Sistemas Inteligentes - Algoritmos de Pesquisa

Alberto Barbosa

Universidade da Maia

alberto.barbosa@umaia.pt

February 17, 2025

Resolução de problemas

Um **agente** passa por um processo de 4 fases para resolver um problema:

- **Objectivo**: Temos de definir o objectivo que pretendemos atingir, definindo as **acções** para lá chegar.
- **Formulação do problema**: O agente formula uma descrição dos **estados** e acções necessárias para atingir um objectivo.
- **Pesquisa**: Antes de agir, o agente simula **sequências de acções** no seu **modelo** até atingir o objectivo. A esta sequência chamamos **solução**. No final, o agente pode chegar a uma possível solução ou concluir que não existe uma solução possível.
- **Execução**: O agente agora pode **executar** as acções da solução.

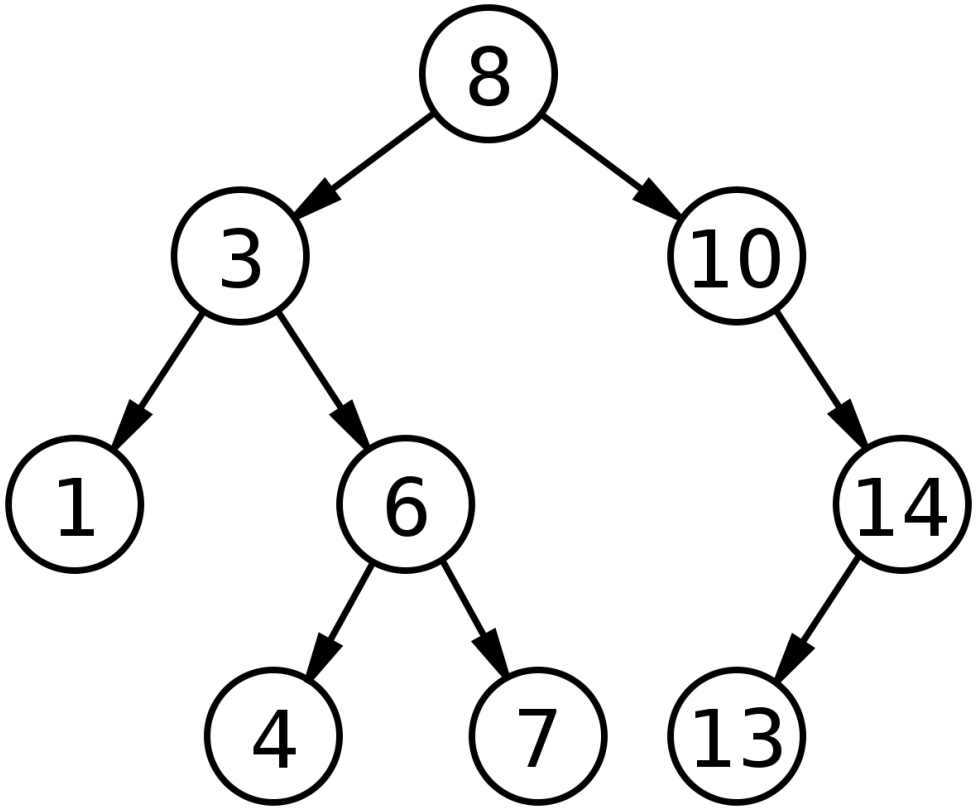
Problemas de pesquisa

Podemos definir um processo de pesquisa da seguinte forma:

- **O espaço de estados**: conjunto de estados possíveis.
- Um **estado inicial**.
- Um ou mais **estados finais**.
- Uma **função de actuação**, onde $Action(s)$ retorna um conjunto finito de acções que podem ser executadas no estado s.
- Um **modelo de transição** que descreve o que cada acção faz. $Result(s, a)$ retorna o estado que resulta de executar a acção a no estado s.
- Uma **função de custo** $c(s, a, s')$ que nos dá o custo numérico de aplicar a acção a no estado s para chegar ao estado s'.

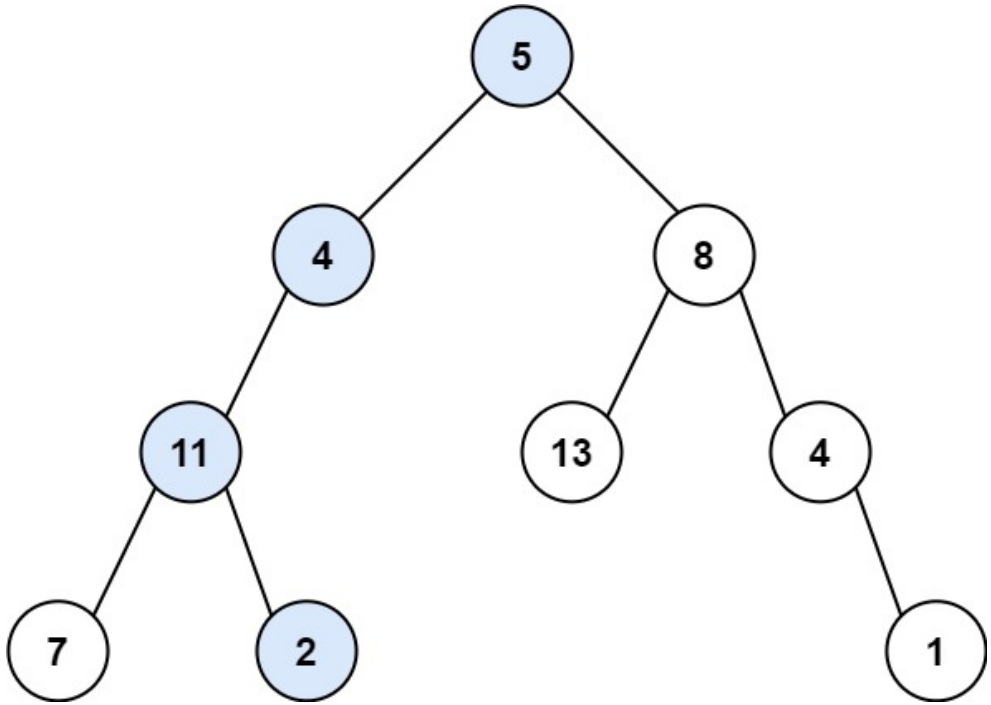
Espaço de Estados

Um **espaço de estados** pode ser representado como um **grafo**, onde cada **vértice** é um **estado** e cada **aresta** entre vértices representa uma **acção**. Uma **árvore de pesquisa** pode ser representada como um grafo onde não existem **ciclos**.



Soluções

A uma **sequência de acções** chamamos caminho e uma **solução** é um caminho desde o **estado inicial** até ao **estado final**.
Podemos assumir que o **custo total** de um caminho é a **soma** dos **custos individuais** de cada acção.
Uma **solução óptima** tem o menor custo entre todas as soluções.



Jogo dos 8

Um puzzle onde dado um estado inicial queremos deslizar as peças para o quadrado em branco até chegarmos ao estado final.

Initial State

1	2	3
8		4
7	6	5

Goal State

2	8	1
	4	3
7	6	5

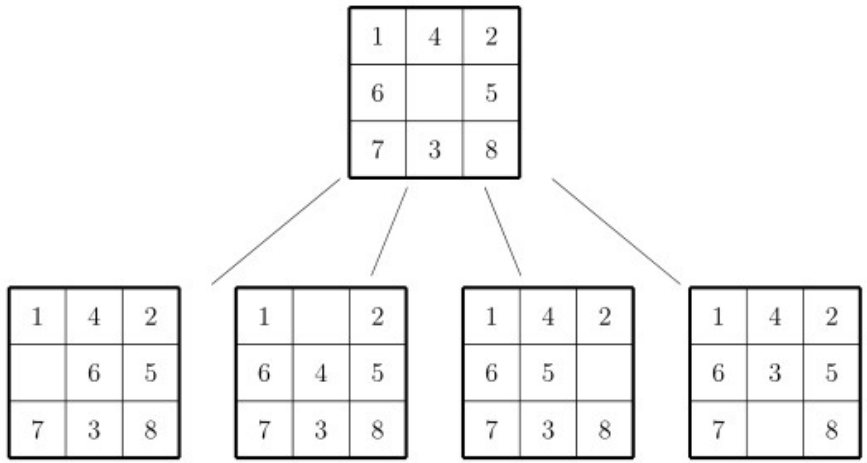
Jogo dos 8

O **jogo dos 8** pode ser formalizado desta forma:

- **Estados**: Cada estado especifica a posição de cada um dos **tiles**.
- **Estado inicial**: Qualquer configuração **válida** pode ser um estado inicial.
- **Acções**: A forma mais simples será do ponto de vista do espaço em branco: *Esquerda*, *Direita*, *Cima* e *Baixo*.
- **Modelo de Transição**: Mapeia um **estado** e **acção** a um **estado resultante**.
- **Objectivo**: Qualquer configuração **válida** pode ser um estado final.
- **Custo**: Cada acção tem **custo 1**.

Algoritmos de pesquisa

Um **algoritmo de pesquisa** aceita um **problema** de pesquisa como *input* e retorna uma **solução** ou indicação de falha como *output*.
Vamos considerar algoritmos que modelam o espaço de estados como uma **árvore de pesquisa**, onde queremos encontrar um caminho entre o estado **inicial** e o estado **final**.
Cada **nó** na árvore corresponde a um **estado** e cada **aresta** corresponde a uma **acção**.
A **raiz** da árvore é o estado inicial.



Algoritmos de pesquisa - Implementação

Algumas estruturas de dados relevantes para implementação de cada nó:

- *node*.**STATE**: estado **representado** pelo nó.
- *node*.**PARENT**: nó na árvore que **gerou** este nó.
- *node*.**ACTION**: **acção** que levou à geração deste nó.
- *node*.**COST**: **custo** desde o estado inicial até este nó.
- *node*.**CHILDREN**: nós **gerados** por este nó.

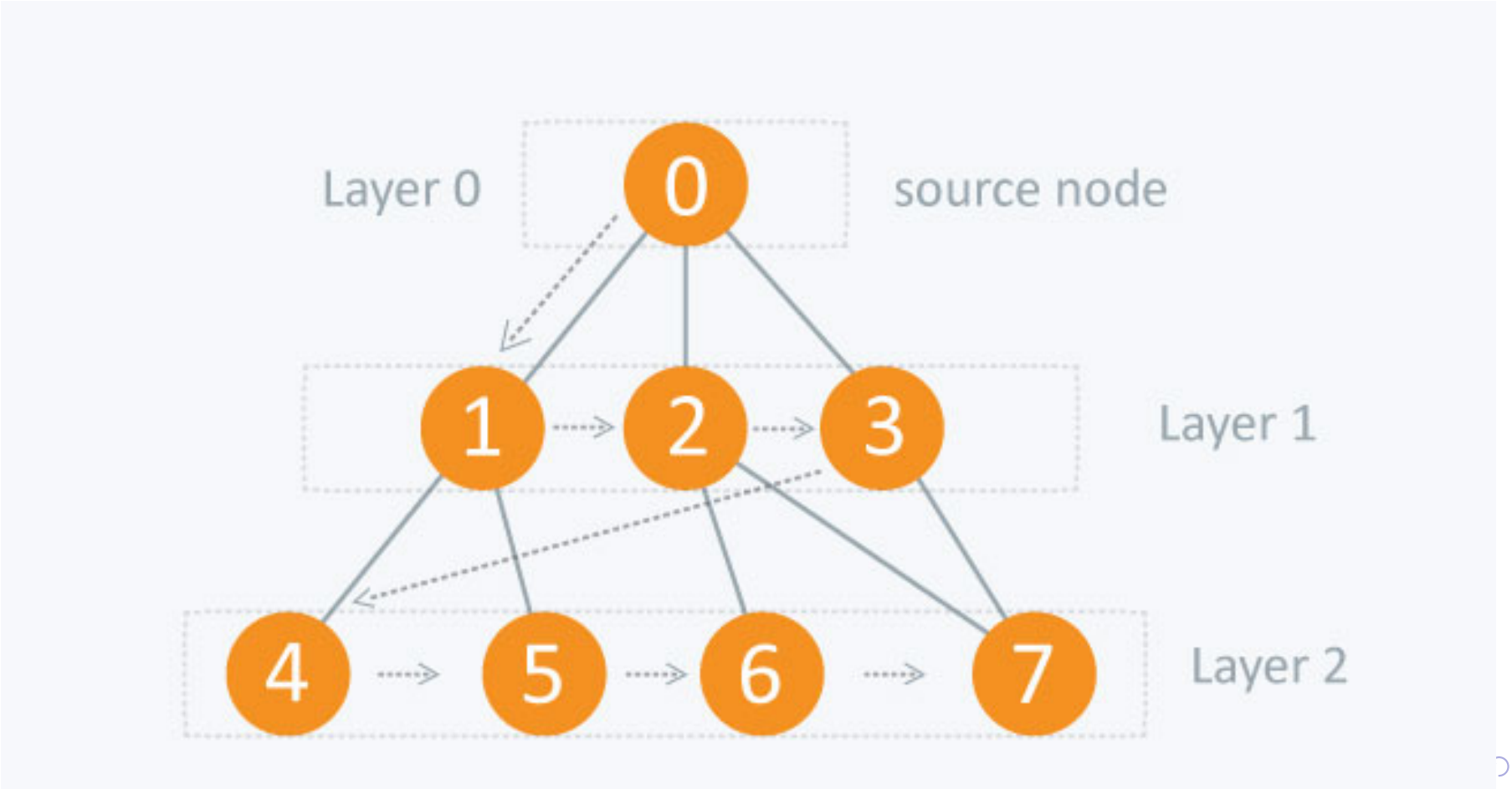
Algoritmos de pesquisa - Implementação

Estruturas de dados relevantes para representar nós a expandir (**fronteira**):

- *empty*(*frontier*): verdade se **não existirem** nós a expandir.
- *pop*(*frontier*): **remove** o próximo nó a expandir da fronteira.
- *top*(*frontier*): **retorna** (mas não remove) o próximo nó a expandir.
- *add*(*node*,*frontier*): **insere** o nó na estrutura na posição correcta.

Algoritmos cegos - BFS

Um algoritmo **cego** não tem qualquer pista sobre quão perto está de uma possível solução.
Um destes algoritmos é a **pesquisa em largura** ou **BFS (breadth first search)**.



Algoritmos cegos - BFS

Na pesquisa em largura, a **raíz** é expandida em **primeiro lugar**, depois **todos** os seus **sucessores** são expandidos, depois **todos** os **sucessores** destes são expandidos, e assim sucessivamente.
A BFS encontra **sempre** a **solução óptima** uma vez que quando geramos os nós com profundidade d, todos os nós com profundidade $d-1$ já foram gerados.
No entanto, apenas para problemas onde todas as **acções** têm o **mesmo custo** e encontra sempre uma solução, caso exista (é **completa**).
Complexidade **temporal** e **espacial** é $\mathcal{O}(b^d)$, sendo b o número de sucessores que cada nó pode gerar e d a profundidade da árvore.

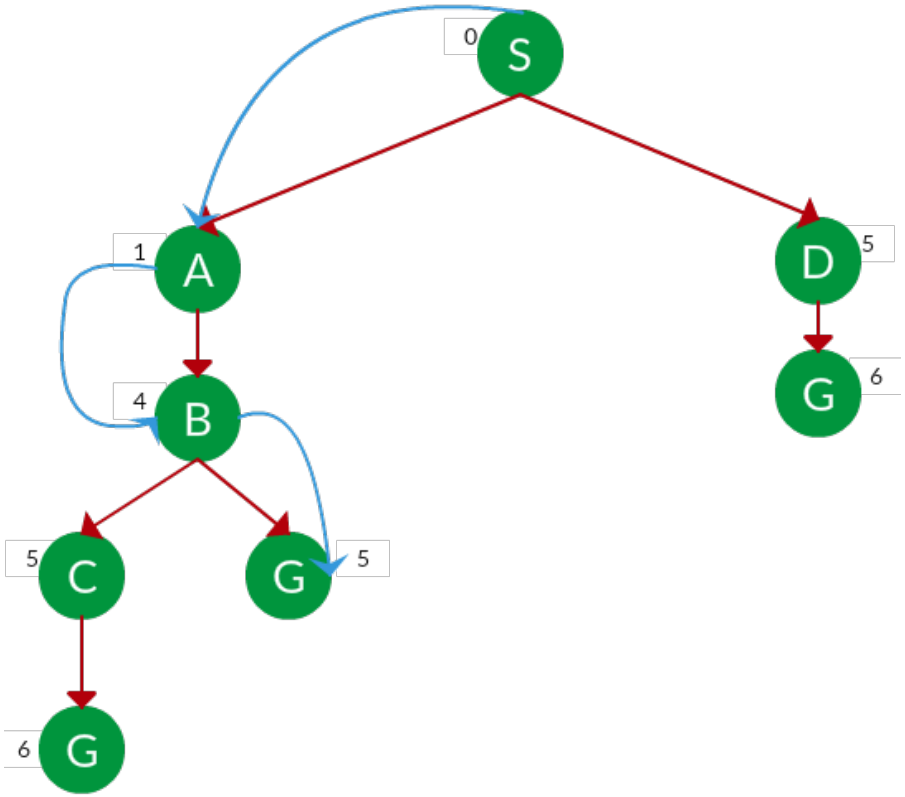
Algoritmos cegos - BFS

Algorithm 1 BFS(problem)

```
node ← NODE(problem.INITIAL)
if problem.IS-GOAL(node.STATE) then
    return node
end if
frontier ← FIFO queue with node
reached ← {problem.INITIAL}
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for all child in EXPAND(problem,node) do
        s ← child.STATE
        if problem.IS-GOAL(s) then
            return child
        end if
        if s is not in reached then
            add s to reached
            add child to frontier
        end if
    end for
end while
return failure
```

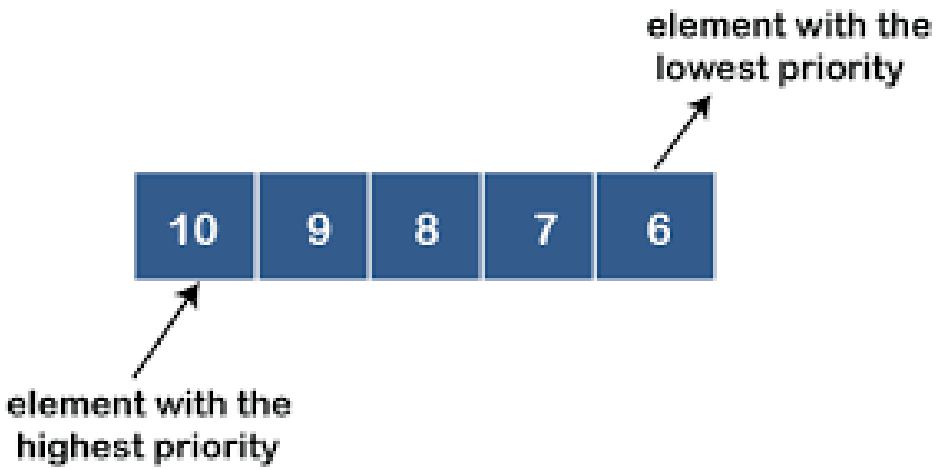
Algoritmos cegos - Pesquisa com custo uniforme (Dijkstra)

Caso tenhamos **custos diferentes** associados a cada acção, uma forma de solucionar o problema será utilizar uma variação do BFS, onde, ao invés de expandirmos os nós pela sua **profundidade** (isto é: expandir primeiro os com profundidade 1, depois com profundidade 2, etc), expandimos pelo **custo** (primeiro com custo 1, depois com custo 2, etc).



Algoritmos cegos - Pesquisa com custo uniforme (Dijkstra)

O algoritmo é então muito semelhante ao BFS, no entanto o que dita o próximo nó a ser expandido é o **custo** associado ao mesmo.
Na prática, teremos de mudar a estrutura de dados que serve como implementação da fronteira. Deixará de ser FIFO para passar a ser uma Priority Queue (Heap).



Algoritmos cegos - Pesquisa com custo uniforme (Dijkstra)

Caso todas as acções tenham o mesmo custo, o algoritmo tem a mesma complexidade espacial e temporal que o BFS.
Caso contrário tem pior complexidade temporal e espacial $\mathcal{O}(b^{1+C^*/\epsilon})$.
O algoritmo é **completo** e **óptimo**, porque a primeira solução encontrada terá sempre o menor custo, uma vez que todos os caminhos são percorridos por ordem crescente de custo.

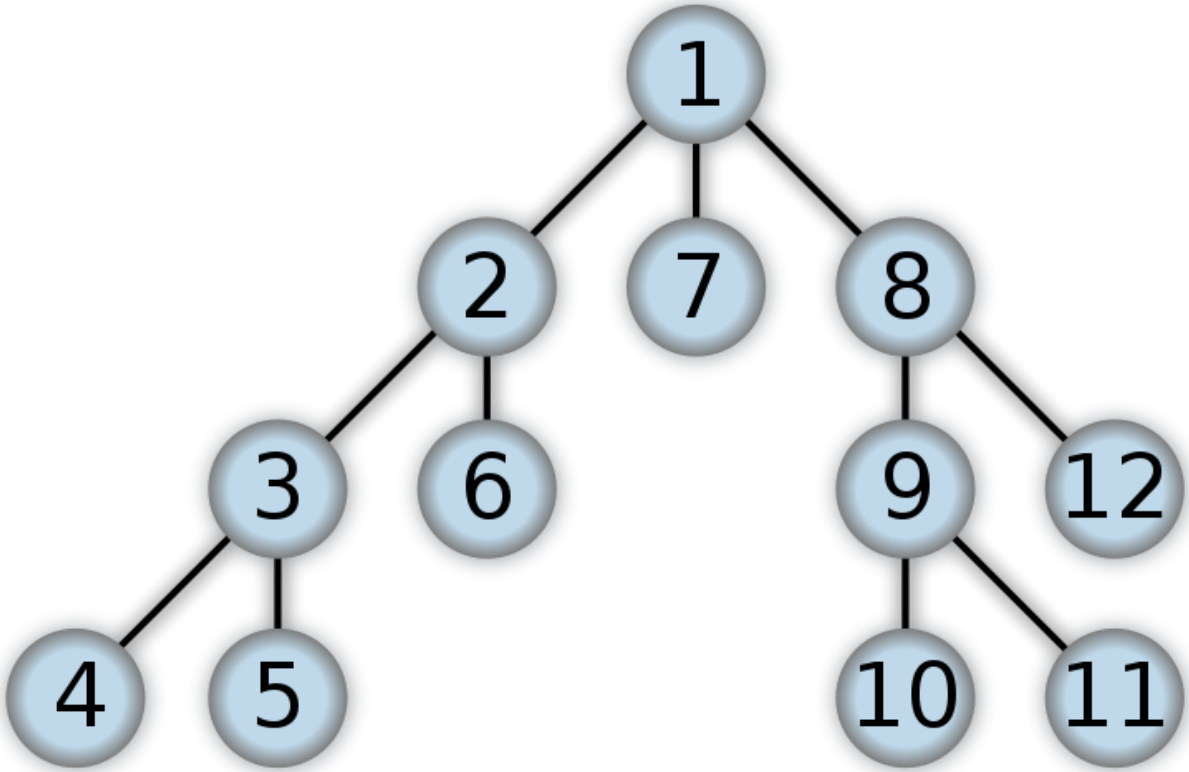
Algoritmos cegos - UCS

Algorithm 2 UCS(problem)

```
node ← NODE(problem.INITIAL)
if problem.IS-GOAL(node.STATE) then
    return node
end if
frontier ← Priority Queue with node
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then
        return node
    end if
    for all child in EXPAND(problem,node) do
        add child to frontier
    end for
end while
return failure
```

Algoritmos cegos - DFS

O algoritmo **DFS** (Depth-first search), irá explorar o algoritmo mais profundo primeiro. Isto é, aquele, dos que estão na fronteira, que se encontra a **maior profundidade** na árvore de pesquisa.



Algoritmos cegos - DFS

A implementação é muito semelhante ao BFS. No entanto, é necessário mudar a estrutura de dados que serve de apoio à fronteira. Deixará de ser **FIFO** e passará a ser **LIFO** (uma pilha por exemplo).
O DFS **não é óptimo**. A solução retornada não é necessariamente a que custa menos.
Em espaços de estados finitos, é **eficiente** e **completo**.
Em espaços de estados cíclicos, pode ficar preso num ciclo, pelo que se torna fundamental a **verificação de estados repetidos** aquando da implementação do DFS.
Em espaços de estados infinitos, pode nunca encontrar uma solução, deixando de ser um algoritmo completo.

Algoritmos cegos - DFS

No entanto, consome menos memória que o BFS (caso não seja necessário fazer verificação de repetidos) e tem fronteiras mais pequenas.
Tem complexidade temporal $O(b^m)$ e complexidade espacial $O(bm)$.
Devido à forma como utiliza pouca memória, acaba por estar na base de muitos algoritmos clássicos de diferentes áreas de IA, como **satisfação de restrições** e **programação lógica**.

Algoritmos cegos - DFS

Algorithm 3 DFS(problem)

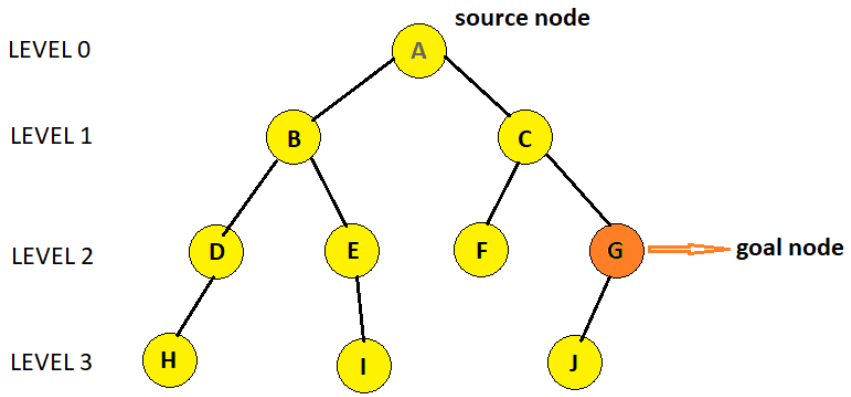
```
node ← NODE(problem.INITIAL)
if problem.IS-GOAL(node.STATE) then
    return node
end if
frontier ← LIFO stack with node
reached ← {problem.INITIAL}
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for all child in EXPAND(problem,node) do
        s ← child.STATE
        if problem.IS-GOAL(s) then
            return child
        end if
        if s is not in reached then
            add s to reached
            add child to frontier
        end if
    end for
end while
return failure
```

Algoritmos cegos - IDFS

Para impedirmos que o DFS corra **infinitamente** ao longo de um caminho, podemos alterar um pouco a sua implementação, de modo a que **limitemos** a **profundidade máxima** que a árvore pode atingir.
Isto significa que iremos determinar uma profundidade **limite** l e que iremos tratar todos os nós na profundidade l como se não tivessem descendentes.
A complexidade temporal é $\mathcal{O}(b^l)$ e espacial é $\mathcal{O}(bl)$.
O mais difícil neste caso é acertar na profundidade máxima. Isto é, como é que sabemos qual é a profundidade **certa** para cortar o nosso algoritmo?

Algoritmos cegos - IDDFS

Para tal, podemos utilizar o IDDFS (Iterative Deepening Depth-first search), onde vamos **incrementando** o limite máximo de profundidade a cada passo. Isto é, primeiro executamos um DFS com limite de profundidade igual a 1, depois com limite igual a 2, depois com limite igual a 3, etc.
Podemos aumentar este limite até que encontremos a mínima profundidade para a qual temos uma solução.



IDDFS with max depth-limit = 3
Note that iteration terminates at depth-limit=2
Iteration 0: A
Iteration 1: A->B->C
Iteration 2: A->B->D->E->C->F->G

Algoritmos cegos - IDFS

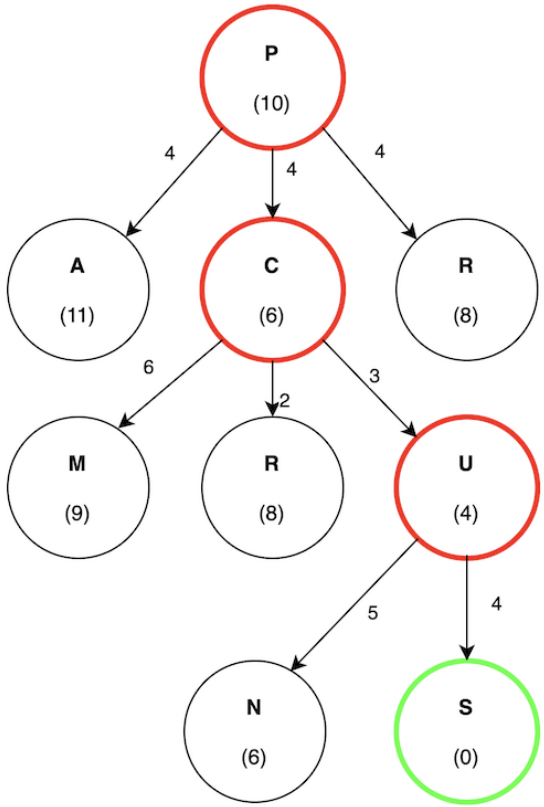
A complexidade espacial é reduzida: $\mathcal{O}(bd)$ quando existe uma solução ou $\mathcal{O}(bm)$ quando é um estado de espaços finito sem solução.
IDFS é **óptimo** para problemas onde todas as acções tenham o mesmo custo e é **completo** (com verificação de repetidos, caso existam ciclos).
Complexidade temporal é $\mathcal{O}(b^d)$ quando há solução e $\mathcal{O}(b^m)$ quando não há.

Algoritmos informados - Heurísticas

Os **algoritmos informados** têm por base uma função que tenta prever **quão longe** estamos de uma possível solução.
A essa função chamamos **função heurística**, denotada $h(n)$.
No jogo dos 8, uma função heurística pode ser o **número de peças fora do sítio**, que nos indica par um dado estado, quantas peças estão fora da sua posição final. Quanto maior o número de peças fora do sítio, mais longe estamos de uma solução.

Algoritmos informados - Greedy Best First

Um algoritmo **Greedy best-first** é um algoritmo que vai expandir sempre o nó com **menor valor** de $h(n)$ dentro dos nós na fronteira.



Node[cost]
N[6]
S[0]

Closed List
P
C
U

Algoritmos informados - Greedy Best First

Esta estratégia é **completa** em espaços finitos, na medida em que retorna sempre solução, no entanto **não é óptima** (depende muito da qualidade da heurística).
A complexidade espacial e temporal é $O(|V|)$, podendo ser reduzida até para $O(bm)$ com uma boa escolha de função heurística.

Algoritmos cegos - Greedy

Algorithm 4 Greedy(problem)

```
node ← NODE(problem.INITIAL)
if problem.IS-GOAL(node.STATE) then
    return node
end if
frontier ← Priority Queue with node and h(node)
reached ← {problem.INITIAL}
while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for all child in EXPAND(problem,node) do
        s ← child.STATE
        if problem.IS-GOAL(s) then
            return child
        end if
        add child to frontier with h(child)
    end for
end while
return failure
```

Algoritmos informados - A*

O algoritmo A* é um **algoritmo best-first** que utiliza a seguinte função de avaliação:

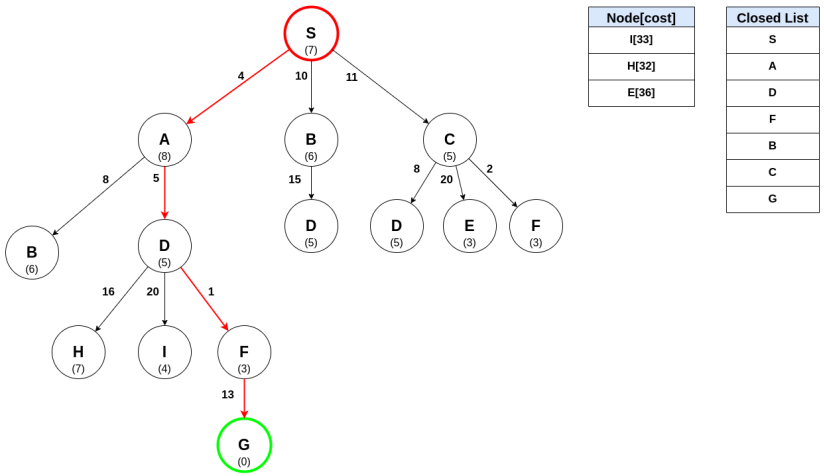
$f(n) = g(n) + h(n)$

onde $g(n)$ é o **custo** desde o nó inicial até ao estado n e $h(n)$ é o **custo estimado** desde n até ao estado final.

O algoritmo A* é **completo**. No entanto, a questão da optimalidade está dependente de certas características da função heurística.

Para tal acontecer, a heurística deve ser **admissível**, isto é, nunca deve superestimar o custo para chegar a uma solução. Deve ser **optimista**.

Caso se utilize uma heurística admissível, A* é **óptimo**.



Algoritmos informados - A* (Demonstração)

Se existe um caminho com custo C^* mas o algoritmo retorna um caminho com custo $C > C^*$, então existe um nó n que está no caminho óptimo, mas não é expandido.
Então usando $g^*(n)$ como o custo óptimo desde o nó inicial até n e $h^*(n)$ como o custo óptimo desde n até ao final, temos:

1. $f(n) > C^*$
2. $f(n) = g(n) + h(n)$
3. $f(n) = g^*(n) + h(n)$ (porque n está no caminho óptimo)
4. $f(n) \leq g^*(n) + h^*(n)$ (porque h é admissível)
5. $f(n) \leq C^*$ (por definição $C^* = g^*(n) + h^*(n)$

A primeira linha e a última entram em **contradição**, pelo que é impossível que A* não retorne o caminho óptimo nestas condições.

Algoritmos informados - A*

Uma propriedade muito importante é também a **consistência**.
Uma heurística $h(n)$ é consistente se para cada nó n e cada sucessor de n, n', gerado por uma acção a, temos:

$$h(n) \leq c(n, a, n') + h(n') \text{ (desigualdade triangular)} \quad (1)$$

Todas as heurísticas consistentes são admissíveis, mas uma heurística admissível pode não ser consistente.
Se a heurística for consistente, a primeira vez que cada nó é expandido, já será quando o mesmo faz parte de um **caminho óptimo**, fazendo com que não seja necessário expandir várias vezes o mesmo nó.

Algoritmos informados - A*

O mesmo não acontece com uma heurística **inconsistente**, podendo ter maiores custos de tempo e memória.
Com uma heurística não admissível, A* pode ou não ser óptimo.
No entanto, pode ser óptimo se $h(n)$ for admissível para todos os inconsistente ou se o problema tiver uma solução óptima com custo C^* e a segunda melhor solução tiver custo C_2 e a superestimativa de $h(n)$ nunca é superior a $C_2 - C^*$.

Algoritmos informados - A*

À medida que estendemos um caminho, os custos de g vão aumentando, uma vez que todas as acções têm um custo positivo. Isto significa que $g(n)$ é uma função **monótona**.
No entanto, não é óbvio que $f(n) = g(n) + h(n)$ vá aumentar monotonamente. Ao expandir um caminho de n para n', o custo vai de $g(n) + h(n)$ para $g(n) + c(n, a, n') + h(n')$. Então o caminho aumentará monotonamente se e só se $h(n) <= c(n, a, n') + h(n')$, ou seja, se $h(n)$ for **consistente**.
Se C^* é o custo do caminho óptimo:

- A* expande todos os nós que podem ser alcançados a partir do nó inicial num caminho onde cada nó tem $f(n) < C^*$.
- A* pode expandir alguns nós com $f(n) = C^*$, caso a diminuição em h corresponda ao aumento em g.
- A* não irá expandir nenhum nó com $f(n) > C^*$.

Algoritmos informados - A*

Dizemos que A*, com uma heurísitca **consistente**, é **optimamente eficiente**: qualquer algoritmo que estenda caminhos a partir do nó inicial com a mesma heurística, irá expandir todos os nós com $f(n) < C^*$ expandidos por A*.
Nos nós com $f(n) = C*$, pode haver mais variabilidade (não relevante para a questão da eficiência óptima).
A* é **completo**, **óptimo** e **optimamente eficiente**. No entanto não é a resposta para todos os problemas de pesquisa. Isto porque para muitos problemas, o número de nós expandidos pode ser exponencial no comprimento da solução.

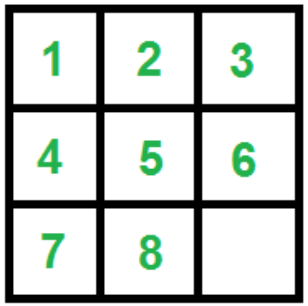
Algumas heurísticas - Jogo dos 8

Existem $9!/2 = 181400$ estados alcançáveis no jogo dos 8.
Facilmente conseguimos **manter tudo** em memória.
No jogo dos 15, por exemplo, existem $16!/2$ (mais de 10 biliões). Já não seria possível...
Duas heurísticas usadas são:

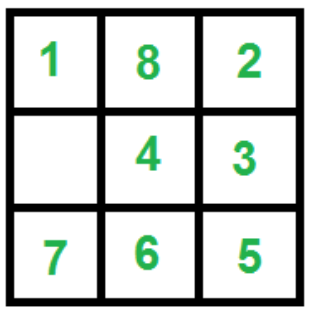
- h_1 = número de peças fora do sítio. A heurística é admissível porque qualquer peça que esteja fora do sítio irá necessitar de pelo menos uma acção para mudar de sítio.
- h_2 = soma das distâncias das peças à sua posição final. Neste caso, como não são permitidos movimentos diagonais, usamos a **Manhattan distance** (*city-block distance*). h_2 é admissível pois uma acção irá significar a aproximação ao objectivo em 1.

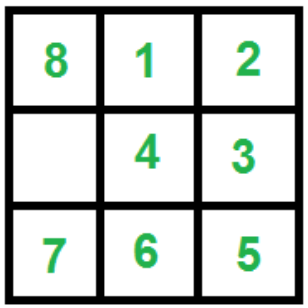
Jogo dos 8 - Verificação de Solução

Tendo em conta o estado final:



Temos que dois estados diferentes poderão ter ou não uma solução possível:





Jogo dos 8 - Verificação de Solução

Neste caso isto acontece porque um dos estados tem um número de inversões par (10) e outro tem um número de inversões ímpar (11).
Uma vez que no jogo dos 8 temos dois espaços de resultados que não se interceptam: estados cujo número de inversões é par e estados cujo número de inversões é ímpar.
Assim sendo, apenas conseguimos navegar entre estados que pertençam ao mesmo "universo" de possíveis estados.

Jogo dos 8 - Verificação de Solução

Um par de peças forma uma inversão se os seus valores estão em ordem inversa relativamente ao estado final.
No jogo dos 8, a paridade das inversões é mantida após qualquer movimento legal: caso no estado inicial seja par, continuará a ser par; se for ímpar, será sempre ímpar.
Apenas podemos fazer 2 tipos de movimentos: ou movemos uma peça ao longo das colunas ou ao longo das linhas.
Caso movamos ao longo das linhas, o número de inversões não é alterado (apenas o espaço em branco muda de posição).

```
 1   2   3    Row Move     1   2   3
 4   _   5   ---------->    _   4   5
 8   6   7                 8   6   7
Inversion count remains 2 after the move

 1   2   3    Row Move     1   2   3
 4   _   5   ---------->    4   5   _
 8   6   7                 8   6   7
Inversion count remains 2 after the move
```

Jogo dos 8 - Verificação de Solução

Caso movamos a peça ao longo das colunas, podem acontecer três cenários:

1. Aumentar as inversões em 2.

```
  1   2   3   Column Move     1   _   3
  4   _   5   ------------>    4   2   5
  8   6   7                    8   6   7
Inversion count increases by 2 (changes from 2 to 4)
```

2. Diminuir as inversões em 2.

```
  1   3   4    Column Move     1   3   4
  5   _   6   ------------>    5   2   6
  7   2   8                    7   _   8
Inversion count decreases by 2 (changes from 5  to 3)
```

3. Mantemos o número de inversões

```
    1   2   3    Column Move     1   2   3
    4   _   5   ------------>    4   6   5
    7   6   8                    7   _   8
  Inversion count remains 1 after the move
```

Jogo dos 8 - Verificação de Solução

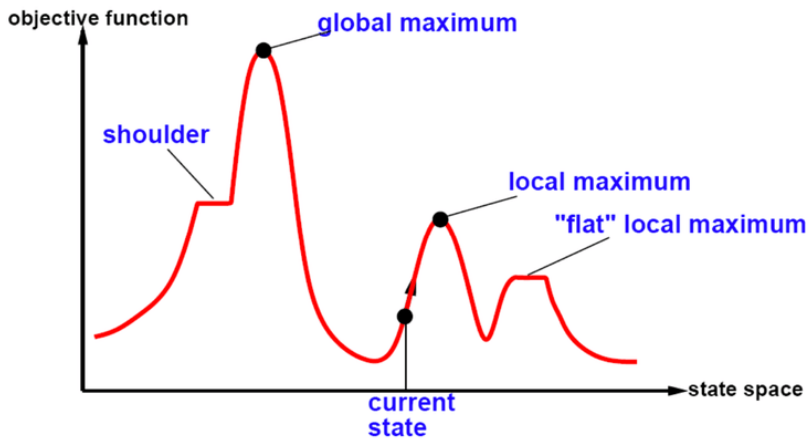
Uma vez que ou o número de inversões se mantém **igual**, ou altera num **valor par**, é impossível que a paridade de inversões de um estado se altere com qualquer operação legal.
Assim sendo, para verificar se podemos chegar de um determinado estado inicial a um estado final, apenas precisamos de verificar a paridade do número de inversões de cada um desses estados. Caso a paridade seja **igual**, então **existe solução**; Caso a paridade seja **diferente**, **não existe solução**.

Pesquisa Local

Por vezes, não estamos necessariamente interessados em descobrir **um caminho** para uma solução, mas apenas em descobrir **uma solução**.
Os algoritmos de **pesquisa local** procuram o espaço de estados a partir de um determinado estado inicial, atingindo estados vizinhos sem registar os caminhos obtidos ou o conjunto de estados visitados.
Isto significa que não são **sistemáticos** (podem nunca explorar uma porção do espaço de estados). No entanto, usam **pouca memória** e conseguem encontrar **soluções razoáveis** em espaços muito grandes (até infinitos) onde algoritmos sistemáticos não seriam uma opção viável.

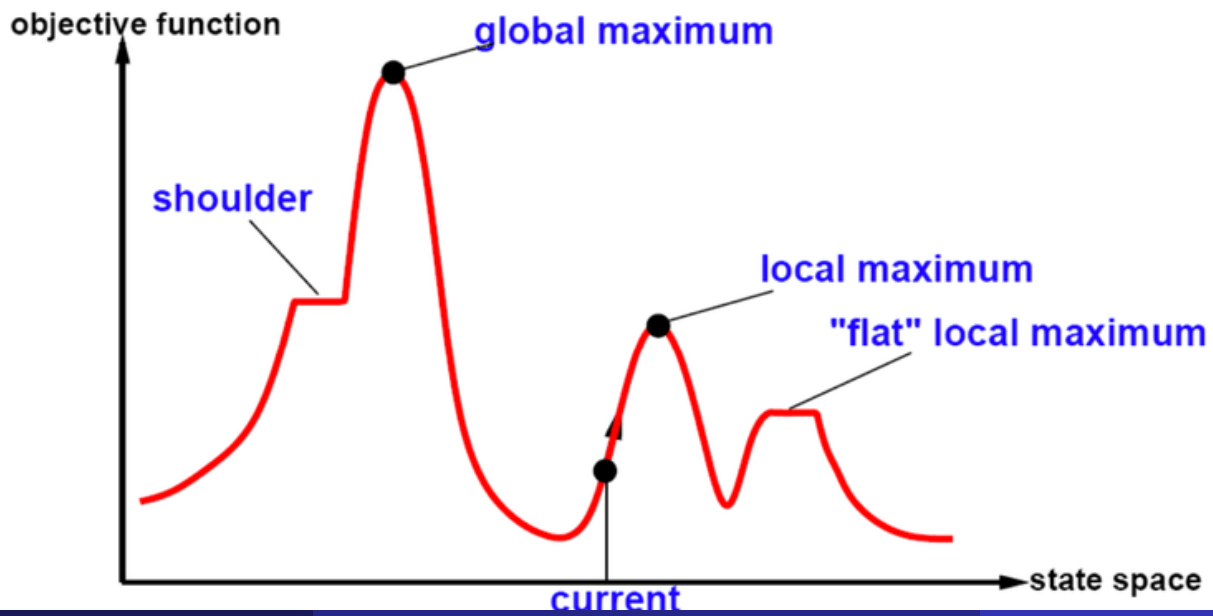
Pesquisa Local

Algoritmos de pesquisa local podem resolver **problemas de optimização**, onde queremos encontrar o melhor estado de acordo com uma **função objectivo**.
Existem alguns pontos fundamentais a ser explorados no *state-space landscape*. Cada ponto representa um **estado** e tem uma **ordenada** (eixo dos yy) correspondente ao **valor** da função objectivo.
Caso o objectivo seja encontrar o **maior pico** (máximo global) chamamos a este processo *Hill Climbing*. Caso queiramos descobrir o **maior vale** (mínimo global) chamamos a este processo *Gradient Descent*.



Hill Climbing

O algoritmo **Hill Climbing** mantém o registo de um estado actual e em cada iteração move-se para o estado vizinho com **maior valor**. Ele termina quando encontra um pico: nenhum vizinho tem maior valor que o estado actual.
O algoritmo não procura informação para além dos **vizinhos imediatos** do estado actual.
Uma forma de utilizar o algoritmo Hill Climbing é negando a **função heurísitca** e usar essa negação como **função objectivo**.



Hill Climbing

É útil em problemas com uma **explosão combinatória** de estados muito grande (não permitem uma pesquisa exaustiva no espaço de estados).
Por vezes é chamado de *greedy local search*, uma vez que apenas olha para os vizinhos imediatos sem pensar mais à frente. Porém, é possível rapidamente chegar a uma **solução**, porque, por norma, é fácil melhorar um estado com mau *score*.
No entanto, o algoritmo pode ficar **preso** por várias razões:

- **Máximos locais**: Caracterizam-se por um pico **maior que todos os vizinhos** mas **menor que o maior pico** presente na função.
- **Cumes**: São uma sequência de **máximos locais** que tornam difícil a navegação por parte de algoritmos *greedy*.
- **Plateaus**: É um espaço **nivelado** (nem sobe, nem desce). Pode representar um máximo local, a partir do qual **não existem** possibilidades de subir, ou um *shoulder* a partir do qual é possível melhorar.

Hill Climbing

Para o problema das 8 rainhas, o algoritmo fica preso 86% das vezes e apenas resolve 14% das instâncias.
No entanto, é bastante rápido a dar uma solução: 4 passos em média quando sucede e 3 quando falha.
O espaço de problemas tem cerca de 17 milhões de estados.
Algumas formas de contornar limitações do Hill Climbing:

- **Permitir a exploração de plateaus** (isto é, permitir movimentações "laterais") para perceber se um determinado *plateau* é um *shoulder*. No entanto, caso não seja, corremos o risco de ficar presos em movimentos laterais para sempre. Torna-se importante **limitar** o número de acções laterais. No problema das 8 rainhas, passamos de um sucesso de 14% para 94%. No entanto, passamos de 21 passos quando sucede e 64 quando falha.

Hill Climbing

Algumas formas de contornar limitações do Hill Climbing:

- **Variação estocástica**: escolhe **aleatoriamente** o próximo estado dentro daqueles que **melhorem** o estado actual. Pode demorar mais tempo a convergir. No entanto, possibilita a descoberta de melhores soluções. *First-choice Hill Climbing* escolhe o primeiro sucessor gerado que melhora o estado actual.
- **Random-Restart Hill Climbing**: conduz uma série de pesquisas Hill Climbing a partir de estados iniciais **aleatoriamente** gerados até chegar a um estado objectivo. O sucesso da aplicação desta estratégia depende muito da forma da *state-space landscape*: se existem poucos máximos locais e/ou *plateaus*, *Random-Restart Hill Climbing* encontrará uma solução rapidamente. No entanto, muitos problemas reais têm uma *landscape* muito errática.

Pesquisa Local - Hill Climbing

Algorithm 5 Hill Climbing(*problem*)

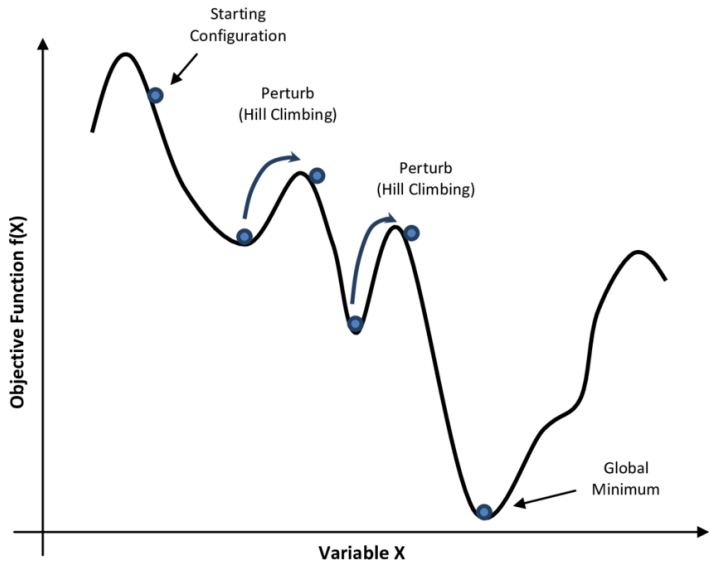
current ← *problem*.INITIAL
while *true* **do**
    neighbor ← highest-value successor of *current*
    if VALUE(*neighbor*) (VALUE(*current*) **then**
        return *current*
    end if
    current ← *neighbor*
end while
return *failure*

Simulated annealing

Um algoritmo de Hill Climbing que nunca se irá movimentar para estados com valor mais baixo, está sempre sujeito a ficar preso em **máximos locais**.
Por outro lado, uma *random walk* que anda de estado em estado sem preocupação com o seu valor, irá eventualmente aterrar num **máximo global**, mesmo que de forma extremamente ineficiente.
Assim sendo, parece razoável tentarmos combinar ambas as abordagens, de forma a conseguirmos uma abordagem completa e eficiente.
Com **Simulated Annealing** podemos fazer isso. Neste algoritmo, passamos do ponto de vista de *Hill Climbing* para *Gradient Descent* (minizamos o custo).

Simulated annealing

Imaginemos uma bola a deslizar por uma descida com várias lombas.
Se deixarmos a bola rolar, ela irá parar num **mínimo local**.
Se **abanarmos** a superfície, podemos **tirar** a bola do mínimo local (possivelmente para outro mínimo).
O truque é agitar a superfície de forma a que a bola saia do mínimo local, mas não com demasiada força de forma a que a mesma nunca atinja o **mínimo global**.
O que pretendemos é começar a agitar **com força** e depois **reduzir** a intensidade.



Simulated annealing

O algoritmo é muito parecido ao *Hill Climbing*.
Em vez de escolhermos o melhor estado vizinho, escolhemos um estado vizinho **aleatório**.
Se o novo estado for melhor que o actual, então **aceitamos a mudança**.
Caso contrário, aceitamos a mudança com uma **probabilidade** (menor que 1).

Simulated annealing

A probabilidade **piora exponencialmente** com a diferença de qualidade entre os dois estados (ΔE).
A probabilidade também **diminui** com a temperatura T: mudanças que **piorem** o estado actual são aceites com **maior probabilidade** no início (quando T é maior) e à medida que T fica mais **pequeno**, a probabilidade de aceitarmos estados piores também **decresce**.
Se T descer para 0 **lentamente**, o algoritmo irá encontrar com probabilidade próxima de 1 ($e^{\Delta E/T}$) o mínimo global (distribuição de Boltzmann).

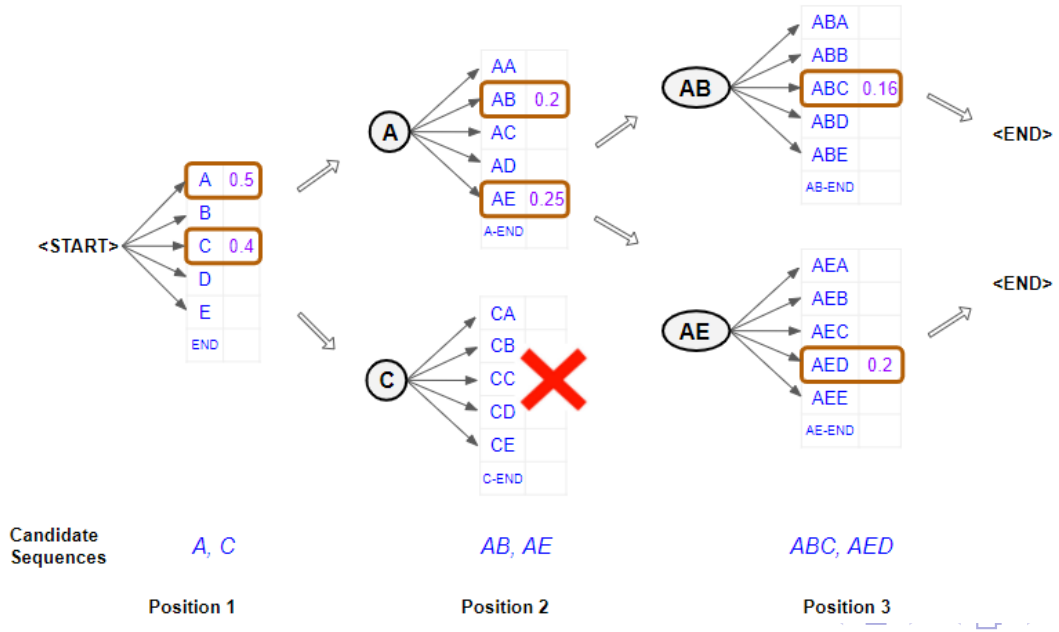
Pesquisa Local - Simulated Annealing

Algorithm 6 Simulated Annealing(*problem*, *schedule*)

current $\leftarrow$ *problem*.INITIAL
for $t = 1$ **to** ∞ **do**
  $T \leftarrow schedule(t)$
  if $T = 0$ **then**
    return *current*
  end if
  next $\leftarrow$ random successor of current
  $\Delta E \leftarrow$ VALUE(*current*) $-$ VALUE(*next*)
  if $\Delta E > 0$ **then**
    current $\leftarrow$ *next*
  else
    current $\leftarrow$ *next* with probability $e^{\Delta E / T}$
  end if
end for
return *failure*

Local Beam Search

Na maioria dos casos, não precisamos de manter apenas **um nó** em memória. O algoritmo local *beam search* mantém k estados em memória, em vez de apenas um.
Começa com k estados gerados **aleatoriamente** e em cada passo, todos os sucessores de cada um dos k estados são gerados.
Se algum deles for a solução, então **retornamos**. Caso contrário, selecionamos os melhores k estados dessa **lista completa** (estados iniciais + sucessores).



Local Beam Search

Difere de uma paralelização do *Random-Restart Hill Climbing* na medida em que cada um dos nós não irá ser expandido **em paralelo** e de forma **independente**.
Pelo contrário, informação útil irá ser passada e partilhada **por toda a execução** do algoritmo. O algoritmo assim foca-se mais na parte **mais promissora** do espaço de estados.
Pode, no entanto, sofrer de alguma **falta de diversidade** dentro dos k estados (podemos aprofundar muito na mesma região do espaço de estados).
Uma variante **estocástica** do algoritmo ajuda na resolução desta limitação: em vez de escolhermos os melhores k sucessores, escolhemos k sucessores aleatoriamente, cuja probabilidade de escolha está relacionada com a qualidade do mesmo, aumentando assim a diversidade.