DCC207 – Algoritmos 2

Aula 11 – Soluções exatas para problemas difíceis

Professor Renato Vimieiro

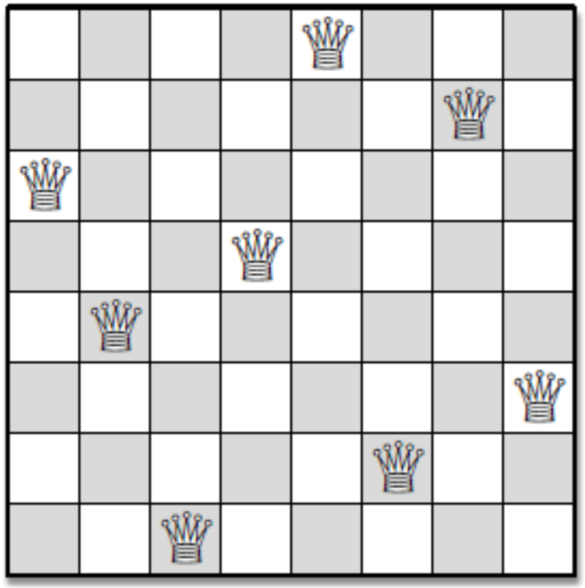
DCC/ICEx/UFMG

Introdução

- As últimas aulas mostraram que diversos problemas, ainda que computáveis, são bastante complexos para serem resolvidos em tempo razoável
- Embora os problemas NP-difíceis e/ou PSPACE-difíceis gerem um certo ‘desânimo’ por, supostamente, não apresentarem soluções eficientes, esses ainda podem ser atacados de maneira inteligente, reduzindo a complexidade das soluções e permitindo resolver instâncias maiores
- Nessa aula, veremos uma técnica que permite desenhar soluções exatas para problemas difíceis que são mais eficientes que soluções ingênuas (força-bruta) para eles
- Essa técnica é conhecida como ***backtracking*** e, como veremos, ela serve de base para vários algoritmos

O problema das 8 rainhas

- O problema das 8 rainhas foi apresentado por um alemão entusiasta de xadrez, Max Bezzel, em 1848
- O problema consiste em posicionar 8 rainhas em um tabuleiro de xadrez de forma que nenhuma rainha possa atacar outra
  - As rainhas em xadrez podem se movimentar horizontal, vertical, e diagonalmente para qualquer casa do tabuleiro
- O problema parecia tão desafiador que chamou a atenção de matemáticos importantes como Gauss
  - A solução na figura ao lado foi proposta por ele em 1850



O problema das 8 rainhas

- Qual seria uma solução trivial para o problema?
  - Podemos, de forma exaustiva, avaliar cada um dos posicionamentos das 8 rainhas nas 64 casas do tabuleiro
  - Essa solução, claramente ineficiente, requer a avaliação de $\binom{64}{8}$ ~ 4G posicionamentos
- Gauss sugeriu uma representação do problema através de vetores
  - Cada posição do vetor corresponde a uma linha do tabuleiro, e seu valor a coluna em que uma rainha foi posicionada
  - A solução da figura anterior é: [5, 7, 1, 4, 2, 8, 6, 3]
  - Essa representação permite uma implementação ingênua com loops aninhados com um custo 8^8 ~ 16M
- Esse ganho é fruto de restringirmos posicionamentos somente a linhas distintas
- Podemos fazer o mesmo com as colunas! Isto é, restringimos as soluções a não conterem colunas iguais.
  - O vetor, nesse caso, só poderá conter números distintos de 1 a 8
  - Ou seja, a solução do problema pode ser obtida computando as 8! ~ 40K soluções candidatas

O problema das 8 rainhas

```
Procedure permutation(T,n,r)
if r == n then
    valid = true;
    for i = 0 to n − 2 do
        for j = i + 1 to n − 1 do
            if |T[j] − T[i]| = |j − i| then
                valid = false;
                break
            end
        end
    end
    if valid then imprimir T;
end
for i = 0 to n − 1 do
    if i ∉ T then
        T[r] = i;
        permutation(T,n,r+1);
        T[r] = −1
    end
end
```

O problema das 8 rainhas

- O algoritmo anterior ainda pode ser melhorado
- Os candidatos só são avaliados ao posicionar todas as rainhas
- Contudo, se a tentativa de colocar a k-ésima rainha gera conflito com alguma das k-1 anteriores, então sabemos que essa permutação não gerará uma solução válida
- Podemos então abortar a exploração de uma permutação tão logo detectarmos a geração de uma solução inválida
    - Exemplo: a solução parcial [1, 4, 2, 5, 8] pode ser abortada, pois a colocação da 6ª rainha em qualquer posição resulta em conflito
    - Isso implica que o teste pertinência à mesma diagonal deve ser realizado à cada escolha de uma posição para uma nova rainha

O problema das 8 rainhas

```
Procedure permutation(T,n,r)
if r == n then imprimir T ;
else
    for i = 0 to n − 1 do
        valid = true;
        for j = 0 to r − 1 do
            if T[j] == i or |T[j] − i| == |j − r| then
                valid = false;
                break
            end
        end
        if valid then
            T[r] = i;
            permutation(T,n,r+1);
        end
    end
end
```

O problema das 8 rainhas

- Embora seja difícil encontrar uma fórmula fechada para o número de avaliações do algoritmo anterior, é perceptível que há um ganho em relação à versão inicial
- Podemos perceber que a ideia do algoritmo é explorar um grafo implícito formado pelas soluções parciais, cujas arestas representam as transições de uma solução parcial para outra
  - Esse grafo, na verdade, é uma árvore em que as folhas são soluções válidas, ou soluções inválidas que não podem ser refinadas
  - Sempre que o algoritmo encontra uma solução inválida, ele retrocede (backtrack) a um nó anterior e continua a busca dali.
  - Nesse sentido, o algoritmo consiste em uma busca em profundidade no grafo implícito dos espaço de busca do problema

Algoritmo geral para backtracking

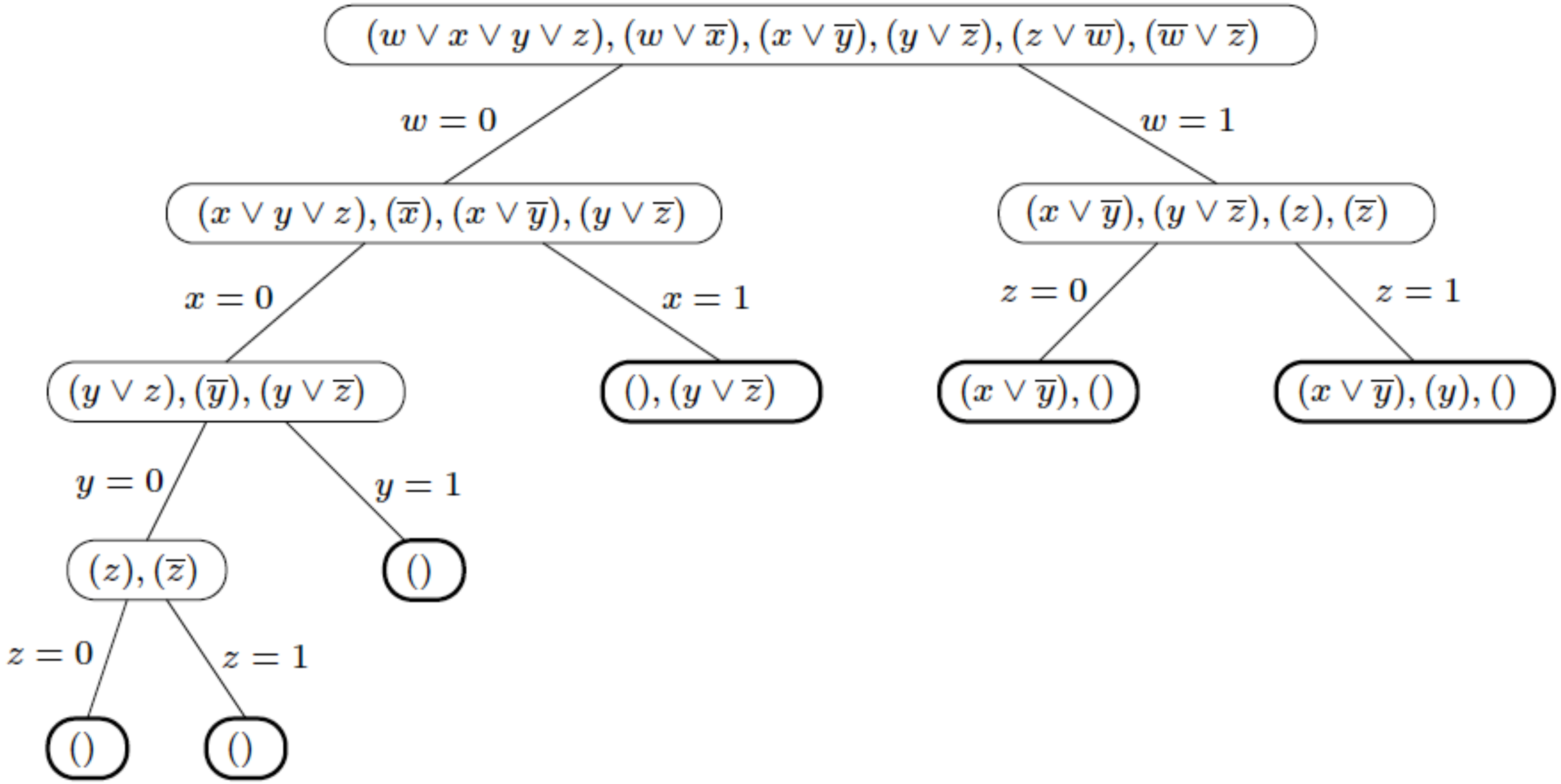
- A ideia geral do algoritmo é explorar sistematicamente os vértices da árvore de busca
- A solução é construída de forma incremental
- A cada iteração (passo recursivo), um novo elemento de um conjunto de candidatos é escolhido para compor a seleção
- O processo segue até que uma folha da árvore seja encontrada
  - Nesse momento, verifica-se se ela é uma solução ou não

Backtrack-DFS(A, k)
  if $A = (a_1, a_2, ..., a_k)$ is a solution, report it.
  else
    $k = k + 1$
    compute S_k
    while $S_k \neq \emptyset$ do
      a_k = an element in S_k
      $S_k = S_k - a_k$
      Backtrack-DFS(A, k)

Backtracking para SAT

- Podemos desenhar uma solução para o SAT baseada na estratégia de backtracking
- Considere uma função $\varphi(w,x,y,z) = (w \vee x \vee y \vee z) \wedge (w \vee \neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg w) \wedge (\neg w \vee \neg z)$
- Da mesma forma que com o problema das 8 rainhas, podemos construir soluções parciais, dessa vez, atribuindo valores às variáveis
- Uma atribuição de valor a uma variável simplifica a função
  - Exemplo: w=falso -> $(x \vee y \vee z) \wedge (\neg x) \wedge (x \vee \neg y) \wedge (y \vee \neg z)$
  - W é excluído das cláusulas, pois elas não dependem mais dele
  - Cláusulas em que ¬w podem ser excluídas, pois já foram satisfeitas
- Note que funções sem cláusulas são sempre satisfeitas, mas cláusulas sem variáveis nunca são
  - Atribuição de x=verdadeiro torna a função falsa, já que teríamos como resultado $() \wedge (y \vee \neg z)$, e a primeira cláusula não pode ser satisfeita

Backtracking para SAT



Backtracking para Subset Sum

- Tendo mostrado uma solução para SAT, sabemos que temos soluções baseadas em backtracking para qualquer problema NP
- Contudo, muitas vezes é mais fácil desenhar soluções específicas para um problema que implementar as reduções vistas anteriormente
- Assim, veremos como desenhar um algoritmo específico para o problema Subset Sum
- A solução trivial seria gerar todos os 2^n subconjuntos e testar aqueles que são soluções válidas
- Alternativamente, podemos empregar o princípio usado nas soluções dos problemas anteriores e interromper a expansão de uma solução parcial tão logo ela se mostre infrutífera

Backtracking para Subset Sum

- As soluções parciais nesse caso são subconjuntos de X
- Considerando que X tenha somente inteiros positivos, existem dois casos triviais:
  - Se a soma requerida T=0, então podemos retornar imediatamente a resposta 'sim'
  - Em contrapartida, se $T < 0$ ou $X=\emptyset$ e $T\neq 0$, então podemos retornar imediatamente a resposta 'não'
- Cada elemento $x \in X$ pode ou não fazer parte da solução. Assim, devemos considerar dois refinamentos para uma solução parcial Y
  - $x \in Y$: nesse caso, existe solução para o problema somente se o subproblema S-{x} admite solução T-x
  - $x \notin Y$: nesse caso, o problema admite solução sse existe um subconjunto de S-{x} que soma a T

Backtracking para Subset Sum

- O algoritmo continua tendo complexidade exponencial no pior caso (T > sum(X))
  - Nesse caso, a árvore de busca é uma árvore binária completa
- Porém, em alguns casos não-triviais, ele certamente é mais eficiente que a implementação ingênua
  - Exemplo, T < x, para x ∈ X
  - Nesse caso, ele retorna a resposta correta em tempo O(n)

```
SubsetSum(X, T):
  if T = 0
      return True
  else if T < 0 or X = ∅
      return False
  else
      x ← any element of X
      with ← SubsetSum(X \ {x}, T − x)
      wout ← SubsetSum(X \ {x}, T)
      return (with ∨ wout)
```

Backtracking para Subset Sum

- Exemplo: $X = \{8, 6, 7, 5, 3, 10, 9\}$, $T = 15$

Backtracking para k-coloração de grafos

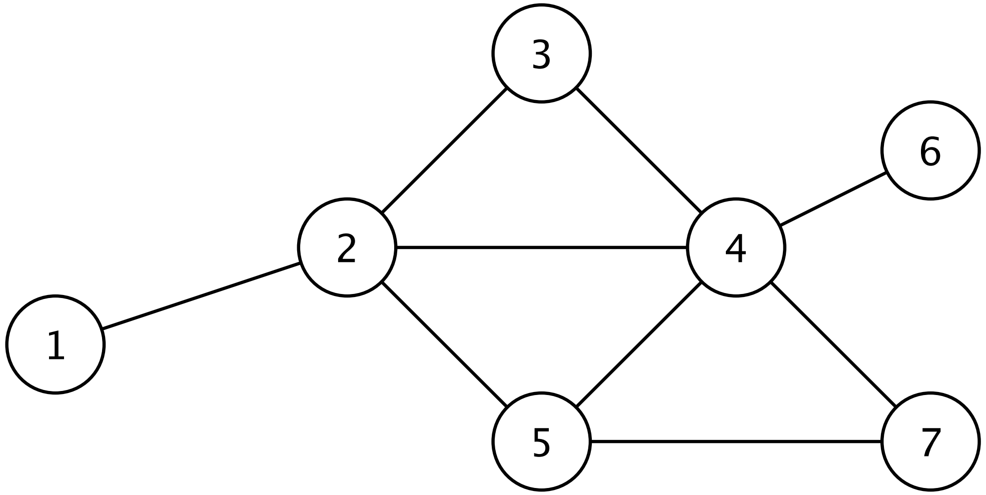
- Vamos considerar agora o problema de coloração de grafos
- Relembrando: o problema de k-coloração de grafos consiste em atribuir cores aos vértices de tal forma que nenhum par de vértices adjacentes possua a mesma cor
- A solução trivial é gerar as k^n colorações possíveis e verificar sua validade
- Novamente, essa solução pode ser melhorada abandonando ramos não promissores tão logo eles sejam detectados
- As soluções parciais são construídas, atribuindo-se uma cor ao i-ésimo vértice
  - Verifica-se se essa atribuição não viola nenhuma restrição (compara-se com a cor dos i-1 anteriores que são adjacentes a i), e, caso não ocorra violação, segue-se expandindo o espaço de busca
  - Caso não seja possível atribuir nenhuma das k cores ao vértice i, retrocede-se ao anterior e testa-se uma nova coloração

Backtracking para k-coloração de grafos

```
Procedure coloring(A,colors,v,n,m)
if v ≥ n then imprimir colors ;
else
|   for c = 0 to m − 1 do
|   |   valid = true;
|   |   for u = 0 to v − 1 do
|   |   |   if colors[u] == c and A[v][u] == 1 then
|   |   |   |   valid = false;
|   |   |   |   break
|   |   |   end
|   |   end
|   |   if valid then
|   |   |   colors[v] = c;
|   |   |   coloring(A,colors,v+1,n,m);
|   |   end
|   end
end
```

Backtracking para k-coloração de grafos

- Exemplo (k=4):



Backtracking para Circuito Hamiltoniano

- A solução ingênua do circuito hamiltoniano consiste em avaliar todas as permutações dos n-1 vértices não iniciais
- De forma análoga ao problema das 8 rainhas, novamente devemos avaliar permutações e decidir se continuamos ou não expandindo soluções parciais
- Dessa forma, nossa solução para o problema das 8 rainhas pode ser reaproveitada aqui
- Devemos alterar a verificação de viabilidade de soluções
  - Deve haver aresta entre os vértices i e i-1
  - Deve haver aresta entre n-1 e 0

Backtracking para Circuito Hamiltoniano

Procedure hamiltonian-circuit(A,P,n,i)

```
if i == n then imprimir P ;
else
    for v = 1 to n − 1 do
        valid = true;
        if A[v][P[i − 1]] == 0 or (i == n − 1 and A[v][P[0]] == 0)
         then valid = false;
        else
            for j = 0 to i − 1 do
                if P[j] == v then
                    valid = false;
                    break
                end
            end
            if valid then
                P[i] = v;
                hamiltonian-circuit(A,P,n,i+1);
            end
        end
    end
end
```

Leitura

- Seção 7.1 Skiena (The Algorithm Design Manual)
- Seções 2.1 a 2.4 Jeff Erickson (Algorithms http://jeffe.cs.illinois.edu/teaching/algorithms/)
- Capítulo 5 Neapolitan (Foundations of Algorithms)