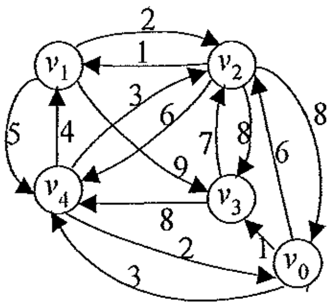
Data Structures (CS 1520) Lecture 28 Name:_______________

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) $[v_0, v_3, v_4, v_1, v_2, v_0]$ = 23

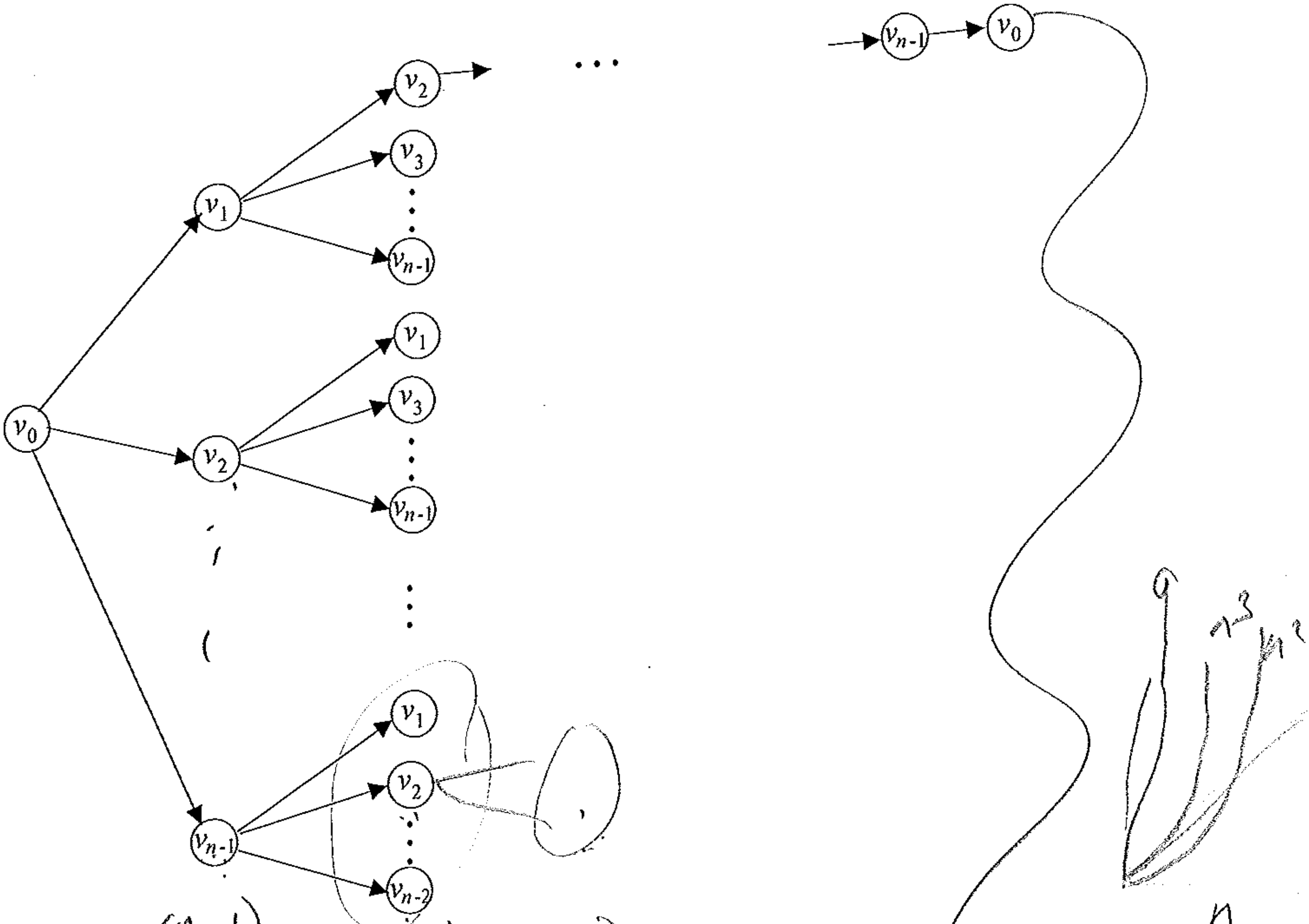
1 8 4 2 8

edge from v_i to v_j ≤ edge(v_i to v_k) + (edge v_k to v_j)

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

$V_0 V_3 V_2 V_1 V_4 V_0 = 16$

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.



$(n-1) \times (n-2) \times (n-3) \times \ldots \times 1 = (n-1)!$

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

$O(n^2)$

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
```

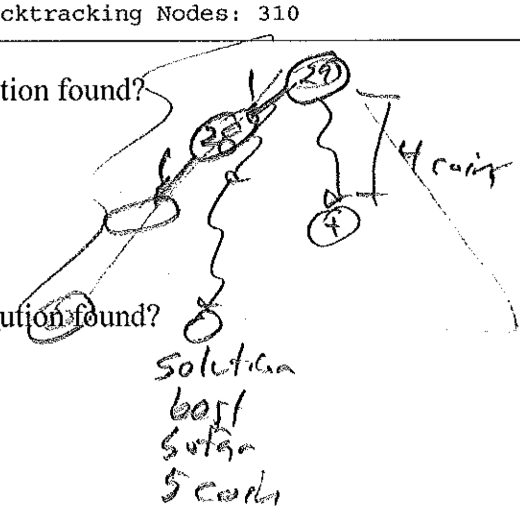
Results of running this code:

```
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?

all pennies

b) How useful is the solution found in (a) for pruning?

not very

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

greedy solution

d) How useful is the solution found in (c) for pruning?

very good

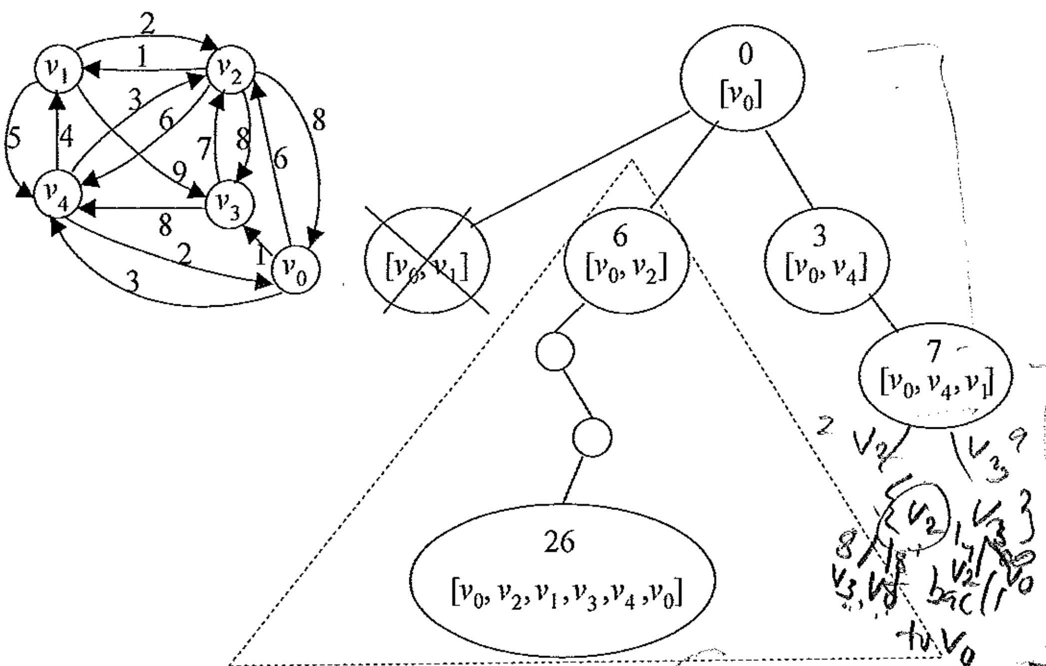
e) For the coin-change problem, backtracking is not the best problem-solving technique. What technique was better?

dyn. programming to avoid recalculating the same problem many times

3. a) For the TSP problem, why is backtracking the best problem-solving technique?

For TSP, dyn. programming solutions are $O(2^n)$

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])?



7

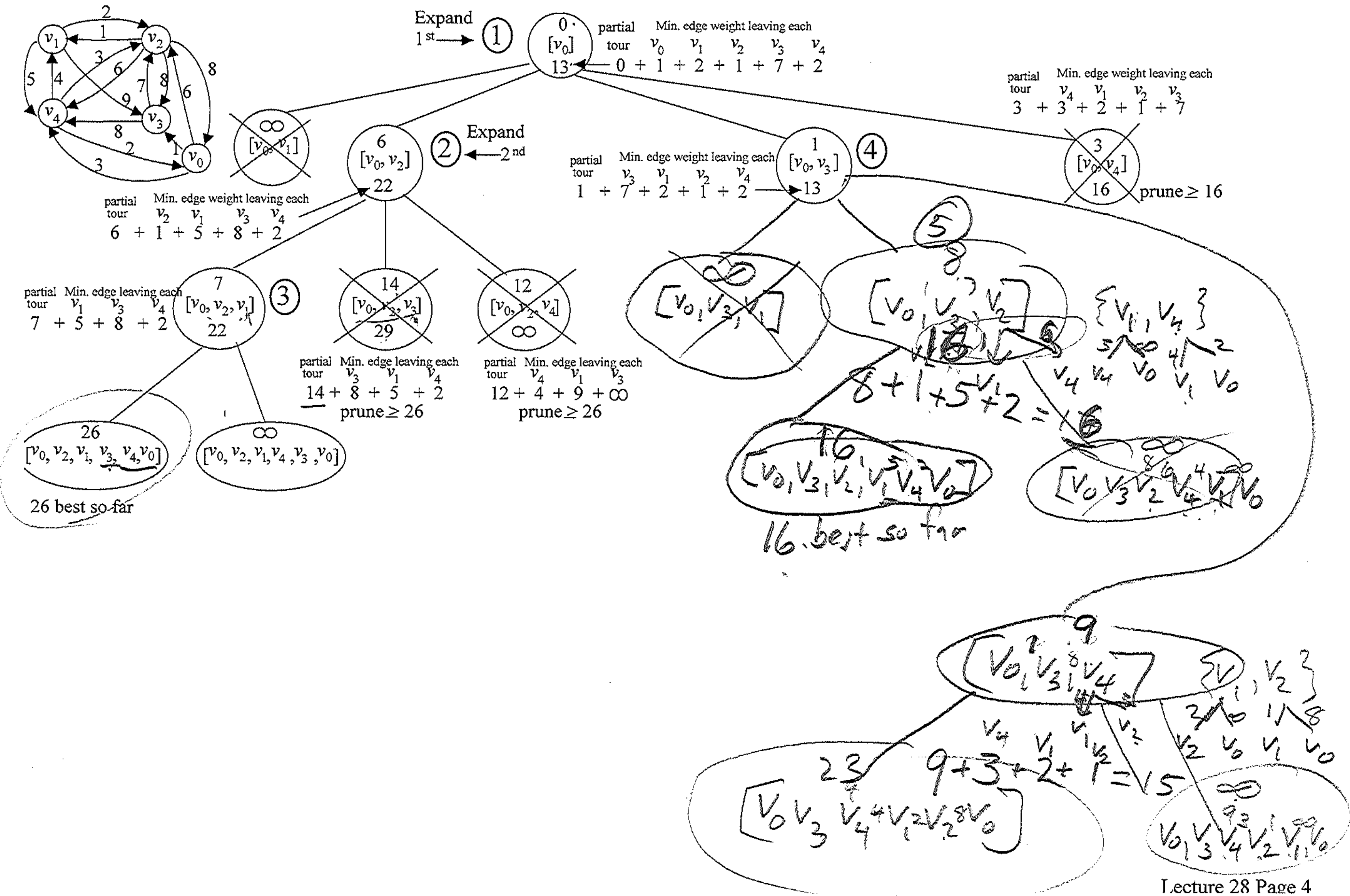
2 + 8 + 7

estimate on best completion

24

promising

2. 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: $[v_0, v_4, v_1]$) 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.



3. An alternative to backtracking, is the *Best-First search with Branch-and-Bound* approach:

- It does not limit us to any particular search pattern in the state-space tree like backtracking's left-most branch to right-most branch
- It 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
- It 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?

Put unexpanded nodes in BinHeap so finding smallest-bound node is fast,

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