Busca clássica e heurística

Emílio Bergamim Júnior

Instituto de Geociências e Ciências Exatas - UNESP

2025

Conteúdo da aula

- Princípio de formulação de problemas
- Resolução de problemas via algoritmos de busca
- Busca clássica
- Busca heurística

Resolução de problemas

- Por vezes, a solução de um problema passa pela execução de uma sequência de ações e não de uma única ação do agente.
- Na aula passada vimos o exemplo do aspirador de pó, no qual um conjunto de ações que faz o agente percorrer toda a sala resolvia o problema de limpá-la.
- Veja que nesse exemplo há uma **meta** a ser atingida, que é limpar a sala. Uma vez consideradas as informações disponíveis sobre o ambiente e as limitações do agente, passa-se à **formulação do problema**, que considera quais ações e estados devem ser considerados de forma a atingir a meta.
- A procura por uma sequência de ações que resolve um problema é denominada **busca**.

A formulação dos problemas I

Um problema pode ser definido formalmente através de cinco propriedades:

1. O estado inicial S_0 do agente
2. Um conjunto de ações determinado por um estado S. Isto é, existe uma função $Actions(S)$ que retorna aquilo que é possível ao agente em um determinado estado
3. Um modelo de transição que especifica quais as consequências de uma ação. Ou seja, uma função $T(S, A)$ que retorna o resultado da ação A em um estado S.
   - Aqui estabelece-se um **espaço de estados** que pode ser representado por um grafo: os estados são os nós deste grafo, enquanto as ações são as arestas que conectam os estados.
4. Uma **função de teste para a meta** na qual verifica-se se a meta foi atingida ou não, de forma a prosseguir na busca ou pará-la.

A formulação dos problemas II

5. Um **custo de caminho** para uma sequência de ações. Isto é, para uma sequência de ações $(A_1, A_2, ..., A_N)$ existe um número real G que determina o custo de executar estas ações. Como cada ação leva a um novo estado, pode-se pensar num custo $g(S_i, A_i, S_{i+1})$ de tal forma que o custo total seja

$$G(S_{N+1}) = \sum_{i=1}^{N} c(S_i, A_i, S_{i+1}). \quad (1)$$

Uma **solução** de um problema é então uma sequência de ações que atinge o estado-meta. A qualidade de uma solução é medida pelo custo de caminho, que consiste de uma métrica de performance interna do agente. Uma **solução ótima** tem o menor custo de caminho entre todas as soluções.

Modelos e abstrações

- No exemplo da aula passada, a sala foi descrita como um ambiente retangular e sem obstáculos. Isso não é nada parecido com cômodos de uma casa real, que podem ter formas geométricas variadas e objetos situados no percurso do agente.
- Para alguns ambientes, no entanto, essa formulação pode ser adequada (o dono do aspirador pode retirar os móveis de um cômodo para facilitar a operação do agente).
- A formulação do problema e o comportamento do agente são, portanto, modelos que podem ser úteis em determinadas situações, mas que **NUNCA** podem ser vistos como uma síntese da realidade.
- O processo de descartar detalhes de uma representação é chamado de **abstração**.

Uma frase para a vida

"Todos os modelos estão errados, mas alguns são úteis." - *George Box*

Um problema para praticar

- Problemas reais são aqueles de fato importantes para alguém. Geralmente, no entanto, podem ser muito difíceis e demandar esforços que não são cabíveis a uma disciplina de graduação.
- Um exemplo clássico de problema real e extremamente difícil é o do caixeiro viajante, no qual dada uma lista de cidades e distâncias entre estas deseja-se encontrar o menor caminho possível para visitar todas as cidades sem repetir nenhuma.
  - Encontrar um caminho capaz de realizar a tarefa pode ser até simples, mas a condição de optimalidade torna o problema computacionalmente intratável (dito *NP-Hard*).
- Nessa aula, vamos nos ater a um problema simples, mas que já é suficientemente complicado: *8-puzzle*.

Um problema para praticar

No jogo chamado *8-puzzle* número de 1 a 8 são distribuídos em um *grid* 3×3, restando um espaço vazio que permite o movimento das peças adjacentes. O objetivo é então ordenar os números de 1 a 8 conforme mostrado na figura.

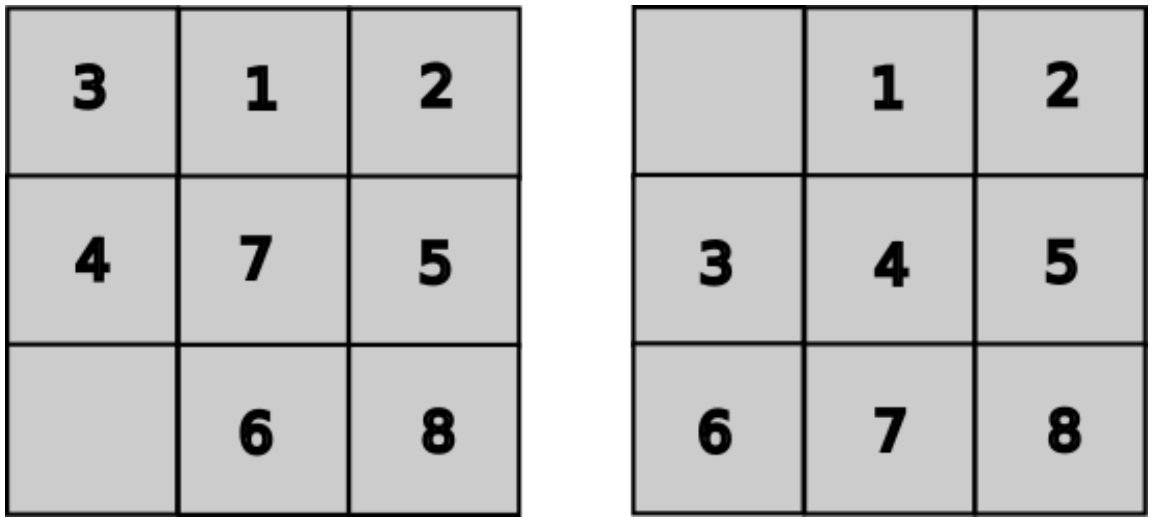

Figura: Exemplo de uma configuração inicial (esq.) e da configuração meta para o *8-puzzle*.

Um problema para praticar

1. Um estado do jogo é descrito unicamente pela configuração das peças.
2. O estado inicial poderia, *a priori* ser qualquer um. No entanto, há questões combinatórias envolvidas: para um dado estado inicial, apenas metade das configurações possíveis é atingível a partir do mesmo. Logo, **existem configurações para as quais o jogo nunca pode ser vencido**.
3. As ações possíveis a um agente que deseja resolver o problema são os movimentos possíveis do espaço vazio.
4. Para uma dada configuração do *grid*, o modelo de transição é o resultado de uma ação.
5. A checagem da meta consiste em verificar se o *grid* está na configuração desejada.
6. Cada movimento tem custo 1, então o custo de um caminho é igual ao número de passos no caminho.

Exercício sugerido

Faça a formulação do problema para o exemplo do aspirador bidimensional visto na última aula, descrevendo as 5 propriedades discutidas nesta aula.

Buscando por soluções

- Como dito anteriormente, o modelo de transição permite ver o espaço de estados como um grafo no qual as ações conectam os estados.
- Assim, a solução de um problema passa pela busca em um grafo: começando de um estado inicial, deseja-se encontrar um caminho até a solução.
  - No que encontrar a melhor solução pode tornar-se excepcionalmente difícil caso seja necessário checar todos os caminhos possíveis. Vamos nos ater a encontrar apenas uma solução do *8-puzzle*, sem demandar optimalidade desta.
- Vamos explorar primeiro métodos clássicos de **força-bruta** que, no pior dos casos, checam todos os caminhos possíveis.

Medidas de performance para algoritmos de busca

Para algoritmos de busca, existem quatro questões de interesse:

- **Completude:** o algoritmo consegue sempre encontrar uma solução quando esta existe?
- **Optimalidade:** o algoritmo consegue uma solução ótima?
- **Complexidade temporal:** quanto tempo leva para encontrar uma solução?
- **Complexidade de armazenamento:** quanta memória é necessária para encontrar uma solução?

Uma base para os algoritmos de busca

- Vamos nos ater ao caso em que o grafo em questão é uma árvore, mas o algoritmo descrito a seguir serve para qualquer grafo. A implementação trazida combina algumas facilidades da topologia de árvores, mas o algoritmo em questão é geral.
- A estrutura básica é um nó (*node*), que armazena o estado atual do jogo, o estado anterior (o pai do nó), a ação executada anteriormente para gerar o estado atual e o custo de caminho para chegar no estado atual.
- Dado um conjunto de ações possíveis em um estado, um nó filho é gerado para cada um destas.
- Estruturas de fila e pilha são úteis nesse caso para auxiliar na implementação dos algoritmos. Isso será discutido a seguir.

Busca em largura

- A ideia da busca em largura consiste de expandir (isto é, gerar filhos) dos nós em uma mesma altura da árvore de busca.
- Assim, dado um nó pai n_P com um conjunto de filhos $\{n_i(n_P)\}_{i=1}^{N_p}$, para cada um destes faz se a expansão

$$Exp(n_i) = \{n_j(n_i)\}_{j=1}^{N_i} \tag{2}$$

  e então prossegue-se para expansão dos filhos dos filhos e assim por diante.
- Há dois critérios de parada para o algoritmo:
  - O estado buscado é encontrado e então retorna-se o nó que armazena o mesmo.
  - Todos os estados são explorados e não encontra-se o estado buscado.

Busca em largura - a implementação

- Ao fazer a expansão de um nó, seus filhos são inseridos em uma estrutura de **fila**
- Uma fila é uma lista com uma dinâmica do tipo **FIFO** (*First In First Out*), na qual os nós que foram inseridos primeiro, são retirados primeiro. Na implementação discutida, são necessárias apenas algumas das funções desta estrutura de dado:
  - *insert*, que insere um nó na última posição da fila.
  - *pop*, que retorna o nó que foi inserido primeiro, removendo-o da fila.
  - *isIn*, que checa se um nó já foi inserido na fila, de forma a evitar a expansão de um estado já visitado.

Busca em largura - o algoritmo

Busca em largura

Entrada: estado inicial S_0. **Saída:** nó que resolve o problema ou erro.

- nd $\leftarrow$ nóFilho(NULL,S_0,NULL)
- **Se** S_0 é a meta **então** retorna nd
- Inicializa duas filas: F (fronteira) e E (explorados) e adiciona nd a ambas
- **Enquanto** F não estiver vazia, faça
  - p = $pop(F)$
  - **Para cada** ação A possível no estado p.S
    - filho = nóFilho(p,p.S,A)
    - **Se não** $isIn(E, filho)$:
      $\rightarrow$ **Se** filho.S é a meta **então** retorna filho
      $\rightarrow$ $insert(F$,filho), $insert(E$,filho)
- $Print$("Estado não encontrado")

Busca em largura - o algoritmo I

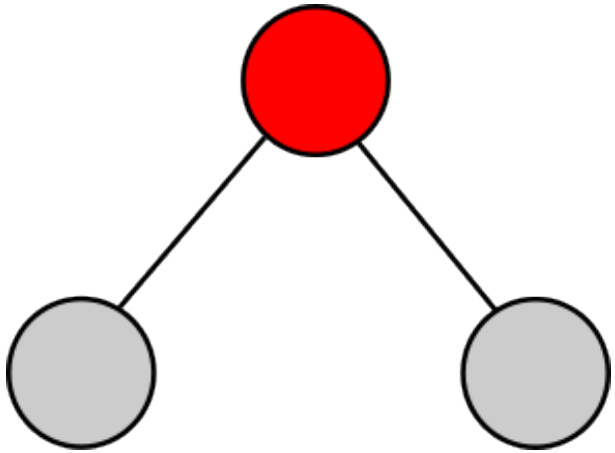

Figura: Primeira expansão de uma busca em largura.

Busca em largura - o algoritmo II

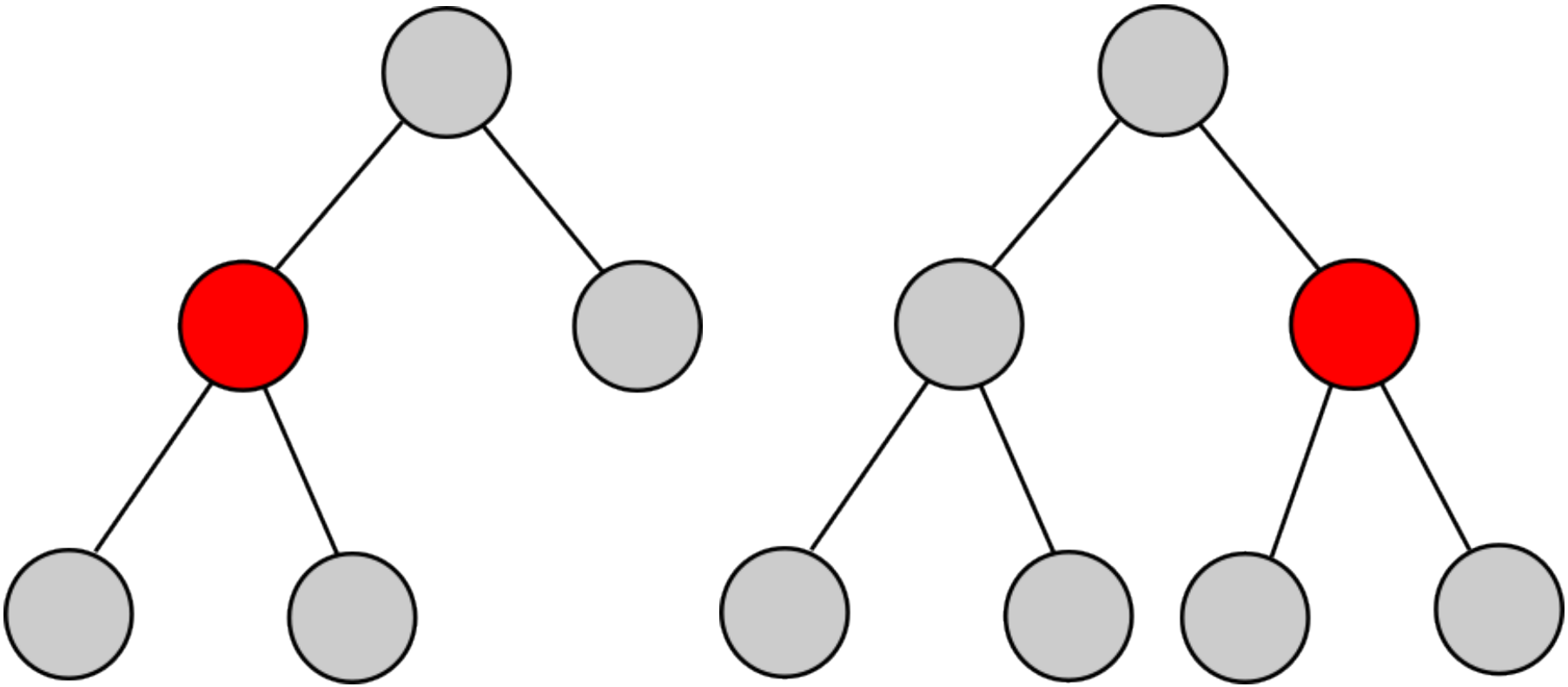

Figura: Expansão dos dois primeiros filhos.

Busca em largura - o algoritmo III

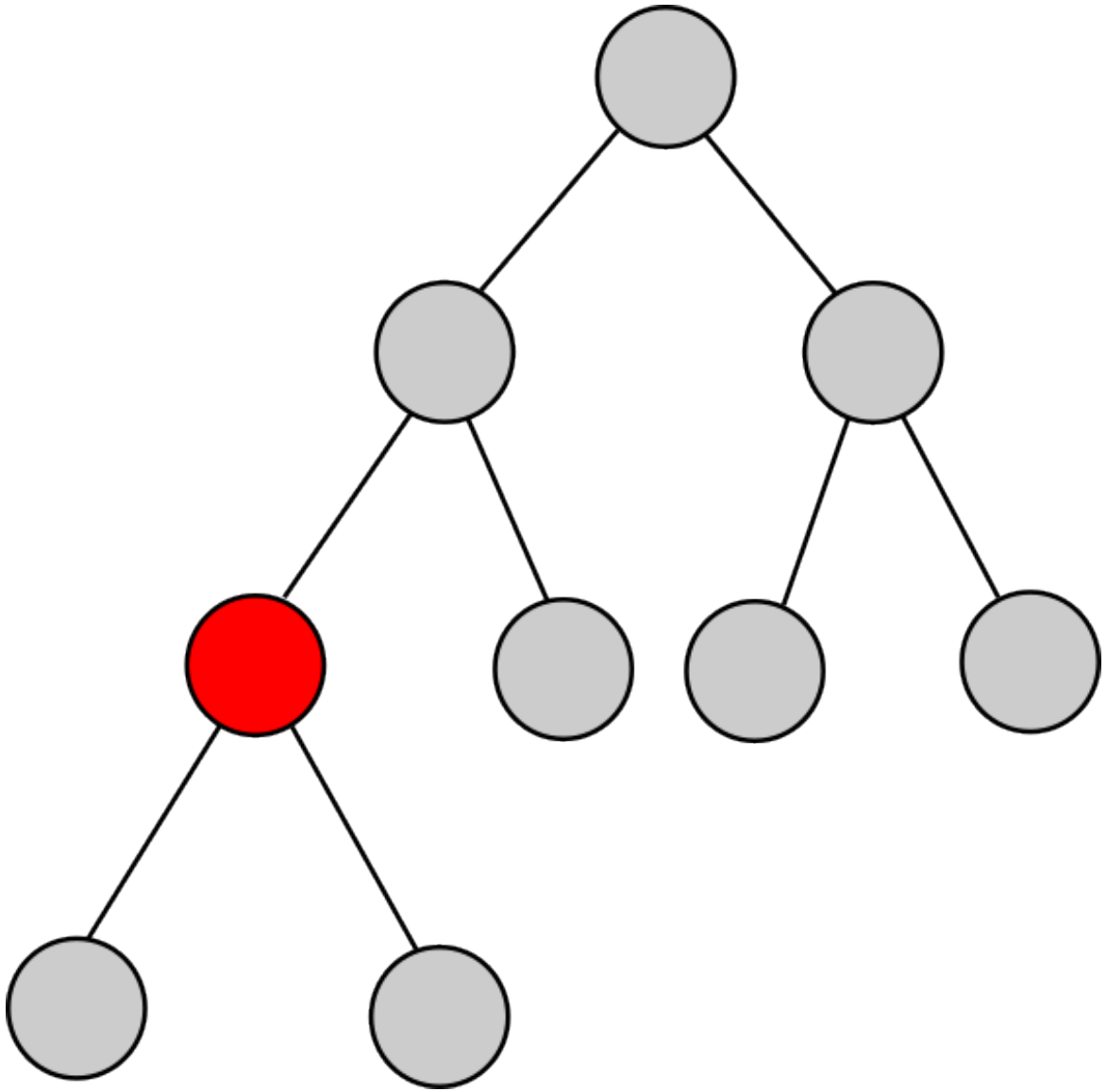

Figura: E assim sucessivamente.

Busca em largura - a análise I

- O número de filhos gerados por cada nó é dito o fator de ramificação do problema e será denotado por b.
- Assim, a primeira expansão gera b filhos, enquanto a segunda gerará b^2, a terceira b^3 e, para um nível d, b^d filhos serão gerados.
- Assim, o custo computacional para chegar ao nível d da árvore de busca é

$$K = \sum_{i=1}^{d} b^i \quad \rightarrow \quad K \in O(b^d) \tag{3}$$

- Aqui, d representa a quantidade passos necessários para atingir a meta de busca e, caso seja suficientemente grande, pode demandar uma quantidade de tempo e memória exorbitante para resolução do problema.
- O algoritmo sempre encontra um solução?

Busca em largura - a análise II

- Sim, desde que esta de fato exista no espaço de busca. Como já discutido para o *8-puzzle*, para determinadas configurações o estado meta é inatingível.
- Outras questão sobre a busca em largura é que o armazenamento dos nós já explorados pode demandar uma quantidade exorbitante de memória em função do tamanho do espaço de estados. Isso acaba sendo um impeditivo maior que o tempo de execução, pois é mais viável deixar um computador ligado por um longo período de tempo, mas raramente será possível conseguir mais memória para uma máquina, especialmente se a demanda por memória cresce exponencialmente a cada nível da árvore.
- A busca em largura é ótima (no sentido de retornar o caminho de menor custo) caso o custo de caminho seja uma função não-decrescente da profundidade d.

Outros algoritmos de busca clássica I

Os algoritmos a seguir são todos modificações da busca em largura.

- **Busca por custo uniforme** troca a estrutura de fila para F e a substitui por uma **fila de prioridade** indexada pelo custo do caminho até cada nó. Assim, os nós de menor custo são expandidos primeiro.
- Em uma ideia similar, **busca em profundidade** visa expandir um nó até chegar no nível mais profundo possível. Para implementar esse algoritmo, basta trocar a fila F por uma pilha, de forma que o primeiro a ser retirado será aquele que foi inserido por último (LIFO - *Last In, First Out*).
  - A complexidade temporal deste algoritmo é então limitada pelo tamanho do espaço de estados. Isto pode vir a ser um problema já que na prática espaços de estados infinitamente grandes são possíveis.
  - Uma alternativa a esse problema é impor um limite L para a profundidade de busca, ignorando as soluções que estão a uma profundidade $d > L$.

Outros algoritmos de busca clássica II

- Na **busca bidirecional** começa-se a busca por um estado S_0 e uma segunda busca em paralelo a partir do estado meta S_M. A expansão de ambos os nós é feita então de forma a encontrar um caminho que passe por um estado comum a ambas as buscas. Ao encontrar tal estado, encontra-se um caminho de S_0 a S_M. Note, no entanto, que tal algoritmo só pode ser aplicável caso saiba-se que S_M está no espaço de busca inicializado por S_0 e vice-versa.

Um exemplo da busca em profundidade

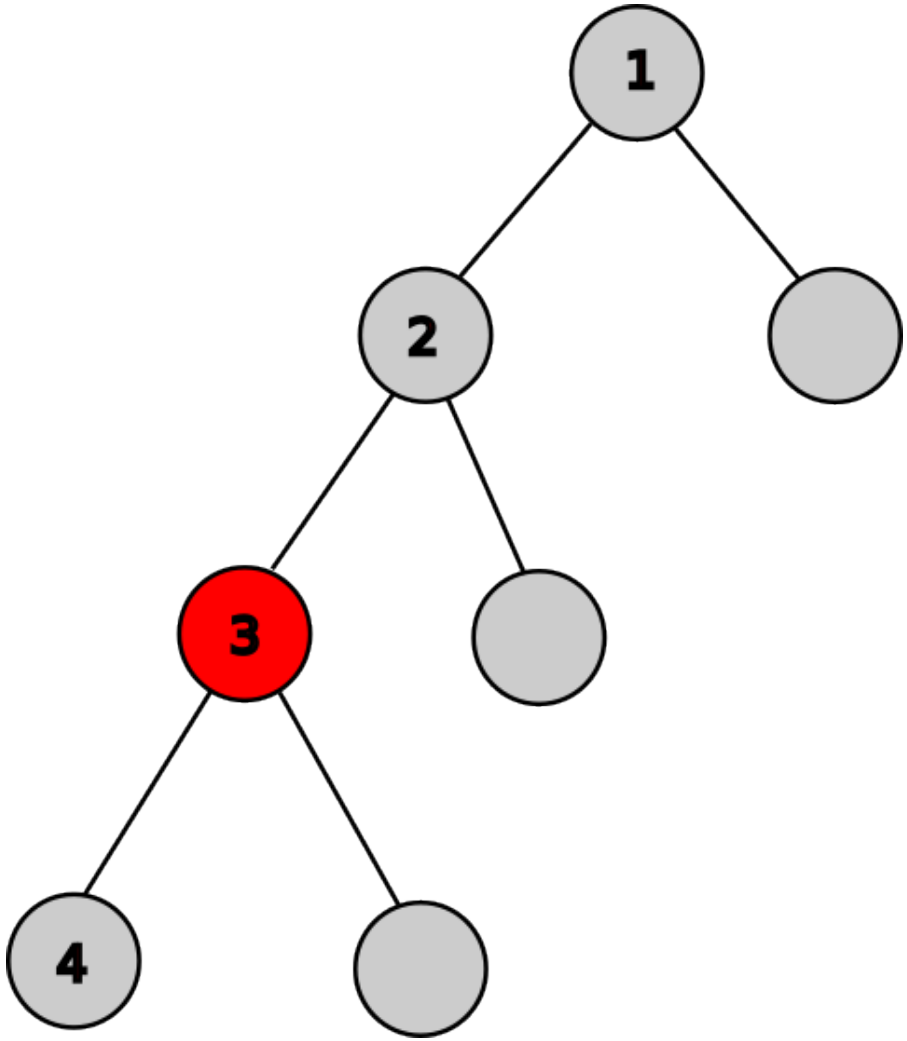

Figura: Exemplo visual do algoritmo de busca em profundidade.

Busca heurística

- Nos algoritmos de busca clássica a questão essencial é achar uma forma de explorar o espaço de busca de forma eficiente, independentemente das especificidades do problema.
- A busca heurística ou busca informada utiliza-se de propriedades do problema de forma a guiar o algoritmo pelo espaço de busca.
- Em espaços de estados grandes demais, esta pode ser a única forma possível de realizar uma busca que retornará algum resultado em um tempo hábil.
- De forma geral, nossa abordagem será a de procurar pelas "melhores soluções"de acordo com alguma métrica e usá-las para orientar o caminho de busca.

Busca heurística

Função heurística

Uma função heurística é uma estimativa do menor custo possível para ir do estado atual para o estado meta do problema.
Será aqui considerada uma estratégia chamada de **melhor-primeiro**, na qual a cada nó expandido escolhe-se o filho mais próximo do objetivo de acordo com uma dada heurística $h(n)$, sendo n um nó filho.

Busca melhor-primeiro gulosa I

- Neste caso, o custo estimado de um nó será igual à heurística imposta:

$$f(n) = h(n) \tag{4}$$

- No problema *8-puzzle*, tomamos $h(n)$ como o **número de peças fora do lugar em relação à meta**
- Na figura mostrada anteriormente, a configuração inicial exibida possui 5 peças fora da lugar em relação à metal. Logo, partiria de $h(n_0) = 5$. Ao expandir esse nó, o próximo seria aquele em que

$$n_1 = \min_{n' \in Exp(n_0)} f(n'), \tag{5}$$

sendo esta a estratégia geral da busca melhor-primeiro e que nesse caso reduz-se a

$$n_1 = \min_{n' \in Exp(n_0)} h(n'). \tag{6}$$

Busca melhor-primeiro gulosa II

- Essa estratégia é completa?
  - Não. O melhor caminho até uma solução nem sempre passa por uma sucessão de soluções **localmente ótimas**. Adotar essa hipótese pode inclusive levar o algoritmo a um *loop* infinito, como por exemplo acontece partindo de

$$\{[0, 2, 5], [1, 3, 8], [6, 4, 7]\}. \tag{7}$$

Busca A*

- Nesta estratégia, o custo estimado corresponde ao custo de caminho adicionado da heurística $h(n)$:

$$f(n) = g(n) + h(n). \tag{8}$$

- Esta abordagem é optimal (em árvores) se a heurística $h(n)$ for **admissível**, que significa que esta **nunca superestima o custo de alcançar a meta**.
- Para optimalidade em grafos, uma condição suficiente é a de **consistência**, na qual, para todo nó n e todo filho n' gerado por uma ação A, a desigualdade triangular abaixo é satisfeita:

$$h(n) \leq c(n, A, n') + h(n'). \tag{9}$$

Distância de Manhattan

- Uma outra heurística possível é a da distância de Manhattan $h_M(n)$ que é a **soma das distâncias de cada peça até cada uma de suas posições no estado meta**.
- Esta é a soma das distâncias horizontais e verticais no *8-puzzle*, já que cada peça só pode se mover nesta direção.
- Exercício sugerido: mostre que, para o *8-puzzle*, ambas as heurísticas $h_M(n)$ e $h_{fora}(n)$ são admissíveis, sendo h_{fora} o número de peças fora do lugar em relação à meta.

Heurísticas em espaços contínuos

- Em espaços contínuos a possibilidade de executar uma busca clássica com força bruta inexiste
- Isso pode ficar claro ao considerar o seguinte resultado: existe uma função bijetora de qualquer intervalo (a, b) no conjunto dos reais $\mathbb{R}$
  - Isso pode ser interpretado como: para cada elemento de (a, b) existe um elemento de $\mathbb{R}$ e vice-versa
  - Ou seja, por "menor"que um conjunto contínuo pareça, encontrar um elemento específico neste possui a mesma dificuldade de encontrar um elemento em $\mathbb{R}$
- Por isso, problemas de busca contínua são usualmente expressos na forma de equações como

$$\boldsymbol{F}(\boldsymbol{x}) = \mathbf{0} \tag{10}$$

Heurísticas em espaços contínuos

- Em espaços contínuos, a busca pode ser orientada por meio das derivadas de uma função.
- Por exemplo, para uma função $f : V \in \mathbb{R} \rightarrow \mathbb{R}$ que seja, para além de contínua, diferenciável, existe o conhecido método de Newton-Raphson

$$x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}. \tag{11}$$

Este pode ainda ser generalizado para o caso multidimensional, no qual $\boldsymbol{F} : V \in \mathbb{R}^p \rightarrow \mathbb{R}^q$ possui sua derivada na forma de uma matriz jacobiana $\boldsymbol{J}_{q \times p}$

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \boldsymbol{J}^{-1}(\boldsymbol{x}_t)\boldsymbol{F}(\boldsymbol{x}_t) \tag{12}$$

- Esse tipo de algoritmo possui diversas questões de convergência, como impossibilidade de inverter $\boldsymbol{J}$ ou uma escolha inadequada da condição inicial $\boldsymbol{x}_0$.

Heurísticas em espaços contínuos

- No caso particular em que $F : A \in \mathbb{R}^p \rightarrow \mathbb{R}$ e deseja-se encontrar um extremo de F,

$$\min_{\boldsymbol{x}} F(\boldsymbol{x}) \quad \text{ou} \quad \max_{\boldsymbol{x}} F(\boldsymbol{x}) \tag{13}$$

  o algoritmo mais comum é chamada descida (subida) do gradiente.

$$\boldsymbol{x}_{t+1} = \boldsymbol{x}_t - \eta \nabla F(\boldsymbol{x}_t), \tag{14}$$

  sendo ∇F o vetor gradiente de F, no qual cada entrada corresponde à derivada parcial de F em relação à variável cabível.

- Essa abordagem também possui problemas. Como os extremos de F situam-se em pontos nos quais $\nabla F = 0$, caso exista mais de um ponto no qual essa propriedade seja satisfeita (extremos locais, por exemplo), o algoritmo ficará parado em uma solução sub-optimal.

Atividade

Implemente a busca A* com a heurística h_{fora} e com a heurística h_M.

Trabalho I

Para o problema do *8-puzzle*, implemente e compare os seguintes algoritmos em termos de tempo de execução e do número de passos fornecidos para cada solução. Faça essa avaliação para diferentes configurações iniciais do problema.

1. Implemente a busca em profundidade.
2. Implemente a busca em profundidade limitada.
3. Implemente a busca bidirecional.
4. Sendo $w \in \mathbb{R}$, o algoritmo do caminho heurístico utiliza a função de avaliação

$$f(n) = (2 - w)g(n) + wh(n). \tag{15}$$

Trabalho II

- Supondo que h seja admissível, para quais valores de w esse algoritmo é completo? Isso significa mostrar que, sendo n_{filho} um nó-filho de n, então um dado w implica que

$$h(n) > h(n_{filho}) \tag{16}$$

  Mostre também que o mesmo generaliza outras abordagens descritas na aula.
- Implemente o algoritmo para as heurísticas h_{fora} e h_M.

Trabalho - análise

- Compare os algoritmos em termos de tempo de execução para encontrar a solução ótima. Considere também a busca em largura e a busca $A*$.
- Para os métodos heurísticos, avalie tanto h_{fora} como h_M.
- Para a busca em profundidade limitada, escolha três limites distintos.
- Para o algoritmo do caminho heurístico, escolha pelo menos três valores distintos de w, sendo ao menos um no regime em que o algoritmo é completo e um no regime incompleto para ilustrar as diferenças.
- Realize pelo menos 10 experimentos para cada método, calcule média e desvio padrão dos resultados. Em cada experimento, utilize uma configuração inicial aleatória (porém, solúvel).
- Apresente seus resultados na forma de tabelas ou gráficos.

Exercício sugerido

- Suponha que duas pessoas vivam em diferentes cidades e desejam se encontrar em uma cidade no meio do caminho. A cada turno, cada uma das pessoas é movida para uma cidade vizinha e isso é repetido de forma que estas se encontrem percorrendo a menor distância possível. Sendo $d(i, j)$ a distância entre cidades i e j, julgue quais (ou qual) das heurísticas abaixo são consistentes para o problema:
  - $d(i, j)$
  - $2d(i, j)$
  - $d(i, j)/2$
- Mostre que toda heurística consistente é admissível (ver material complementar da aula).