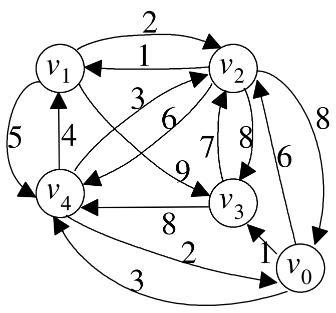
Data Structures (CS 1520) Lecture 28 Name:_________________

1. *Traveling Salesperson Problem* (TSP) -- Find an optimal (i.e., minimum length) tour when at least one tour exists. A *tour* (or *Hamiltonian circuit*) is a path from a vertex back to itself that passes through each of the other vertices exactly once. (Since a tour visits every vertice, it does not matter where you start, so we will generally start at v_0.)



What are the length of the following tour?

a) $[v_0, v_3, v_4, v_1, v_2, v_0]$

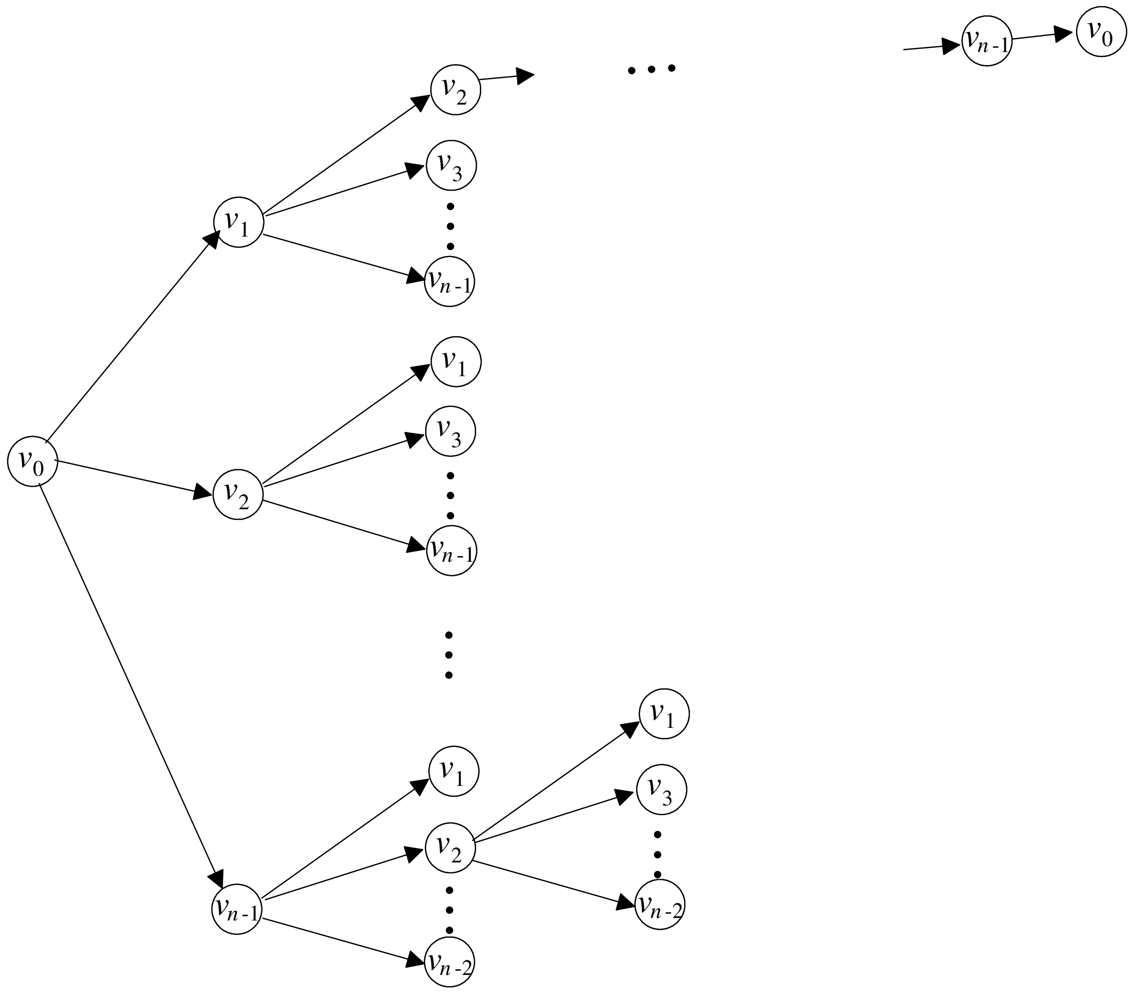
1+8+4+2+8 = 23

b) List another tour starting at v_0 and its length.

$[v_0, v_2, v_1, v_3, v_4, v_0]$

6+1+9+8+2 = 26 (others possible too)

c) For a graph with "n" vertices ($v_0, v_1, v_2, \ldots, v_{n-1}$), one possible approach to solving TSP would be to brute-force generate all possible tours to find the minimum length tour. "Complete" the following decision tree to determine the number of possible tours.



(n-1) nodes at this level

(n-1)*(n-2) nodes at this level

(n-1)*(n-2)*(n-3) nodes at this level

(n-1)! branches in whole brute-force decision tree O((n-1)!) is very bad!!!

Unfortunately, TSP is an "*NP-hard*" problem, i.e., no known polynomial-time algorithm.

2. **Handling "Hard" Problems**: For many optimization problems (e.g., TSP, knapsack, job-scheduling), the best known algorithms have run-time's that grow exponentially ($O(2^n)$ or worse). Thus, you could wait centuries for the solution of all but the smallest problems!

Ways to handle these "hard" problems:

- Find the best (or a good) solution "quickly" to avoid considering the vast majority of the 2^n worse solutions, e.g, Backtracking (section 4.6) and Best-first-search-branch-and-bound
- See if a restricted version of the problem meets your needed that might have a tractable (polynomial, e.g., $O(n^3)$) solution. e.g., TSP problem satisfying the triangle inequality, Fractional Knapsack problem
- Use an approximation algorithm to find a good, but not necessarily optimal solution

Backtracking general idea: (Recall the coin-change problem from lectures 10 and 13)

- Search the "*state-space tree*" using depth-first search to find a suboptimal solution quickly
- Use the best solution found so far to prune partial solutions that are not "promising,", i.e., cannot lead to a better solution than one already found.

The goal is to prune enough of the state-space tree (exponential is size) that the optimal solution can be found in a reasonable amount of time. However, in the worst case, the algorithm is still exponential.

My simple backtracking solution for the coin-change problem **without pruning**:

```
def recMC(change, coinValueList):
    global backtrackingNodes
    backtrackingNodes += 1
    minCoins = change
    if change in coinValueList:
       return 1
    else:
       for i in coinValueList:
           if i <= change:
               numCoins = 1 + recMC(change - i, coinValueList)
               if numCoins < minCoins:
                   minCoins = numCoins
    return minCoins
```

Results of running this code:

```
Change Amount: 63 Coin types: [1, 5, 10, 25]
Run-time: 45.815 seconds
Fewest number of coins 6
Number of Backtracking Nodes: 67,716,925
```

Consider the output of running the backtracking code **with pruning** twice with a change amount of 63 cents.

```
Change Amount: 63  Coin types: [1, 5, 10, 25]
Run-time: 0.036 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 1-cent coins is 3
number of 5-cent coins is 0
number of 10-cent coins is 1
number of 25-cent coins is 2
Number of Backtracking Nodes: 4831
```

```
Change Amount: 63  Coin types: [25, 10, 5, 1]
Run-time: 0.003 seconds
Fewest number of coins 6
The number of each type of coins is:
number of 25-cent coins is 2
number of 10-cent coins is 1
number of 5-cent coins is 0
number of 1-cent coins is 3
Number of Backtracking Nodes: 310
```

a) With the coin types sorted in ascending order what is the first solution found? 63 pennies/coins

b) How useful is the solution found in (a) for pruning? Not useful at all!

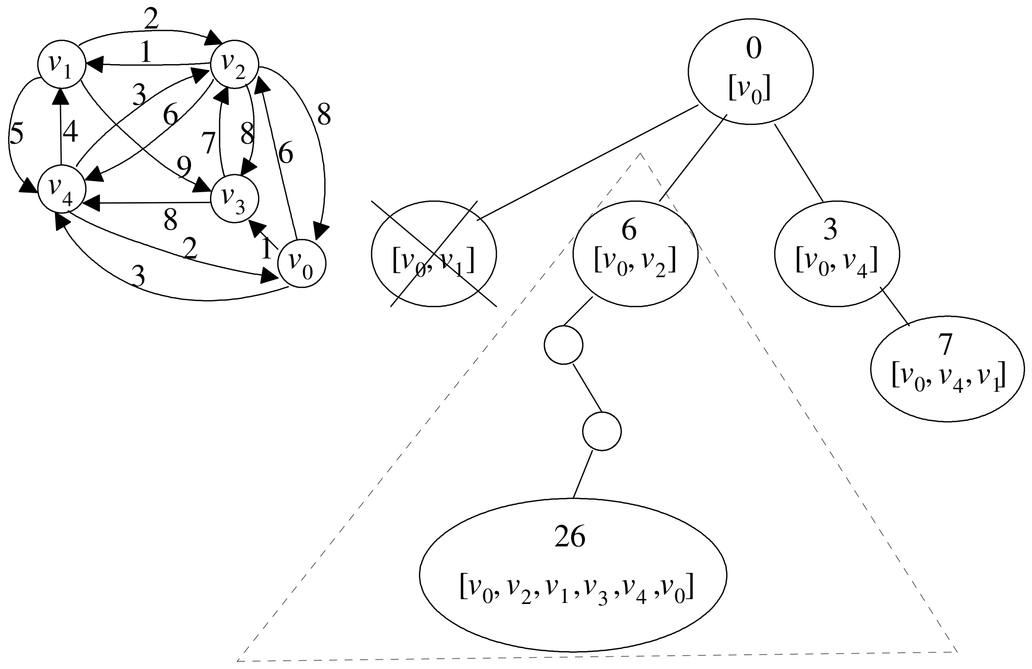
c) With the coin types sorted in descending order what is the first solution found? 6 coin solution which also is optimal

d) How useful is the solution found in (c) for pruning? Very useful since we never need to pursue a branch of the tree past 6 coins

e) For the coin-change problem, backtracking is not the best problem-solving technique. What technique was better?
Dynamic-programming is much better

3. a) For the TSP problem, why is backtracking the best problem-solving technique?
Even the dynamic-programming solutions are $O(2^n)$ where n is the number of cities.

b) To prune a node in the search-tree, we need to be certain that it cannot lead to the best solution. How can we calculate a "bound" on the best solution possible from a node (e.g., say node with partial tour: $[v_0, v_4, v_1]$)?



best tour so far = 26

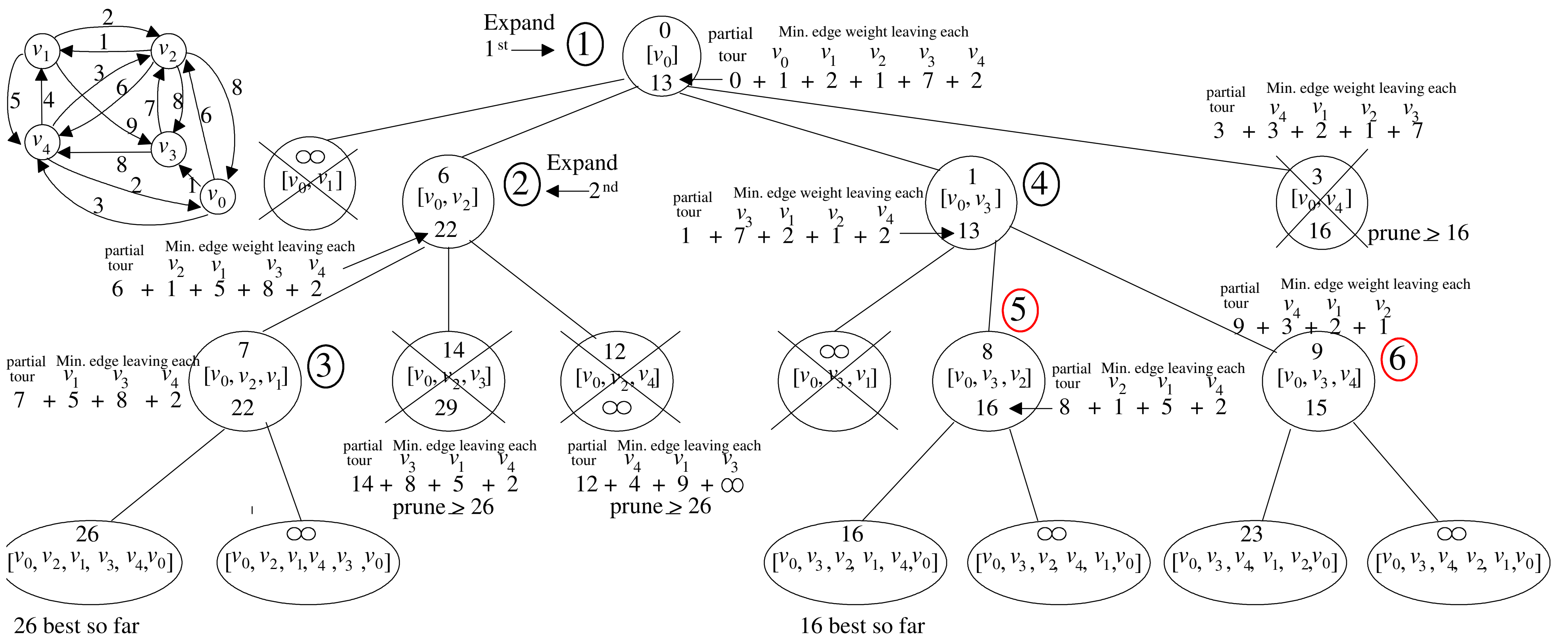
To complete the tour from the partial path $[v_0, v_4, v_1]$ each node v_1, v_2, v_3 must be left going someplace reasonable.

- For node v_1 going someplace reasonable is either v_2 or v_3. It cannot go to v_0 yet since the tour must include all nodes before going back to v_0.
- For node v_2 going someplace reasonable is either v_3 or v_0. With v_0 being an option if v_2 is the last node on the tour before going back to v_0.
- For node v_3 going someplace reasonable is either v_2 or v_0. With v_0 being an option if v_3 is the last node on the tour before going back to v_0.

A *bound* on the best we could do to complete the tour would be to sum the minimum edges leaving each of v_1, v_2, v_3 going someplace reasonable.

partial path length		min(2,9) leaving v_1		min(8,8) leaving v_2		min(∞,7) leaving v_3	
7	+	2	+	8	+	7	= 24

4. To prune a node in the search-tree, we need to be certain that it cannot lead to the best solution. We can calculate a "bound" on the best solution possible from a node (e.g., say node with partial tour: [v₀, v₄, v₁]) by summing the partial tour with the minimum edges leaving the remaining nodes going some place reasonable. Complete the backtracking state-space tree with pruning.



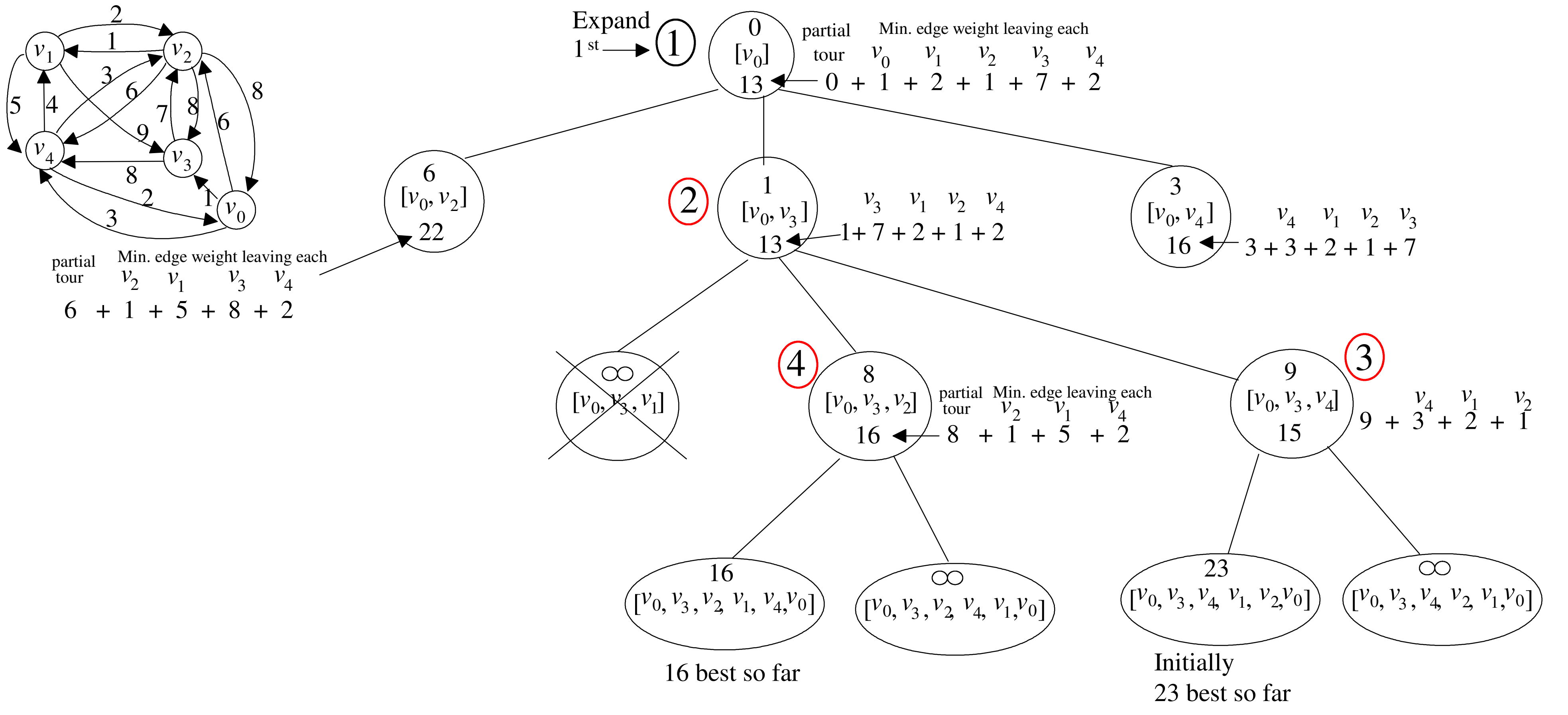
Expand 1st
0 [v0] 13
partial tour Min. edge weight leaving each v0 v1 v2 v3 v4
0 + 1 + 2 + 1 + 7 + 2
∞ [v0, v1]
6 [v0, v2] 22
Expand 2nd
partial tour Min. edge weight leaving each v2 v1 v3 v4
6 + 1 + 5 + 8 + 2
1 [v0, v3] 13
partial tour Min. edge weight leaving each v3 v1 v2 v4
1 + 7 + 2 + 1 + 2
3 [v0, v4] 16 prune ≥ 16
partial tour Min. edge weight leaving each v4 v1 v2 v3
3 + 3 + 2 + 1 + 7
7 [v0, v2, v1] 22
partial tour Min. edge leaving each v1 v3 v4
7 + 5 + 8 + 2
14 [v0, v2, v3] 29
partial tour Min. edge leaving each v3 v1 v4
14 + 8 + 5 + 2 prune ≥ 26
12 [v0, v2, v4] ∞
partial tour Min. edge leaving each v4 v1 v3
12 + 4 + 9 + ∞ prune ≥ 26
∞ [v0, v3, v1]
8 [v0, v3, v2] 16
partial tour Min. edge leaving each v2 v1 v4
8 + 1 + 5 + 2
9 [v0, v3, v4] 15
partial tour Min. edge weight leaving each v4 v1 v2
9 + 3 + 2 + 1
26 [v0, v2, v1, v3, v4, v0]
∞ [v0, v2, v1, v4, v3, v0]
16 [v0, v3, v2, v1, v4, v0]
∞ [v0, v3, v2, v4, v1, v0]
23 [v0, v3, v4, v1, v2, v0]
∞ [v0, v3, v4, v2, v1, v0]
26 best so far
16 best so far
1 2 3 4 5 6

5. In the *Best-First search with Branch-and-Bound* approach:

- does not limit us to any particular search pattern in the state-space tree
- calculates a "bound" estimate for each node that indicates the "best" possible solution that could be obtained from any node in the subtree rooted at that node, i.e., how "promising" following that node might be
- expands the most promising ("best") node first by visiting its children

a) What type of data structure would we use to find the most promising node to expand next? min. heap

b) Complete the best-first search with branch-and-bound state-space tree with pruning. Indicate the order of nodes expanded.



2
1
3
6
5
4
7
8
6
8
9
8
2
1
3
v_0 v_1 v_2 v_3 v_4
Expand 1st
1
0 $[v_0]$ 13
partial tour
Min. edge weight leaving each
v_0 v_1 v_2 v_3 v_4
0 + 1 + 2 + 1 + 7 + 2
6 $[v_0, v_2]$ 22
partial tour
Min. edge weight leaving each
v_2 v_1 v_3 v_4
6 + 1 + 5 + 8 + 2
2
1 $[v_0, v_3]$ 13
v_3 v_1 v_2 v_4
1+ 7 + 2 + 1 + 2
3 $[v_0, v_4]$ 16
v_4 v_1 v_2 v_3
3 + 3 + 2 + 1 + 7
∞ $[v_0, v_3, v_1]$
4
8 $[v_0, v_3, v_2]$ 16
partial tour
Min. edge leaving each
v_2 v_1 v_4
8 + 1 + 5 + 2
3
9 $[v_0, v_3, v_4]$ 15
v_4 v_1 v_2
9 + 3 + 2 + 1
16 $[v_0, v_3, v_2, v_1, v_4, v_0]$
16 best so far
∞ $[v_0, v_3, v_2, v_4, v_1, v_0]$
23 $[v_0, v_3, v_4, v_1, v_2, v_0]$
Initially 23 best so far
∞ $[v_0, v_3, v_4, v_2, v_1, v_0]$