Inteligência Artificial
Profa. Heloisa Camargo
Breno da Silveira Souza - 551481
Gabriela Vanessa Pereira Alves Mattos - 551570

Documentação do trabalho 1 de Inteligência Artificial

Neste trabalho foi implementado o algoritmo de busca heurística A aplicado ao problema do mundo do aspirador de pó.

- **Representação do Problema:**

O problema do mundo do aspirador de pó consiste de um cenário representado por uma grade 2X2, onde cada quadrado pode conter sujeira ou não. O objetivo é que a grade ao final esteja limpa, e que seja encontrada a solução com menor custo no caminho, observando para isso o custo de cada ação e a função heurística aplicada ao estado.

Considerando a seguinte grade:

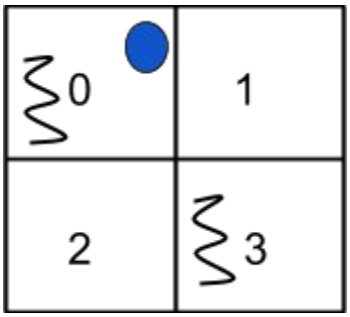
0	1
2	3

Nela, temos um identificador para cada posição: um número inteiro entre 0 e 3.

Os estados em Prolog foram implementados a partir de uma lista composta por 5 elementos: elemento inicial determina a posição na grade em que se encontra o aspirador, e os outros quatro elementos representam cada uma das posições da grade, sendo que esses elementos podem assumir dois valores:

- 0: significa que o quadrado dessa posição (no caso, posição - 1, pois o elemento 0 da lista representa a posição do aspirador na grade, assim , o elemento 1 na verdade se refere ao quadrado 0, o elemento 2 se refere ao quadrado 1, o elemento 3 ao quadrado 2 e o elemento 4 ao quadrado 3) está limpo;
- 1: o quadrado está sujo.

Por exemplo, a representação do seguinte estado, no qual o círculo azul representa o aspirador e o rabisco representa a sujeira será da seguinte forma em uma lista em Prolog:



[0, 1, 0, 0, 1]

O aspirador se encontra na posição 0, a posição 0 e a posição 3 estão sujas, as posições 1 e 2 estão limpas.

As ações possíveis são mover e aspirar, sendo que o aspirador pode apenas mover-se para a direita e esquerda ou cima e baixo. Cada ação tem um custo específico.

Para o trabalho, foi criada uma base de dados com todos os movimentos possíveis, representados pelo predicado move.

- **Alguns movimentos possíveis presentes na base de dados:**

move([0,1,1,1,1], [0,0,1,1,1]).

O aspirador se encontra na posição 0 da grade e todas as posições estão sujas. A ação executada foi limpar a posição em que ele se encontra, resultando no estado em que o aspirador se encontra na posição 0, e a posição 0 está limpa, enquanto que as outras continuam sujas.

move([0,0,1,1,1], [2,0,1,1,1]).

O aspirador se encontra na posição 0 da grade e essa posição já está limpa, enquanto que as outras estão sujas. A ação executada foi mover para baixo, colocando o aspirador na posição 2.

move([1,0,1,1,1], [0,0,1,1,1]).

O aspirador se encontra na posição 1 da grade e a posição 0 já está limpa, enquanto que as outras estão sujas. A ação executada foi mover para a esquerda, colocando o aspirador na posição 0.

- **Alguns movimentos não permitidos que não estão representados na base de dados:**

move([0,1,1,1,1], [0,1,0,1,1]).

O aspirador se encontra na posição 0 e todas as posições estão sujas. No estado seguinte, o aspirador ainda se encontra na posição 0, mas a posição 1 está limpa. Não é possível essa transição, pois para limpar uma posição, o aspirador deve estar presente nessa posição no estado anterior.

move([3,1,1,1,1], [0,1,1,1,1]).

O aspirador se encontra na posição 3 e todas as posições estão sujas. No estado seguinte, ele se encontra na posição 0. Esse movimento não é permitido, pois é um movimento em diagonal.

- **Avaliação do Custo e da Heurística:**

Para determinar o custo de cada ação e a heurística de cada estado foram definidos dois predicados: calculaG e calculaH.

O predicado calculaG se baseia apenas no primeiro elemento da lista, observando as possíveis transições entre os valores desse elemento e atribuindo o custo dessa transição a uma variável.
Por exemplo:

calculaG([A|Y],[C|Z],B,D):-D is B + 1, A == C, !.

Se o primeiro elemento de ambas as listas forem iguais, isso significa que nessa transição ocorreu a ação de limpeza. Desse modo, o custo total é acrescido de 1.

calculaG([A|Y],[C|Z],B,D):-D is B + 2, A == 1, C == 0, !.

Se o primeiro elemento da primeira lista é 1 e o primeiro elemento da segunda lista é 0, então a ação foi um movimento para a esquerda. Desse modo, o custo total é acrescido de 2. O mesmo vale para os movimentos similares.

calculaG([A|Y],[C|Z],B,D):-D is B + 3, A == 0, C == 2, !.

Se o primeiro elemento da primeira lista é 0 e o primeiro elemento da segunda lista é 2, então a ação foi um movimento para a baixo. Desse modo, o custo total é acrescido de 3. O mesmo vale para os movimentos similares.

O predicado calculaH analisa, em cada lista que representa o estado, a quantidade de números 1 presentes nessa lista, do segundo elemento em diante.

calculaH([X|Y],H):-conta1s(Y,H).

Como pode ser observado, a contagem de números 1 é feita por um predicado auxiliar, e é aplicada a cauda da lista que representa o estado.

- **Algoritmo de busca best-first:**

Algumas modificações foram feitas no algoritmo que foi disponibilizado para que seu comportamento fosse equivalente ao comportamento do algoritmo de busca best-first. A primeira delas foi a retirada das seguintes condições do predicado moves:

```
not(member([A,_,_,_], T)),
not(member([A,_,_,_], C)),
```

Essas condições impediam que estados que já tivessem sido gerados anteriormente fossem gerados novamente. Dessa maneira não seria possível encontrar um caminho melhor se esse caminho melhor passasse por algum estado que já estivesse em Open ou em Closed.

Após os estados serem gerados novas verificações foram adicionadas ao predicado best-first. Essas verificações tem o intuito de verificar se algum dos estados gerados já está presente ou na lista Closed ou na lista Open. Se ele já estiver é necessário verificar o custo para chegar nesse estado e fazer as alterações necessárias.

A seguir o código do best-first, com destaque aos novos predicados adicionados:

```
best_first([[F,D,H,S]|T], C, [_,_,_,G]) :-
```

```
write('open: '), printlist([ S | T ] ),nl,
write('closed: '), printlist(C),nl,
findall(X, moves([F,D,H,S], T, C, X), List),
write('nos gerados: '),printlist(List),nl,
verificaOpen(List, T, NT),
verificaClosed(List, C, NC,NT,NewT),
verificaIgual(NewT, List, NList),
mergeList(NewT, NList, NewList),
sort(NewList,O),
best_first(O, [[F,D,H,S] | NC], [_,_,_,G]).
```

O predicado verificaOpen recebe como parâmetro a lista de estados gerados (List), a lista de nós que estão em Open (T), e gera uma nova lista de nós abertos, a NT.

A lógica utilizada no verificaOpen é a seguinte:

verificaOpen([ ],T,T):-!.

A lista que será usada para fazer a recursão é a lista de nós gerados que é passada no primeiro parâmetro. Quando essa lista estiver vazia, todos os elementos dela foram analisados e a lista resultante nessa resursão será uma cópia da lista do segundo parâmetro.

verificaOpen([[A,G1,H1,B]|Z],T,Y):- member([_,_,_,B],T), troca([A,G1,H1,B],T,E), verificaOpen(Z, E, Y),!.

Se a primeira lista não for vazia, é verificado se a representação do estado contido no nó do primeiro elemento dessa lista está presente em T. Se estiver, o predicado *troca,* que faz a troca se o valor de custo (variável A) for menor que o valor de custo que está associado ao estado em Open, é chamado, e realiza essa troca, retornando uma nova lista de abertos Open representada na variável E. Depois o predicado é chamado para a cauda da primeira lista, passando o novo Open como parâmetro.

verificaOpen([[A,_,_,B]|Z],T,Y):-not(member([_,_,_,B],T)), verificaOpen(Z, T, Y).

Se o estado não está em Open, então continua-se a verificação com a cauda da primeira lista para ver se os outros estados estão em Open.

Predicado troca:

troca([A,G1,H1,B],[[C,_,_,B]|Z],[[A,G1,H1,B]|Z]):-B==B,A<C.
troca([A,_,_,B],[[C,G1,H1,B]|Z],[[C,G1,H1,B]|Z]):-B==B, A>=C.
troca([A,G1,H1,B],[[C,G2,H2,D]|Z],[[C,G2,H2,D]|W]):-B\==D,troca([A,G1,H1,B], Z, W).

Como pode ser observado, ocorre a troca dos nós apenas quando o custo associado ao estado é menor, se não, o valor continua o mesmo.

A lógica utilizada no verificaClosed é a seguinte:

Após feita a verificação na lista Open, e gerado uma nova lista de nós Open, é feita uma verificação na lista closed, para gerar uma nova lista Open novamente e uma nova lista Closed. Se o estado gerado estiver presente em Closed, e tiver uma avaliação melhor do que a avaliação que tinha anteriormente, então ele deve ser retirado de Closed e recolocado em Open, com esse novo valor de avaliação.

verificaClosed([],C,C,T,T).

Assim como no anterior, a lista de nós gerados é utilizada para a recursão, sendo que a condição de parada acontece quando essa lista estiver vazia. É importante notar que não é possível que um mesmo estado esteja em Open ou Closed ao mesmo tempo, mas o teste é feito com a mesma lista de nós gerados para ambos predicados por simplificação.

verificaClosed([[A,G1,H1,B]|Z], C, D, T, [[A,G1,H1,B]|Cauda]):-member([_,_,_,B],C), ehMenor([A,_,_,B],C), del([A,_,_,B],C,Y), verificaClosed(Z,Y,D,T,Cauda).

Nesse caso, O novo Open (representado pela última lista) vai ser composto pelo nó gerado, se for verdade que ele pertence a Closed seu custo é menor que o custo armazenado em Closed (verificação realizada pelo predicado auxiliar ehMenor). Esse estado então será eliminado da lista closed, pelo del, e a recursão será chamada para a cauda da lista

verificaClosed([[A,_,_,B]|Z], C, D, T, Cauda):-member([_,_,_,B],C), not(ehMenor([A,_,_,B],C)), verificaClosed(Z, C, D, T, Cauda) .

Se o primeiro estado da lista de nós gerados for membro da lista de fechados, mas se o custo para chegar a esse estado não for menor que o que está armazenado na lista, então não se modifica nada na lista Closed ou na

lista Open, e a recursão é chamada para a cauda da lista de nós gerados, para continuar as verificações com os outros nós que foram gerados nessa etapa.

verificaClosed([[A,_,_,B]|Z], C, D, T,Cauda):-not(member([_,_,_,B],C)), verificaClosed(Z, C, D, T, Cauda).

A última possibilidade em verificaClosed, é o caso de quando o estado gerado não pertence a lista Closed. Nesse caso, então, deve apenas ser chamada a recursão para a cauda dos nós gerados, para que se continue as comparações.

Predicado ehMenor:

```
ehMenor([A,_,_,B],[[C,_,_,B]|T]):-A<C.
ehMenor([A,_,_,B],[[_,_,_,D]|T]):-B\==D,ehMenor([A,_,_,B],T).
```

O predicado ehMenor tem apenas o objetivo de retornar verdadeiro ou falso. Desse modo, um nó é passado no primeiro argumento e uma lista no segundo. É feita a recursão nessa lista até que se encontre o nó com o estado equivalente. Encontrando esse nó, o primeiro elemento desse nó é comparado ao primeiro elemento do nó passado, e ele retornará verdadeiro se esse primeiro elemento for menor que o valor do primeiro elemento do nó passado.

Depois de realizada a filtragem nas listas Open e Closed, é necessário realizar uma filtragem na lista de nós gerados naquele momento. A lista de nós Open, representada pela variável NewT, foi preenchida com todos os estados que já estavam em Open ou em Closed e tiveram alguma avaliação melhor, se teve alguma avaliação pior, então o estado não é modificado nessas listas. Então para cada estado da lista de nós gerados, é verificado se ele já está presente tanto em Open (no caso de ele já existir em Open anteriormente, e ter uma avaliação melhor ou pior, ou no caso de ele já existir anteriormente em Closed e ter tido uma avaliação melhor), quanto em Closed (no caso de o estado já existir anteriormente em Closed e não ter tido uma avaliação melhor).

A seguir, a lógica do predicado verificaIgual:

verificaIgual(NT, [], []).

A lista usada na recursão é novamente a lista de nós gerados, quando ela for vazia, resultará em uma lista vazia.

verificaIgual(NT, [[_,_,_,A]|B], Z) :- member([_,_,_,A], NT), verificaIgual(NT,B,Z).

Se ela não for vazia e possuir um primeiro elemento, o estado desse elemento é procurado na lista NT (que na chamada pode ser tanto Open, quanto Closed). Se ele estiver presente, esse estado não é colocado na lista de retorno, e é realizada a mesma operação para a cauda da lista de nós gerados, retornando Z.

verificaIgual(NT, [[F1,G1,H1,A]|B], [[F1,G1,H1,A]|Z]) :- not(member([_,_,_,A], NT)), verificaIgual(NT,B,Z).

A última possibilidade acontece quando o elemento da lista de nós gerados não está na lista que foi passada para comparação. Nesse caso o elemento deve ser mantido na lista, e a recursão é chamada pra cauda para verificar os outros elementos da lista.

- **Aplicação do algoritmo em um exemplo:**

A seguir, a representação da execução do algoritmo, para o estado inicial [0,0,0,1,1] e o estado final desejado [3,0,0,0,0]. Esse exemplo mostra quais estados estão sendo expandidos, os estados possíveis de serem gerados a partir do estado expandido, e aqueles que efetivamente são gerados. Logo em seguida, encontra-se a representação gráfica da aplicação desse algoritmo.

| best_first([[2,0,2,[0,0,0,1,1]]], [ ], [T,D,B,[3,0,0,0,0]]).
open: [0,0,0,1,1]

closed:
nos gerados: [4,2,2,[1,0,0,1,1]]
[5,3,2,[2,0,0,1,1]]

open: [1,0,0,1,1]
[5,3,2,[2,0,0,1,1]]

closed: [2,0,2,[0,0,0,1,1]]

nos gerados: [6,4,2,[0,0,0,1,1]]
[7,5,2,[3,0,0,1,1]]

open: [2,0,0,1,1]

[7,5,2,[3,0,0,1,1]]

closed: [4,2,2,[1,0,0,1,1]]
[2,0,2,[0,0,0,1,1]]

nos gerados: [5,4,1,[2,0,0,0,1]]
[7,5,2,[3,0,0,1,1]]
[8,6,2,[0,0,0,1,1]]

open: [2,0,0,0,1]
[7,5,2,[3,0,0,1,1]]

closed: [5,3,2,[2,0,0,1,1]]
[4,2,2,[1,0,0,1,1]]
[2,0,2,[0,0,0,1,1]]

nos gerados: [7,6,1,[3,0,0,0,1]]
[8,7,1,[0,0,0,0,1]]

open: [3,0,0,1,1]
[7,6,1,[3,0,0,0,1]]
[8,7,1,[0,0,0,0,1]]

closed: [5,4,1,[2,0,0,0,1]]
[5,3,2,[2,0,0,1,1]]
[4,2,2,[1,0,0,1,1]]
[2,0,2,[0,0,0,1,1]]

nos gerados: [7,6,1,[3,0,0,1,0]]
[9,7,2,[2,0,0,1,1]]
[10,8,2,[1,0,0,1,1]]

open: [3,0,0,0,1]
[7,6,1,[3,0,0,1,0]]
[8,7,1,[0,0,0,0,1]]

closed: [7,5,2,[3,0,0,1,1]]
[5,4,1,[2,0,0,0,1]]
[5,3,2,[2,0,0,1,1]]
[4,2,2,[1,0,0,1,1]]
[2,0,2,[0,0,0,1,1]]

nos gerados: [7,7,0,[3,0,0,0,0]]
[9,8,1,[2,0,0,0,1]]
[10,9,1,[1,0,0,0,1]]

open: [3,0,0,1,0]
[7,7,0,[3,0,0,0,0]]
[8,7,1,[0,0,0,0,1]]
[10,9,1,[1,0,0,0,1]]

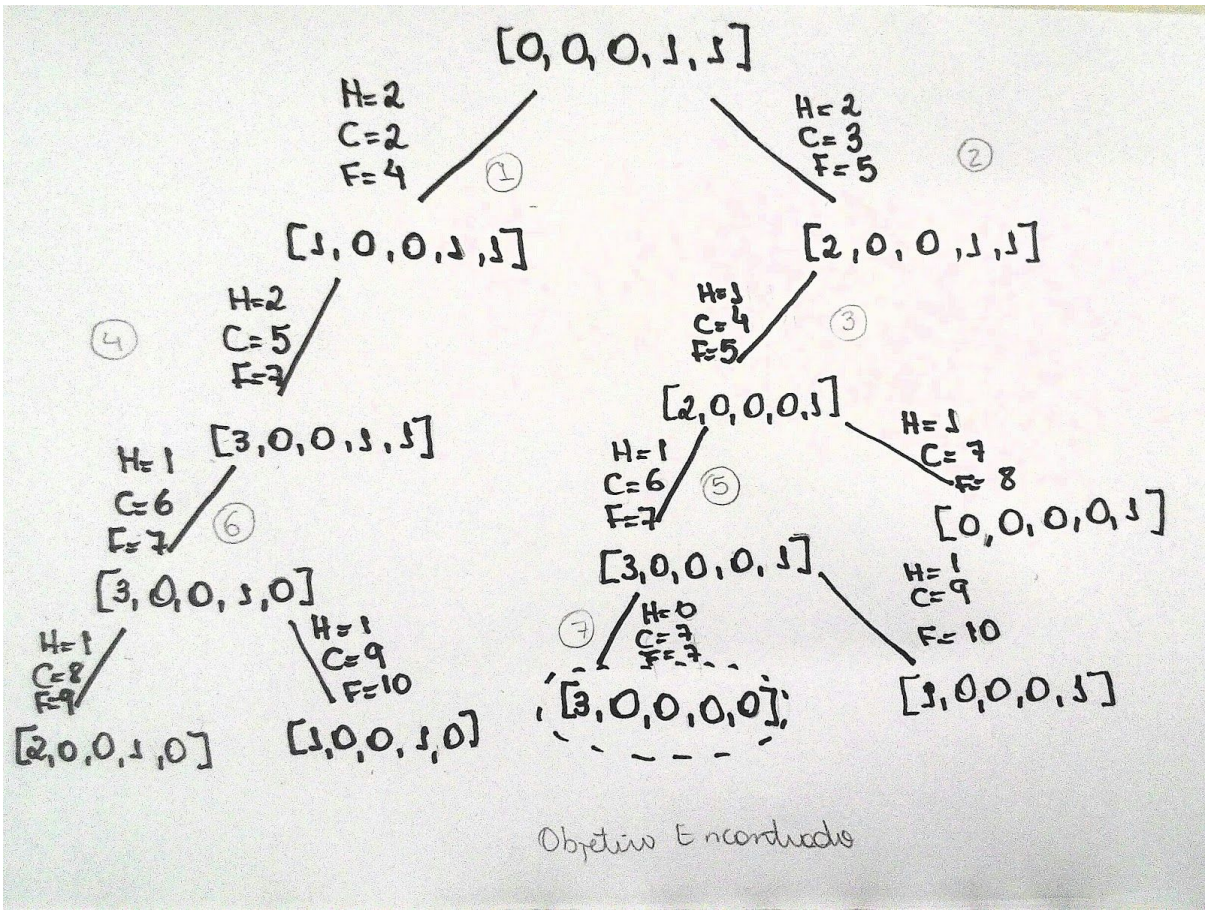
closed: [7,6,1,[3,0,0,0,1]]
[7,5,2,[3,0,0,1,1]]
[5,4,1,[2,0,0,0,1]]
[5,3,2,[2,0,0,1,1]]
[4,2,2,[1,0,0,1,1]]
[2,0,2,[0,0,0,1,1]]

nos gerados: [9,8,1,[2,0,0,1,0]]
[10,9,1,[1,0,0,1,0]]

open: [3,0,0,0,0]
[8,7,1,[0,0,0,0,1]]
[9,8,1,[2,0,0,1,0]]
[10,9,1,[1,0,0,0,1]]
[10,9,1,[1,0,0,1,0]]

closed: [7,6,1,[3,0,0,1,0]]
[7,6,1,[3,0,0,0,1]]
[7,5,2,[3,0,0,1,1]]
[5,4,1,[2,0,0,0,1]]
[5,3,2,[2,0,0,1,1]]
[4,2,2,[1,0,0,1,1]]
[2,0,2,[0,0,0,1,1]]

OBJETIVO: [3,0,0,0,0]
Solução encontrada



A ordem das ações pode ser resumida da seguinte forma:

Expande [0,0,0,1,1]
Os nós gerados são[1,0,0,1,1] e [2,0,0,1,1]
O próximo nó escolhido para ser expandido é o [1,0,0,1,1], pois seu custo de avaliação é menor.
Expande [1,0,0,1,1]
Os nós gerados são [3,0,0,1,1] e [0,0,0,1,1]. O estado gerado é o [3,0,0,1,1], pois o outro estado já está em Closed, e o valor de avaliação não foi menor.
O próximo nó escolhido para ser expandido é o [2,0,0,1,1], pois seu custo de avaliação é menor.
Expande[2,0,0,1,1]
Os nós gerados são [0,0,0,1,1], [2,0,0,0,1] e [3,0,0,1,1]. O único que é efetivamente gerado é [2,0,0,0,1], pois o [0,0,0,1,1] já está em em Closed e o valor da avaliação não é menor, e o estado [3,0,0,1,1] já está em Open, e o valor da avaliação também não é menor, o que faz com que seu pai continue sendo o nó [1,0,0,1,1].
Continuando esse raciocínio, temos de forma resumida:
Expande [3,0,0,1,1]
Nós gerados: [3,0,0,1,0]

Expande [2,0,0,0,1]
Nós gerados: [3,0,0,0,1],[0,0,0,0,1]

Expande [3,0,0,1,0]
Nós gerados: [2,0,0,1,0],[1,0,0,1,0]

Expande [3,0,0,0,1]
Nós gerados: [3,0,0,0,0],[1,0,0,0,1]

Achou a solução! Sucesso.