Sistemas Inteligentes

Busca - Funções Heurísticas e Algoritmos de Melhorias Interativas

Ao final desta aula, a gente deve...

- Especificar boas funções heurísticas para o nosso problema
- Conhecer os algoritmos de melhorias Interativas e suas aplicações

Inventando Funções Heurísticas

- Como escolher uma boa função heurística *h* ?
  - *h* depende de cada problema particular.
  - *h* deve ser *admissível*
    - i.e., não superestimar o custo real da solução
- Existem estratégias genéricas para definir *h* :

1) Relaxar restrições do problema

2) "Aprender" a heurística pela experiência
    - Aprendizagem de máquina

Estratégias para definir *h*
(1) Relaxando o problema

- Problema Relaxado:
  - versão simplificada do problema original, onde os operadores são menos restritivos
- Exemplo: jogo dos 8 números
  - Operador original
    - um número pode mover-se de A para B se A é adjacente a B e B está vazio
    - busca exaustiva $\approx 3^{22}$ estados possíveis
  - Operadores relaxados:
    1. um número pode mover-se de A para B se A é adjacente a B (*h2*)
    2. um número pode mover-se de A para B se B está vazio
    3. um número pode mover-se de A para B (*h1*)

Estratégias para definir *h*

(1) Relaxando o problema

5	4	
6	1	8
7	3	2

Start State

1	2	3
8		4
7	6	5

Goal State

Heurísticas para o jogo dos 8 números

h1 = no. de elementos fora do lugar (*h1*=7)

h2 = soma das distâncias de cada número à posição final

(*h2* = 2+3+3+2+4+2+0+2=18)

Estratégias para definir *h*
(1) Relaxando o problema

- O custo de uma solução ótima para um problema relaxado é sempre uma heurística admissível para o problema original.
- Existem softwares capazes de gerar automaticamente problemas relaxados
  - Se o problema for definido em uma linguagem formal
- Existem também softwares capazes de gerar automaticamente funções heurísticas para problemas relaxados

Escolhendo Funções Heurísticas

- É sempre melhor usar uma função heurística com valores mais altos
  - i.e., mais próximos do valor real do custo de caminho
  - ** contanto que ela seja admissível **
- No exemplo anterior, h_2 é melhor que h_1
  - $\forall n,\ h_2(n) \geq h_1(n)$
  - A* com h_2 expande menos nós do que com h_1
- h_i domina $h_k \Rightarrow h_i(n) \geq h_k(n)\ \forall$n no espaço de estados
  - h_2 domina h_1

Escolhendo Funções Heurísticas

- Caso existam muitas funções heurísticas para o mesmo problema,
  - e nenhuma delas domine as outras,
  - usa-se uma heurística composta:
    - $h(n) = max\ (h_1(n), h_2(n), \ldots, h_m(n))$
- Assim definida, h é admissível e domina cada função h_i individualmente
- Existem software capazes de gerar automaticamente problemas relaxados
  - Se o problema for definido em uma linguagem formal

Estratégias para definir *h*
(2) Aprendendo a heurística

- Definindo *h* com aprendizagem automática

(1) Criar um corpus de exemplos de treinamento

- Resolver um conjunto grande de problemas
  - e.g., 100 configurações diferentes do jogo dos 8 números
- Cada solução ótima para um problema provê exemplos
  - Cada exemplo consiste em um par
  - (estado no caminho "solução", custo real da solução a partir daquele ponto)

Estratégias para definir *h*

(2) Aprendendo a heurística

(2) Treinar um algoritmo de aprendizagem indutiva

- Que então será capaz de prever o custo de outros estados gerados durante a execução do algoritmo de busca

Qualidade da função heurística

- Medida através do fator de expansão efetivo (b*)
  - b* é o fator de expansão de uma árvore uniforme com N nós e nível de profundidade d
  - $N = 1 + b^* + (b^*)^2 + ... + (b^*)^d$, onde
    - N = total de nós expandidos para uma instância de problema
    - d = profundidade da solução
- Mede-se empiricamente a qualidade de *h* a partir do conjunto de valores experimentais de *N* e *d*.
  - uma boa função heurística terá o b* muito

Qualidade da função heurística

- Observações:
  - Se o custo de execução da função heurística for maior do que expandir os nós, então ela *não* deve ser usada.
  - uma boa função heurística deve ser *eficiente* e *econômica*.

Experimento com 100 problemas

	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Uma boa função heurística terá o b^* muito próximo de 1.

Na sequencia....

- Algoritmos de Melhorias Iterativas

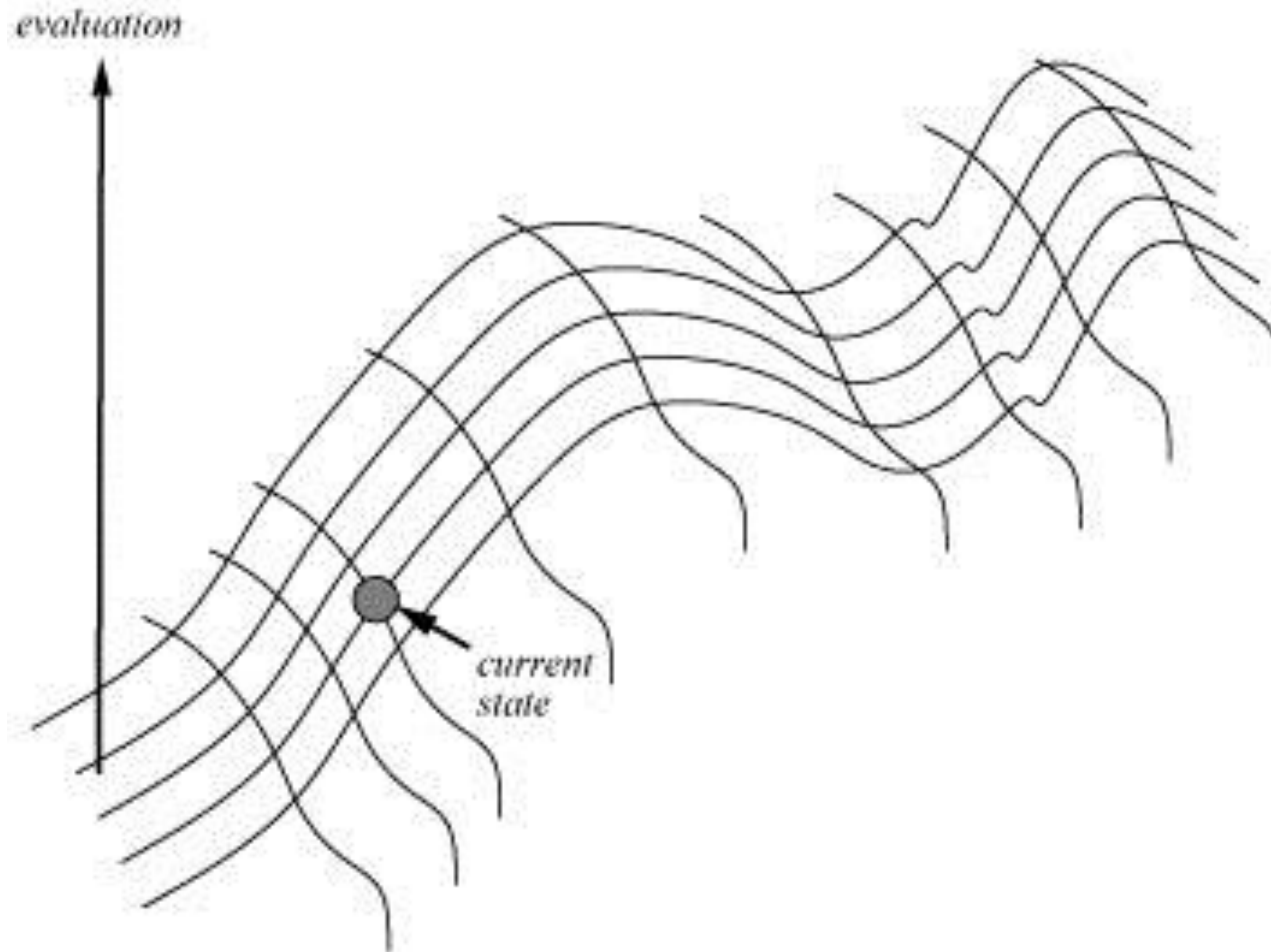
Algoritmos de Melhorias Iterativas

- Dois exemplos clássicos
  - Subida da encosta
  - Têmpera simulada

Algoritmos de Melhorias Iterativas
Iterative Improvement Algorithms

- Idéia geral
  - começar com um estado inicial
    - configuração completa, solução aceitável
  - e tentar melhorá-lo iterativamente
  - E.g., ajustar a imagem da TV com antena interna
- Os estados são representados sobre uma superfície (gráfico)
  - a altura de qualquer ponto na superfície corresponde à função de avaliação do estado naquele ponto

Exemplo de Espaço de Estados



Algoritmos de Melhorias Iterativas

- O algoritmo se “move” pela superfície em busca de pontos mais altos
  - Objetivos (onde a função de avaliação é melhor)
    - Objetivos são estados mais adequados
- O ponto mais alto corresponde à solução ótima
  - máximo global
    - nó onde a função de avaliação atinge seu valor máximo
- Aplicações: problemas de otimização
  - por exemplo, linha de montagem, rotas, etc.

Algoritmos de Melhorias Iterativas

- Esses algoritmos guardam apenas o estado atual, e não vêem além dos vizinhos imediatos do estado
  - Contudo, muitas vezes são os melhores métodos para tratar problemas reais muito complexos.
- Duas classes de algoritmos:
  - Subida da Encosta ou Gradiente Ascendente
    - *Hill-Climbing*
    - só faz modificações que melhoram o estado atual.
  - Têmpera Simulada
    - *Simulated Annealing*
    - pode fazer modificações que pioram o estado temporariamente para fugir de máximos locais

Subida da Encosta - *Hill-Climbing*

- O algoritmo não mantém uma árvore de busca:
  - guarda apenas o estado atual e sua avaliação
- É simplesmente um "loop" que se move
  - na direção crescente da função de avaliação
    - para maximizar
  - ou na direção decrescente da função de avaliação
    - para minimizar
    - Pode ser o caso se a função de avaliação representar o custo, por exemplo...

Subida da Encosta: algoritmo

- função **<u>Hill-Climbing</u> (*problema*)** retorna **uma solução**

  variáveis locais: *atual* (o nó atual), *próximo* (o próximo nó)

atual ← Estado-Inicial do *Problema*

loop do

próximo ← **sucessor** do nó *atual* **de maior/menor valor**

(i.e., expande nó *atual* e seleciona seu melhor filho)

se Valor[*próximo*] < Valor[*atual*] (ou >, para minimizar)

então retorna nó *atual* (o algoritmo pára)

atual ← *próximo*

end

Exemplo de Subida da Encosta

Cálculo da menor rota com 5 nós

- estado inicial = (N1, N2, N3, N4, N5)
- f = soma das distâncias diretas entre cada nó, na ordem escolhida (admissível!)
- operadores = permutar dois nós quaisquer do caminho
- restrição = somente caminhos conectados são estados válidos
- estado final = nó onde valor de f é mínimo

- e1 = {N1, N2, N3, N4, N5}
  - f(N1, N2, N3, N4, N5) = 10
- e2 = {N2, N1, N3, N4, N5}
  - f(N2, N1, N3, N4, N5) = 14
- e3 = {N2, N1, N4, N3, N5}
  - f(N2, N1, N3, N4, N5) = 9!!!

Subida da Encosta
Problemas

- O algoritmo move-se sempre na direção que apresenta maior taxa de variação para *f*
- Isso pode levar a 3 problemas:
    1. Máximos locais
    2. Planícies (platôs)
    3. Encostas e picos

Subida da Encosta

Máximos locais

- Os máximos locais são picos mais baixos do que o pico mais alto no espaço de estados
  - máximo global - solução ótima
- Nestes casos, a função de avaliação leva a um valor máximo para o caminho sendo percorrido
  - a função de avaliação é menor para todos os filhos do estado atual, apesar de o objetivo estar em um ponto mais alto
    - essa função utiliza informação “local”
  - e.g., xadrez:
    - eliminar a Rainha do adversário pode levar o jogador a perder o jogo.

Subida da Encosta
Máximos locais

- O algoritmo pára no máximo local
  - só pode mover-se com taxa crescente de variação de f
    - restrição do algoritmo
  - Exemplo de taxa de variação negativa
    - Jogo dos 8 números:
      - mover uma peça para fora da sua posição correta para dar passagem a outra peça que está fora do lugar tem taxa de variação negativa!!!

Subida da Encosta

Platôs (Planícies)

- Uma região do espaço de estados onde a função de avaliação dá o mesmo resultado
  - todos os movimentos são iguais (taxa de variação zero)
    - $f(n) = f(filhos(n))$
- O algoritmo pára depois de algumas tentativas
  - Restrição do algoritmo
- Exemplo: jogo 8-números
  - em algumas situações, nenhum movimento possível vai influenciar no valor de *f*, pois nenhum número vai chegar ao seu objetivo.

Subida da Encosta
Encostas e Picos

- Apesar de o algoritmo estar em uma direção que leva ao pico (máximo global), não existem operadores válidos que conduzam o algoritmo nessa direção
  - Os movimentos possíveis têm taxa de variação zero ou negativa
    - restrição do problema e do algoritmo
- Exemplo: cálculo de rotas
  - quando é necessário permutar dois pontos e o caminho resultante não está conectado.

Subida da Encosta

Problemas - solução

- Nos casos apresentados, o algoritmo chega a um ponto de onde não faz mais progresso
- Solução: reinício aleatório *(random restart)*
  - O algoritmo realiza uma série de buscas a partir de estados iniciais gerados aleatoriamente
  - Cada busca é executada
    - até que um número máximo estipulado de iterações seja atingido, ou
    - até que os resultados encontrados não apresentem melhora significativa
  - O algoritmo escolhe o melhor resultado obtido com as diferentes buscas.
    - Objetivo!!!

Subida da Encosta: análise

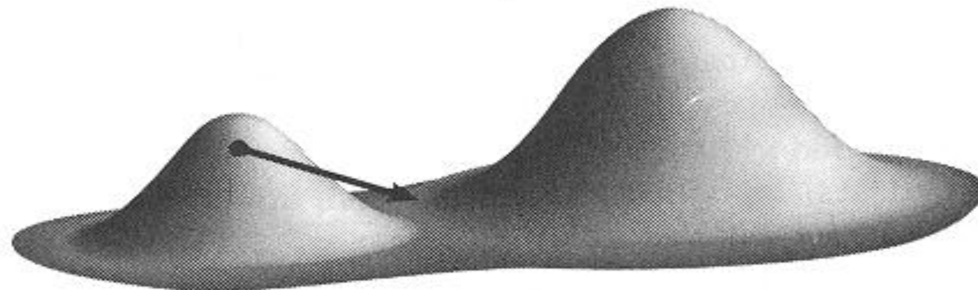
- O algoritmo é completo?
  - **SIM,** para problemas de *otimização*
    - uma vez que cada nó tratado pelo algoritmo é sempre um estado completo (uma solução)
  - **NÃO,** para problemas onde os nós não são estados completos
    - e.g., jogo dos 8-números
    - semelhante à busca em profundidade
- O algoritmo é ótimo?
  - **TALVEZ,** para problemas de *otimização*
    - quando iterações suficientes forem permitidas...
  - **NÃO,** para problemas onde os nós não são estados completos

Subida da Encosta: análise

- O sucesso deste método depende muito do formato da superfície do espaço de estados:
  - se há poucos máximos locais, o reinício aleatório encontra uma boa solução rapidamente
  - caso contrário, o custo de tempo é exponencial.

Têmpera Simulada -*Simulated Annealing*

- Este algoritmo é semelhante à Subida da Encosta, porém oferece meios para escapar de máximos locais
  - quando a busca fica “presa” em um máximo local, o algoritmo não reinicia a busca aleatoriamente
  - ele retrocede para escapar desse máximo local
  - esses retrocessos são chamados de passos indiretos
- Apesar de aumentar o tempo de busca, essa estratégia consegue escapar dos máximos locais



Têmpera Simulada

- Analogia com cozimento de vidros ou metais:
  - processo de resfriar um líquido gradualmente até ele se solidificar
- O algoritmo utiliza um mapeamento de resfriamento de instantes de tempo (*t*) em temperaturas (*T*).

Têmpera Simulada

- Nas iterações iniciais, não escolhe necessariamente o "melhor" passo, e sim um movimento aleatório:
  - se a situação melhorar, esse movimento será sempre escolhido posteriormente;
  - caso contrário, associa a esse movimento uma probabilidade de escolha menor do que 1.
- Essa probabilidade depende de dois parâmetros, e decresce exponencialmente com a piora causada pelo movimento,
  - $e^{\Delta E/T}$, onde:

    ΔE = Valor[próximo-nó] - Valor[nó-atual]

    T = Temperatura

Têmpera Simulada: algoritmo

função **Anelamento-Simulado** (***problema, mapeamento***)
retorna **uma solução**
variáveis locais: *atual*, *próximo*, T (temperatura que controla a probabilidade de passos para trás)

atual ← Faz-Nó(Estado-Inicial[*problema*])

for $t \leftarrow 1$ **to** ∞ **do**

$T \leftarrow$ mapeamento[t]

Se $T = 0$

então retorna *atual*

próximo ← um sucessor de *atual* escolhido aleatoriamente

$\Delta E \leftarrow$ Valor[*próximo*] - Valor[*atual*]

Se $\Delta E > 0$

então *atual* ← *próximo*

senão *atual* ← *próximo com probabilidade* = $e^{-\Delta E/T}$

Têmpera Simulada

- Para valores de T próximos de zero
  - a expressão $\Delta E/T$ cresce
  - a expressão $e^{-\Delta E/T}$ tende a zero
  - a probabilidade de aceitar um valor de próximo menor que corrente tende a zero
  - o algoritmo tende a aceitar apenas valores de próximo maiores que corrente
- Conclusão
  - com o passar do tempo (diminuição da temperatura), este algoritmo passa a funcionar como Subida da Encosta

Têmpera Simulada

- Implementação (dica)
  - Gerar número aleatório entre (0,1) e comparar com o valor da probabilidade
  - Se número sorteado < probabilidade, aceitar movimento para trás
- Análise
  - O algoritmo é completo
  - O algoritmo é ótimo se o mapeamento de resfriamento tiver muitas entradas com variações suaves
    - isto é, se o mapeamento diminui *T* suficientemente devagar no tempo, o algoritmo vai encontrar um máximo global ótimo.

Próxima aula

- Tira dúvidas!
- Depois disso.... Passamos para a parte II do curso – Representação de Conhecimento e Raciocínio!