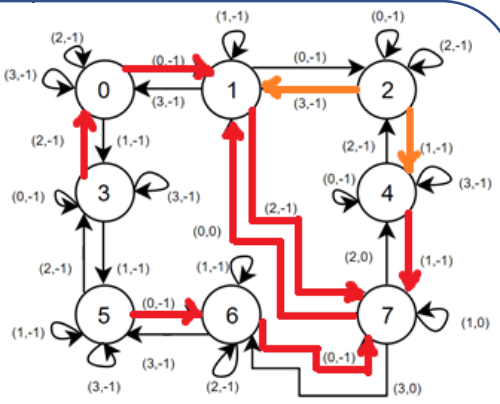
A função de recompensa é dada pela análise de cada estado da trajetória - para cada ação executada no ambiente, está previsto um valor de recompensa na transição para o estado. Na representação dos dados no projeto, este valor de recompensa consistia no 4º elemento do *array* representativo da trajetória (no ambiente 2) ou nos parâmetros de input do *array Rl* (no ambiente 1). Estados mais aprazíveis têm valores de recompensa maiores e estados menos úteis valores mais baixos. O agente é encorajado a maximizar um valor final de recompensa que vai atualizando pela soma das recompensas nos vários estados.

Quanto à determinação da política ótima e sabendo, por definição, que é a política que procura retornar o melhor valor em cada estado executando a ação que levará à maximização da recompensa, então a política ótima para cada um dos ambientes com o numero de iterações dadas é dada pela matriz Q que assume os seguintes valores para cada um dos ambientes sobre análise:

Ambiente 1		Ambiente 2			
Q =	[[9.09999212	9.99999137]	Q= [[-1.9	-3.43899984	-2.70999993	-2.71]
	[7.28999304	8.99999189]	[-2.71	-1.9	-1.	-2.71]
	[6.56099383	8.09999251]	[-2.71	-1.9	-2.71	-1.9]
	[7.22948596	7.28999323]	[-3.4389412	-2.71	-2.70999997	-3.43789128]
	[8.01656072	6.56099361]	[-1.9	-1.	-2.71	-1.9]
	[8.91315968	7.23206459]	[-1.9	-2.70999999	-3.43899994	-2.71]
	[9.99999583	9.01578061]]	[-1.	-1.9	-1.9	-2.71]
			[-0.9	0.	-0.9	-0.9]]

Por interpretação dos dados da matriz, vendo em cada linha (estado) qual a ação (0 ou 1, no caso do primeiro ambiente e 0,1,2 ou 3 no segundo ambiente) que tem maior recompensa, podemos concluir que a política ótima em cada um é:





Legenda: Política Ótima para o Ambiente 1 e 2.
A encarnado está a escolha da política, sendo que a laranja, no Ambiente 2, foram representadas as duas alternativas de ação para esse estado (têm o mesmo valor Q associado)

A matriz Q é responsável por registar em cada entrada (cada uma corresponde a um estado e uma ação que pode ser feita no estado) uma ponderação de valores resultantes da execução dessa ação no estado. Quando o agente se movimenta, ou seja, escolhe num estado executar uma determinada ação, este vai recalcular a entrada correspondente a essa transição na matriz Q. Heis a forma como é feita a atualização da matriz:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$$ *, s-estado; a- ação executada*

Esta revisão é aplicada pela função *traces2Q* do projeto, discutida mais à frente. A entrada (s,a) é uma ponderação de vários fatores, sendo especialmente relevantes de abordar a parcela de ponderação *alfa e* um parâmetro *gama*, sendo que R(s), a função de recompensa, já foi abordada.

O parâmetro *alfa* ($0 \leq \alpha \leq 1$) vai medir o peso que o agente deve dar, ao aprender a função de política ótima, às experiências passadas consultadas na matriz Q aquando a sua atualização. O valor usado foi de 0.2 no projeto tendendo a beneficiar experiências recentes. Quanto mais próximo de 0 for o valor de alfa, maior terá de ser o número de iterações para chegar a uma política ótima confortável, uma vez que beneficiando mais experiências recentes, valores que podem divergir da norma que se pretende encontrar em termos de dados, podem influenciar fortemente os cálculos, necessitando de se amortizar estas discordâncias com o maior número de acontecimentos possíveis.

Em relação ao parâmetro *gama* ($0 \leq \gamma \leq 1$), este determina o desconto que se deve dar a recompensas futuras. Nos testes, este foi de 0.9. Quanto mais próximo de 1 - como se verifica – mais tendencioso fica o agente a fazer escolhas que apesar de terem uma menor recompensa no imediato o poderão levar a estados de recompensa maiores.

Quanto aos mecanismos que vão mimicar estes conceitos, temos a implementação de uma classe finiteMDP e várias funções.

A classe finiteMDP contem, para além de outros parâmetros, a função Q. Poder-se-iam dispensar alguns deles, uniformizando a obtenção da informação dos ambientes, na medida em que as recompensas e valores de probabilidade poderiam provir todos da mesma fonte de input de um ficheiro com a trajetória.

Existe uma função *runPolicy*, que corre um dado número de iterações **n** uma política dada como input também, aplicada a um agente finiteMDP. A função recorre a *policy*, que por sua vez especifica duas abordagens de políticas: *exploration* (escolha aleatória de uma ação para um estado na esperança de chegar a estados com maiores valores de recompensa, ainda que não óbvios) e *exploitation* (escolha de ações que se mostraram mais promissoras em experiências passadas em termos de obtenção de recompensa). Na primeira, a complexidade será de **O(1)** e na segunda **O(a),** em que **a** é o número de ações disponíveis em cada estado para o agente. Assim sendo, assumindo o pior caso para a função *runPolicy* (*exploitation*) este terá complexidade **O(n*a)**, sendo n é o número de iterações e a o tal número de ações em cada estado.

Já a função *traces2Q* vai percorrer o *array* de trajetória tantas vezes quantas a norma da diferença da matriz Q com a matriz inicial ultrapasse um valor específico. Assumindo que vai demorar N iterações para convergir, a complexidade será **O(N*nA+nE),** sendo nA sendo o número de ações e nE o número de exemplos.

Uma alternativa à forma de atualização da matriz Q seria a de implementar um modelo SARSA. Através dessa abordagem, o agente estaria a escolher o valor da entrada Q não em função da ação escolhida, mas em função da ação que realmente foi efetuada no estado, beneficiando de um comportamento mais realista em termos de resultados (ainda que possivelmente piores) e um tratamento do caso do primeiro ambiente mais em conta à sua particularidade não determinística. Uma vez que o agente pode, no ambiente 1, apesar de escolher uma ação não a efetuar, casos deste género seriam absorvidos, já que mesmo que o agente não cumprisse a escolha de ação que teria escolhido, seria só contabilizada a que realmente foi consumada, dando maior robustez ao modelo e, consequentemente, resultados mais fidedignos para o mesmo. A abordagem feita no projeto é, portanto, *off-policy:* como a matriz retém o melhor valor de Q, não tem em grande consideração a política que está a ser seguida.

Em jeito de conclusão, o agente *Q-learning* poderia comportar-se melhor sabendo o tipo de ambiente em que opera *a priori*, podendo especializar o algoritmo e a política segundo as particularidades que cada um apresenta. Não sendo fornecidas pistas do meio de atuação, pode-se dizer que o agente vai na mesma encontrar uma política ótima para um ambiente, indo em cada estado atualizar a sua matriz Q, auxiliar nesta busca, e apresentar valores para a mesma que parecem coincidir com as condições do problema.