Universidade Federal de São Carlos
Disciplina de Inteligência Artificial
Trabalho 1

Resolução de problema por busca: “mundo do aspirador de pó”

Wilton Vicente Gonçalves da Cruz RA:586889
Inteligência Artificial
2º semestre de 2016
Ciência da Computação

1. Introdução

Na Inteligência Artificial, uma das formas de se resolver um determinado problema é através de mecanismos de busca. Assim, dado um determinado problema, deve-se modelá-lo de forma a se explicitar informações como a representação de seus estados mediante à aplicação de determinados operadores, custos associados à aplicação destas operações, estado inicial e o estado objetivo para a resolução do problema.

Após a modelagem do problema, deve-se aplicar uma busca sobre o espaço de estados do problema modelado. As estratégias de busca, porém, podem ser divididas em dois grandes grupos: busca desinformada e busca informada. A principal diferença entre os algoritmos integrantes destas duas categorias está no critério utilizado para a seleção do próximo nó a ser expandido. Basicamente, em algoritmos de busca desinformada, como busca em largura e busca em profundidade, não se tem informações adicionais sobre o problema, prevalecendo a ordem dos nós como critério para expansão. Já em algoritmos de busca informada, como o “greedy” e o “A*”, têm-se informações prévias sobre o problema representadas através de heurísticas.

Neste trabalho, será apresentada um possível resolução para um determinado problema proposto. Será apresentada uma possível modelagem para o mesmo, bem como os resultados obtidos a partir da aplicação de algoritmos de busca informada e desinformada sobre o espaço de estados do problema.

2. Desenvolvimento

2.1 Definição do problema

O problema proposto, chamado de “Mundo do aspirador de pó”, consiste em um cenário representado por uma grade de dimensão 2x2. No início da resolução do problema o aspirador deve estar em algumas destas posições. As quatro posições podem ou não ter sujeira. O aspirador pode se mover ou aspirar a sujeira. Espera-se como resultado final a grade com todas as posições sem sujeira. Abaixo, uma definição mais precisa dos operadores e custos associados ao problema.

“Mundo do Aspirador de Pó - Um cenário é representado por uma grade de 2X2, sendo que cada quadrado pode ter ou não sujeira. Um aspirador de pó pode aspirar sujeira ou se mover nesse cenário, com os seguintes operadores: 1) aspirar sujeira do quadrado em que se encontra; 2) mover para o quadrado da direita; 3) mover para o quadrado de baixo; 4) mover para o quadrado da esquerda; 5) mover para o quadrado de cima. Esses operadores têm custos diferentes que são: aspirar a sujeira - custo 2; andar para a esquerda ou para a direita - custo 1; andar para cima ou para baixo - custo 3. Encontre a sequência de movimentos para chegar a um estado onde todos os quadrados estão limpos, a partir de uma situação inicial com pelo menos 2 quadrados sujos.”

2.2 Modelagem conceitual do problema

Basicamente um problema deve ser modelado de tal forma que fiquem explícitas informações como representação do estados, ações que podem ser tomadas na transição de um estado a outro, estado inicial, estado final e custos associados à tomada de ações.

Antes da apresentação da modelagem feita para o problema do aspirador de pó, foi definida uma convenção para o cenário. Assim, o cenário será representado por uma grade de 4 posições, onde cada posição é associada à uma localização de coordenadas X e Y. Abaixo, a relação entre a posição na grade e o ponto ao qual está associada.

M3 **(2,1)**	**M4** **(2,2)**
M1 **(1,1)**	**M2** **(1,2)**

Figura 1. Cenário do “Mundo do aspirador de pó”

A representação dos estados foi definida como uma tupla de seis posições. As quatro primeiras posições indicam se há (valor 1) ou não (valor 0) sujeira nas posições M1, M2, M3 e M4, respectivamente. As duas últimas posições da tupla indicam a posição em que o aspirador se encontra, X e Y, respectivamente.

Para a definição do estado inicial, levou-se em conta a especificação do problema. Assim, o estado inicial é qualquer combinação de tal forma que ao menos duas posições da grade estejam sujas. Entre os operadores possíveis, estão cinco possibilidades. São elas: aspirar sujeira de uma determinada posição, mover aspirador para esquerda, direita, para cima e para baixo. Os custos associados à tomada de ações a partir dos operadores definidos são: aspirar sujeira, custo 2; mover aspirador para direita ou esquerda, custo 1 e movimento para cima ou para baixo, custo 3.

Por fim, o estado final, ou seja, o estado objetivo a ser alcançado no processo de busca, consiste em uma combinação de tal forma que as posições de M1 a M4 estejam limpas, ou seja, como valor zero, independente de posição do aspirador. No quadro abaixo, um resumo da modelagem feita.

Estados	Um estado é representado por uma tupla de seis posições. M1 a M4 representam se há (valor 1) ou não (valor 0) sujeira em uma dada posição da grade com coordenadas X e Y, conforme convenção acima. Assim, um estado E tem a forma: E = (M1,M2,M3,M4,X,Y)
Estado inicial	Qualquer estado com a restrição de que ao menos duas posições da grade estejam com sujeira. Exemplo: I = (1,1,0,0,1,1)
Operadores	1) aspirar sujeira; 2) mover para esquerda; 3) mover para direita; 4) mover para cima; 5) mover para baixo
Estado final	Qualquer combinação de tal forma que o estado contenha todas posições de M1 a M4 limpas, ou seja, com valor zero. Exemplo: F = (0,0,0,0,1,2)
Custo do caminho	1) aspirar sujeira – custo 2 2) e 3) mover esquerda e direita – custo 1 4) e 5) mover cima e baixo – custo 3

2.3 Modelagem do problema em linguagem R

Um estado como definido acima, foi implementado em linguagem R através do uso de uma estrutura de dados de vetor. Assim, um estado é um vetor de valores inteiros , conforme especificação conceitual mostrada acima. Abaixo, um exemplo de um estado “E” cujos valores são M1=1, M2=0,M3=1,M4=0,X=1,Y=2.

```
E ← c(1,0,1,0,1,2)
```

Utilizando-se do sistema S3 para definição de classes, criou-se uma classe “AspiradorDePo”. Assim, consegue-se instanciar um objeto do tipo “AspiradorDePo”, objeto este que contém o estado em que encontra, objeto que o gerou, caso exista, valor de custo “g”, valor de heurística “h” e valor “f” relativo aos dois valores de custo anteriores. Além disso, esta classe oferece métodos para a geração de objetos filhos de um dado objeto “AspiradorDePo”, método para cálculo de heurística de um objeto (será explicado mais adiante), método para comparação de estados, além de um método para impressão no terminal dos atributos de um dado objeto. No processo de busca, os nós da árvores de busca serão objetos do tipo “AspiradorDePo”, de forma que se tenha não apenas o estado, mas também seu nó pai e valores de custo associados àquele nó.

No método de comparação de estados da classe “AspiradorDePo” considerou-se que dois estado são iguais se os valores referentes à existência de sujeira em uma dada posição são iguais. Desprezou-se os dois valores referentes às coordenadas de localização do aspirador de pó, visto que para obtenção do estado objetivo basta que todas as posições estejam limpas, independentemente da localização do aspirador de pó. Assim, dois estados E = (A1,B1,C1,D1,E1,F1) e F = (A2,B2,C2,D2,E2,F2) são iguais se:

A1=A2 e B1=B2 e C1=C2 e D1=D2

Abaixo o código em R da sobrecarga do operador “==” para a comparação de estados. Para acesso ao estado de um dado objeto “AspiradorDePo” “obj”, acessa-se o campo “desc” com a sintaxe obj$desc.

```
## Sobrecarregando o operador "==" para comparação entre estados
Ops.AspiradorDePo = function(obj1,obj2){
        if(.Generic == "=="){
                return(all(obj1$desc[1:4] == obj2$desc[1:4]))
        }
}
```

Outro método implementado foi o “geraFilhos”. Sua função é gerar os filhos de um dado nó (objeto “AspiradorDePo”) através da aplicação dos operadores definidos na modelagem do problema. Seu código está abaixo.

```
#Metodo para gerar filhos de um dado no
geraFilhos.AspiradorDePo <- function(obj) {

  #Um estado e representado : (M1,M2,M3,M4,X,Y) => Mi = 1, posicao suja, Mi=0, limpa
  #                           => X,Y , posicao atual do aspirador

  filhosDesc <- list() #lista de filhos inclusive incompativeis
  filhos <- list() # lista de filhos sem incompativeis

  desc <- obj$desc #desc recebe estado a ter filhos gerados
```

```
#operadores

#Caso aspirador em (1,1)
if(desc[5]==1 && desc[6]==1){
   operadores <- list(c(-1,0,0,0,0,0), c(0,0,0,0,-1,0), c(0,0,0,0,1,0), c(0,0,0,0,0,-1), c(0,0,0,0,0,1))
   #custo aplicacao dos operadores
   custo <- c(2,1,1,3,3)

#Caso operador em (1,2)
}else if(desc[5]==1 && desc[6]==2){
  operadores <- list(c(0,-1,0,0,0,0), c(0,0,0,0,-1,0), c(0,0,0,0,1,0), c(0,0,0,0,0,-1), c(0,0,0,0,0,1))
  #custo aplicacao dos operadores
  custo <- c(2,1,1,3,3)

  }#Caso operado em (2,1)
else if(desc[5]==2 && desc[6]==1){
  operadores <- list(c(0,0,-1,0,0,0), c(0,0,0,0,-1,0), c(0,0,0,0,1,0), c(0,0,0,0,0,-1), c(0,0,0,0,0,1))
  #custo aplicacao dos operadores
  custo <- c(2,1,1,3,3)

 }#Caso operador em (2,2)
else if(desc[5]==2 && desc[6]==2){
  operadores <- list(c(0,0,0,-1,0,0), c(0,0,0,0,-1,0), c(0,0,0,0,1,0), c(0,0,0,0,0,-1), c(0,0,0,0,0,1))
  #custo aplicacao dos operadores
  custo <- c(2,1,1,3,3)

}

#geracao de filhos inclusive com incompativeis
filhosDesc <- lapply(operadores, function(op) desc+op)

#remocao de estados incompativeis => quando alguma posicao de celula for <0 OU quando
posicao X,Y for fora dos limites 2 X 2
incompativeis <- sapply(1:length(filhosDesc),
                        function(i){
                          fDesc = filhosDesc[[i]] #pega estado

                          retorno <- 0

                          #verifica se posicao celula negativa
                          for(k in 1:4){
                            if(fDesc[k]<0){
                              retorno <- i
                            }
                          }

                          #verifica limites de posicao em grid 2X2
                          if(fDesc[5]<1 || fDesc[5]>2 || fDesc[6]<1 || fDesc[6]>2){
                            retorno <- i
                          }
```

```
                return(retorno)


            }) #fim sapply

  #Manter no vetor incompativeis apenas indices dos incompativeis => != 0
  incompativeis <- incompativeis[incompativeis!=0]

  #Colocar no vetor FilhosDesc apenas filhos compativeis
  filhosDesc <- filhosDesc[-incompativeis]
  custo <- custo[-incompativeis]

  #Geracao de objetos AspiradorDePo para filhos
  i <- 1 #contador
  for(filhoDesc in filhosDesc){
    filho <- AspiradorDePo(desc=filhoDesc,pai=obj)
    filho$h <- heuristica(filho)
    filho$g <- obj$g + custo[i] #pega custo gasto na geracao deste estado filho
    i <- i + 1
    filhos <- c(filhos, list(filho))
  }


  return(filhos)

}
```

A lista “filhosDesc” será utilizada para armazenar todos os estados gerados a partir da aplicação dos operadores a serem definidos, inclusive estados incompatíveis com a definição do problema. A lista “filhos”, por sua vez, conterá apenas nós (objetos AspiradorDePo) cujos estados sejam compatíveis e ela é que será o retorno deste método. Posteriormente, foram definidos os operadores a serem aplicados sobre o estado passado como parâmetro no método. Um operador foi definido como um vetor de seis posições. Na quatro primeiras posições deste vetor estão as ações referentes à aspirar sujeira em uma dada posição. Para aspirar sujeira, o valor -1 foi definido. Para os operadores de movimento, considerando que as duas últimas posições deste vetor de operação sejam as coordenadas X e Y, foi definido que caso o movimento seja para direita, o valor de X é incrementado em uma unidade, para esquerda decrementado em uma unidade, para cima o valor de Y é aumentado em uma unidade e para baixo é diminuído em uma unidade. Desta forma, a aplicação de um operador sobre um estado corresponde a soma entre os vetores destas duas entidades.

Uma questão foi considerada. Supondo um estado no qual a posição (1,1) esteja suja, o operador referente a aspirar sujeira só pode aspirar sujeira nesta posição. Em virtude disto, foram construídas estruturas condicionais de forma a ser garantir que uma posição da grade só seja aspirada caso o aspirador esteja nesta posição. Os

operadores de movimento, por sua vez, podem ser aplicados em qualquer posição da grade.Assim, a lista “operadores” guarda os operadores possíveis (inclusive incompatíveis) para uma dada posição do cenário.

Porém, tanto na aplicação do operador de aspirar sujeira quanto no de movimentos, estados incompatíveis podem ser gerados. Por exemplo, supondo que na posição (1,1) não exista sujeira, o vetor que representa seu estado seria (0,?,?,?,1,1). O operador para aspirar sujeira nesta posição seria (-1,0,0,0,0,0). Aplicando este operador sobre este estado, ou seja, somando-os, o vetor resultante seria (-1,?,?,?,1,1). Percebe-se uma inconsistência, visto que nas posições do vetor relativas a ter ou não sujeira em uma dada posição os valores possíveis são 0 ou 1. Outra inconsistência, seria na extrapolação dos limites da grade definida como cenário. Supondo a posição (1,2), um possível vetor de estado para esta posição seria (?,?,?,?,1,2). Um operador de deslocamento para direita seria (0,0,0,0,0,1). Aplicando este operador ao estado acima o vetor estado resultante seria (?,?,?,?,0,3). Como os limites da grade, considerando as definições dadas acima, são 2 <= X <= 1 e 2 <= Y <= 1, o valor referente à coordenada Y extrapolou os limites definidos. Desta forma, o estado resultante é inconsistente.

Para a remoção destes estados inconsistentes, utilizou-se a função “sapply”. As duas possibilidades de inconsistência citadas acima são verificadas. Obtém-se os indíceis dos estados incompatíveis, posteriormentes removidos, de forma que a lista “filhosDesc” passa a ter apenas estados consistentes com a definição do problema.

Como os custos associados à aplicação dos operadores são diferentes, foi necessário armazenar em um vetor “custo” os custos referentes à aplicação destes operadores. No momento em que os estados incompatíveis devem ser removidos, os custos associados aos operadores que geraram tais operadores também foram removidos do vetor “custo”.

Apenas com os estados consistentes e seus respectivos custos, foram gerados os nós filhos, ou seja, objetos do tipo AspiradorDePo. Além de receberem o estado, foram “setadas” seu valor “g” de custo e seu valor “h” de heurística. Estes filhos gerados foram armazenados em uma lista “filhos” e esta foi retornada.

O método “heuristica.AspiradorDePo” tem como função gerar a heurística de um dado nó. Foi estabelecida que a heurítica de um dado nó, com estado E = (M1,M2,M3,M4,X,Y), corresponde à soma M1+M2+M3+M4. Como Mi representa se há (1) ou não sujeira (0) numa posição da grade, quando menos células sujas houverem no cenário, menor será a heurística e mais próximo do estado objetivo o estado atual estará.

```
## Sobrecarga da função genérica "heuristica", definida por Estado.R
heuristica.AspiradorDePo <- function(atual){

  if(is.null(atual$desc))
    return(Inf)
  ## h(obj) = M1 + M2 + M3 + M4
  return(sum(atual$desc[1:4]))
```

}

2.4 Resultados

Para a aplicação dos algoritmos de busca foi criado um arquivo "iniciaAspirador.R". Neste arquivo, além da inclusão dos outros arquivos necessários para criação de objetos do tipo "AspiradorDePo" e dos algoritmos de busca, foram criados um nó inicial e um nó objetivo. Estes nós serão passados para os algoritmos de busca. Como na especificação do problema apenas foi exigido que ao menos duas posições do cenário estivessem sujas, foi definido que as posições (1,1), (1,2) e (2,1) estariam com sujeira e que o aspirador iniciaria seu trabalho na posição (1,1). Assim, o nó inicial possuiria o seguinte estado: (1,1,1,0,1,1).

inicial** <- **AspiradorDePo(desc = c(1,1,1,0,1,1)) #dois quadrados sujos com inicio em (1,1)

Pela especificação, o objetivo seria limpar todas as células do cenário. Assim, a única restrição para o estado final seria que as posições referentes à estas células deveriam ter valor zero, ou seja, estarem sem sujeira, independentemente da posição em que o aspirador se encontra. Assim, foi definida uma sobrecarga de comparação entre estados que levem em conta esta restrição, como já explicado acima. Para a aplicação dos algoritmos foi estabelecido um nó com células limpas com a posição (1,1).

objetivo$desc <- c(0,0,0,0,1,1) #possilidade com objetivo em posicao (1,1).

Com os nós inicial e objetivo definidos, foram aplicados os algoritmos de busca informada e desinformada. As saídas completas obtidas pela aplicação destes algoritmos encontram-se em arquivos anexos.

2.4.1 Busca em largura

Com base no caminho retornado pela busca em largura, as ações tomadas pelo aspirador foram: aspirou sujeira da posição (1,1), movimentou-se para (2,1), aspirou sujeira em (2,1), movimentou-se para posição (1,1), movimentou-se para posição (1,2), aspirou sujeira em (1,2), alcançou objetivo. O custo total foi 11.

2.4.2 Busca em profundidade

Com base no caminho retornado pela busca em profundidade, as ações tomadas pelo aspirador foram: aspirou sujeira da posição (1,1), movimentou-se para (2,1), aspirou sujeira em (2,1), movimentou-se para posição (1,1), movimentou-se para posição (1,2), aspirou sujeira em (1,2), alcançou objetivo. O custo total foi 11.

2.4.3 Busca de custo uniforme

Com base no caminho retornado pela busca com custo uniforme, as ações tomadas pelo aspirador foram: movimentou-se para (2,1), aspirou sujeira em (2,1), movimentou-se para (1,1), aspirou sujeira em (1,1), movimentou-se para (1,2), aspirou sujeira em (1,2), alcançou objetivo. O custo total desse caminho foi 11.

2.4.4 Busca gulosa

Com base no caminho retornado pela busca gulosa, as ações tomadas pelo aspirador foram: aspirou sujeira da posição (1,1), movimentou-se para (2,1), aspirou sujeira em (2,1), movimentou-se para posição (1,1), movimentou-se para posição (1,2), aspirou sujeira em (1,2), alcançou objetivo. O custo total foi 11.

2.4.5 Busca A*

Com base no caminho retornado pela busca pelo algoritmo A*, as ações tomadas pelo aspirador foram: aspirou sujeira da posição (1,1), movimentou-se para (2,1), aspirou sujeira em (2,1), movimentou-se para posição (1,1), movimentou-se para posição (1,2), aspirou sujeira em (1,2), alcançou objetivo. O custo total foi 11.

2.5 Comparação entre resultados obtidos

Na aplicação de algoritmos de busca desinformada, neste caso o de busca em largura, em profundidade e de custo uniforme, os custos totais dos caminhos obtidos foram iguais. No caso da busca em largura e profundidade, as ações tomadas foram sequencialmente idênticas. Já no de busca de custo uniforme, houve mudanças na sequência de movimentos feitos, porém foram iguais aos movimentos feitos no dois outros algoritmos. Isto ocorreu pelo fato do algoritmo de busca uniforme levar em conta o custo associado a um determinado movimento, prevalecendo o menor, diferentemente dos algoritmos de busca em largura e profundidade, onde a ordem dos nós prevalece.

Já na aplicação dos algoritmos de busca informada, o de busca gulosa e o A*, as ações tomadas para se alcançar o estado objetivo foram idênticas em termos de valor e de sequência, semelhantes às ações tomadas pelos algoritmos de busca em largura e profundidade.

Apesar destes algoritmos variarem em termos dos critérios a serem levados em conta para a definição do próximo nó a ser expandido, os resultados obtidos foram em termos de custo iguais, além do caminho variar em apenas um deles. Talvez isto tenha ocorrido pelo pequeno espaço de estados possíveis para este problema.

Apenas para ilustrar isso, foi mudado o estado inicial de forma que todas as células estivessem com sujeira e o aspirador inicia-se na posição (2,1). Após a execução dos algoritmos, o custo retornado pelos algoritmos de busca em largura,

custo uniforme e A* foi de valor treze, enquanto que os outros algoritmos retornaram custo quatorze. As saídas de execução destes algoritmos estão em arquivos anexos.

3. Conclusão

Com este trabalho, foi possível entender o processo de resolução de um problema por mecanismos de busca. Inicialmente, deve-se fazer uma modelagem deste problema, de forma que fiquem explícitos aspectos como estados, ações, custos associados à essas ações, estado inicial, estado final, entre outros. Posteriormente, deve-se aplicar algoritmos de busca sobre uma árvore que corresponde ao espaço de estados possíveis obtido através da aplicação de operadores sobre outros estados. Os mecanismos de busca variam pelo critério que possuem para a escolha do próximo nó a ser expandido e ter seu estado gerado. Assim, foi necessário definir quando dois estados são iguais e uma função heurística quando esta for exigida.

Como apoio, um ambiente de desenvolvimento foi utilizado munido na linguagem de programa R e seu vasto repositório de recursos. Assim, foi possível também entender aspectos básicos deste ambiente de desenvolvimento.