DCC207 – Algoritmos 2

Aula 12 – Soluções exatas para problemas difíceis (branch-and-bound)

Professor Renato Vimieiro

DCC/ICEx/UFMG

Introdução

- Vimos que o paradigma de desenho de algoritmos backtracking permite resolver instâncias ‘pequenas’ de problemas difíceis
- A ideia do paradigma é explorar o espaço de busca de forma sistemática
- Os algoritmos exploram a árvore de soluções candidatas, expandindo soluções parciais, e podando soluções inviáveis tão cedo elas sejam detectadas
- Os nós representando soluções parciais viáveis são chamados de promissores, e os com soluções inviáveis não-promissores
- O critério usado para definir se um nó era promissor ou não foi simplesmente avaliar se ele feria ou não uma restrição do problema (e.g. gerava uma soma com valor acima de T no Subset Sum)
- Podemos estender essa ideia para problemas de otimização, eliminando ramos da árvore de busca que representam soluções piores que a de outros ramos já explorados
- Essa técnica de desenho de algoritmos é conhecida como branch-and-bound

Branch and bound

- De uma forma geral, a técnica de desenho branch-and-bound é uma modificação de backtracking que passa a incluir dois outros itens:
  - Capacidade de calcular, para cada nó da árvore de busca, um **limiar (bound)** para o valor da **função objetivo** de todas as soluções que puderem ser obtidas a partir desse nó
    - Esse limiar é uma estimativa do valor máximo/mínimo das soluções obtidas a partir do nó, e deve ser estabelecida com base na natureza do problema (maximização/minimização)
  - Armazenar o valor da melhor solução vista até o momento
    - Opcionalmente, pode-se também armazenar a própria solução

Problema de designação

- Vamos usar o problema de designação (matching de peso mínimo) como exemplo de construção de um algoritmo branch-and-bound
  - O problema possui solução polinomial. Essa solução é simplesmente didática!
- O problema consiste em designar n pessoas a n funções com o menor custo possível
  - Exemplo:

$$C = \begin{array}{c} \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$

Levitin, Algorithms Design

Problema de designação

- Similar ao problemas da n-rainhas, podemos representar as soluções como vetores em que cada célula contém a função designada à pessoa
  - Ou seja, devemos buscar uma permutação das colunas com o menor valor associado
- O problema agora é como detectar soluções não-promissoras
  - Isto é, como definir uma estimativa mínima para as soluções construídas a partir de um nó na árvore?
- Note que queremos uma estimativa realista, mas não queremos gastar recursos computacionais com essa estimativa (a melhor estimativa seria computar o valor da melhor solução obtenível a partir do nó)
  - Em outras palavras, qualquer estimativa é válida, mas ela tem que ser computada rapidamente
- Dessa forma, podemos usar a soma dos menores custos das pessoas ainda não designadas (soma das menores colunas de cada linha) como estimativa
  - Exemplo: considerando a matriz de custo anterior, se alocarmos a pessoa ***a*** à função ***1***, o menor custo de uma solução obtida a partir desse nó é: 17 = 9+3+1+4
  - Se o menor custo obtenível a partir de um nó for maior que a melhor solução atual, então esse nó não é promissor
  - Veja que a estimativa pode não representar uma solução viável; as pessoas ***b*** e ***c*** foram ambas designadas à função 3.

Problema de designação

- Outra questão importante que surge nessa abordagem é a ordem com que devemos explorar os nós da árvore
- Ao contrário do que acontecia com backtracking, os filhos de um nó possuem diferentes prioridades
  - No nosso caso, poderíamos priorizar a expansão do filho com a menor estimativa, o qual seria o ‘mais’ promissor dentre eles
- Sendo assim, ao invés de explorar os filhos usando uma ordem arbitrária, vamos explorá-los conforme suas prioridades
  - Logo, devemos verificar todos os filhos de um nó, antes de decidir qual será expandido
  - Essa busca é conhecida como ***best-first search***; ela se diferencia da BFS por usar uma fila de prioridades para armazenar os nós que serão visitados

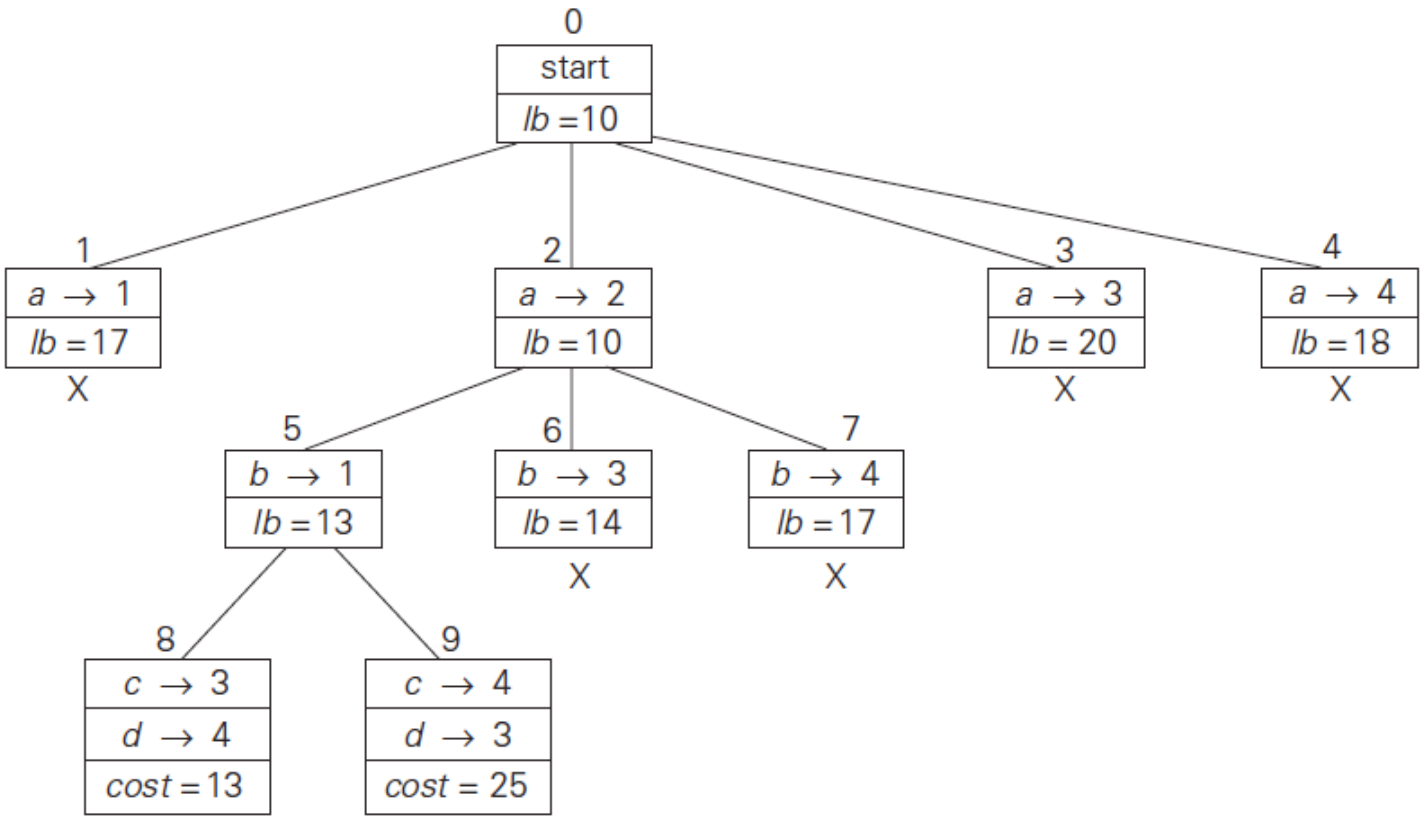
Problema de designação

Procedure bnb-assignment(C,n)

```
root = (0, bound(C, 0, []), ∞, []);
queue = heap([root]);
while queue ≠ ∅ do
    node = queue.pop();
    if node.level == n − 1 then
        if best > node.cost then best = node.cost; sol = node.s;
    end
    else if node.bound < best then
        for k = 0 to n − 1 do
            if k ∉ node.s and bound(C, node.level + 1, node.s ∪ {k})
             then
                newsol = node.s ∪ {k};
                child = (node.level + 1, bound(C, node.level +
                  1, newsol), node.cost + C[node.level + 1][k], newsol));
                queue.push(child)
            end
        end
    end
end
```

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

Problema de designação



Problema da mochila binário

- Vamos agora ver como construir um algoritmo branch-and-bound para o problema da mochila binário
- Relembrando: dados n itens, cada um com um peso e valor associado, e uma mochila com capacidade W quilos, deseja-se maximizar o lucro total, respeitando a capacidade da mochila.
- Para resolver o problema, vamos considerar os itens ordenados pelo valor relativo (valor/peso)

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

W=10

Problema da mochila binário

- Como vimos anteriormente, podemos construir uma solução com backtracking para o problema
  - Cada nó gera duas novas soluções parciais: incluindo; e excluindo um item
- Podemos, então, partir dessa solução e acrescentar podas baseadas nas estimativas de lucro
  - Por se tratar de um problema de maximização, devemos estimar o lucro máximo possível a ser obtido a partir do nó
- Novamente, qualquer estimativa simples de ser computada pode ser usada na construção do algoritmo
- Vamos considerar a seguinte:
  - O i-ésimo nível da árvore contém nós com subconjuntos de elementos até i
  - Se o lucro do nó atual é v e o peso total w, então o lucro máximo obtenível a partir desse nó é W-w frações do i+1-ésimo item; isto é W-w * (v_{i+1}/w_{i+1})
  - A estimativa do lucro na raiz é 100

Problema da mochila binário

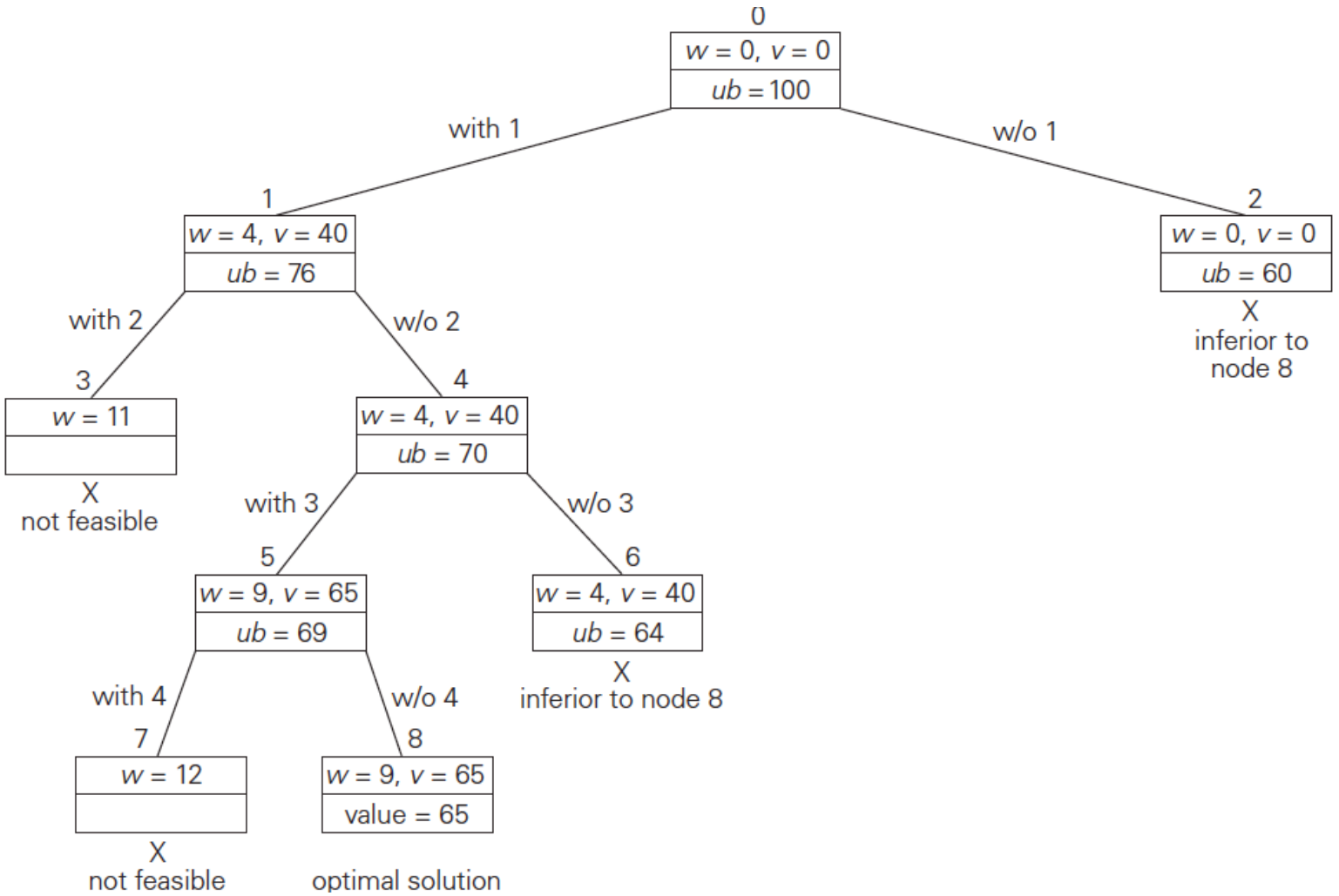
Procedure bnb-knapsack(items,W)

```
root = (0, 0, 0, W * items[0][2], []);
queue = heap([root]);
best = 0;
while queue ≠ ∅ do
    node = queue.pop();
    if node.level == n − 1 then
        if best < node.value then best = node.value; sol = node.s;
    end
    else if node.bound > best then
        with = node.value + items[node.level][1] + (W − w −
          items[node.level][0]) * items[level + 1][2];
        wout = node.value + (W − w) * items[level + 1][2];
        if node.weight + items[node.level + 1] < W and with > best
          then
            queue.push((node.level + 1, node.value +
              items[node.level][1], w + items[node.level][0], with, s ∪
              {node.level})
        end
        if wout > best then
            queue.push((node.level + 1, node.value, w, wout, s)
        end
    end
end
```

item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

W=10

Problema da mochila binário

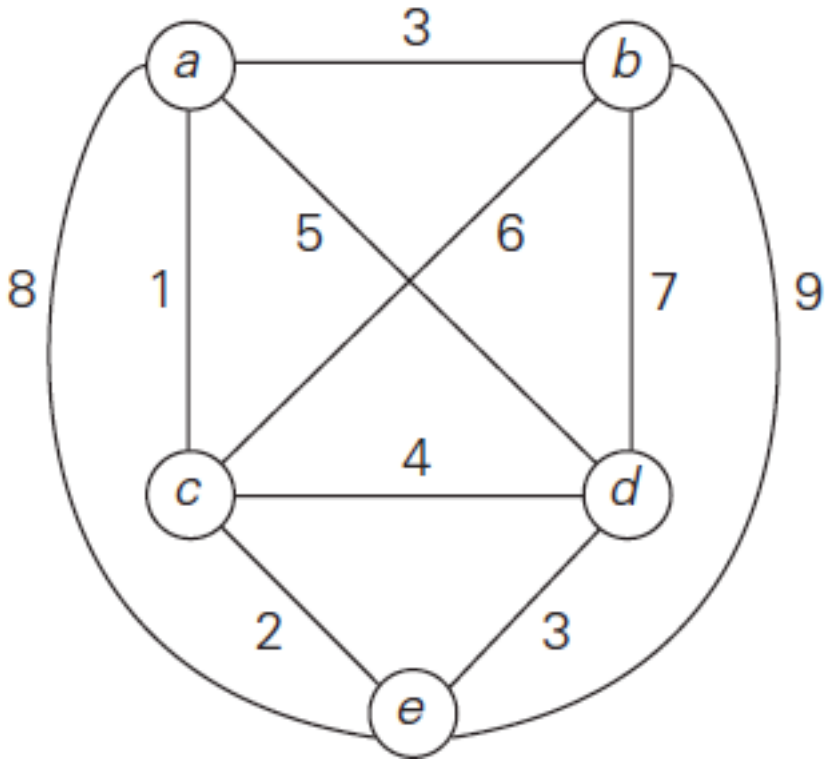


Problema do Caixeiro Viajante

- O problema do caixeiro viajante é outro candidato a uma solução com branch-and-bound
- A adaptação é direta do algoritmo backtracking para computação do circuito hamiltoniano
- Devemos incluir, agora, uma forma de podar ramos que levem a caminhos mais longos/custosos que os já vistos
- Como devemos sempre chegar e sair de um vértice, podemos usar como estimativa as somas das duas arestas de menor peso incidentes em cada vértice
  - Como cada aresta seria contada duas vezes, dividimos esse valor por 2

Problema do Caixeiro Viajante

- Por exemplo, sem incluir aresta alguma no caminho, temos uma estimativa de:
  - 14 = teto([(1+3)+(3+6)+(1+2)+(4+3)+(2+3)]/2)
- O mesmo pode ser feito caso arestas já tenham sido incluídas no caminho
  - Considerando a aresta (a,d) como pertencente
  - 16 = teto([(1+5)+(3+6)+(1+2)+(3+5)+(2+3)]/2)

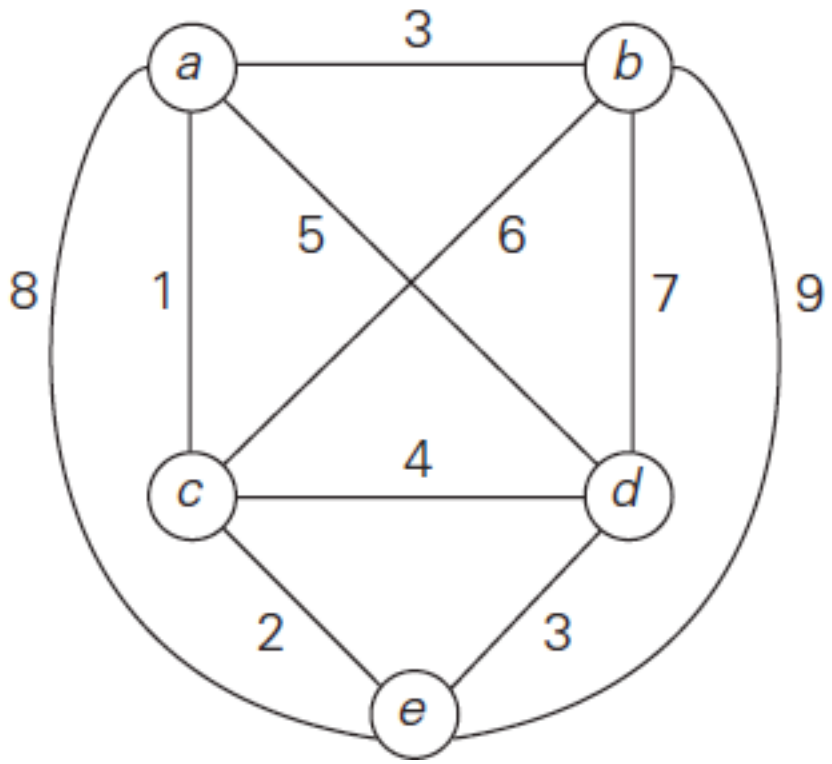


Problema do Caixeiro Viajante

Procedure bnb-tsp(A,n)

```
root = (bound([0]), 0, 0, [0]);
queue = heap([root]);
best = ∞;
sol = ∅;
while queue ≠ ∅ do
    node = queue.pop();
    if node.level > n then
        if best > node.cost then best = node.cost; sol = node.s;
    end
    else if node.bound < best then
        if node.level < n then
            for k = 1 to n − 1 do
                if k ∉ node.s and A[node.s[−1]][k] ≠ ∞ and
                  bound(node.s ∪ {k}) < best then
                    queue.push((bound(node.s ∪ {k}, node.level +
                      1, node.cost + A[node.s[−1]][k], node.s ∪ {k}))
                end
            end
        end
        else if A[node.s[−1]][0] ≠ ∞ and bound(node.s ∪ {0}) < best
          and ∀i ∈ node.s then
            queue.push((bound(node.s ∪ {0}, node.level + 1, node.cost +
              A[node.s[−1]][0], node.s ∪ {0}))
        end
    end
end
```

Problema do Caixeiro Viajante



Leitura

- Seção 12.2 (Levitin)