Alberto Barbosa

Universidade da Maia *alberto.barbosa@umaia.pt*

Sistemas Inteligentes - Algoritmos de Pesquisa

February 17, 2025

# Resolu¸c˜ao 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 acc¸˜oes para l´a chegar.



Formulac¸˜ao do problema: O agente formula uma descri¸c˜ao dos estados e ac¸c˜oes necess´arias para atingir um objectivo.

Pesquisa: Antes de agir, o agente simula sequˆencias de acc¸˜oes no seu modelo at´e atingir o objectivo. A esta sequˆencia chamamos soluc¸˜ao. No final, o agente pode chegar a uma poss´ıvel solu¸c˜ao ou concluir que n˜ao existe uma soluc¸˜ao poss´ıvel.

Execuc¸˜ao: O agente agora pode executar as ac¸c˜oes da soluc¸˜ao.



# Problemas de pesquisa

Podemos definir um processo de pesquisa da seguinte forma:

O espac¸o de estados: conjunto de estados poss´ıveis.



Um estado inicial.

Um ou mais estados finais.

Uma func¸˜ao de actua¸c˜ao, onde *Action*(*s*) retorna um conjunto finito de acc¸˜oes que podem ser executadas no estado *s*.

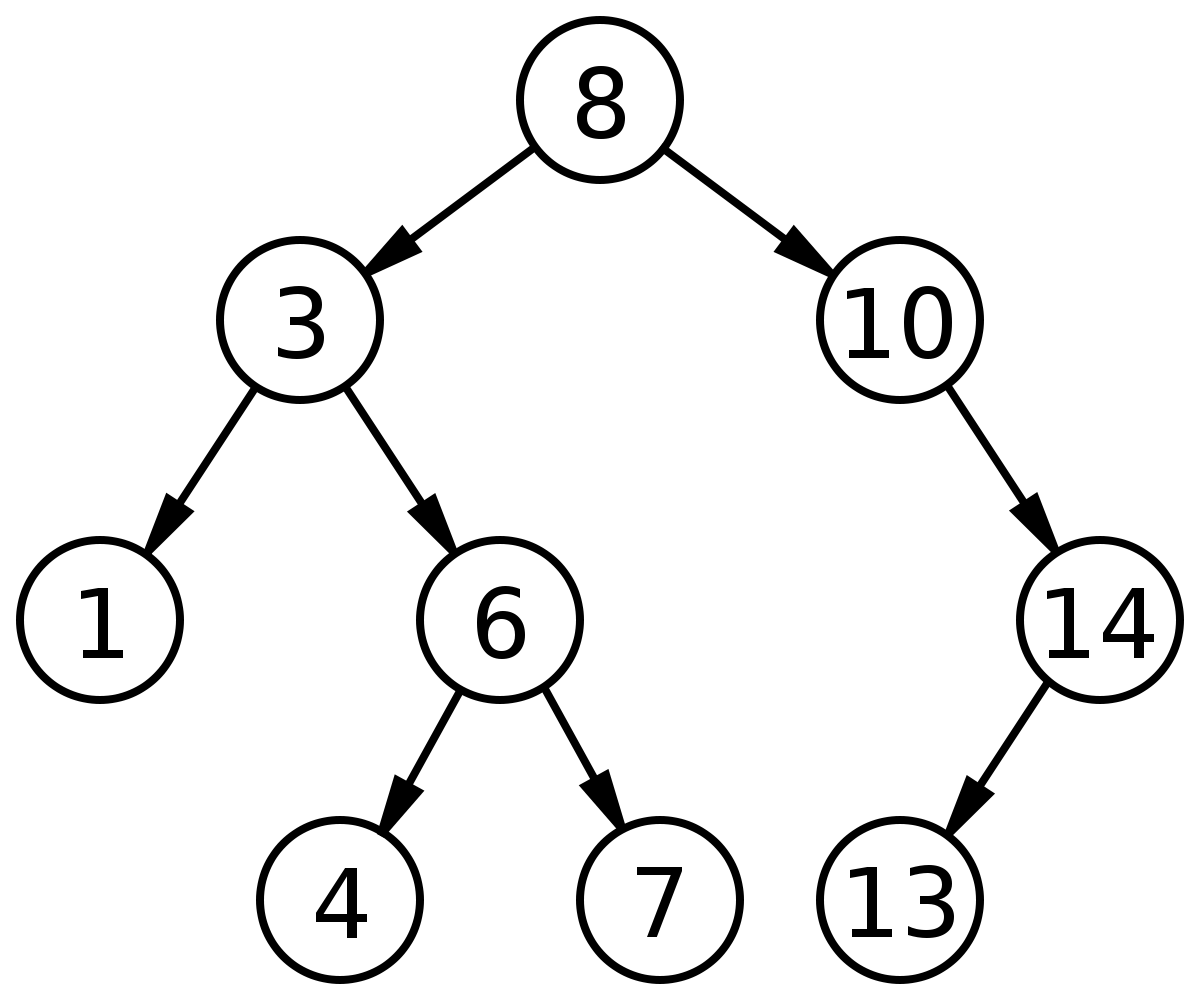
Um modelo de transic¸˜ao que descreve o que cada acc¸˜ao faz. *Result*(*s,a*) retorna o estado que resulta de executar a ac¸c˜ao *a* no estado *s*.

Uma func¸˜ao de custo *c*(*s,a,s*′) que nos d´a o custo num´erico de aplicar a acc¸˜ao *a* no estado *s* para chegar ao estado *s*′.



# Espa¸co de Estados

Um espac¸o de estados pode ser representado como um grafo, onde cada v´ertice ´e um estado e cada aresta entre v´ertices representa uma acc¸˜ao. Uma ´arvore de pesquisa pode ser representada como um grafo onde n˜ao existem ciclos.

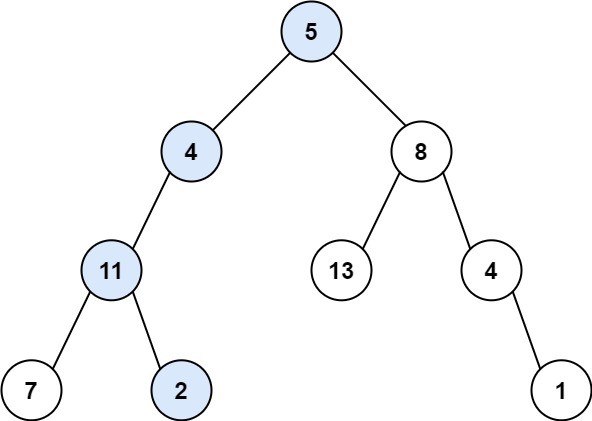


# Solu¸co˜es

A uma sequˆencia de acc¸˜oes chamamos caminho e uma soluc¸˜ao ´e um caminho desde o estado inicial at´e ao estado final.

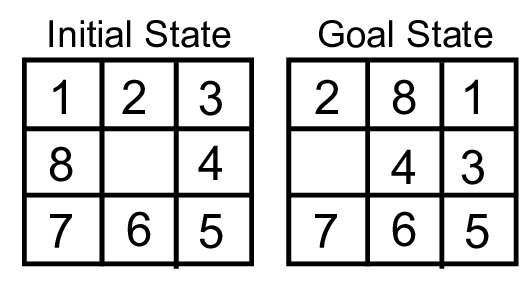
Podemos assumir que o custo total de um caminho ´e a soma dos custos individuais de cada acc¸˜ao.

Uma soluc¸˜ao ´optima tem o menor custo entre todas as solu¸c˜oes.



# Jogo dos 8

Um puzzle onde dado um estado inicial queremos deslizar as pe¸cas para o quadrado em branco at´e chegarmos ao estado final.



# Jogo dos 8

O jogo dos 8 pode ser formalizado desta forma:

Estados: Cada estado especifica a posi¸c˜ao de cada um dos tiles.



Estado inicial: Qualquer configurac¸˜ao v´alida pode ser um estado inicial.

Acc¸˜oes: A forma mais simples ser´a do ponto de vista do espac¸o em branco: *Esquerda*, *Direita*, *Cima* e *Baixo*.

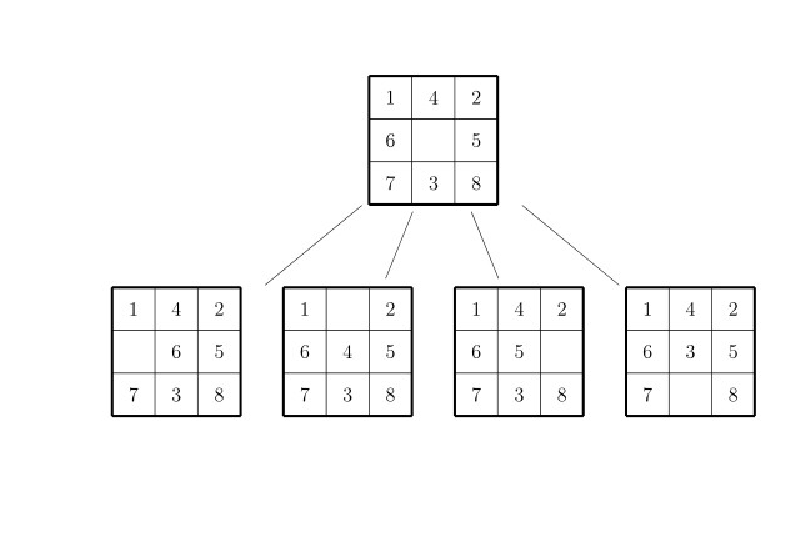
Modelo de Transic¸˜ao: Mapeia um estado e acc¸˜ao a um estado resultante.

Objectivo: Qualquer configurac¸˜ao v´alida pode ser um estado final.

Custo: Cada acc¸˜ao tem custo 1.

# Algoritmos de pesquisa

Um algoritmo de pesquisa aceita um problema de pesquisa como *input* e retorna uma soluc¸˜ao ou indica¸c˜ao de falha como *output*.

Vamos considerar algoritmos que modelam o espa¸co de estados como uma ´arvore de pesquisa, onde queremos encontrar um caminho entre o estado inicial e o estado final.

Cada n´o na ´arvore corresponde a um estado e cada aresta corresponde a uma acc¸˜ao.

A raiz da ´arvore ´e o estado inicial.

# Algoritmos de pesquisa - Implementac¸˜ao

Algumas estruturas de dados relevantes para implementa¸c˜ao de cada n´o:

*node*.STATE: estado representado pelo n´o. *node*.PARENT: n´o na ´arvore que gerou este n´o.



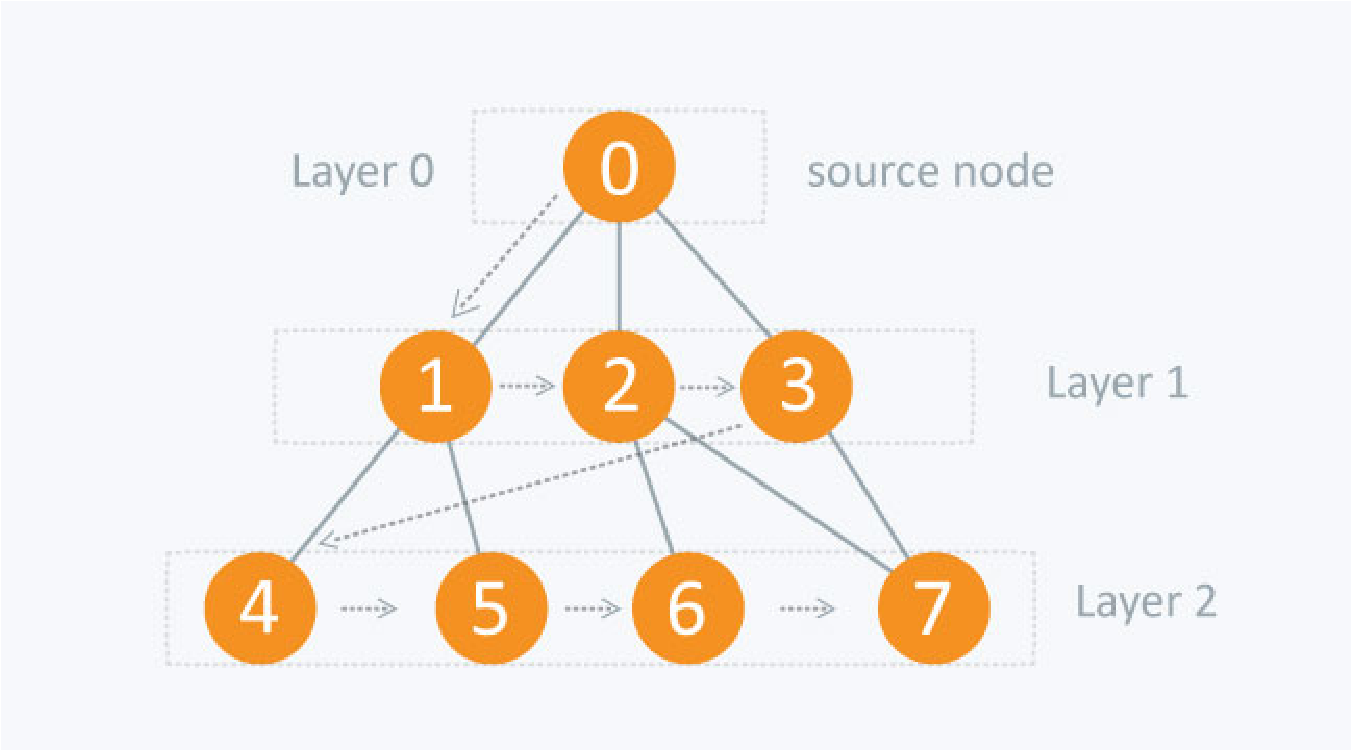
*node*.ACTION: acc¸˜ao que levou `a gera¸c˜ao deste n´o. *node*.COST: custo desde o estado inicial at´e este n´o. *node*.CHILDREN: n´os gerados por este n´o.

# Algoritmos de pesquisa - Implementac¸˜ao

Estruturas de dados relevantes para representar n´os a expandir (fronteira):

*empty*(*frontier*): verdade se n˜ao existirem n´os a expandir. *pop*(*frontier*): remove o pr´oximo n´o a expandir da fronteira. *top*(*frontier*): retorna (mas n˜ao remove) o pr´oximo n´o a expandir. *add*(*node*,*frontier*): insere o n´o na estrutura na posi¸c˜ao correcta.



Um algoritmo cego n˜ao tem qualquer pista sobre qu˜ao perto est´a de uma poss´ıvel solu¸c˜ao.

Um destes algoritmos ´e a pesquisa em largura ou BFS (breadth first search).

Na pesquisa em largura, a ra´ız ´e expandida em primeiro lugar, depois todos os seus sucessores s˜ao expandidos, depois todos os sucessores destes s˜ao expandidos, e assim sucessivamente.

A BFS encontra sempre a soluc¸˜ao ´optima uma vez que quando geramos os n´os com profundidade *d*, todos os n´os com profundidade *d* − 1 j´a foram gerados.

No entanto, apenas para problemas onde todas as acc¸˜oes tˆem o mesmo custo e encontra sempre uma soluc¸˜ao, caso exista (´e completa). Complexidade temporal e espacial ´e O(*bd*), sendo *b* o nu´mero de sucessores que cada n´o pode gerar e *d* a profundidade da ´arvore.

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*

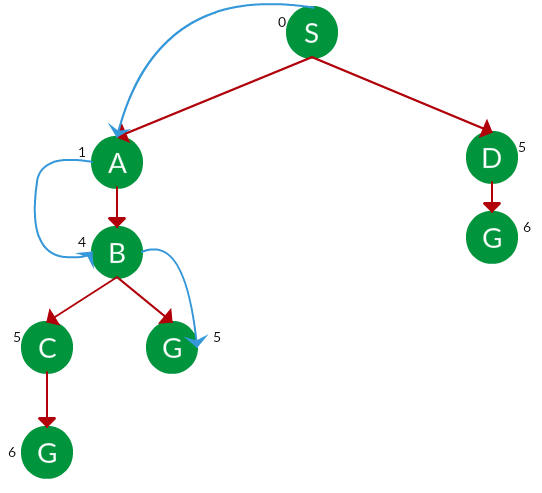
end if if *s* is not in *reached* then add *s* to *reached* add *child* to *frontier*

end if

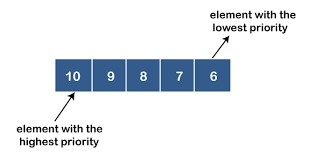
end for

end while return *failure*

Caso tenhamos custos diferentes associados a cada acc¸˜ao, uma forma de solucionar o problema ser´a utilizar uma varia¸c˜ao do BFS, onde, ao inv´es de expandirmos os n´os pela sua profundidade (isto ´e: expandir primeiro os com profundidade 1, depois com profundidade 2, etc), expandimos pelo custo (primeiro com custo 1, depois com custo 2, etc).



O algoritmo ´e ent˜ao muito semelhante ao BFS, no entanto o que dita o pr´oximo n´o a ser expandido ´e o custo associado ao mesmo. Na pr´atica, teremos de mudar a estrutura de dados que serve como implementa¸c˜ao da fronteira. Deixar´a de ser FIFO para passar a ser uma Priority Queue (Heap).



Caso todas as acc¸˜oes tenham o mesmo custo, o algoritmo tem a mesma complexidade espacial e temporal que o BFS.

Caso contr´ario tem pior complexidade temporal e espacial O(*b*1+*C*∗*/ϵ*). O algoritmo ´e completo e ´optimo, porque a primeira solu¸c˜ao encontrada ter´a sempre o menor custo, uma vez que todos os caminhos s˜ao 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

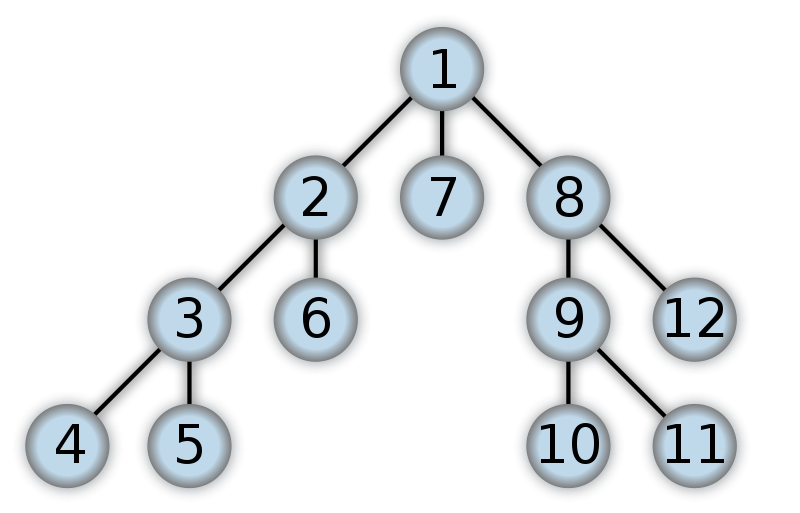
for all *child* in EXPAND(*problem*,*node*) do

add *child* to *frontier*

end for

end while return *failure*

O algoritmo DFS (Depth-first search), ir´a explorar o algoritmo mais profundo primeiro. Isto ´e, aquele, dos que est˜ao na fronteira, que se encontra a maior profundidade na ´arvore de pesquisa.



A implementa¸c˜ao ´e muito semelhante ao BFS. No entanto, ´e necess´ario mudar a estrutura de dados que serve de apoio `a fronteira. Deixar´a de ser FIFO e passar´a a ser LIFO (uma pilha por exemplo).

O DFS n˜ao ´e ´optimo. A soluc¸˜ao retornada n˜ao ´e necessariamente a que custa menos.

Em espa¸cos de estados finitos, ´e eficiente e completo.

Em espa¸cos de estados c´ıclicos, pode ficar preso num ciclo, pelo que se torna fundamental a verificac¸˜ao de estados repetidos aquando da implementa¸c˜ao do DFS.

Em espa¸cos de estados infinitos, pode nunca encontrar uma solu¸c˜ao, deixando de ser um algoritmo completo.

No entanto, consome menos mem´oria que o BFS (caso n˜ao seja necess´ario fazer verifica¸c˜ao de repetidos) e tem fronteiras mais pequenas.

Tem complexidade temporal *O*(*bm*) e complexidade espacial *O*(*bm*). Devido `a forma como utiliza pouca mem´oria, acaba por estar na base de muitos algoritmos cl´assicos de diferentes ´areas de IA, como satisfac¸˜ao de restric¸˜oes e programac¸˜ao l´ogica.

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 implementa¸c˜ao, de modo a que limitemos a profundidade m´axima que a ´arvore pode atingir. Isto significa que iremos determinar uma profundidade limite *l* e que iremos tratar todos os n´os na profundidade *l* como se n˜ao tivessem descendentes.

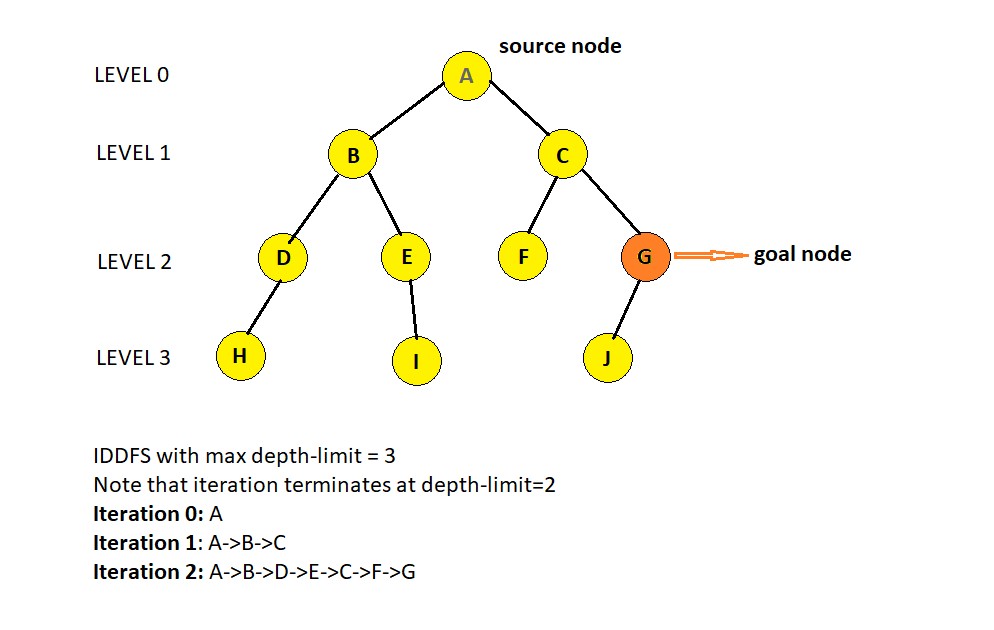
A complexidade temporal ´e O(*bl*) e espacial ´e O(*bl*).

O mais dif´ıcil neste caso ´e acertar na profundidade m´axima. Isto ´e, como ´e que sabemos qual ´e 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 m´aximo de profundidade a cada passo. Isto ´e, 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 at´e que encontremos a m´ınima profundidade para a qual temos uma solu¸c˜ao.



# Algoritmos cegos - IDFS

A complexidade espacial ´e reduzida: O(*bd*) quando existe uma solu¸c˜ao ou O(*bm*) quando ´e um estado de espa¸cos finito sem soluc¸˜ao.

IDFS ´e ´optimo para problemas onde todas as acc¸˜oes tenham o mesmo custo e ´e completo (com verifica¸c˜ao de repetidos, caso existam ciclos). Complexidade temporal ´e O(*bd*) quando h´a soluc¸˜ao e O(*bm*) quando n˜ao h´a.

# Algoritmos informados - Heur´ısticas

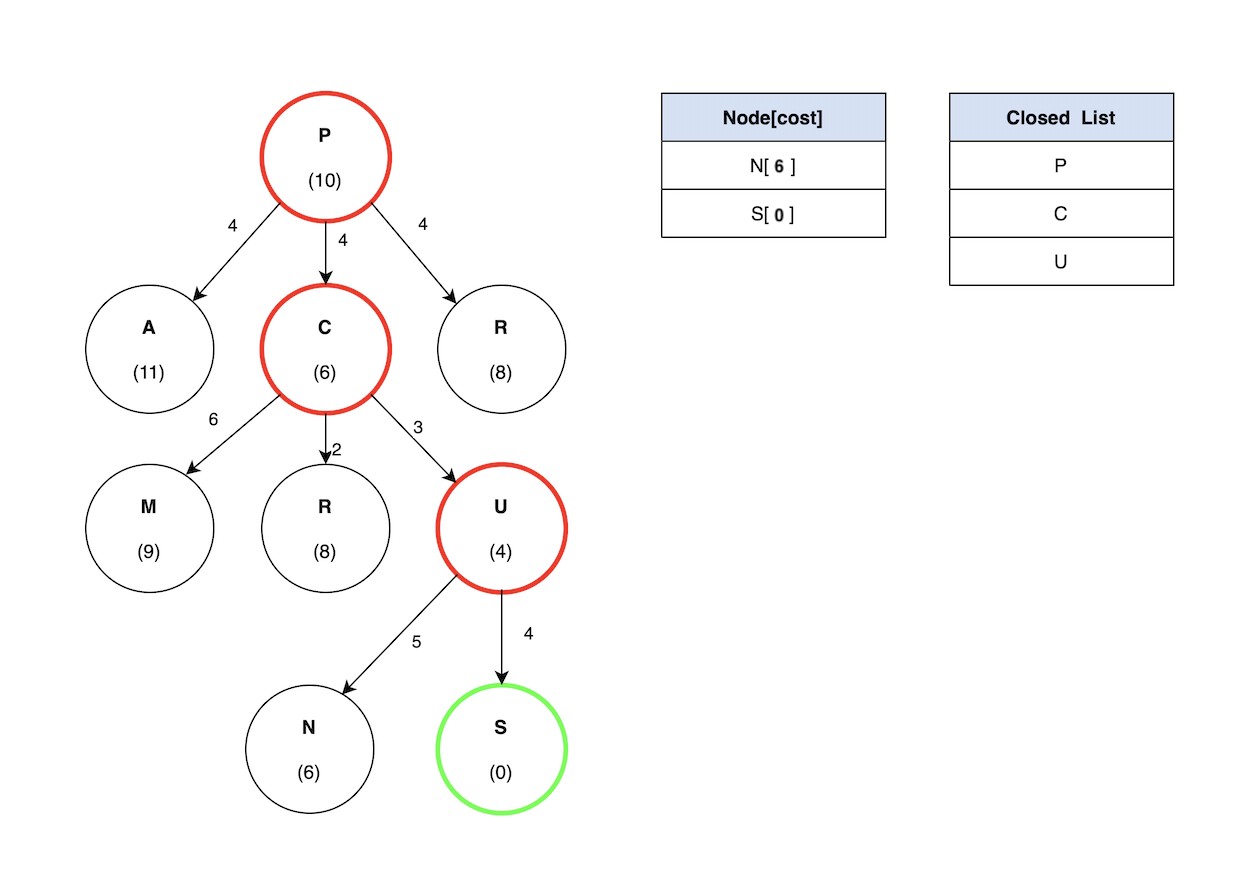
Os algoritmos informados tˆem por base uma fun¸c˜ao que tenta prever qu˜ao longe estamos de uma poss´ıvel solu¸c˜ao.

A essa func¸˜ao chamamos func¸˜ao heur´ıstica, denotada *h*(*n*).

No jogo dos 8, uma func¸˜ao heur´ıstica pode ser o nu´mero de pec¸as fora do s´ıtio, que nos indica par um dado estado, quantas pec¸as est˜ao fora da sua posi¸c˜ao final. Quanto maior o nu´mero de pe¸cas fora do s´ıtio, mais longe estamos de uma solu¸c˜ao.

# Algoritmos informados - Greedy Best First

Um algoritmo Greedy best-first ´e um algoritmo que vai expandir sempre o n´o com menor valor de *h*(*n*) dentro dos n´os na fronteira.



# Algoritmos informados - Greedy Best First

Esta estrat´egia ´e completa em espa¸cos finitos, na medida em que retorna sempre soluc¸˜ao, no entanto n˜ao ´e ´optima (depende muito da qualidade da heur´ıstica).

A complexidade espacial e temporal ´e *O*(|*V*|), podendo ser reduzida at´e para *O*(*bm*) com uma boa escolha de func¸˜ao heur´ıstica.

# 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*

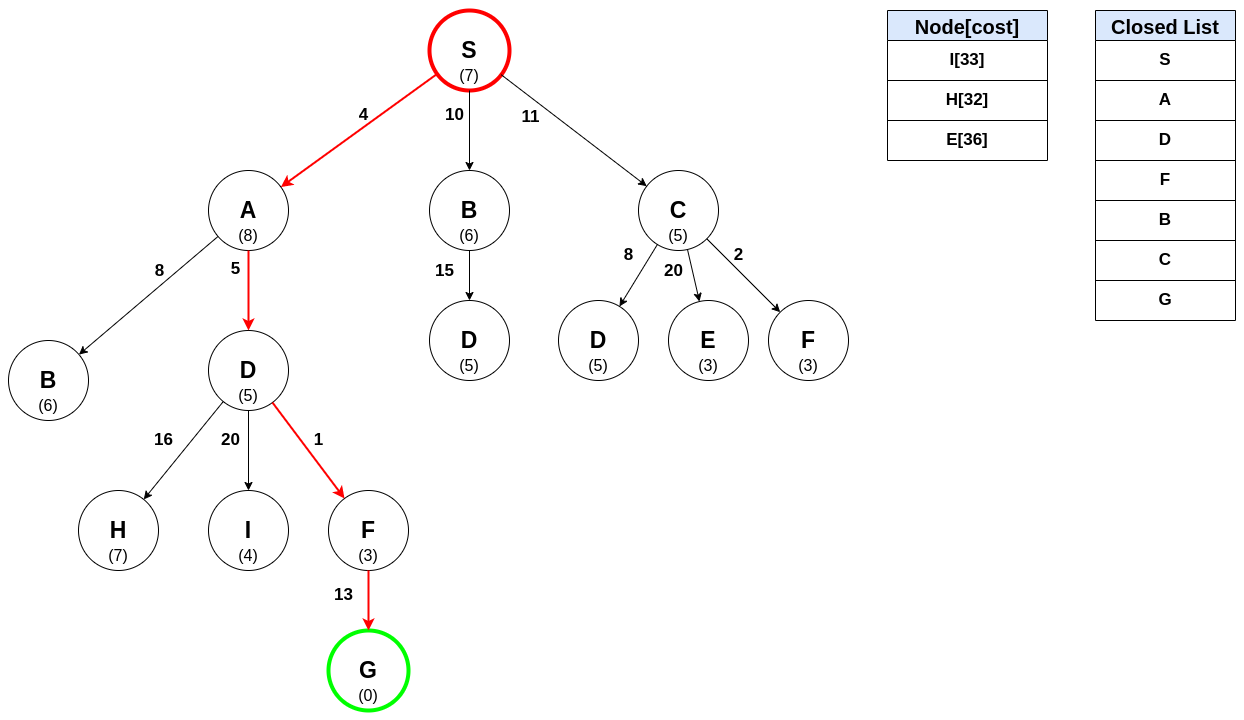
O algoritmo A\* ´e um algoritmo best-first que utiliza a seguinte fun¸c˜ao de avalia¸c˜ao:

*f* (*n*) = *g*(*n*) + *h*(*n*) onde *g*(*n*) ´e o custo desde o n´o inicial at´e ao estado *n* e *h*(*n*) ´e o custo estimado desde *n* at´e ao estado final.

O algoritmo A\* ´e completo. No entanto, a quest˜ao da optimalidade est´a dependente de certas caracter´ısticas da func¸˜ao heur´ıstica.

Para tal acontecer, a heur´ıstica deve ser admiss´ıvel, isto ´e, nunca deve superestimar o custo para chegar a uma soluc¸˜ao. Deve ser optimista.

Caso se utilize uma heur´ıstica admiss´ıvel, A\* ´e ´optimo.



# Algoritmos informados - A\* (Demonstra¸c˜ao)

Se existe um caminho com custo *C*∗ mas o algoritmo retorna um caminho com custo *C > C*∗, ent˜ao existe um n´o *n* que est´a no caminho ´optimo, mas n˜ao ´e expandido.

Ent˜ao usando *g*∗(*n*) como o custo ´optimo desde o n´o inicial at´e *n* e *h*∗(*n*) como o custo ´optimo desde *n* at´e ao final, temos:

1. *f* (*n*) *> C*∗
2. *f* (*n*) = *g*(*n*) + *h*(*n*)
3. *f* (*n*) = *g*∗(*n*) + *h*(*n*) (porque *n* est´a no caminho ´optimo)
4. *f* (*n*) ≤ *g*∗(*n*) + *h*∗(*n*) (porque *h* ´e admiss´ıvel)
5. *f* (*n*) ≤ *C*∗ (por defini¸c˜ao *C*∗ = *g*∗(*n*) + *h*∗(*n*)

A primeira linha e a u´ltima entram em contradic¸˜ao, pelo que ´e imposs´ıvel que A\* n˜ao retorne o caminho ´optimo nestas condic¸˜oes.

Uma propriedade muito importante ´e tamb´em a consistˆencia.

Uma heur´ıstica *h*(*n*) ´e consistente se para cada n´o *n* e cada sucessor de *n*, *n*′, gerado por uma ac¸c˜ao *a*, temos:

*h*(*n*) ≤ *c*(*n,a,n*′) + *h*(*n*′) (desigualdade triangular) (1)

Todas as heur´ısticas consistentes s˜ao admiss´ıveis, mas uma heur´ıstica admiss´ıvel pode n˜ao ser consistente.

Se a heur´ıstica for consistente, a primeira vez que cada n´o ´e expandido, j´a ser´a quando o mesmo faz parte de um caminho ´optimo, fazendo com que n˜ao seja necess´ario expandir v´arias vezes o mesmo n´o.

O mesmo n˜ao acontece com uma heur´ıstica inconsistente, podendo ter maiores custos de tempo e mem´oria.

Com uma heur´ıstica n˜ao admiss´ıvel, A\* pode ou n˜ao ser ´optimo. No entanto, pode ser ´optimo se *h*(*n*) for admiss´ıvel para todos os inconsistente ou se o problema tiver uma soluc¸˜ao ´optima com custo *C*∗ e a segunda melhor soluc¸˜ao tiver custo *C*2 e a superestimativa de *h*(*n*) nunca ´e superior a *C*2 − *C*∗.

A medida que estendemos um caminho, os custos de` *g* v˜ao aumentando, uma vez que todas as ac¸c˜oes tˆem um custo positivo. Isto significa que *g*(*n*) ´e uma fun¸c˜ao mon´otona.

No entanto, n˜ao ´e ´obvio que *f* (*n*) = *g*(*n*) + *h*(*n*) v´a 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*′). Ent˜ao o caminho aumentar´a monotonamente se e s´o se *h*(*n*) *<*= *c*(*n,a,n*′) + *h*(*n*′), ou seja, se *h*(*n*) for consistente.

Se *C*∗ ´e o custo do caminho ´optimo:

A\* expande todos os n´os que podem ser alcan¸cados a partir do n´o inicial num caminho onde cada n´o tem *f* (*n*) *< C*∗.



A\* pode expandir alguns n´os com *f* (*n*) = *C*∗, caso a diminuic¸˜ao em *h* corresponda ao aumento em *g*.

A\* n˜ao ir´a expandir nenhum n´o com *f* (*n*) *> C*∗.

Dizemos que A\*, com uma heur´ısitca consistente, ´e optimamente eficiente: qualquer algoritmo que estenda caminhos a partir do n´o inicial com a mesma heur´ıstica, ir´a expandir todos os n´os com *f* (*n*) *< C*∗ expandidos por A\*.

Nos n´os com *f* (*n*) = *C*∗, pode haver mais variabilidade (n˜ao relevante para a quest˜ao da eficiˆencia ´optima).

A\* ´e completo, ´optimo e optimamente eficiente. No entanto n˜ao ´e a resposta para todos os problemas de pesquisa. Isto porque para muitos problemas, o nu´mero de n´os expandidos pode ser exponencial no comprimento da solu¸c˜ao.

# Algumas heur´ısticas - Jogo dos 8

Existem 9!*/*2 = 181400 estados alcan¸c´aveis no jogo dos 8.

Facilmente conseguimos manter tudo em mem´oria.

No jogo dos 15, por exemplo, existem 16!*/*2 (mais de 10 bili˜oes). J´a n˜ao seria poss´ıvel...

Duas heur´ısticas usadas s˜ao:

*h*1 = nu´mero de pec¸as fora do s´ıtio. A heur´ıstica ´e admiss´ıvel porque qualquer pe¸ca que esteja fora do s´ıtio ir´a necessitar de pelo menos uma ac¸c˜ao para mudar de s´ıtio.

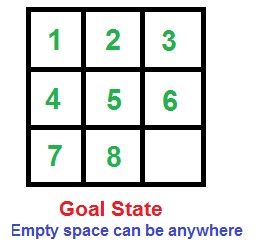


*h*2 = soma das distˆancias das pe¸cas `a sua posic¸˜ao final. Neste caso, como n˜ao s˜ao permitidos movimentos diagonais, usamos a

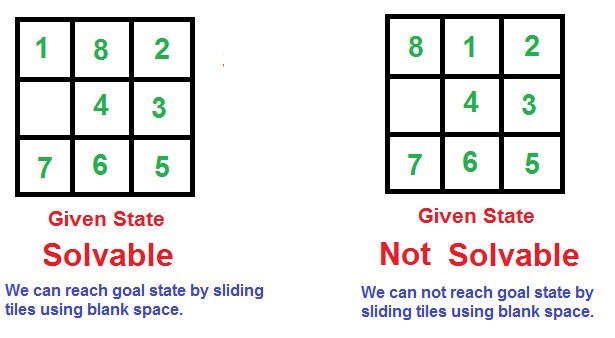


Manhattan distance (*city-block distance*). *h*2 ´e admiss´ıvel pois uma ac¸c˜ao ir´a significar a aproxima¸c˜ao ao objectivo em 1.

Tendo em conta o estado final:



Temos que dois estados diferentes poder˜ao ter ou n˜ao uma solu¸c˜ao poss´ıvel:



Neste caso isto acontece porque um dos estados tem um nu´mero de invers˜oes par (10) e outro tem um nu´mero de invers˜oes ´ımpar (11). Uma vez que no jogo dos 8 temos dois espa¸cos de resultados que n˜ao se interceptam: estados cujo nu´mero de invers˜oes ´e par e estados cujo nu´mero de invers˜oes ´e ´ımpar.

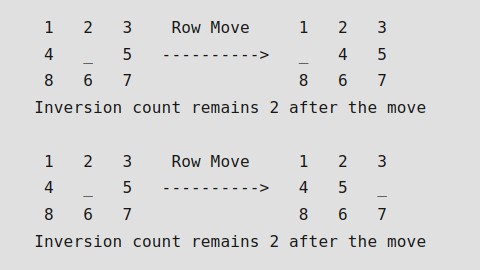
Assim sendo, apenas conseguimos navegar entre estados que perten¸cam ao mesmo ”universo” de poss´ıveis estados.

Um par de pe¸cas forma uma invers˜ao se os seus valores est˜ao em ordem inversa relativamente ao estado final.

No jogo dos 8, a paridade das invers˜oes ´e mantida ap´os qualquer movimento legal: caso no estado inicial seja par, continuar´a a ser par; se for ´ımpar, ser´a sempre ´ımpar.

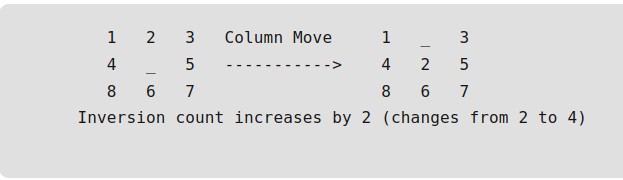
Apenas podemos fazer 2 tipos de movimentos: ou movemos uma pec¸a ao longo das colunas ou ao longo das linhas.

Caso movamos ao longo das linhas, o nu´mero de invers˜oes n˜ao ´e alterado (apenas o espa¸co em branco muda de posic¸˜ao).

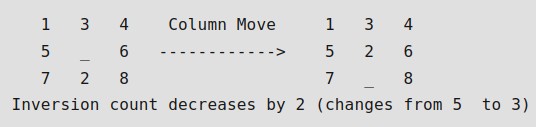


Caso movamos a pec¸a ao longo das colunas, podem acontecer trˆes cen´arios:

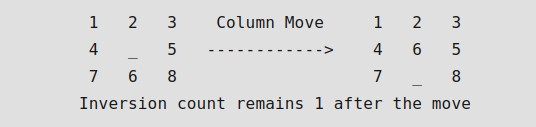
1. Aumentar as invers˜oes em 2.



1. Diminuir as invers˜oes em 2.



1. Mantemos o nu´mero de invers˜oes



Uma vez que ou o nu´mero de invers˜oes se mant´em igual, ou altera num valor par, ´e imposs´ıvel que a paridade de invers˜oes de um estado se altere com qualquer opera¸c˜ao legal.

Assim sendo, para verificar se podemos chegar de um determinado estado inicial a um estado final, apenas precisamos de verificar a paridade do nu´mero de invers˜oes de cada um desses estados. Caso a paridade seja igual, ent˜ao existe soluc¸˜ao; Caso a paridade seja diferente, n˜ao existe soluc¸˜ao.

# Pesquisa Local

Por vezes, n˜ao estamos necessariamente interessados em descobrir um caminho para uma soluc¸˜ao, mas apenas em descobrir uma soluc¸˜ao. Os algoritmos de pesquisa local procuram o espa¸co de estados a partir de um determinado estado inicial, atingindo estados vizinhos sem registar os caminhos obtidos ou o conjunto de estados visitados.

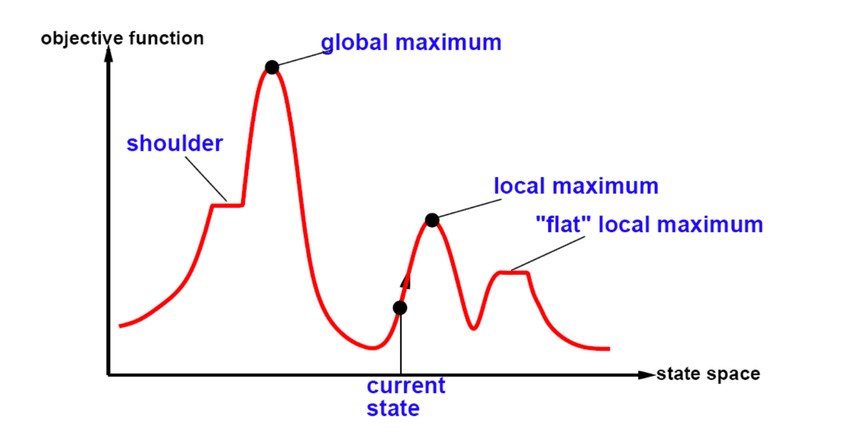
Isto significa que n˜ao s˜ao sistem´aticos (podem nunca explorar uma por¸c˜ao do espa¸co de estados). No entanto, usam pouca mem´oria e conseguem encontrar soluc¸˜oes razo´aveis em espa¸cos muito grandes (at´e infinitos) onde algoritmos sistem´aticos n˜ao seriam uma opc¸˜ao vi´avel.

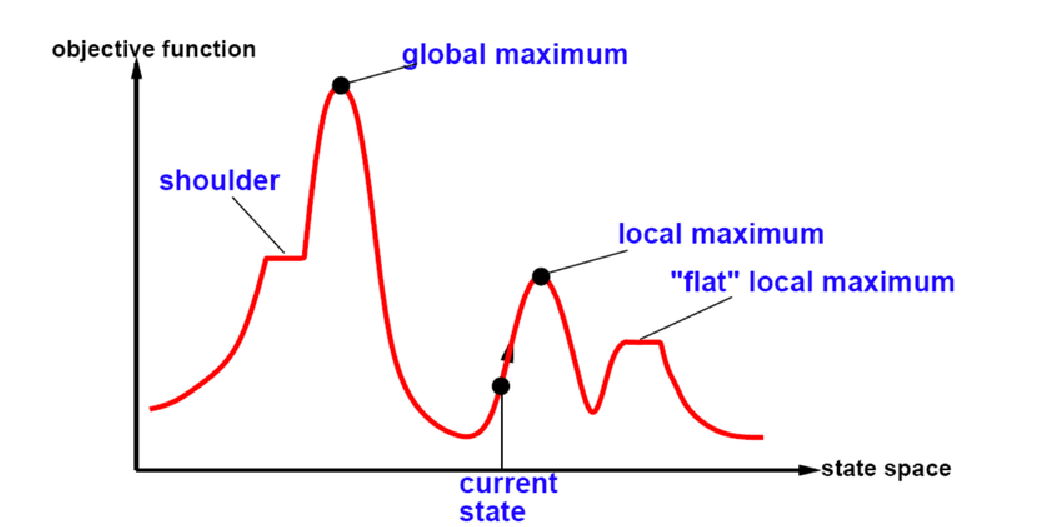
# Pesquisa Local

Algoritmos de pesquisa local podem resolver problemas de optimizac¸˜ao, onde queremos encontrar o melhor estado de acordo com uma func¸˜ao 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 fun¸c˜ao objectivo.

Caso o objectivo seja encontrar o maior pico (m´aximo global) chamamos a este processo *Hill Climbing*. Caso queiramos descobrir o maior vale (m´ınimo global) chamamos a este processo *Gradient Descent*.



O algoritmo Hill Climbing mant´em o registo de um estado actual e em cada itera¸c˜ao 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 n˜ao procura informa¸c˜ao para al´em dos vizinhos imediatos do estado actual.

Uma forma de utilizar o algoritmo Hill Climbing ´e negando a func¸˜ao heur´ısitca e usar essa nega¸c˜ao como func¸˜ao objectivo.

E u´til em problemas com uma´ explos˜ao combinat´oria de estados muito grande (n˜ao permitem uma pesquisa exaustiva no espac¸o de estados). Por vezes ´e chamado de *greedy local search*, uma vez que apenas olha para os vizinhos imediatos sem pensar mais `a frente. Por´em, ´e poss´ıvel rapidamente chegar a uma soluc¸˜ao, porque, por norma, ´e f´acil melhorar um estado com mau *score*.

No entanto, o algoritmo pode ficar preso por v´arias raz˜oes:

M´aximos locais: Caracterizam-se por um pico maior que todos os vizinhos mas menor que o maior pico presente na func¸˜ao.



Cumes: S˜ao uma sequˆencia de m´aximos locais que tornam dif´ıcil a navega¸c˜ao por parte de algoritmos *greedy*.

Plateaus: E um espac¸o´ nivelado (nem sobe, nem desce). Pode representar um m´aximo local, a partir do qual n˜ao existem possibilidades de subir, ou um *shoulder* a partir do qual ´e poss´ıvel melhorar.

Para o problema das 8 rainhas, o algoritmo fica preso 86% das vezes e apenas resolve 14% das instˆancias.

No entanto, ´e bastante r´apido a dar uma soluc¸˜ao: 4 passos em m´edia quando sucede e 3 quando falha.

O espa¸co de problemas tem cerca de 17 milh˜oes de estados. Algumas formas de contornar limitac¸˜oes do Hill Climbing:

Permitir a explorac¸˜ao de plateaus (isto ´e, permitir movimentac¸˜oes ”laterais”) para perceber se um determinado *plateau* ´e um *shoulder*. No entanto, caso n˜ao seja, corremos o risco de ficar presos em movimentos laterais para sempre. Torna-se importante limitar o nu´mero de acc¸˜oes 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.



Algumas formas de contornar limitac¸˜oes do Hill Climbing:

Variac¸˜ao estoc´astica: escolhe aleatoriamente o pr´oximo estado dentro daqueles que melhorem o estado actual. Pode demorar mais tempo a convergir. No entanto, possibilita a descoberta de melhores solu¸c˜oes. *First-choice Hill Climbing* escolhe o primeiro sucessor gerado que melhora o estado actual.



Random-Restart Hill Climbing: conduz uma s´erie de pesquisas Hill Climbing a partir de estados iniciais aleatoriamente gerados at´e chegar a um estado objectivo. O sucesso da aplica¸c˜ao desta estrat´egia depende muito da forma da *state-space landscape*: se existem poucos m´aximos locais e/ou *plateaus*, *Random-Restart Hill Climbing* encontrar´a uma soluc¸˜ao rapidamente. No entanto, muitos problemas reais tˆem uma *landscape* muito err´atica.



# 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*

Um algoritmo de Hill Climbing que nunca se ir´a movimentar para estados com valor mais baixo, est´a sempre sujeito a ficar preso em m´aximos locais.

Por outro lado, uma *random walk* que anda de estado em estado sem preocupa¸c˜ao com o seu valor, ir´a eventualmente aterrar num m´aximo global, mesmo que de forma extremamente ineficiente.

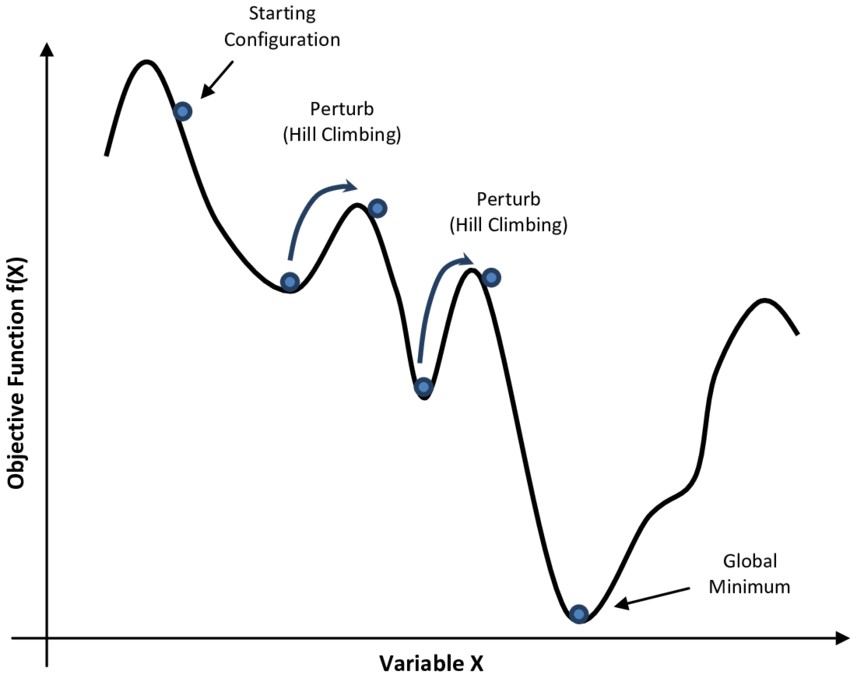
Assim sendo, parece razo´avel tentarmos combinar ambas as abordagens, de forma a conseguirmos uma abordagem completa e eficiente. Com Simulated Annealing podemos fazer isso. Neste algoritmo, passamos do ponto de vista de *Hill Climbing* para *Gradient Descent* (minizamos o custo).

Imaginemos uma bola a deslizar por uma descida com v´arias lombas.

Se deixarmos a bola rolar, ela ir´a parar num m´ınimo local. Se abanarmos a superf´ıcie, podemos tirar a bola do m´ınimo local (possivelmente para outro m´ınimo).

O truque ´e agitar a superf´ıcie de forma a que a bola saia do m´ınimo local, mas n˜ao com demasiada for¸ca de forma a que a mesma nunca atinja o m´ınimo global.

O que pretendemos ´e comec¸ar a agitar com forc¸a e depois reduzir a intensidade.



O algoritmo ´e muito parecido ao *Hill Climbing*.

Em vez de escolhermos o melhor estado vizinho, escolhemos um estado vizinho aleat´orio.

Se o novo estado for melhor que o actual, ent˜ao aceitamos a mudanc¸a. Caso contr´ario, aceitamos a mudanc¸a com uma probabilidade (menor que

1).

A probabilidade piora exponencialmente com a diferenc¸a de qualidade entre os dois estados (∆*E*).

A probabilidade tamb´em diminui com a temperatura *T*: mudan¸cas que piorem o estado actual s˜ao aceites com maior probabilidade no in´ıcio (quando *T* ´e maior) e `a medida que *T* fica mais pequeno, a probabilidade de aceitarmos estados piores tamb´em decresce.

Se *T* descer para 0 lentamente, o algoritmo ir´a encontrar com probabilidade pr´oxima de 1 (*e*∆*E/T* ) o m´ınimo global (distribui¸c˜ao de Boltzmann).

# Pesquisa Local - Simulated Annealing

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

*current* ← *problem*.INITIAL

for *t* = 1 to ∞ do

*T* ← *schedule(t)* if *T* = 0 then

return *current*

end if

*next* ← random successor of current ∆*E* ← VALUE(*current*) − VALUE(*next*)

if ∆*E >* 0 then

*current* ← *next*

else

*current* ← *next* with probability *e*∆*E/T*

end if

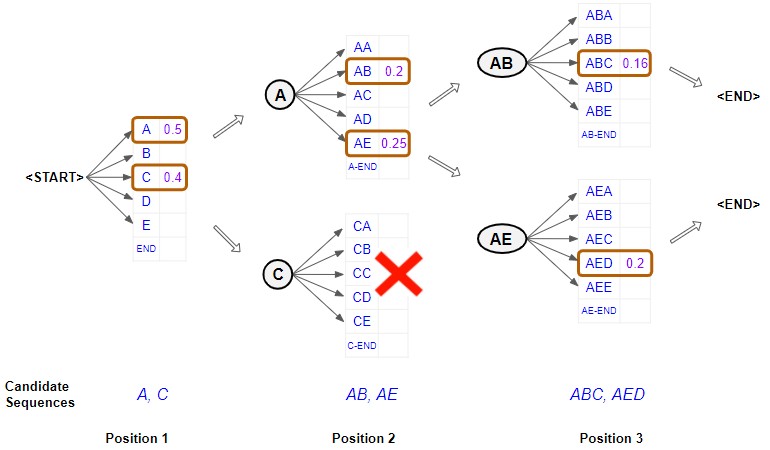
end for return *failure*

# Local Beam Search

Na maioria dos casos, n˜ao precisamos de manter apenas um n´o em mem´oria. O algoritmo local *beam search* mant´em *k* estados em mem´oria, em vez de apenas um.

Come¸ca com *k* estados gerados aleatoriamente e em cada passo, todos os sucessores de cada um dos *k* estados s˜ao gerados.

Se algum deles for a soluc¸˜ao, ent˜ao retornamos. Caso contr´ario, selecionamos os melhores *k* estados dessa lista completa (estados iniciais + sucessores).



# Local Beam Search

Difere de uma paraleliza¸c˜ao do *Random-Restart Hill Climbing* na medida em que cada um dos n´os n˜ao ir´a ser expandido em paralelo e de forma independente.

Pelo contr´ario, informa¸c˜ao u´til ir´a ser passada e partilhada por toda a execuc¸˜ao do algoritmo. O algoritmo assim foca-se mais na parte mais promissora do espa¸co de estados.

Pode, no entanto, sofrer de alguma falta de diversidade dentro dos *k* estados (podemos aprofundar muito na mesma regi˜ao do espa¸co de estados).

Uma variante estoc´astica do algoritmo ajuda na resolu¸c˜ao desta limita¸c˜ao: em vez de escolhermos os melhores *k* sucessores, escolhemos *k* sucessores aleatoriamente, cuja probabilidade de escolha est´a relacionada com a qualidade do mesmo, aumentando assim a diversidade.