# CS472 Module 8 Part C - Circling back to a sad sack

Athens State University

March 13, 2016

### Let's try backtracking on some other problems

- So, we've seen how backtracking can help us cut back on the amount of work we have to do on a brute force algorithm
- OK... we've been looking at the TSP and 0-1 Knapsack problems
- Is it a good idea to use backtracking with those algorithms?

### Hamiltonian Circuits

- For TSP, one searches for an optimal tour in a graph
- We have a dynamic programming algorithm that can find such an optimal tour with $T(n) = (n-1) * (n-2) * 2^{n-3}$
- This blows up quickly as the size of the graph increases
- Why don't we "simplify" the problem by looking for any tour in a graph?
- A **Hamiltonian Circuit** for a given connected and undirected graph is a path that starts with a vertex, goes through every other vertex exactly once, and ends at the starting point
- Suppose we just try to find a single Hamiltonian Circuit...

### State space tree for Hamiltonian Circuits problem

- Put starting vertex at level 0 in the state space tree
- At level 1, consider each vertex other than the starting vertex
- At level 2, consider each of those vertexes as the second vertex
- At level $n-1$, consider each of these same vertexes as the $(n-1)$st vertex

### Backtracking in this state space

- The $i$th vertex on the path must be adjacent to the $(i-1)$st vertex on the path
- The $(n-1)$st vertex must be adjacent to the 0th vertex (the starting point)
- The $i$th vertex cannot be one of the first $(i-1)$ vertices
- We will, without loss of generality, always assume that $v_1$ will be the starting point (just relabel to change)

**Backtracking Hamiltonian Circuits Algorithm**

**Algorithm 1:** hamil(): backtracking Hamiltonian Circuits Algorithm

**Input**: An adjacency matrix $W$ and a positive integer $n$ that is number of nodes in $W$
**Output**: For all paths, an array *vindex* of node indexes with entries indicating the node at that location of the path

```
if promising(i) then
    if i = n-1 then
        Output vindex;
    else
        for j ∈ [2, n] do
            vindex[i+1] ← j;
            hamil( i+1 );
```

**Backtracking Hamiltonian Circuits Algorithm**

**Algorithm 2:** promising(): evaluation function for BT Hamiltonian Circuits algorithm

**Input**: Index node $i$ to evaluate
**Output**: Return true if moves towards solution, false otherwise

```
if (i = n-1) and not W[vindex[n-1]][vindex[0]] then
    switch ← false;
else if (i>0) and not W[vindex[i]][vindex[i - 1]] then
    switch ← false;
else
    switch ← true;
    j ← 1;
    while (j < i) and switch is true do
        if vindex[i] = vindex[j] then
            switch ← false;
        j ← j+1;
return switch;
```

**So, what do we see?**

- Note the similarity in structure to other BT algorithms
- The number of nodes in the state space tree is

$$1 + (n-1) + (n-1)^2 + \ldots + (n-1)^{(n-1)} = \frac{(n-1)^n - 1}{n-2}$$

- Note that is worse than exponential in rate-of-growth
- So.. maybe this wasn't a good idea after all

**How about the 0-1 Knapsack problem?**

We have a set of $n$ items where we know each items weight $w_i$ and profit $p_i$ (for $i$ from 0 to $n-1$). We want to put a subset of these items into a sack that can hold $W$ units of weight. We wish to find the subset of items that gets us as close to $W$ as possible with the maximum profit.

**A state space for the 0-1 Knapsack problem?**

- Build the tree by picking an item as the root node
- We select the left child of this node if we include the item and the right node if we exclude that item
- Label each edge in the tree with the contribution of that item's weight to the total
- Each path from the root to a leaf of the tree is a candidate solution for the problem

**The pruning process**

- Pruning the tree is complicated by the fact that we are looking for a solution to an optimization problem
- This means that we don't know if a node contains a solution until we complete the search and we must adjust our backtracking accordingly
- If we find that items up to a node have a greater total profit than the best solution found so far, then we update our guess of best solution
- But we may find a better solution later in the traversal and so must continue to check a node's descendants

**Algorithm 3:** btknapsack(): A backtracking solver for the 0-1 knapsack problem

```
Input: Positive integers n and W and arrays w and p, with the contents of the arrays sorted by p_i/w_i
Output: A boolean array bestset with elements set to true if that item is in the optimal set and an
        integer maxprofit
if (weight <= W) and (profit > maxprofit) then
    maxprofit ← profit;
    numbest ← i;
    bestset ← include;
if promising(i) then
    include[i+1] ← true;
    btknapsack(i+1, profit + p[i+1], weight + w[i+1]);
    include[i+1] ← false;
    btknapsack(i+1, profit, weight);
```

**Algorithm 4:** promising(i): Check to see if we should prune 0-1Knapsack search tree

```
Input: An index i
Output: True if promising, false otherwise
if weight >= W then
    return false;
else
    j ← i + 1;
    bound ← profit;
    totweight ← weight;
    while (j<=n) and ((totweight + w[j]) <= W) do
        totweight ← totweight + w[j];
        bound ← bound + p[j];
        j ← j + 1;
    k = j; if (k <= n) then
        bound ← bound + (W - totweight)*(p[k]/w[k]);
    return (bound > maxprofit);
```

### Backtracking vs. dynamic programming

- We have a DP algorithm for 0-1 Knapsack that has a rate of growth $O(\min 2^n, nW)$
- In the worst case, the BT algorithm will check $O(2^n)$ nodes
- It would appear that the BT algorithm might be better since it doesn't include the $nW$ bound in its rate-of-growth
- But in reality, it is hard to tell exactly how many nodes get pruned from the search tree
  - Have to just use experimental data to determine if you get any gain by selecting one algorithm over another