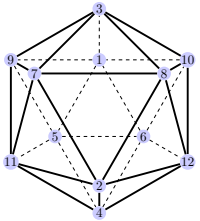
4 - Algoritmos gulosos/semi-gulosos (*greedy/semi-greedy*)

Alexandre Checoli Choueiri

19/01/2025



Introdução

Introdução

A importância da solução inicial

1. As metaheuristicas são procedimentos iterativos que tentam melhorar uma **solução já existente**.
2. Ou seja, antes de aplicarmos qualquer metaheuristica precisamos pelo menos ter uma solução inicial (não importa a sua qualidade, e em alguns casos nem a sua factibilidade!).
3. Uma família de algoritmos muito utilizada para geração de soluções inicias é chamada de: Algoritmos Construtivos Gulosos.

Intuição

Algoritmos gulosos

Por quê **construtivos**?

> Construtivos pois a solução é inciada de um conjunto vazio, e a cada iteração um valor é designado para uma variável de decisão (a solução é construída por partes).

Por quê **gulosos**?

> O termo guloso diz respeito a forma de escolha do próximo elemento que fará parte da solução. Os algoritmos gulosos sempre escolhem a **opção mais vantajosa naquele momento** (ou seja, somente olhando a solução parcialmente construída e o próximo movimento).

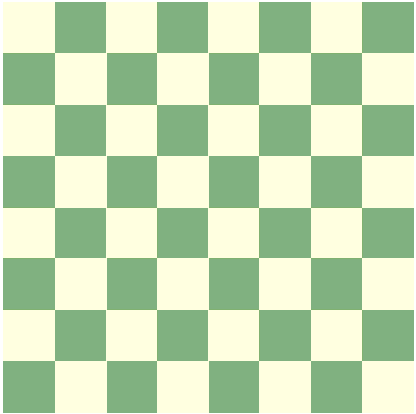
Mas se a cada iteração escolhemos a solução mais vantajosa, por quê esse algoritmo não nos leva a solução ótima?

> Embora existam algoritmos gulosos que gerem a solução ótima (algoritmo de Prim e Kruskal para árvores geradoras mínimas), uma decisão ótima tomada a cada estágio de decisão do algoritmo não garante que o ótimo global seja atingido no final. Você consegue pensar em um exemplo onde isso pode ocorrer?

Vamos pensar no jogo de xadrez.

Intuição

Algoritmos gulosos

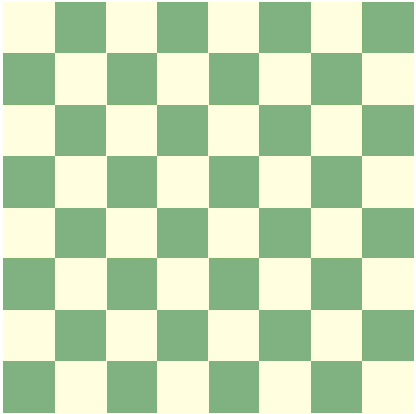


Sabemos que no xadrez todas as peças possuem uma pontuação:

- Peão:1
- Cavalo/Bispo:3
- Torre:5
- Dama:9

Intuição

Algoritmos gulosos



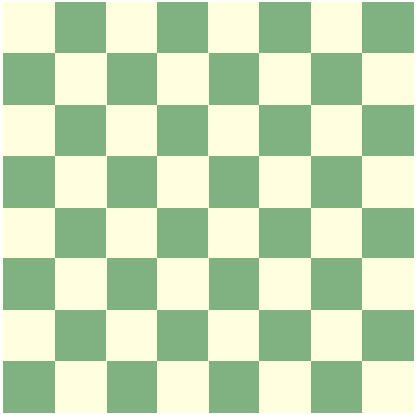
Ainda, podemos pensar na "solução" de um jogo de xadrez como uma sequência de movimentos

$$e_1 \rightarrow e_2 \rightarrow ... \rightarrow e_n \qquad (1)$$

Que levam ao fim do jogo.

Intuição

Algoritmos gulosos

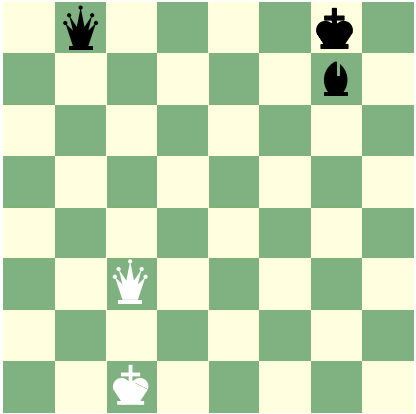


Um algoritmo guloso para o xadrez poderia ter o seguinte design:

1. Quando for a sua vez, faça:
2. Faça o movimento e_k que minimize o $\sum$ de pontos das peças do adversário.

Intuição

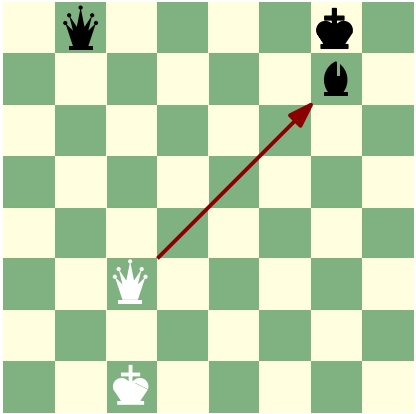
Algoritmos gulosos



Considerando que as brancas jogam no tabuleiro ao lado. O que o algoritmo guloso faria?

Intuição

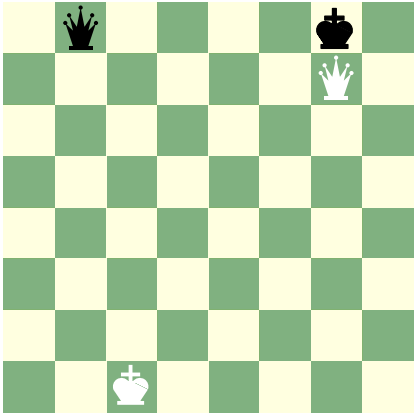
Algoritmos gulosos



Obviamente o algoritmo eliminaria o bispo com a dama, **reduzindo os pontos do adversário em 3 unidades**!

Intuição

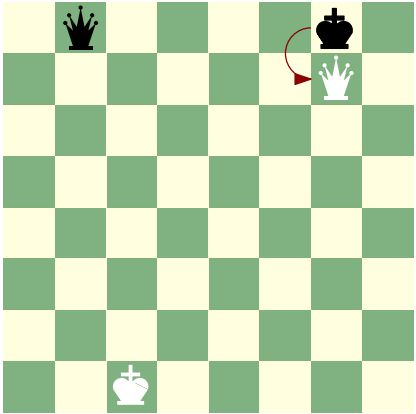
Algoritmos gulosos



Note que a jogada foi de fato a melhor a ser realizada naquele momento (de forma gulosa, não tem outra jogada e_k que reduz mais os pontos do adversário).

Intuição

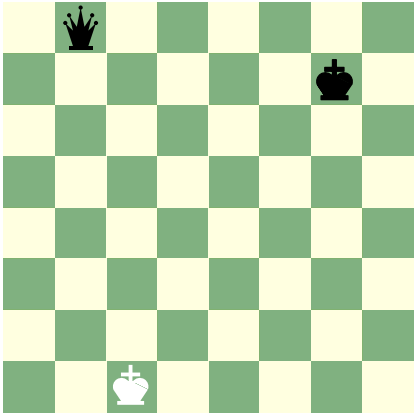
Algoritmos gulosos



Mas o que vai acontecer agora? Pretas jogam...

Intuição

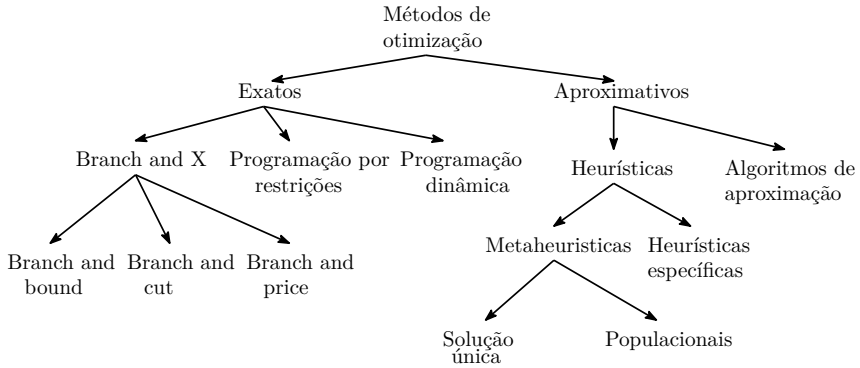
Algoritmos gulosos



PERDEU A DAMA!

Onde eles se encaixam

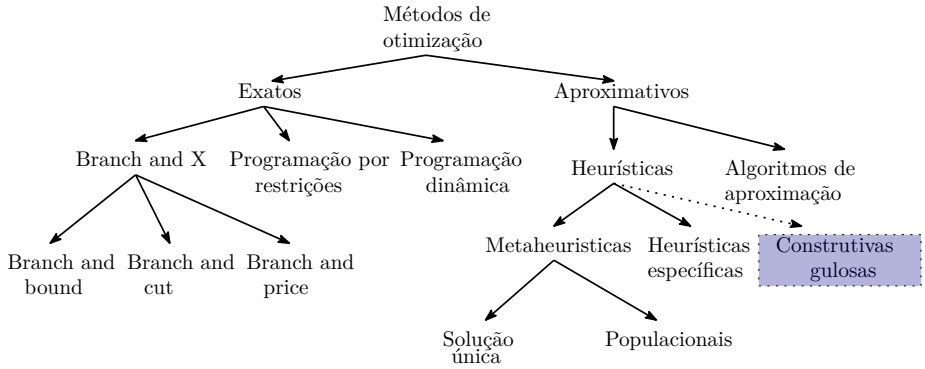
Algoritmos gulosos



Onde eles se encontram na nossa árvore de métodos de resolução?

Onde eles se encaixam

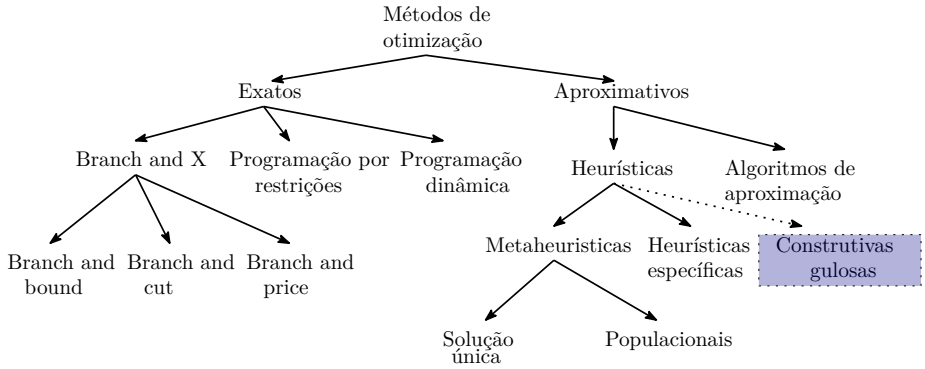
Algoritmos gulosos



A verdade é que eles são difíceis de classificar (gerando disputas)...eu me sinto confortável colocando-os na categoria de heurísticas (embora possam ser exatos!).

Onde eles se encaixam

Algoritmos gulosos



Eles são específicos...mas também existe um *template* genérico!

Conclusões

Vantagens: os algoritmos gulosos (greedy) são muito populares pois são fáceis de se criar. Além do mais, sua complexidade é geralmente reduzida em relação aos algoritmos iterativos, o que é um atrativo quando se precisa de uma solução rápida (porém não tão boa).

Conclusões

Desvantagens: os algoritmos gulosos (greedy), de forma geral são considerados míopes na construção de uma solução. Alguns deles podem incluir capacidades de previsão (*look-ahead*), em que as consequências futuras de uma decisão são estimadas e levadas em consideração na tomada de decisão atual.

Definições e terminologia

Definição

Terminologia

Considere um problema de otimização em que a solução pode ser definida pela presença/ausência de um conjunto finito de elementos:

$$E = \{e_1, e_2, ..., e_n\} \tag{2}$$

Uma solução parcial s pode ser vista como um subconjunto

$$\{e_1, e_2, ..., e_k\} \tag{3}$$

de elementos de e_i do conjunto E. O conjunto inicial é $s = \emptyset$ (solução vazia).

Definição

Terminologia

Considere um problema de otimização em que a solução pode ser definida pela presença/ausência de um conjunto finito de elementos:

$$E = \{e_1, e_2, ..., e_n\} \tag{2}$$

Uma solução parcial s pode ser vista como um subconjunto

$$\{e_1, e_2, ..., e_k\} \tag{3}$$

de elementos de e_i do conjunto E. O conjunto inicial é $s = \emptyset$ (solução vazia). A cada iteração uma heurística é usada para selecionar um novo elemento para ser adicionado no conjunto s. Quando um elemento é selecionado para entrar em s ele **nunca mais é removido**.

Definição

Pseudocódigo

Abaixo é mostrado um pseudocódigo genérico de algoritmos gulosos.

Algorithm 1 Template de algoritmo guloso

$s = \emptyset$ ▷ Solução inicia sem elementos
while Solução s for incompleta **do**
  e_i = seleciona-elemento($E \setminus \{e : e \in s\}$) ▷ Elementos que ainda não estão em s
  if $s \cup e_i$ é factível **then**
    $s = s \cup e_i$ ▷ Adiciona e_i em s
  end if
end while
return s.

Definição

Design

As duas maiores questões de design para os algoritmos gulosos são então:

1. **A definição do conjunto de elementos:** Para um problema específico, é preciso conseguir identificar uma solução como um subconjuntos de elementos.
2. **A heurística de seleção de elementos:** A cada iteração essa heurística é a responsável por determinar o melhor elemento do conjunto. Portanto, ela calcula o "custo/lucro"da inserção de cada elemento e seleciona o que mais contribui para a função objetivo. Esse cálculo não precisa ser em termos da contribuição do elemento para a função objetivo, qualquer quantificador pode ser criado para a seleção do elemento.

Exemplo - o problema da mochila

Definição

Exemplo - problema da mochila

EXERCÍCIO: Resolva as duas questões básicas do design de um algoritmo guloso considerando o problema da mochila. Escreva um pseudo-código para o algoritmo.
Relembrando o problema da mochila:

	Ipod	Abobrinha	H_2O	Canivete	Carne	Arroz	Aveia	PS4
Valor	10	8	5	15	25	17	8	30
Peso	50	55	60	45	15	25	35	25

Quais itens selecionar de forma a maximizar a utilidade (Valor), respeitando o limite de peso da mochila?

Exemplo - mochila

Exemplo - problema da mochila

1. **A definição do conjunto de elementos:** Para um problema específico, é preciso conseguir identificar uma solução como um subconjunto de elementos.
2. **A heurística de seleção de elementos:** A cada iteração essa heurística é a responsável por determinar o melhor elemento do conjunto. Portanto ela calcula o "custo/lucro" da inserção de cada elemento e seleciona o que mais contribui para a função objetivo. Esse cálculo não precisa ser em termos da contribuição do elemento para a função objetivo, qualquer quantificador pode ser criado para a seleção do elemento.

Exemplo - problema da mochila

1. **A definição do conjunto de elementos:** Para o problema da mochila essa questão é fácil. O conjunto E é o próprio conjunto de elementos que podem ser adicionados à mochila. A cada iteração um novo item é adicionado à mochila.

Exemplo - problema da mochila

1. **A definição do conjunto de elementos:** Para um problema específico, é preciso conseguir identificar uma solução como um subconjunto de elementos.
2. **A heurística de seleção de elementos:** A cada iteração essa heurística é a responsável por determinar o melhor elemento do conjunto. Portanto ela calcula o "custo/lucro"da inserção de cada elemento e seleciona o que mais contribui para a função objetivo. Esse cálculo não precisa ser em termos da contribuição do elemento para a função objetivo, qualquer quantificador pode ser criado para a seleção do elemento.

Exemplo - problema da mochila

1. **A definição do conjunto de elementos:** Para o problema da mochila essa questão é fácil. O conjunto E é o próprio conjunto de elementos que podem ser adicionados à mochila. A cada iteração um novo item é adicionado à mochila.
2. **A heurística de seleção de elementos:** Podemos selecionar os elementos de forma gulosa escolhendo aquele que mais contribui para a função objetivo do problema (ou seja, o elemento com maior "valor/custo").

Exemplo - problema da mochila

Algorithm 2 Algoritmo guloso

$s = \emptyset$
$E_s =$ ordena_por_valor(E) ▷ Ordena os elementos por valor
for $e_i \in E_s$ **do**
  if $s \cup e_i \leq$ C **then** ▷ Verifica se o peso $\leq$ capacidade
    $s = s \cup e_i$
  end if
end for
return s

Exemplo - problema da mochila

EXEMPLO:

1. Implemente o algoritmo proposto e use-o para resolver duas instancias do conjunto: uma grande e uma pequena. Compare o seu resultado com o ótimo.
2. Você consegue pensar em uma outra forma (mais inteligente) de avaliar as elementos do conjunto? Implemente uma nova forma e compare com as soluções encontradas anteriormente.

Exemplo - TSP

O vizinho mais próximo - TSP

Um algoritmo guloso muito famoso para o TSP é chamado de vizinho-mais-próximo. As questões de design do algoritmo guloso são respondidas da seguinte forma:

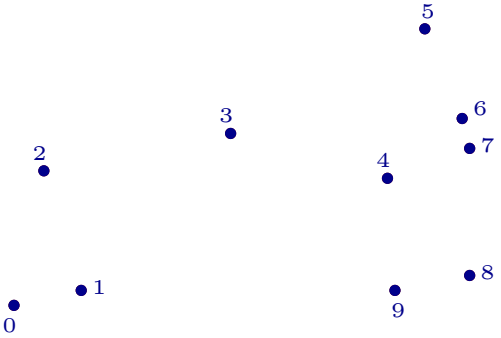
1. **A definição do conjunto de elementos:** Os elementos da solução são definidos como os arcos do grafo.
2. **A heurística de seleção de elementos:** A cada iteração, o arco com a menor distância até o último ponto da rota é selecionado (o ponto inicial é selecionado aleatóriamente).

O vizinho mais próximo - TSP

Vizinho mais próximo simplificado

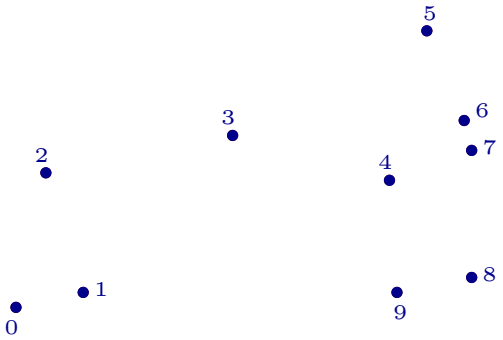
Simplificando: De forma simplificada, a cada iteração o algoritmo seleciona o ponto mais próximo do último selecionado, que ainda não está contido na solução parcial atual.

O vizinho mais próximo - TSP



Considerando o conjunto de 10 pontos acima. Inicialmente inserimos um ponto na solução s (a seleção deste ponto pode ser aleatória).

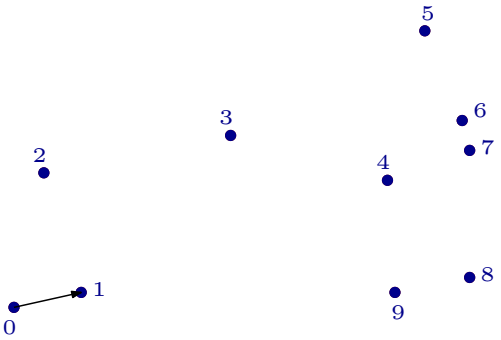
O vizinho mais próximo - TSP



Por exemplo, inserindo o ponto 0 na solução:

$$s = \{0\}. \tag{4}$$

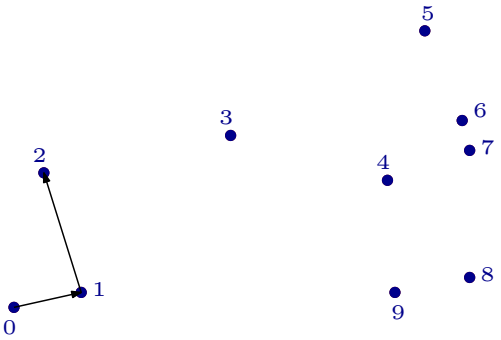
O vizinho mais próximo - TSP



Em seguida selecionamos o ponto (arco) com a menor distância até o ponto 0, nesse caso o ponto 1:

$$s = \{0, 1\}. \tag{5}$$

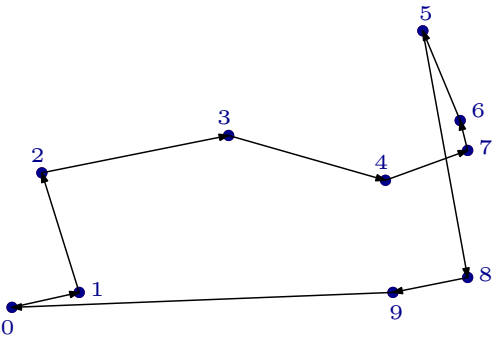
O vizinho mais próximo - TSP



Da mesma forma agora selecionamos o ponto mais perto de 1 que ainda não está na rota! Neste caso 2.

$$s = \{0, 1, 2\}. \tag{6}$$

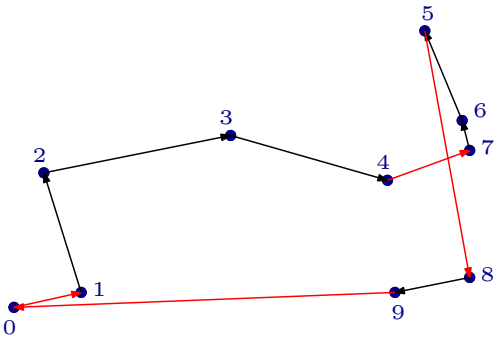
O vizinho mais próximo - TSP



E assim sucessivamente, até obtermos a rota final:

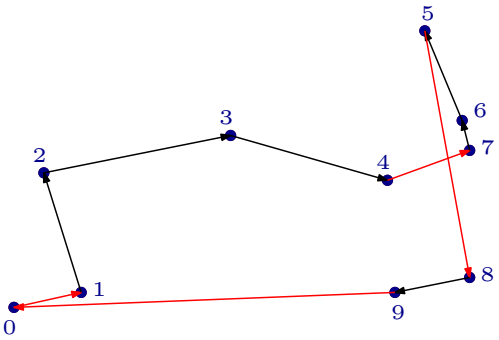
$$s = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}. \tag{7}$$

O vizinho mais próximo - TSP



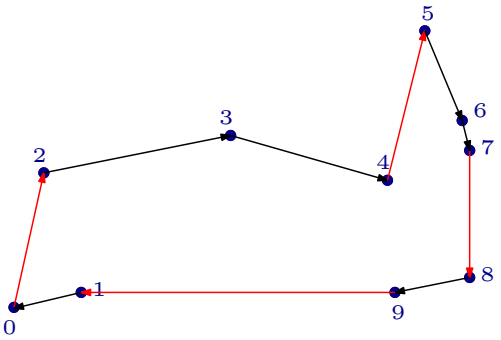
Note que a miopia do método gerou arcos ruins (sempre que uma rota possui cruzamentos, ela pode ser melhorada - a solução ótima nunca têm cruzamentos).

O vizinho mais próximo - TSP



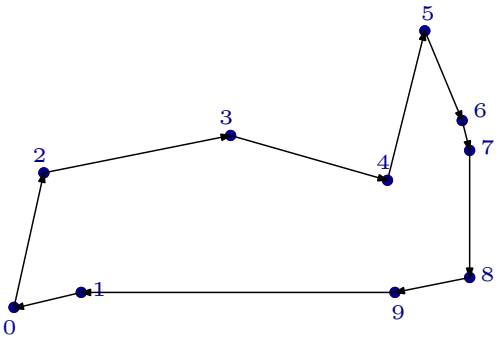
Note que a miopia do método gerou arcos ruins (sempre que uma rota possui cruzamentos, ela pode ser melhorada - **a solução ótima nunca têm cruzamentos**).

O vizinho mais próximo - TSP



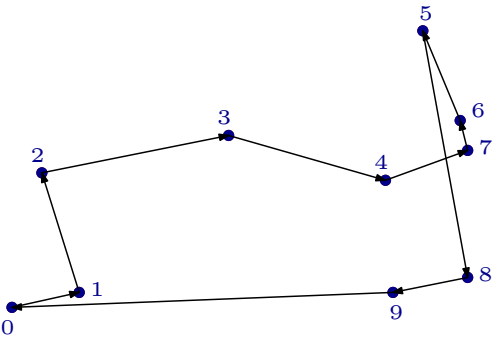
Uma rota melhorada fica como mostrado acima.

O vizinho mais próximo - TSP



Uma rota melhorada fica como mostrado acima.

O vizinho mais próximo - TSP



OBSERVAÇÃO: Note que o ponto inicial escolhido **altera a solução gerada**!

Algoritmos semi-gulosos (*semi-greedy*)

Idéia

A formalização de algoritmos semi-gulosos foi feita por Hart (*1987 - Semi-greedy heuristics: An empirical study - Hart, J Pirie and Shogan, Andrew W*).

Idéia

A formalização de algoritmos semi-gulosos foi feita por Hart (*1987 - Semi-greedy heuristics: An empirical study - Hart, J Pirie and Shogan, Andrew W*).

Idéia

A formalização de algoritmos semi-gulosos foi feita por Hart (*1987 - Semi-greedy heuristics: An empirical study - Hart, J Pirie and Shogan, Andrew W*).

Sua ideia central é muito simples, e conta com os seguintes pontos chave:

1. Algoritmos gulosos são **determinísticos**.

Idéia

A formalização de algoritmos semi-gulosos foi feita por Hart (*1987 - Semi-greedy heuristics: An empirical study - Hart, J Pirie and Shogan, Andrew W*).

Sua ideia central é muito simples, e conta com os seguintes pontos chave:

1. Algoritmos gulosos são **determinísticos**.
2. Portanto, se executados várias vezes, sempre resultam nas mesmas soluções.

Idéia

A formalização de algoritmos semi-gulosos foi feita por Hart (*1987 - Semi-greedy heuristics: An empirical study - Hart, J Pirie and Shogan, Andrew W*).

Sua ideia central é muito simples, e conta com os seguintes pontos chave:

1. Algoritmos gulosos são **determinísticos**.
2. Portanto, se executados várias vezes, sempre resultam nas mesmas soluções.
3. Se for possível torná-los **estocásticos...**.

Idéia

A formalização de algoritmos semi-gulosos foi feita por Hart (*1987 - Semi-greedy heuristics: An empirical study - Hart, J Pirie and Shogan, Andrew W*).

Sua ideia central é muito simples, e conta com os seguintes pontos chave:

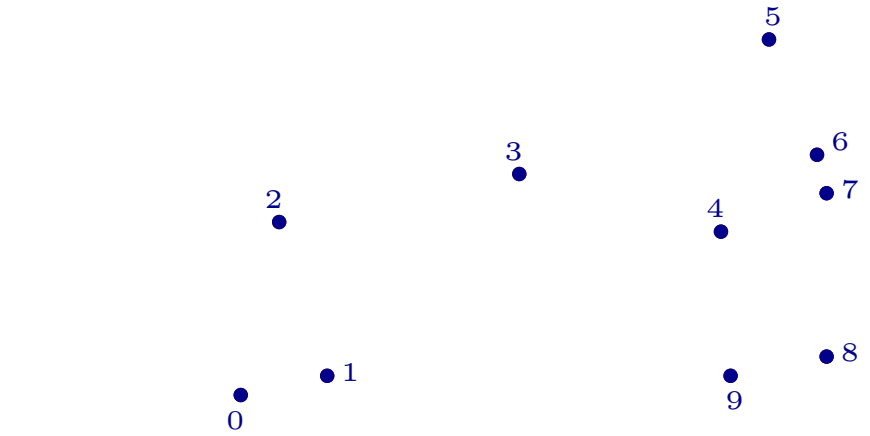
1. Algoritmos gulosos são **determinísticos**.
2. Portanto, se executados várias vezes, sempre resultam nas mesmas soluções.
3. Se for possível torná-los **estocásticos...**.
4. Será possível executar o algoritmo várias vezes, e coletar a melhor solução de todas no final.

Idéia

Ou seja, a ideia principal é conseguir gerar soluções iniciais gulosas diferentes umas das outras. Para que isso seja possível, é necessário uma componente de **aleatoriedade** no algoritmo. É possível pensar em diversas formas de aleatorizar um algoritmo guloso, Hart propõe duas formas, uma dela é:

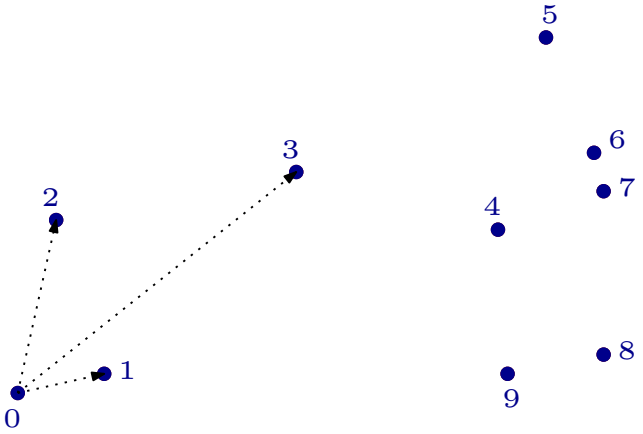
Heurística baseada em porcentagem p: A cada iteração do algoritmo guloso, ao invés de selecionar a melhor opção, escolhe-se uma soluções aleatória dentre as $p\%$ melhores.

Exemplo TSP



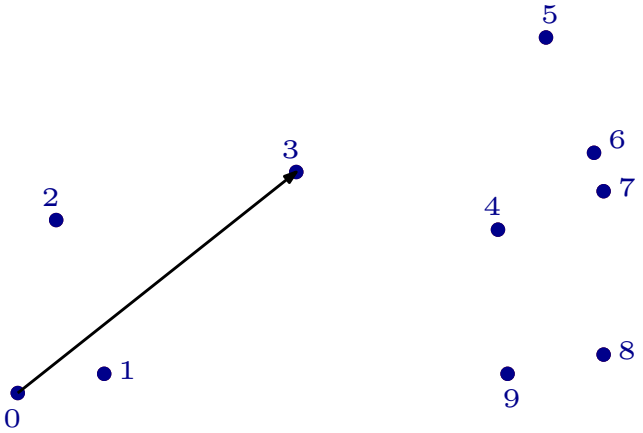
Considerando o exemplo anterior do TSP.

Exemplo TSP



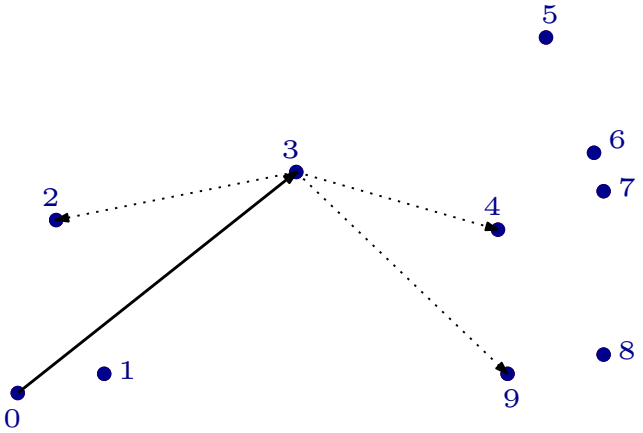
Selecionamos os $p\%$ pontos mais próximos do pontos inicial.

Exemplo TSP



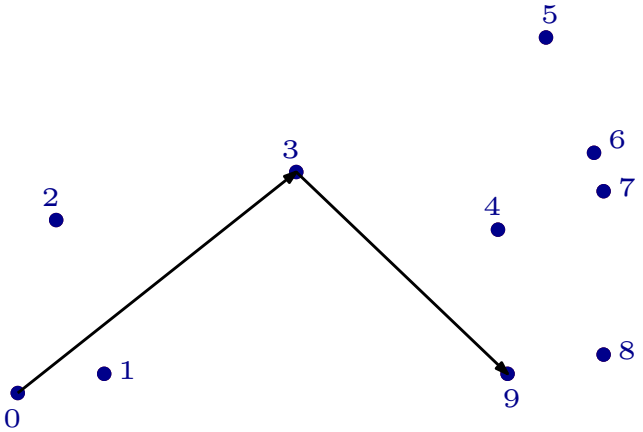
E um desses é selecionado de forma aleatória (todos com a mesma probabilidade).

Exemplo TSP



E o processo é repetido até que o último ponto seja selecionado.

Exemplo TSP



E o processo é repetido até que o último ponto seja selecionado.

Pseudocódigo Algoritmo semi-guloso

Se pensarmos que temos um algoritmo semi-guloso (semi_guloso(c)) implementado com as idéias acima, podemos executá-lo várias vezes (K), como mostrado no pseudo-código abaixo:

Algorithm 3 Algoritmo semi-guloso

$s_c = semi\_guloso(c)$
$s_b = s_c$
for $k = 1, ..., K$ **do**
    $s_c = semi\_guloso(c)$
    if $f(s_c) < f(s_b)$ **then**
        $s_b = s_c$
    end if
end for
return s_b

Conclusão

Conclusão

1. Se considerarmos o valor do parâmetro c muito baixo, o algoritmo semi-guloso se transforma no guloso original, ou seja, **o semi-guloso é uma versão mais geral do guloso**.
2. Dessa forma, sempre que for desenvolver um algoritmo guloso, na etapa de *design* do mesmo, é uma boa prática **tentar desenvolvê-lo como um semi-guloso** (ou mesmo pensar se é possível ou não).
3. Isso vai possibilitar uma otimização já em etapas inicias de desenvolvimento de códigos (na criação de soluções iniciais).

Atividade

Atividade 1

1. O que é um algoritmo guloso? Quais as suas vantagens e desvantagens? Quais as questões chave de *design* do algoritmo que devemos determinar?
2. Implemente o algoritmo guloso do vizinho mais próximo para o TSP, e use as instancias *dantzig42* e *gr17* como teste. Compare as suas soluções com os ótimos.
3. É possível melhorar a eficácia do vizinho-mais-próximo? Como? Implemente e teste novamente nas instâncias.

Atividade 2

Considerando o problema que você escolheu para tratar na disciplina.

1. Pense no design de um algoritmo guloso para o problema.
2. Busque referências na literatura (pelo menos 3) sobre a geração de soluções iniciais para o problema de otimização que você escolheu na disciplina. Escolha pelo menos um deles e o entenda.
3. Implemente um algoritmo para a geração de uma solução inicial para o seu problema em VBA (pode ser o guloso desenvolvido, algum dos artigos ou mesmo uma combinação)
4. Anote a ideia geral da solução no formato de relatório que será entregue no fim da disciplina.