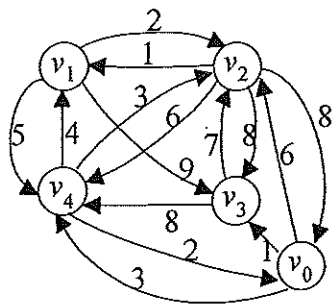


1. *Traveling Salesperson Problem (TSP)* -- Find an optimal (i.e., minimum length) tour when at least one tour exists. A *tour* (or *Hamiltonian circuit*) is a path from a vertex back to itself that passes through each of the other vertices exactly once. (Since a tour visits every vertex, it does not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

What are the length of the following tours?

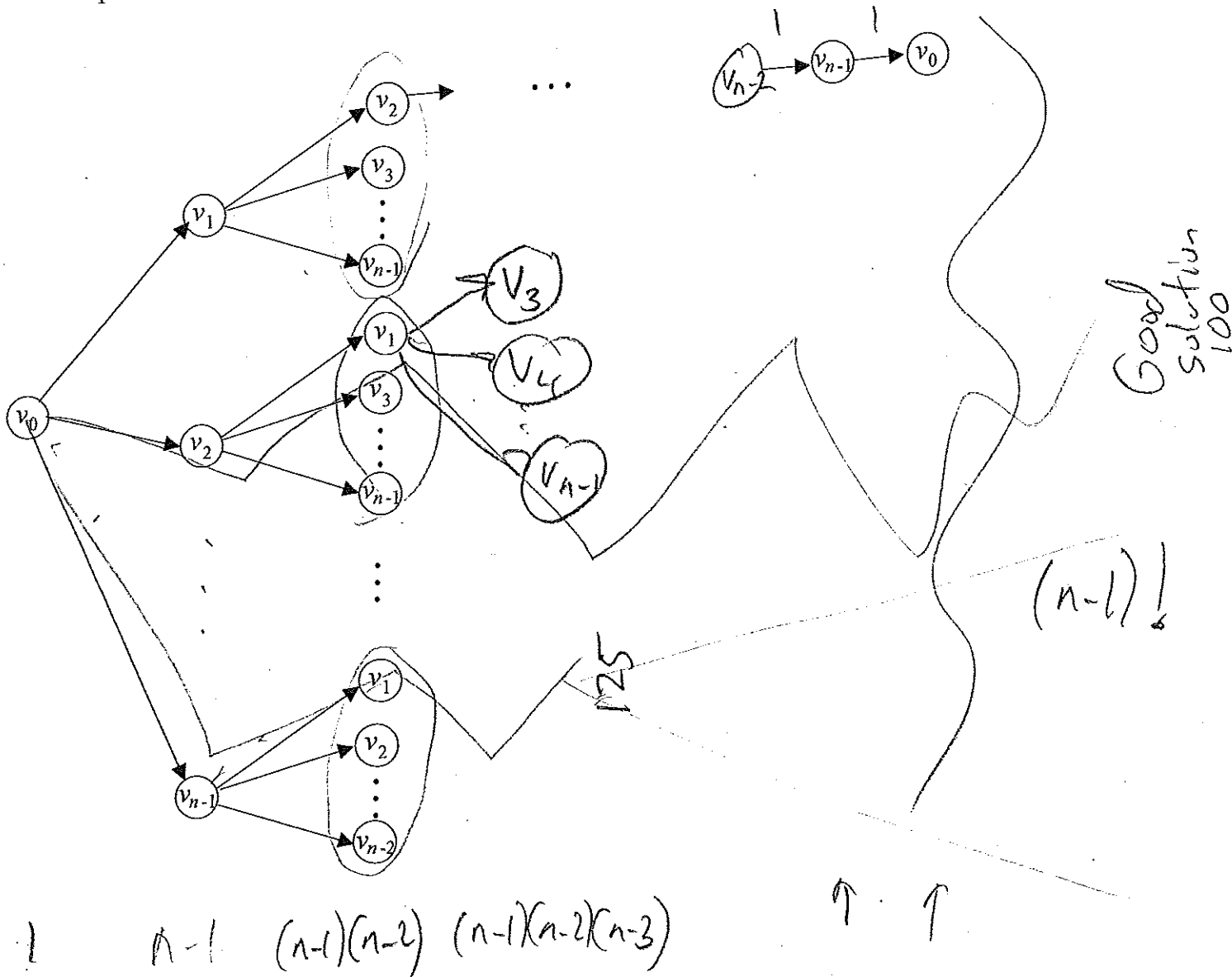


a) $[v_0, v_3, v_4, v_1, v_2, v_0]$ 23
 $1 \cdot 8$

b) List another tour starting at v_0 and its length.
 $v_0, v_3, v_2, v_1, v_4, v_0$ 16

$[v_0, v_4, v_1, v_3, v_2, v_0]$ 31
 $3 \quad 4 \quad 9 \quad 7 \quad 8$

c) For a graph with "n" vertices ($v_0, v_1, v_2, \dots, v_{n-1}$), 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.



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

$O(n^2)$ $O(n^2)$ $O(n^2)$

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 in 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*

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? *25, 25, 10, 1, 1, 1
6 coin solution*

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

Very useful

Coin - change - fewest coins

29¢ 1 5 10 12 25 50
 - 25¢

 4

3 5 coin
 1.

2
 1.

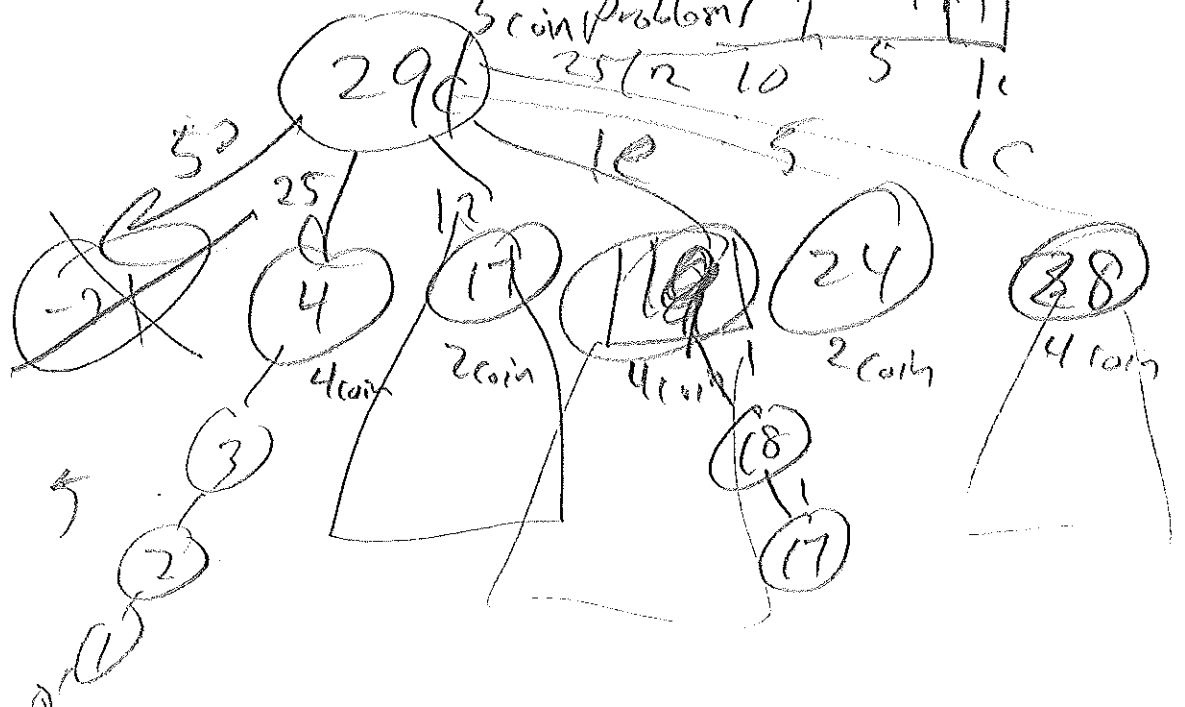
1
 1.

0

Dynamic Programming
 smallest 24 28 29 Amt change

0	1	2	3
---	---	---	---

3 coin problem
 25 (12 10 5) 10 5 10



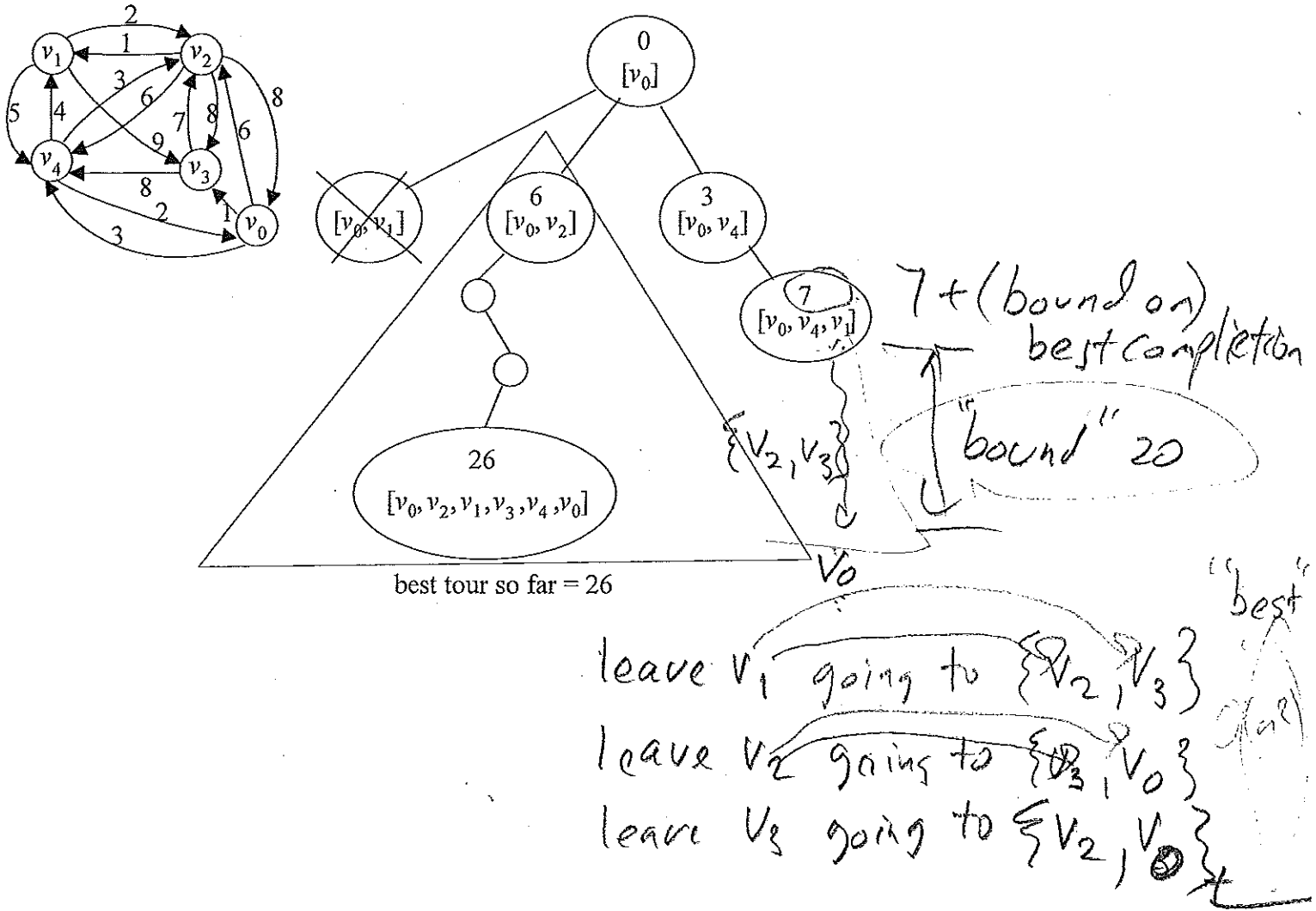
e) For the coin-change problem, backtracking is not the best problem-solving technique. What technique was better?

dynamic programming

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

dynamic programming still leads to $O(n^2^n)$ algorithm

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: $[v_0, v_4, v_1]$)?

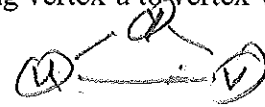


Approximation Algorithm for TSP with Triangular Inequality

Restrictions on the weighted, undirected graph $G=(V, E)$:

1. There is an edge connecting every two distinct vertices.
2. Triangular Inequality: If $W(u, v)$ denotes the weight on the edge connecting vertex u to vertex v , then for every other vertex y ,

$$W(u, v) \leq W(u, y) + W(y, v).$$



NOTES:

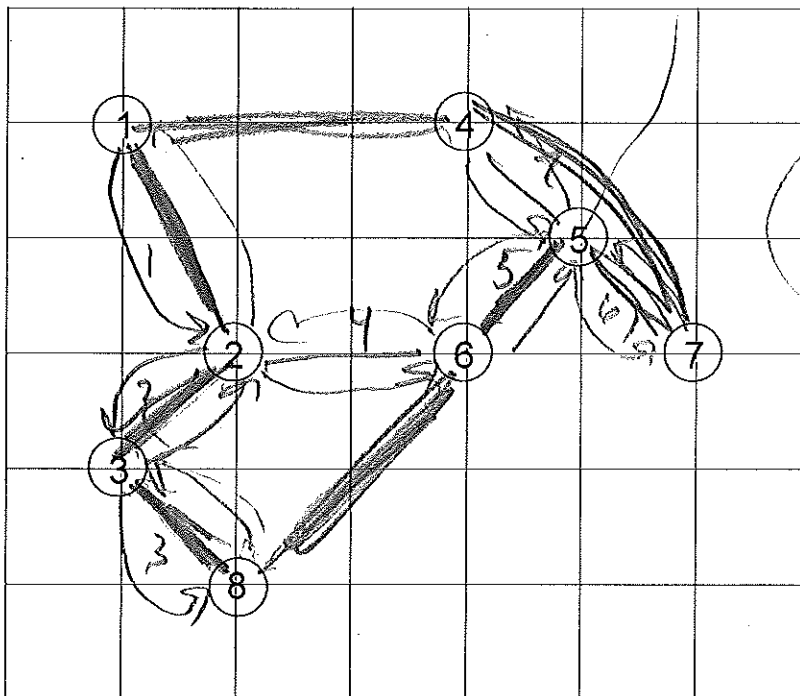
- These conditions satisfy automatically by a lot of natural graph problems, e.g., cities on a planar map with weights being as-the-crow-flies (Euclidean distances).
- Even with these restrictions, the problem is still NP-hard.

A simple TSP approximation algorithm:

1. Determine a Minimum Spanning Tree (MST) for G (e.g., Prim's Algorithm section 4.1)
2. Construct a path that visits every node by performing a preorder walk of the MST. (A preorder walk lists a tree node every time the node is encounter including when it is first visited and "backtracked" through.)
3. Create a tour by removing vertices from the path in step 2 by taking shortcuts.

2MST
4
1991/9/10

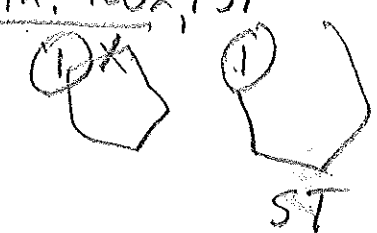
Determine a Minimum Spanning Tree (MST) for G (e.g., Prim's Algorithm) if we start with vertex 1 in the MST. (Assume edges connecting all vertices with their Euclidean distances)



preorder walk:
1, 2, 3, 8, X, X, 6, 5,
7, X, 4, X, X, X, 1

MST 2MST

Mia. tour, TSP



Prim's algorithm is a greedy algorithm that performs the following:

- a) Select a vertex at random to be in the MST.
- b) Until all the vertices are in the MST:
 - Find the closest vertex not in the MST, i.e., vertex closest to any vertex in the MST
 - Add this vertex using this edge to the MST