SCC0220 - Laboratório de Introdução à Ciência de Computação II

Paradigmas de Projeto de Algoritmos

Prof.: Leonardo Tórtoro Pereira

leonardop@usp.br

Paradigma de Projetos de Algoritmos

- 8 paradigmas e técnicas de projeto de algoritmos
  - Indução
  - Recursividade
  - Algoritmos de tentativa e erro
  - Divisão e conquista
  - Balanceamento
  - Programação dinâmica
  - Algoritmos gulosos
  - Algoritmos aproximados

Indução

Indução

- É usada para descobrir correção e eficiência de algoritmos
- Inferir uma lei geral a partir de instâncias particulares
- É o que usamos para equações de recorrência

Recursividade

Recursividade

- Um procedimento que chama a si mesmo
  - Diretamente
  - Indiretamente
- Descrição mais clara e concisa dos algoritmos
  - Especialmente quando o problema é recursivo por natureza ou usa estruturas recursivas, como árvores
- Usa uma pilha que armazena os dados de cada chamada de um procedimento que ainda não terminou

Recursividade

- É importante lembrar que todos os caminhos possíveis do problema precisam de um critério de terminação
  - Evitar laços infinitos
- Projetos com muitas chamadas recursivas podem acabar com a memória
  - Ver “Balanceamento”

Recursividade

- Caso o programa seja facilmente convertido para uma versão iterativa, evitar usar recursão
  - “Se condição então faça algo”
  - $x = x_0$; *while* condição *do* comandos
- Programas assim tendem a ter chamadas recursivas lineares, ocupando muita memória. Como *Fibonacci*!
- A versão iterativa tem a mesma complexidade de tempo, mas com memória *O(1)*, enquanto a recursiva *O(n)*

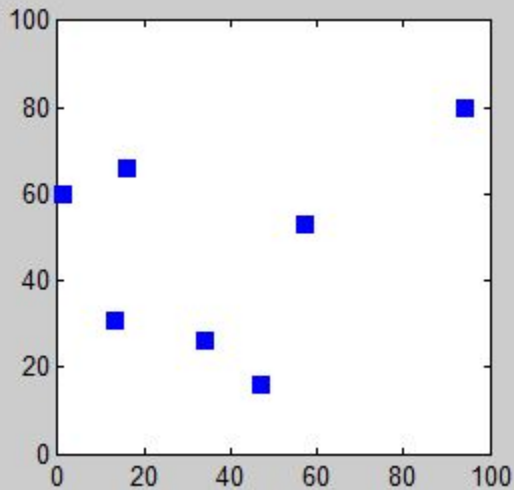
Algoritmos de Tentativa e Erro

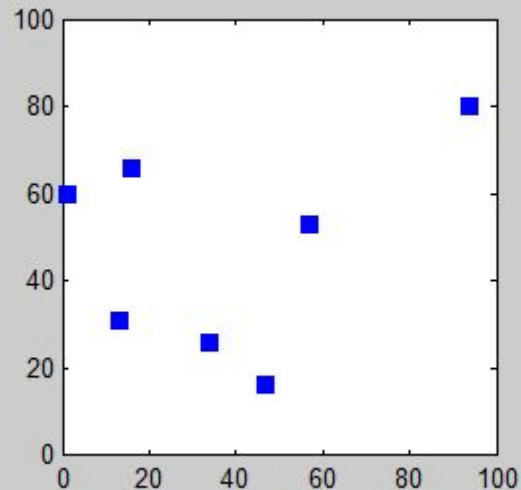
Algoritmos de Tentativa e Erro

- Uma certa particularidade de algoritmos recursivos
- Tentar todas as alternativas possíveis para resolver o problema
- Decompor o processo em número finito de sub-tarefas parciais que devem ser exploradas exaustivamente
- Uma pesquisa/tentativa que gradualmente constrói e percorre uma árvore de subtarefas

Algoritmos de Tentativa e Erro

- Não possuem uma regra fixa
  - Passos em direção à solução final são tentados e registrados
  - Se não levarem à solução, podem ser apagados do registro
- Pesquisas em árvore podem crescer rapidamente (exponencialmente às vezes)
  - Necessita de Algoritmos aproximados ou heurísticas





Fonte:
https://www.docsity.com/en/news/hacking/5-hacking-techniques-presented-interesting-gifs/

Algoritmos de Tentativa e Erro

```
int main(int argc, char const *argv[]) {
    char *password = "abc";
    char guess[6] = {'\0'};
    for(guess[0] = 33; guess[0] < 127; guess[0]++) {
        for(guess[1] = 33; guess[1] < 127; guess[1]++){
            for(guess[2] = 33; guess[2] < 127; guess[2]++){
                printf("Guess: %s\n", guess);
                if(strcmp(guess, password) == 0){
                    printf("\nSenha encontrada: %s\n", guess);
                    return 0;
                }
            }
        }
    }
    return 0;
}
```

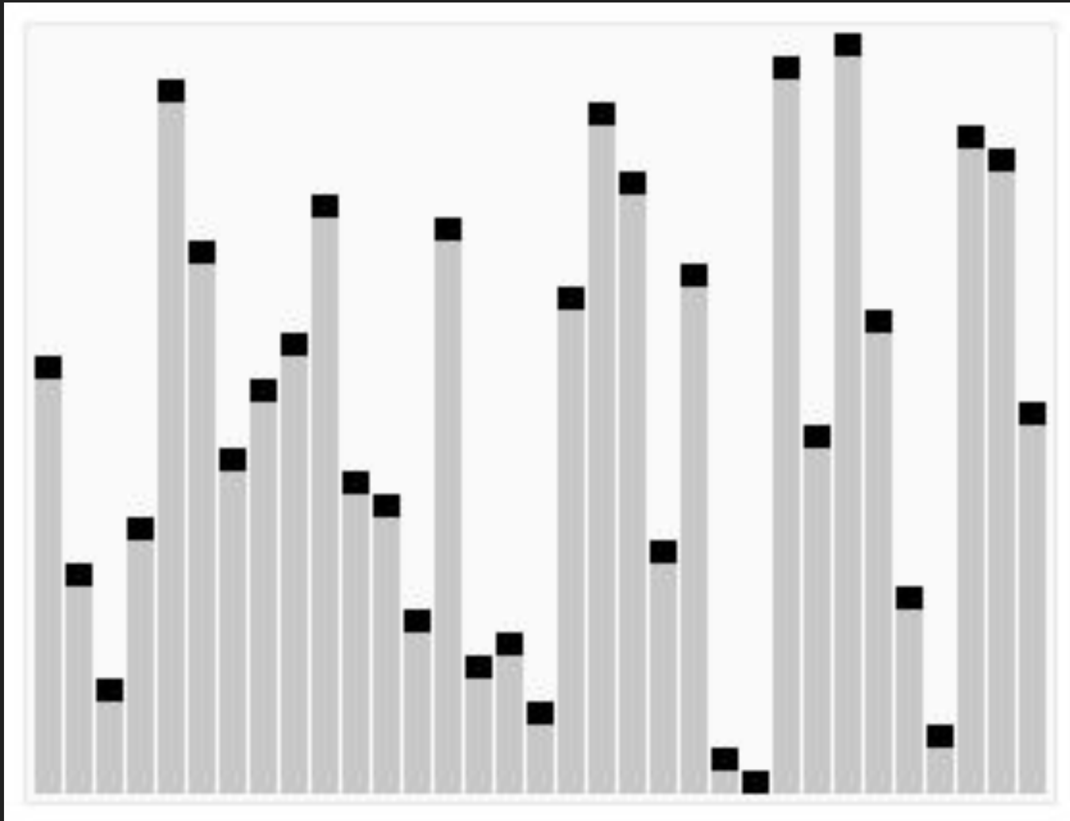
Divisão e Conquista

Divisão e Conquista

- 3 passos básicos:
  - Dividir o problema em partes menores
  - Encontrar soluções para as partes
  - Recombinar as partes em uma solução global
- Recomendado quando é possível criar subproblemas que são versões menores do problema original
  - Soluções eficientes e elegantes
  - Costuma ser usado com recursão

Divisão e Conquista

- Quick sort é um exemplo de algoritmos que usa esse paradigma
- Por normalmente usar recursividade, é usual usar equações de recorrência para provar a complexidade dos algoritmos deste paradigma
- Especialmente o Teorema Mestre



Fonte: https://pt.wikipedia.org/wiki/Quicksort

Balanceamento

Balanceamento

- Durante a divisão e conquista é desejável manter o balanceamento na subdivisão em partes menores
  - Costuma gerar algoritmos mais eficientes e estáveis
  - Evita o “pior caso” do Quicksort existir, por exemplo
- Não dividir o problema em 2, simplesmente, mas buscar dividi-lo em dois subproblemas de tamanhos aproximadamente iguais
- Exemplo: Merge Sort

Fonte: https://imgur.com/gallery/HU2tfzo

Programação Dinâmica

Programação Dinâmica

- Quando a recursividade consegue dividir bem os subproblemas, geralmente em problemas que a **soma** dos subproblemas é *O(n)*, ela tende a ser uma boa solução
- Porém, em casos como, por exemplo, ela divide em *n* subproblemas de tamanho *O(n-1)*, o algoritmo provavelmente será exponencial

Programação Dinâmica

- Nesses casos a Programação Dinâmica pode ajudar!
  - É uma técnica de programação e não um paradigma em si
- Calcula a solução para todos os subproblemas partindo dos menores subproblemas para os maiores
- Armazena os resultados em uma tabela
- Não precisa resolver mais o subproblema pois está salvo!

Programação Dinâmica

- Pode ser usado com sucesso quando o **princípio de otimalidade** se aplica e há **sobreposição no espaço da solução**

Programação Dinâmica

- **Princípio da otimalidade**
  - Em uma sequência de escolhas ou decisões, cada subsequência também deverá ser ótima

- **Sobreposição no espaço da solução**
  - Ocorre quando a solução para subproblemas menores é usada em subproblemas maiores

Fibonacci - https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/

```
int fib(int n) {
  /* Declare an array to store Fibonacci numbers. */
  int f[n+2];   // 1 extra to handle case, n = 0
  int i;
  /* 0th and 1st number of the series are 0 and 1*/
  f[0] = 0;
  f[1] = 1;
  for (i = 2; i <= n; i++)  {
      /* Add the previous 2 numbers in the series
         and store it */
      f[i] = f[i-1] + f[i-2];
  }
  return f[n];
}
```

Programação Dinâmica

- ➔ Outro exemplo simples é o troco em moedas (lembram de Lab de ICC1?)
- ➔ Vamos ver como fazer usando Programação Dinâmica :)

Algoritmos Gulosos

Fonte: https://www.gratispng.com/png-psbpya/

Algoritmos Gulosos

- Usados normalmente em problemas de otimização
- Ex: caminho mais curto entre dois vértices de um grafo
- Escolhe sempre a melhor escolha naquele momento
    - E nunca volta atrás!
    - Não importa quão ruim o resultado!

Algoritmos Gulosos - Quando usar?

- Considere um problema em que a solução ótima deve ser obtida
- Existe uma lista ou conjunto de candidatos para construir a solução
  - Ex: arestas de grafo que constroem um caminho

Algoritmos Gulosos - Quando usar?

- Conforme o algoritmo procede, dois outros conjuntos são acumulados
  - Candidatos analisados e escolhidos
  - Candidatos analisados e descartados

Algoritmos Gulosos - Quando usar?

- Existe uma função que verifica se um conjunto particular de candidatas produz uma solução para o problema
  - Mas sem considerar se é ótima!
  - Ex: as arestas formam um caminho **válido**?

Algoritmos Gulosos - Quando usar?

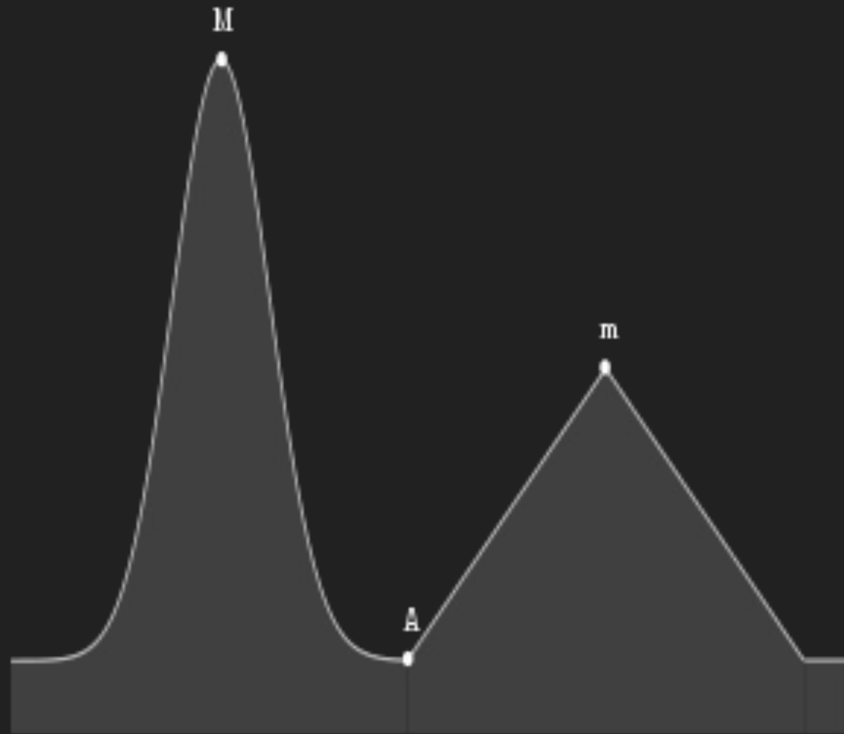
- Outra função verifica se um conjunto de candidatos é **viável**
  - É possível completar o conjunto adicionando mais candidatos? Vai ser encontrada uma solução?
  - Mas sem considerar se é ótima!

Algoritmos Gulosos - Quando usar?

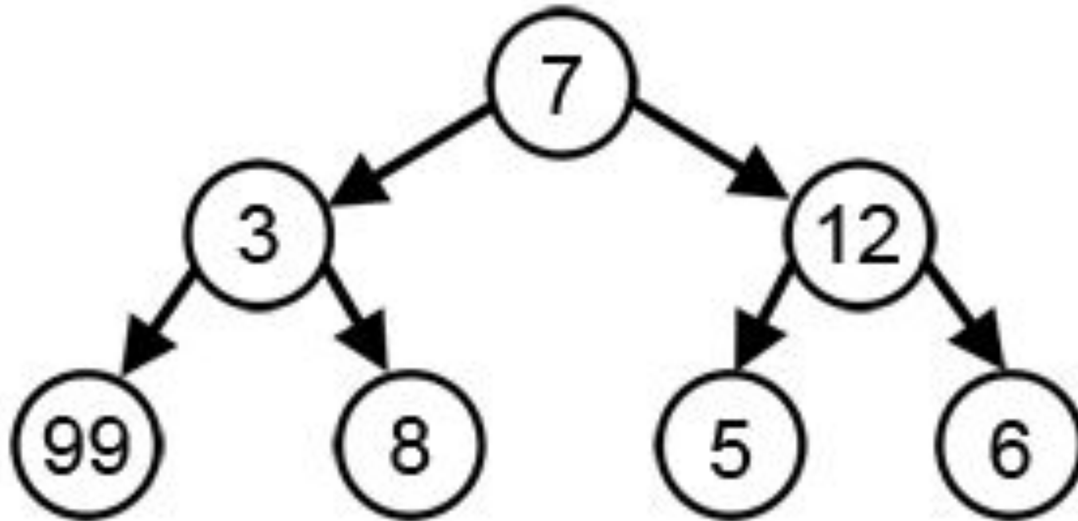
- A função de seleção indica quais dos candidatos restantes (não escolhidos nem rejeitados) são os mais promissores
- A função objetivo fornece o valor da solução
  - Ex: comprimento do caminho
  - Não aparece explicitamente no algoritmo guloso

Algoritmos Gulosos

- Vantagens:
  - São de implementação simples
  - Execução rápida
  - Podem fornecer a melhor solução. Mas só idealmente
- Desvantagens:
  - Nem sempre leva ao ótimo global
  - Pode entrar em loop ou achar caminhos infinitos
- São boas aproximações para problemas complexos!



Fonte: https://en.wikipedia.org/wiki/Greedy_algorithm



Fonte: https://en.wikipedia.org/wiki/Greedy_algorithm

Algoritmos Gulosos

- Exemplo do troco!
  - Discussão interessante sobre casos que a busca gulosa pode falhar :)
  - https://stackoverflow.com/questions/13557979/why-does-the-greedy-coin-change-algorithm-not-work-for-some-coin-sets

Algoritmos Aproximados e Heurísticas

Algoritmos Aproximados e Heurísticas

- Problemas resolvidos por algoritmos polinomiais são "fáceis"
- Problemas resolvidos por algoritmos exponenciais são "difíceis"
- Porém esses problemas difíceis ou intratáveis são bem comuns
- Ex: caixeiro viajante, como completar uma fase do Mario

Algoritmos Aproximados e Heurísticas

- Nesses casos, é comum desistir da ideia de encontrar uma solução ótima
- É preferível um algoritmo rápido o bastante para encontrar uma solução boa o suficiente
- Para isso usamos algoritmos aproximados ou usamos heurísticas

Algoritmos Aproximados e Heurísticas

- Algoritmos Aproximados
  - Algoritmo que gera soluções aproximadas dentro de um limite aceitável entre o ótimo e a solução produzida
  - A qualidade do resultado é monitorada

Algoritmos Aproximados e Heurísticas

- Heurística
  - Algoritmo que pode produzir um bom resultado. Ou até mesmo a solução ótima
  - Mas ao mesmo tempo pode produzir uma solução ruim... ou não encontrar nada
  - É um chute elegante
  - No geral, funciona bem se bem implementado :)

Algoritmos Aproximados e Heurísticas

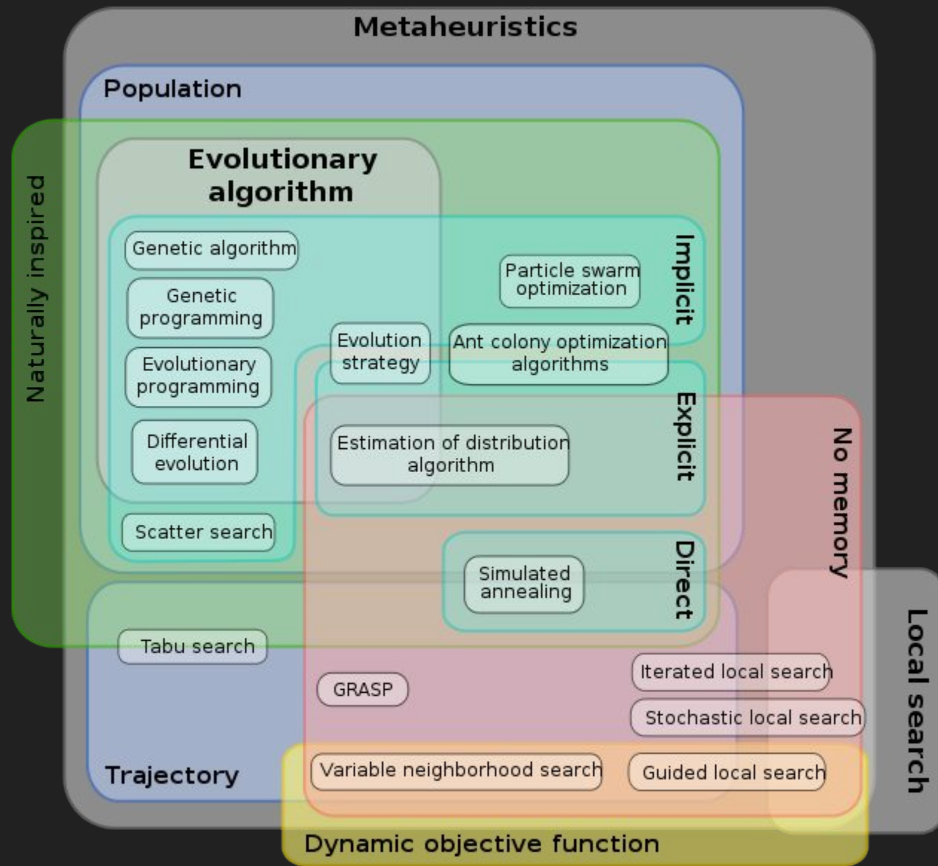
- Heurística
  - A* é um dos melhores algoritmos de busca por caminhos e super usado em jogos e robótica e é uma heurística
  - https://www.youtube.com/watch?v=JAZNPe5OCcU

Algoritmos Aproximados e Heurísticas

- Metaheurística
  - Um procedimento ou heurística de alto nível feito para encontrar, gerar ou selecionar uma heurística que pode providenciar uma solução boa o suficiente para um problema de otimização
  - Muito útil quando não é possível saber tudo sobre o problema ou ele é grande demais para sequer cogitar sua exploração

Algoritmos Aproximados e Heurísticas

- Metaheurística
  - Baseada em testes empíricos
  - Algumas análises matemáticas sobre convergência



Fonte: https://en.wikipedia.org/wiki/Metaheuristic

Referências

- [1] ZIVIANI, N. Projeto de Algoritmos. 3º edição revisada e ampliada. Cengage Learning, 2017.
- https://en.wikipedia.org/wiki/Greedy_algorithm