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

Ways to handle these "hard" problems:

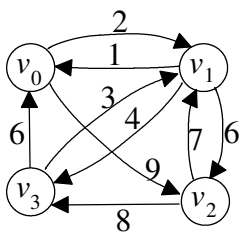
- Find the best (or a good) solution "quickly" to avoid considering the vast majority of the 2^n worse solutions, e.g, Backtracking and Branch-and-Bound
- See if a restricted version of the problem meets your needed that might have a tractable (polynomial, e.g., $O(n^3)$) solution. e.g., Fractional Knapsack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

Backtracking general idea:

- 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.

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 tours?

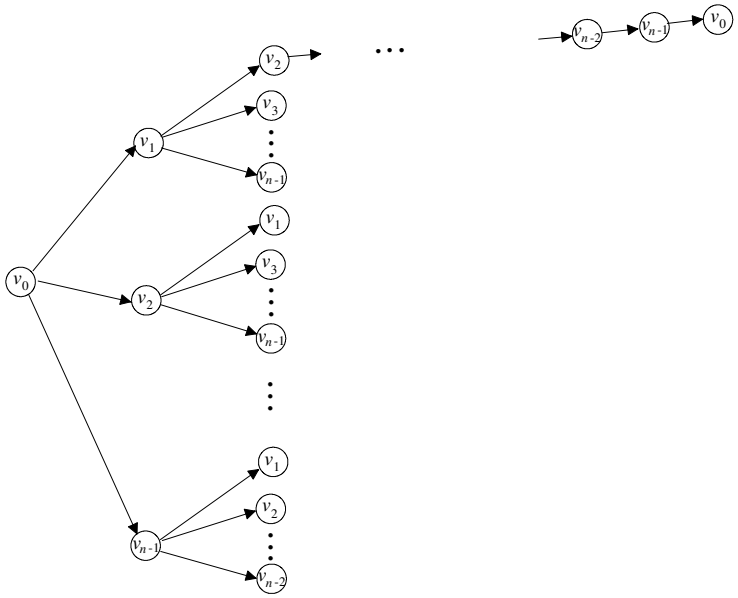


a) $[v_0, v_1, v_2, v_3, v_0]$

b) $[v_0, v_2, v_1, v_3, v_0]$

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

d) 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.



To speed the backtracking algorithm, we can prune unpromising branches. The general recursive backtracking algorithm for optimization problems (e.g., TSP, knapsack, job-scheduling) looks something like:

```
Backtrack( recursionTreeNode p ) {

    treeNode c;
    for each child c of p do                              # each c represents a possible choice
        if promising(c) then                              # c is "promising" if it could lead to a better solution
            if c is a solution that's better than best then   # check if this is the best solution found so far
                best = c                                  # remember the best solution
            else
                Backtrack(c)                              # follow a branch down the tree
            end if
        end if
    end for
} // end Backtrack
```

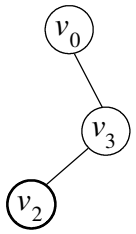
General Notes about Backtracking:

- The depth-first nature of backtracking only stores information about the current branch being explored so the memory usage is “low”
- Each node of the state-space (recursive-call) tree maintains the state of a partial solution. In general the partial solution state consists of potentially large arrays that change little between parent and child. To avoid having multiple copies of these arrays, a single “global” state is maintained which is updated before we go down to the child (via a recursive call) and undone when we backtrack to the parent.
- We could use the concept of backtracking without recursion by using a stack to maintain a collection of unexplored choices. Thus, we would simulate the run-time stack to drive the backtracking algorithm.

2. Consider customizing the above Backtrack template for the TSP problem.

a) What would the "for each child c" loop iterate over? (What problem instance information is needed?)

b) What state information is needed at each node?



c) What criteria can be used to determine if a child node (c) is NOT promising?

d) What information is needed by our promising function?

3. To parallelize the Backtracking "tree" search, we want to eliminate the recursion and replace the run-time stack by our own stack. The TSP algorithm might looks something like:

```
startState = Create partialTour with only v0
push(startState)
while stack is not empty do
   currentPartialTour = pop( )
   if currentPartialTour contains n cities then
      if currentPartialTour distance < bestTour distance then
           bestTour = currentPartialTour
    else
       for city not already on currentPartialTour do
            if  promising( currentPartialTour, city) then
                  push( new lengthened partial tour with city added )
            end if
       end for
    end if
    free( currentPartialTour)
end while
```

a) What information would we need to maintain for each partial tour?

b) If we want to parallelize TSP with pthreads, how might we statically allocate the work to each thread?

c) If we want to parallelize TSP with pthreads, how might we dynamically allocate the work to each thread?