Inteligência Artificial

Prof. Alexsander da Rosa <alexsand@urcamp.tche.br>
http://www.urcamp.tche.br/~alexsand/

Estratégias de Busca

- Procurando soluções de problemas
- Estratégias de Busca (não informada)

VERSÃO 1.1

Procurando Soluções de Problemas

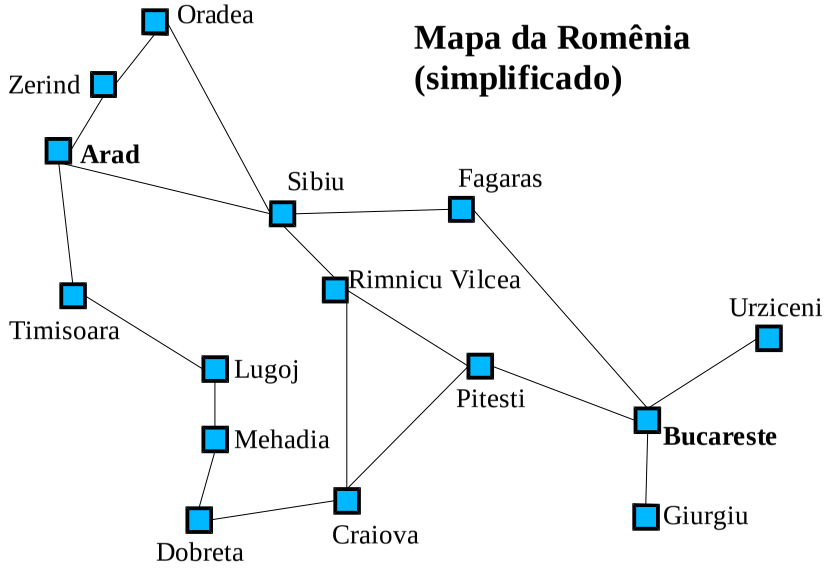
Depois de *definir um problema*, e saber reconhecer uma *solução* para ele, é preciso saber *encontrar* a solução.

Pode−se usar o exemplo do problema da viagem de Arad até Bucareste, na Romênia. O *estado inicial* é Arad, os *operadores* são ir de uma cidade a outra pela estrada e o *estado final* [*objetivo*] é [chegar em] Bucareste.

A procura da solução de um problema é uma *busca no espaço de estados*. A idéia é manter e estender um conjunto de soluções parciais. Para isso, é preciso saber como *gerar sequências de ações* e como rastreá−las através do uso de *estruturas de dados* adequadas.

A ***geração das sequências de ações*** é feita aplicando os *operadores* ao estado atual, gerando um novo conjunto de estados (*expansão* do estado). Por exemplo, expandindo o estado inicial (*Arad*) surgem três outros estados: *Sibiu*, *Timisoara* e *Zerind*. Após a expansão é preciso escolher *que* estado, dentre os gerados, será expandido primeiro; esta escolha é determinada pela *estratégia de busca*.

O processo de busca pode ser visto como a construção de uma *árvore de busca*, e a raiz desta árvore é um *nodo de busca* correspondendo ao *estado inicial*. Os *nodos−folha* são os estados que não têm sucessores, porque não foram expandidos ainda ou geraram um conjunto vazio.



As ***estruturas de dados para árvores de busca*** podem ser várias, mas o *nodo* deve ter pelo menos 5 componentes:

1. O *estado*, no espaço de estados, ao qual o nodo corresponde;
2. O nodo, na árvore de busca, que o gerou (*nodo pai*);
3. O *operador* que foi aplicado para gerá−lo;
4. O número de nodos no *caminho* desde a raiz até o nodo corrente (a *profundidade* do nodo);
5. O custo do caminho desde o estado inicial até ele.

Assim como é importante diferenciar a *árvore de busca* do *espaço de estados*, é preciso diferenciar *nodos* de *estados*.

O conjunto dos nodos que estão esperando a expansão é chamado *fronteira* ou *borda*. Geralmente usa−se uma *fila*.

Estratégias de Busca

Pode−se usar quatro perguntas para avaliar estratégias:

1. A estratégia *garante* encontrar uma solução (se existir)?
2. Quanto *tempo* ela demora para achar uma solução?
3. Quanta *memória* ela precisa para realizar a busca?
4. Ela encontra a *melhor solução* quando há várias?

A pergunta 1 define se uma estratégia é ou não *completa*; a 2 e a 3 definem a *complexidade*, em termos de *O(n)*, de tempo e memória; e a 4 indica se a ela é ou não *ótima*.

As *seis* estratégias de busca que serão vistas agora são chamadas de *busca não−informada*, ou *busca cega*.

Busca em amplitude (*breadth−first*)

Nesta estratégia, o *nodo raiz* é expandido primeiro, depois *todos* os nodos gerados a partir dele, depois os *sucessores* destes, etc. Em geral, *todos* os nodos no nível *n* são expan−didos *antes* dos nodos no nível *n+1*. A busca em amplitude coloca os estados gerados *por último* no final da fila.

Se há uma solução, a busca em amplitude a encontra (logo, é *completa*), e encontra a que tem o caminho mais raso primeiro (*ótima* se o custo é proporcional à profundidade).

Para analisar o tempo e o espaço necessários, usa−se um espaço de estados hipotético, onde todos os estados se expandem em um fator de ramificação constante.

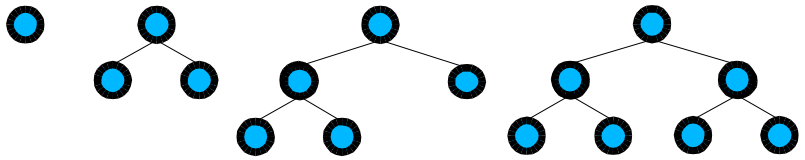


Figura 1: Árvores de busca em amplitude após 0, 1, 2 e 3 expansões

Com um *branching factor b*, a raiz da árvore de busca gera *b* nodos no primeiro nível, e cada um gera outros *b* nodos, num total de b^2 no 2º nível. A complexidade é $O(b^n)$.

Supondo que a solução está no nível *n*, o número máximo de nodos expandidos até que a solução seja encontrada é:

$$max = 1 + b + b^2 + b^3 + ... + b^n$$

A estratégia requer muito tempo e memória, pois todos os *nodos* devem ser mantidos em memória ao mesmo tempo.

Profundidade	***Nº de nodos***	***Tempo***	***Memória***
0	1	1 milisegundo	100 bytes
2	111	0,1 segundo	11 Kb
4	11111	11 segundos	1 Mb
6	10^6	18 minutos	111 Mb
8	10^8	31 horas	11 Gb
10	10^{10}	128 dias	1 Tb
12	10^{12}	35 anos	111 Tb

Tabela 1: Uso de tempo e memória da busca em amplitude, usando *branching factor* 10, 1000 nodos/segundo e 100 bytes/nodo.

Busca com custo uniforme (*uniform cost*)

A busca em amplitude encontra a solução no estado mais raso, mas nem sempre esta é a solução com o menor custo.

A busca com *custo uniforme* modifica a busca em amplitude, sempre expandindo o nodo com o menor custo da *fronteira*, ao invés do nodo com menor profundidade.

Para isso é usada a função de custo do caminho *g(n)*. A busca em amplitude é um caso especial da busca de custo uniforme, onde *g(n)=profundidade(n)*, ou seja é *ótima*.

A primeira solução encontrada é a de menor custo, pois se houvesse outra ela teria sido expandida primeiro.

Por exemplo, no problema de *route−finding* (Figura 2), o objetivo é ir do EI até o EF. O custo de cada *operador* está indicado. Após a expansão inicial, são criados caminhos para A, B e C. O caminho para A é o mais barato, e ele é o próximo a ser expandido, gerando a solução EI−A−EF.

O algoritmo ainda não reconhece EI−A−EF como solução, porque ela tem custo 11, e deve ir para a fila atrás do caminho EI−B, que tem custo 5. Após isso, EI−B é expandida em EI−B−EF, a melhor solução.

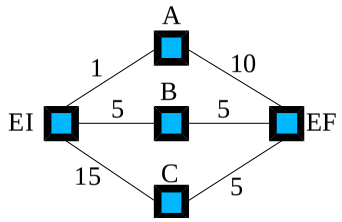


Figura 2: Espaço de estados de um problema de *route−finding*

Busca em Profundidade (*depth−first*)

A busca em profundidade sempre expande um dos nodos do nível mais profundo da árvore. Somente quando a busca atinge um beco sem saída (estado não final sem expansão), a busca volta e expande nodos de níveis mais rasos.

Os estados mais recentes são colocados no início da fila, e serão os primeiros a ser expandidos. Como seus sucessores serão ainda mais *profundos*, também irão para o início.

A busca em profundidade exige bem pouca memória. Só é preciso armazenar um único caminho a partir da raiz até um *nodo−folha*, além dos nodos a expandir pelo caminho.

Para um espaço de estados com um fator de ramificação *b*, e uma profundidade máxima *m*, a busca em *profundidade* requer o armazenamento de apenas *bm* nodos, contra os b^n da busca em amplitude com a solução mais rasa no nível *n*.

Então a complexidade de espaço é *O(bm)*, na busca em profundidade. Já a complexidade de tempo é $O(b^m)$, ou seja, a mesma da busca em amplitude. No entanto, em problemas que aceitam muitas soluções, a busca em profundidade tende a ser mais rápida que a em amplitude.

A desvantagem da busca em profundidade é que ela pode perder muito tempo explorando becos sem−saída, pois muitos problemas têm árvores muito grandes ou infinitas.

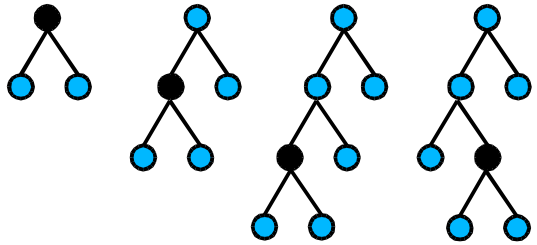


Figura 3: Árvores de busca em profundidade, níveis 0 a 3.

Como a busca em profundidade pode perder−se em um laço infinito, não é *completa*. Também não é *ótima*.

É comum a busca em profundidade ser implementada com uma função recursiva. Neste caso, a fila é implicitamente armazenada na pilha das chamadas de função, no sistema.

Busca com limite de profundidade (*depth−limited*)

Esta estratégia de busca evita as armadilhas da busca em profundidade, impondo um limite à profundidade máxima de um caminho. Esse limite pode ser implementado por um algoritmo específico, ou com operadores que guardam a profundidade. Por exemplo, se no mapa da Romênia existem 20 cidades, sabe−se que uma solução deve ter um comprimento máximo de 19. O operador ficaria: "Se estou na cidade A [e viajei menos que 19 passos], então gero um novo estado na cidade B [e incremento a profundidade]."

A estratégia é *completa**, mas não *ótima.* A complexidade de tempo é $O(b^L)$ e a de espaço é *O(bL)*, onde *L* é o limite.

Busca: Aprofundamento iterativo (*iterative deepening*)

A parte mais difícil da busca *depth−limited* é escolher o limite certo. No problema do mapa da Romênia, um estudo detalhado do mapa mostra que qualquer cidade pode ser atingida, partindo de qualquer cidade, em no máximo 9 passos. Esse número, conhecido como *diâmetro* do espaço de estados, dá um limite melhor que 19.

Na maioria dos problemas, no entanto, não se conhece um bom limite de profundidade antes de resolver o problema.

A busca com aprofundamento iterativo contorna este problema testando todos os limites de profundidade.

Começa tentando executar uma busca *depth−limited* com o limite em 0, depois em 1, em 2, em 3 e assim por diante.

A busca com aprofundamento iterativo combina as vantagens da busca em profundidade com as da busca em amplitude: é *ótima* e *completa*, como a busca em ampli−tude, e tem a complexidade de memória *O(bm)* da busca em profundidade. Em termos de tempo, é mais que $O(b^n)$.

$$(n+1)1 + (n)b + (n-1)b^2 + ... + 3b^{n-2} + 2b^{n-1} + 1b^n$$

O aprofundamento iterativo parece desperdiçar tempo, pois vários estados são expandidos diversas vezes. Na prática, porém, a maioria dos nodos estão no nível mais profundo.

Busca bidirecional (*bidirectional search*)

A idéia por trás da *busca bidirecional* é fazer a busca simultaneamente tanto para frente (*forward*), a partir do estado inicial, quanto para trás (*backward*), a partir da solução. O algoritmo pára quando as duas buscas (usando qualquer estratégia) se encontram no meio do caminho.

Se há uma solução no nível *n*, temos a complexidade de tempo em $O(2b^{n/2})$. Por exemplo, a busca em amplitude da *Tabela 1*, que tem mais de 10^6 nodos no nível 6, se feita de modo bidirecional reduz o número de nodos para 2.222, o que significa demorar 2 segundos ao invés de 18 minutos.

Apesar de ser excelente na teoria, várias questões precisam ser resolvidas antes da implementação do algoritmo:

- Como se faz uma busca para trás? Os *predecessores* de um nodo *x* são todos os nodos que têm *x* como sucessor; se todos os operadores são reversíveis, os conjuntos de *sucessores* e *predecessores* são idênticos, mas se não pode ser muito difícil calcular os *predecessores*;
- E se houverem vários estados finais? Se houver uma lista *explícita* deles, aplica−se uma função predecessor no conjunto, como num problema de múltiplos estados; se houver apenas uma descrição do conjunto, pode ser bem difícil determinar o conjunto de estados finais;

- Deve haver uma maneira eficiente de checar se um novo nodo não aparece na árvore de busca da outra metade;
- É preciso definir que estratégia de busca será usada em cada metade da busca bidirecional.

É preciso também que *todos* os nodos de pelo menos uma das metades da busca sejam guardados em memória (como na busca em amplitude), para que os dois lados possam se encontrar. Assim, a complexidade de espaço é $O(b^{n/2})$.

Comparação de estratégias de busca

Critério	*Amplitude*	*Custo Uniforme*	*Profun−didade*	*Limite de profun−didade*	*Aprofun−damento iterativo*	*Bidire−cional*
Tempo	$O(b^n)$	$O(b^n)$	$O(b^m)$	$O(b^L)$	$O(b^n)$	$O(b^{n/2})$
Espaço	$O(b^n)$	$O(b^n)$	$O(bm)$	$O(bL)$	$O(bn)$	$O(b^{n/2})$
Ótima?	Sim	Sim	Não	Não	Sim	Sim
Completa?	Sim	Sim	Não	Sim*	Sim	Sim

Tabela 2: Quadro comparativo das estratégias de busca (* se $L \geqslant n$)

Onde *b* é o fator de ramificação (*branching factor*), *n* é a profundidade da solução, *m* é a profundidade máxima da árvore de busca, e *L* é o limite de profundidade.

Evitando estados repetidos

O método de busca pode perder tempo expandindo estados que já foram expandidos antes, em um outro caminho.

Há 3 maneiras de lidar com estados repetidos, listadas em ordem crescente de eficiência e esforço computacional:

1. Não voltar ao estado de onde se acaba de chegar (não gerar nenhum sucessor que seja o mesmo estado que o nodo−pai);
2. Não criar caminhos com ciclos (não gerar nenhum sucessor que seja o mesmo estado que um *ancestral* do nodo);
3. Não gerar estados que já foram gerados antes (manter em memória todos os estados já gerados −tabela *hash*).

Satisfação de Restrições (*Constraint Satisfaction*)

Um problema de satisfação de restrições (CSP), é um tipo especial de problema. Os estados são definidos por valores de um conjunto de *variáveis*, e o teste de objetivo é um conjunto de *restrições* que estes valores devem obedecer.

Cada variável $\boldsymbol{V_i}$ em um CSP tem um *domínio* $\boldsymbol{D_i}$, que é o conjunto de valores que a variável pode assumir. No NQP, com N=8, se V_c é a linha em que a rainha da coluna *c* está, os domínios de V_1 e V_2 são {1,2,3,4,5,6,7,8}. A *restrição* de "não−ataque" ligando V_1 e V_2 é o conjunto de pares {(1,3),(1,4),(1,5),...,(2,4),(2,5),...}, com todos os valores permitidos. Isto elimina 22 das 64 combinações possíveis.

A busca em *profundidade* em um CSP desperdiça tempo quando as *restrições* já foram violadas. Por exemplo, se as duas primeiras rainhas estão na mesma linha, o algoritmo vai testar todas as 8^6 combinações possíveis para as seis rainhas restantes, antes de descobrir que não há solução.

O teste de satisfação de restrição deve ser feito antes da geração dos sucessores. Esse tipo de algoritmo, chamado de busca *backtracking*, volta alguns níveis e recomeça.

Se as posições das 6 primeiras rainhas atacam todas as casas da 8ª coluna, o *backtracking* vai examinar todas as casas possíveis da 7ª coluna em vão. O *forward checking* reduz os domínios das variáveis ainda não instanciadas.