Universidade Federal de Santa Maria
Curso: Ciência da Computação
Disciplina: Inteligência Artificial
Notas sobre Programação com Agentes
Professora: Juliana Kaizer Vizzotto

0.1 Buscando Soluções: Busca Cega

Uma vez que o problema foi bem formulado, o estado final deve ser buscado. Em outras palavras, deve-se usar um método de busca para saber a ordem correta de aplicação dos operadores que levará do estado inicial para o estado final.

Isto é feito por um processo de geração de estados possíveis e teste, para ver se o objetivo está entre eles. Uma vez a busca terminada com sucesso, é só executar a solução (que é um conjunto ordenado de operadores a aplicar).

0.1.1 Gerando sequências de ações

Para, por exemplo, resolver o problema de rota visto em aula de ir de Arad para Bucharest, começando com somente o estado inicial, Arad. O primeiro passo é testar se esse é o *goal state*. Claramente, ele não é. Como esse não é o *goal state* , precisamos considerar alguns outros estados. Isto é feito através da aplicação dos *operadores* no estado corrente, e assim gerando um novo conjunto de estados. Esse processo é chamado de *expansão* do estado. Nesse caso específico do exemplo, teríamos três novos estados, “em Sibiu”, “em Timisoara”, e “em Zerind”, pois existe uma rota direta de Arad para essas três cidades. Se existisse somente uma possibilidade, nós simplesmente escolheríamos ele e continuaríamos. Mas sempre que existem múltiplas possibilidades, precisamos fazer uma escolha de qual seguir primeiro.

Essa é a essência da busca: escolher uma opção e colocar as outras de lado por um momento, no caso que a primeira escolha não leve ao objetivo. Suponha que escolhemos Zerind, assim verificamos se este é o *goal state*. Esse processo de escolher, testar e expandir deve ser executado até que a solução seja encontrada, ou até que não exista mais estados para expandir. A escolha de qual estado *expandir* primeiro é determinada pela **estratégia de busca**.

É interessante pensar no processo de procura como construindo uma **árvore de procura**. A raiz da árvore é chamada de nodo de procura correspondendo ao estado inicial. Os nodos folhas da árvore correspondem aos estados que não têm sucessores. A cada passo o algoritmo de procura escolhe um nodo folha para expandir.

É importante distinguir entre o espaço de estados e a árvore de busca. Por exemplo, para esse problema, existem somente 20 estados, um para cada cidade. Mas existem um número infinito de caminhos (paths) nesse espaço de de estados. Assim, a árvore de busca tem um número infinito de nodos. Temos que cuidar com os loops. Obviamente um bom algoritmo de busca evita tais loops. Logo vamos discutir técnicas para isso.

0.1.2 Estrutura de Dados para Árvores de busca

Existem muitas maneiras de representar nodos, mas vamos assumir por enquanto que um nodo é uma estrutura de dados com cinco componentes:

- o estado no espaço de estados o qual o nodo corresponde;
- o nodo *pai* na árvore de busca;
- o operador que foi aplicado para gerar esse nodo;

- o número de nodos no caminho da raiz até este nodo;
- o custo do caminho do nodo inicial até este nodo.

É importante lembrar da diferença entre nodos e estados. Um nodo é uma estrutura de dados usado para representar a árvore de procura para uma instância particular do problema. Um estado representa uma configuração (ou conjunto de configurações) do mundo. Então, nodos têm profundidade e pais, e estados não. Ainda, é bastante possível que dois nodos diferentes contenham o mesmo estado, se tal estado foi gerado por duas sequências diferentes de ações. A função de **expansão** é responsável por calcular cada um dos componentes dos nodos que ela gera.

Também precisamos representar a coleção de nodos que estão esperando para serem expandidos, i.e., as *fronteiras*. A representação mais simples seria um conjunto de nodos. A estratégia de procura então seria uma função que seleciona o próximo nodo a ser expandido desse conjunto. Vamos assumir que a coleção de nodos é implementada como uma **fila**. As operações na fila são:

- `MAKE-QUEUE(elements)`: cria um fila com os dados elementos.
- `EMPTY?(queue)`: retorna *true* se não existem elementos na fila.
- `REMOVE-FRONT(queue)`: remove o elemento no início da fila e retorna-o.
- `QUEUING-FN(elements,queue)`: insere um conjunto de elementos na fila. Essa função depende da estratégia de busca utilizada.

Assim, o algoritmo geral de procura pode ser resumido como:

```
function GENERAL-SEARCH(problem, QUEUING-FN)  returns a solution or failure

   nodes <- MAKE-QUEUE(MAKE-NODE(INITIAL-STATE[problem]))

   loop do
     if nodes is empty then return failure
     node <- REMOVE-FRONT(nodes)
     if GOAL-TEST[problem] applied to STATE(node) suceeds then return node
     nodes <- QUEUING-FN(nodes,EXPAND(node,OPERATORS[problem]))
end
```

0.1.3 Estratégias de Busca

Objetivo: Procurar a **estratégia de busca** certa para um dado problema.

Em geral, avalia-se as estratégias de busca de acordo com quatro critérios:

- **Completeness**: a estratégia garante que vamos encontrar uma solução, quando ela existe?
- **Complexidade de tempo**: quando tempo demora para encontrar a solução?
- **Complexidade de espaço**: quando de memória precisamos para executar a procura?
- **Otimização**: a estratégia encontra a solução de melhor qualidade quando existem diferentes soluções?

Em geral podemos classificar os métodos de busca de acordo com dois tipos:

- Busca exaustiva ou cega: não sabe qual o melhor nó da fronteira a ser expandido = menor custo de caminho desse nó até um nó final (objetivo).

- Busca Heurística ou informada: estima qual o melhor nó da fronteira a ser expandido com base em **funções heurísticas**, i.e., conhecimento.

Vamos estudar esta semana seis estratégias de busca cega. Esse termo significa que não se tem informação sobre o custo ou número de passos do estado inicial até o objetivo. Tudo que se sabe é distinguir entre um estado normal e um estado de objetivo.

Considere novamente o problema de rotas. A partir do estado inicial, Arad, existem três ações levando para três novos estados: Sibiu, Timisoara, Zerind. Uma busca cega não tem preferência entre esses três. Já um agente mais esperto poderia notar que o objetivo, Bucarest, é ao sul de Arad, e então somente ir para Sibiu seria uma opção interessante. Estratégias que usam tais considerações são chamadas de busca heurística. A busca cega é realmente menos efetiva que a busca heurística, entretanto, ela ainda é importante, pois existem diversos problemas para os quais não temos informações adicionais para considerar.

As seis estratégias de busca cega que veremos são distintas pela *ordem* na qual os nodos são expandidos.

As estratégias para determinar a ordem de ramificação são:

- Busca em largura
- Busca de custo uniforme
- Busca em profundidade
- Busca com aprofundamento iterativo

Além disso, a direção da ramificação pode ser:

- Do estado inicial para um estado final
- De um estado final para o estado inicial
- Busca bi-direcional

Busca em Largura

Nessa estratégia a ordem de ramificação dos nós é a seguinte:

1. Nó raiz
2. Todos os nós de profundidade 1
3. Todos os nós de profundidade 2, etc

Em geral, todos os nodos na profundidade d na árvore de busca são expandidos antes dos nodos na profundidade $d+1$. Esse método pode ser implementado chamando o algoritmo de procura geral (`GENERAL-SEARCH`) com uma função de fila que coloca o novo elemento gerado no final da fila, depois de todos os estados gerados previamente.

```
function BL(Estado inicial) : Nodo
  Queue fronteira
  fronteira.add(new Nodo(inicial))
  while not fronteira.isEmpty() do
    Nodo n <- fronteira.remove()
    if n.getEstado().éMeta() then
      return n
```

```
    endif
    fronteira.add(n.sucessores())
 end while
 return null
```

Um problema geral em busca é a geração de estados repetidos, o que torna a árvore de busca potencialmente infinita. **Uma solução é a implementação de lista fechada**.

- Não retornar ao estado "pai"
- Não retornar a um ancestral
- Não gerar qualquer estado que já tenha sido criado antes (em qualquer outro ramo)

Requer que todos os estados gerados permaneçam na memória. Pode ser implementado mais eficientemente com hash tables. Quando entra nó igual tem de escolher o melhor (menor custo de caminho até então).

Assim, a ideia é manter uma **lista fechada**, i.e., uma lista de nós que já foram expandidos. Se o nó atual estiver na **lista fechada**, ele será descartado ao invés de ser expandido.

```
function BL(Estado inicial) : Nodo
  Queue fronteira
  fronteira.add(new Nodo(inicial))
  while not fronteira.isEmpty() do
    Nodo n <- fronteira.remove()
    if n.getEstado().éMeta() then
      return n
    endif
    if n.getEstado() nao esta em fechado then
       fechado.add(n.getEstado())
       fronteira.add(n.sucessores())
 end while
 return null
```

O método de expansão para obter sucessores de um nó pode ser implementado como segue:

```
function sucessores(Nodo n, acao):list de nós
   for (cada acao em n.getEstado()) do
      s <- cria novo no
      s.estado = resultado da acao em n
      s.pai = n
      s.acao = acao
      s.custo-caminho = s.custo-caminho+custo-passo(n,acao,s)
      s.profundidade = n.profundidade+1
      add s a sucessores
retornar sucessores
```

Se existe uma solução, o algoritmo de busca em largura sempre irá encontrá-la, e se existem várias soluções o algoritmo sempre vai encontrar o estado de objetivo de menor profundidade primeiro. O algoritmo de busca em largura é completo.

Em geral o algoritmo de busca em largura parece uma opção atrativa, a menos que consideremos a quantidade de memória e tempo necessários.

Uma consideração é com relação ao fator de ramificação da árvore de busca: número de nós gerados a partir de cada nó (b).

- Custo de tempo: se o fator de ramificação do problema é b, e a primeira solução para o problema está no nível d. Então o número máximo de nós gerados até se encontrar a solução é $1 + b + b^2 + ... + b^d$. **Custo exponencial** = $O(b^d)$
- Custo de memória: todo nó gerado deve permanecer na memória, então também é $O(b^d)$.

Assim, busca em largura só dá bons resultados quando a profundidade da áravore de busca é pequena.

Busca de de Custo Uniforme (Dijkstra Search)

Estende a busca em largura expandindo o nó da fronteira com **menor custo de caminho até o momento**. Cada operador pode ter um custo associado diferente, medido pela função $g(n)$ que dá o custo do caminho da origem ao nó n.

```
function BL(Estado inicial) : Nodo
  Priority Queue(g) fronteira {fila ordenada por g}
  fronteira.add(new Nodo(inicial))
  while not fronteira.isEmpty() do
    Nodo n <- fronteira.remove()
    if n.getEstado().éMeta() then
      return n
    endif
    fronteira.add(n.sucessores())
 end while
 return null
```