Algorithms and Data Structures

Week 12 – Lecture A

The Knapsack Problem

- Some of you may remember the knapsack problem from last year:
  - Given:
    - A knapsack of size *S* (integer);
    - *n* items each having:
      - Size, s_i; (integer)
      - Value, v_i.
  - Find the set of items such that:
    - The total size of the items in the set $\leq S$;
    - The total value of the items in the set is maximized.
- Note: in this version of the problem each item (and the knapsack) must have an integer size.

Dynamic Programming VII
Integer Knapsack

DP VII: Integer Knapsack

- Once again, we will approach this problem using dynamic programming.
- We use the 5 steps from our previous examination:
  1. Define the sub-problems (determine number of sub-problems);
  2. Guess the solution to a sub-problem (determine number of guesses);
  3. Relate sub-problems via recursion (determine time per sub-problem);
  4. Solve the sub-problems systematically, ensure sub-problem dependency is acyclic (#sub-problems $\times$ time per sub-problem);
     1. Recurse and memoize (top down);
     2. Topological sort and loop (bottom up);
  5. Solve original problem by combining sub-problem solutions

1, 2: Sub-problems and Guess

- Although the order of items is of no importance we can still use it to define the sub-problems.
- We need to solve the n sub-problems consisting of each possible suffix $(i, n]$ of the items to be chosen.
- The guess for each sub-problem is "what do we do with item i?"
  - Put it in the knapsack?
  - Leave it out?

3: Relating the Sub-problems

- We can define the solution to the sub-problem in the form of a recursive function $k(i, c)$, representing the greatest value to be obtained by packing items selected from i to $n-1$ into a backpack with capacity $c \leq S$.
  - **Procedure knapsack(i,c)**

```
  if i == n return 0
  best = knapsack(i+1,c)
  if s(i) <= c then
       best = max(best, v(i)+ knapsack(i+1,c-s(i))
  endif
  return best
end knapsack()
```

4a: Memoizing

- You may have noticed a problem in the procedure I just outlined:
  - To save recursion costs I want to memoize the individual knapsack sub-problem solutions;
  - I do not know, in advance, for what values of c I will need solutions.
- What should I do?
  - Solve each sub-problem for all possible values of c!
  - $0 < c \leq S$.
- The number of sub-problems is not O(n):
  - It is O($n \times |S|$)

4b: Topological Sort

- Now we can see why we needed to specify that all of the sizes, s_0, s_1, …, s_{n-1} and S, must be integers:
  - If this is not true there are an infinite number of potential sub-problems!
- The bottom up solution requires that we memoize all possible remaining capacities, as follows:

Bottom Up Knapsack

```
Procedure KnapsackUp(n,S)
     best: array[0..n,0..S] of float
     for j in 0..S
          best[n,j] = 0
     rof
     for i in n-1..0
          for j in 0..S
               best [i,j]= best[i+1,j]
               if s(i)<= j
                    best[i,j] = max(best[i+1,j], v(i) +
                                                  best[i+1,j-
s(i)])
               endif
          rof
     rof
     return best[0,S]
end KnapsackUp
```

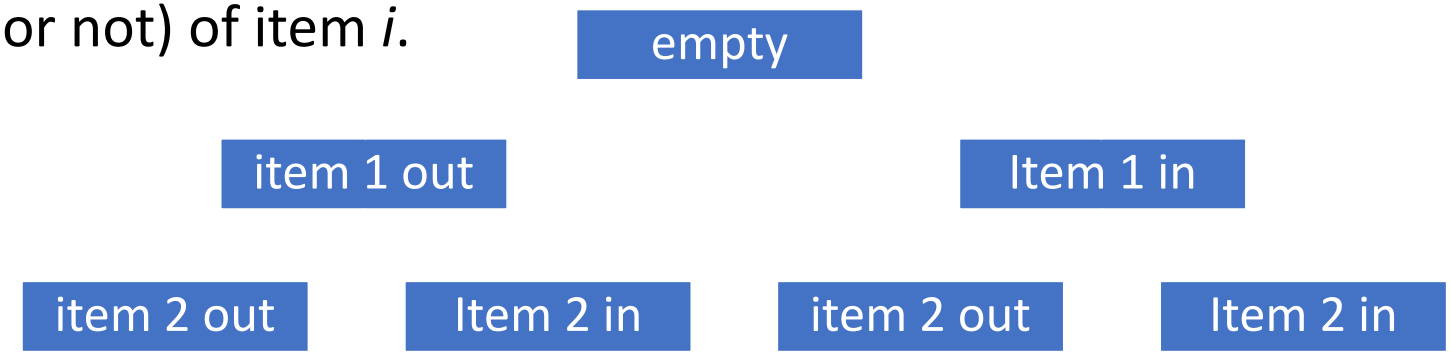
5: The Final Solution

- As can be seen from the bottom up solution the solution to the original problem is **`best[0,S]`**.
- Does this analysis mean that we cannot solve the knapsack problem if the item sizes are not integers?
- No, it just means that we can't use dynamic programming as a solution technique.
- Does this mean that we can only solve it with brute force?
  - Try all 2^n possible combination of items.
- Not necessarily.
- Perhaps we can find a way to reduce the size of the search tree.

Branch and Bound

Pruning the Search Tree

- We can present the knapsack problem in the form of an *n*-deep binary tree where each level, *i*, considers the inclusion (or not) of item *i*.



- If we have to traverse the entire tree this will cost $O(2^n)$.
- We need some way to *prune* the tree:
  - Eliminate sub-trees without fully evaluating them.

Estimating a Value

- Consider an arbitrary node in the search tree.
- For any node we know:
  - What items have been included in the knapsack so far;
  - The total value of the items that have been included;
  - The total size of the items that have been included;
  - The remaining size left in the knapsack.
- We do not know:
  - The best selection of the remaining items to pack in the remaining space in the knapsack.
- If we can estimate this unknown value we can proceed as follows:

A Preliminary Attempt

- **node: record**
  {id: bit string
  value: float:
  capacity: float
  }
- **Procedure knapsackEst(i: node,S: size)**
  reachable: max heap of nodes ordered by value
  next: node
  while reachable is not empty
  add new nodes to the heap where:
  node.id = node.id &(0|1)
  node.value=value(id)+estimate(remainder)
  node.capacity = node.capacity (-0|c(i))
  elihw
- As written, this procedure has a number of problems:

Problems with the Procedure

1. The procedure does not stop:
   - Solution: once we have considered all *n* items we stop adding.
   - Is this the best possible solution?
2. We still have all 2^n nodes in the heap:
   - We need some way to *not* add nodes to the heap.
   - We need to establish *bounds* which bracket the best solution.

- *Upper bound*: an overestimate of the best possible solution starting at a given node.
- *Lower bound*: an underestimate of the best possible solution starting at the same node.

Bounds

- Once we have these we can assert that for any node:
  - Lower bound < actual solution < upper bound.
- For two nodes, *i* and *j*, if upper bound (*j*) < lower bound(i) then the best solution cannot lie in the sub-tree rooted at node *j*.
- This means that we do not need to add node *j* to the heap:
  - We can prune its sub-tree.
- This is the key insight to the branch and bound technique.

Branch and Bound

- Now we need only determine mechanisms for finding these bounds:
- This will vary depending on the particular problem.
- For the knapsack problem:
  - The upper bound can be found by solving the *continuous* problem.
  - A lower bound can be found by removing the fractional part from this upper bound.
- For any node, *i*, we can establish an upper bound as follows:

Evaluating a Node

- The upper bound value of a node is obtained by taking the *known* value of the items already in the knapsack and adding to this the continuous solution to the new problem of packing the best selection from the remaining items into the space we have left.
- Thus, if the parent of node *i* has the following known values:
  - selected_{i-1}, the selected items from the first $i-1$ items;
  - value_{i-1}, the total value of these selected items;
  - size_{i-1}, the total size of these items.
- Then we can estimate the value of node *i* as follows:

Analysis

- Each node requires us to evaluate the continuous solution for some set of items.
- This uses a greedy algorithm and is O(n).
- Thus, the overall solution requires O(n) times the number of nodes we need to solve.
- In the worst case this is O(n)x2^n.
- This is no better than brute force.
- In practice we can usually do much better by considering items in an optimal order.

Best Order

- To solve the continuous problem we need our items in a particular order:
  - Decreasing order of $value_i/size_i$.
- If we sort our items in this order and then add them (or not) to the knapsack in the same order we will typically only need to consider a small subset of the total 2^n nodes in the problem tree.
- Note: In some cases we may still end up needing to evaluate all 2^n nodes in the tree.

Summing Up

- In general:
  - Branch and bound is less efficient than dynamic programming;
  - It can often be used where DP is not an option:
    - The dependency graph is not acyclic;
    - The number of possible sub-solutions is too large to compute.
- If a DP solution is available, it is usually a better choice of approach than branch and bound.