Graph Algorithms (2)
Backtracking and Pruning
Review of first approach:

• find all possible combinations of $i_1, i_2, \cdots$ such that $w_{i_1} + w_{i_2} + \cdots \leq W$

• record the corresponding value $v_{i_1} + v_{i_2} + \cdots$

• finally find the solution

Start procedure by picking up nothing, followed by picking up one item, then add one further item at a time. Stop when the sum of added items would exceed $W$. 
Example: Assume $n = 4$, $W = 8$,

$v_1 = 3$, $v_2 = 5$, $v_3 = 6$, $v_4 = 10$

$w_1 = 2$, $w_2 = 3$, $w_3 = 4$, $w_4 = 5$

Solution: $(x_1, x_2, x_3, x_4) = (0, 1, 0, 1)$
Backtracking...

- A recursive method for building up feasible solutions one at a time.
- Is exhaustive, i.e., all feasible solutions are considered.
- Provides a simple recursive method of generating all possible $n$-tuples. After each $n$-tuple is generated, it is checked for feasibility. If feasible, the solution incurred is compared to the current optimal solution, and this current optimal solution is updated whenever a better feasible solution is found.
- Is a special case of graph algorithm.
Example: 0-1 knapsack

- Given profits $p_i$, weights $w_i (1 \leq i \leq n)$, and a capacity $M$. Aim is to maximise $\sum p_i x_i$, subject to $\sum w_i x_i \leq M$, $x_i = 0$ or $1$ ($1 \leq i \leq n$).

- A feasible solution is an $n$-tuple such that $\sum w_i x_i \leq M$.

- The number of possible $n$-tuples is $2^n$, since there are two choices for each of the $n$ co-ordinates (not all $2^n$ $n$-tuples are feasible solutions).
Example: 0-1 knapsack

- Define $P = \{p_1, \ldots, p_n\}$, $W = (w_1, \ldots, w_n)$, and $M$. $X$ shall denote the current $n$-tuple being constructed, OPTX the current optimal solution, and OPTP its profit.

A possible backtracking solution is given by a recursive procedure named $\text{Knap}$, where

- the parameter “lev” shall denote the current co-ordinate (corresponding to the depth of a node in a tree) being chosen.
- OPTX and OPTP are passed by reference.
- Start with lev = OPTX = OPTP = 0
0-1 knapsack: Backtracking algorithm

Knap (lev, OPTP, OPTX) {
    if (lev = n+1) {
        if ( \( \sum w_i x_i \leq M \) AND \( \sum p_i x_i > OPTP \)) {
            OPTP = \( \sum p_i x_i \);
            OPTX = X;
        }
    }
    else {
        \( x_{lev} = 1; \)
        knap (lev+1, OPTP, OPTX);
        \( x_{lev} = 0; \)
        knap (lev+1, OPTP, OPTX);
    }
}

MA008/MIIZ01 – p.8/28
0-1 knapsack: Backtracking algorithm

- This procedure generates all $2^n$ possible $n$-tuples in reverse lexicographic order. The complexity is $O(n \cdot 2^n)$, since the second if-statement takes $O(n)$.

- The recursive calls generated by $\text{Knap}$ produce a binary tree, called the state space tree of the given problem instance. The backtrack performs a depth-first traversal of the state space tree.

Question: Why is this called *backtracking*?
0-1 knapsack: State space tree

When \( n = 3 \), the following state space tree is produced (each node records \( X \); a hyphen indicates that the specified co-ordinate is undefined.)
0-1 knapsack backtracking solution

Advantage:

• Will find an optimal solution if an optimal solution exists.

Disadvantage:

• Computationally very demanding: $O(n \cdot 2^n)$.
• Very memory intensive. Number of nodes and links both grow with $2^n$, node label grows with $n$. 
Pruning

- The graph algorithms described so far are nothing more than convenient methods of checking all possibilities. But the rate of growth is exponential.
- Pruning: Identify “good” and “bad” branches of the state space tree, then abandon bad branches.
- E.g.: apply feasibility test to interior nodes of the tree, rather than just to leaf nodes. If a feasibility constraint is violated at an interior node, there is no need to look at any descendants of that node, as there can be no feasible solutions in the subtree under the given node.
Pruning the 0-1 knapsack tree

- check that $\sum_{i=1}^{\ell} w_i x_i \leq M$ before performing any recursive calls.
- This is efficiently achieved by defining a parameter CURRW which records the weight $\sum_{i=1}^{lev-1} w_i x_i$ of the partial solution $(x_1, \ldots, x_{lev-1}, -, \ldots, -)$.
- If $x_{lev} = 0$ then CURRW does not change; if $x_{lev} = 1$, then CURRW increases by $w_{lev}$. A partial solution is feasible if and only if CURRW $\leq M$.
- The following procedure `knap1` explores only those nodes in the state space tree which correspond to feasible partial solutions.
knap1 (lev, CURRW, OPTP, OPTX) {
  if (lev = (n+1)) { as before }
else{
    if (CURRW + w_{lev} ≤ M) {
      x_{lev} = 1;
      knap1 (lev+1, CURRW+w_{lev}, OPTP, OPTX)
    }
    x_{lev} = 0;
    knap1(lev+1, CURRW, OPTP, OPTX);
  }
}
Approaches to pruning

- Heuristics: Apply common knowledge to achieve pruning.
- Expert: Apply expert knowledge to achieve pruning.
- Statistics: Use results from statistical analysis to achieve pruning.
- Bounding functions: Define a bound on the (local) state space.
Bounding Functions

- are a generalization of many other approaches to pruning.
- are popularly used to achieve efficient pruning.
- defines a bound (or threshold) on the feasibility of a state.
Bounding Functions: Preliminaries

Let's make a few preliminary definitions. These apply to any backtracking algorithm for a maximisation problem:

- Suppose a node in the state space tree corresponds to a partial solution
  \[ X = (x_1, \ldots, x_{\text{lev}}, -, \ldots, -) \], where \( 0 \leq \text{lev} \leq n \).

- Define \( C(X) \) to be the maximum profit of any feasible solution which is a descendant of \( X \) in the state space tree. So, if \( X \) is itself a feasible solution (i.e. \( \text{lev} = n \)), then \( C(X) = \) the profit incurred by \( X \). Also, if \( X = (-, \ldots, -) \) (i.e. \( \text{lev} = 0 \)), then \( C(X) = \) the optimal profit of the given problem instance.
Bounding Functions: Definition

- In general, $C(X)$ can be computed only by traversing the subtree with root node $X$, which is precisely what we aim to avoid. This motivates the following definition. A bounding function is any function $B$ defined on the set of nodes in the state space tree which satisfies the following properties:

1. If $X$ is a feasible solution (i.e. $lev = n$) then $B(X) = C(X) =$ the profit incurred by $X$

2. For any feasible partial solution $X$, $B(X) \geq C(X)$ (so $B(X)$ provides an upper bound on the profit of any feasible solution which is a descendant of $X$ in the state space tree).
Bounding Functions: Definitions

- $B(X)$ can be used to prune the state space tree. If at some stage of the backtrack $B(X) \leq OPTP$ (the current profit) then $C(X) \leq B(X) \leq OPTP$. Thus, we can ignore all descendants of $X$ in the state space tree since none of them can yield a profit higher than OPTP.

- For a minimisation problem, the definitions are the same, except that all inequalities are reversed.

- We think of $B(X)$ as an approximation to $C(X)$. We want a bounding function to be:
  1. easy to compute, and
  2. close to $C(X)$. 
Finding a Bounding Function

• These two aims work against each other, so it is sometimes hard to find suitable bounding functions. At one extreme, $C(X)$ is itself a bounding function, but it is normally too costly to compute. At the other extreme, we could define $B(X) = 10^{10}$ (i.e. some very large number) for all $X$. This is certainly very easy to compute, but is too far away from $C(X)$ to be of any use.

• In general, it takes some brain-activity to find a good bounding function.
Finding a Bounding Function: Example

Let's try to define a useful bounding function for the 0-1 knapsack problem using the same notation as before.

- Given $0 \leq lev \leq n$, and $0 \leq M' \leq M$, define $RK(lev, M')$ to be the optimal profit of the rational knapsack problem, using only objects $lev + 1, \ldots, n$, and capacity $M'$.

- Recall that the rational knapsack problem can easily be solved by a greedy algorithm. Hence, we may use $RK$ to define a bounding function.
A Bounding Function based on RK solution

- Given a (feasible) partial solution

\[ X = (x_1, \ldots, x_{lev}, -, \ldots, -) (0 \leq lev \leq n), \]

define

\[
B(X) = \sum_{i=1}^{lev} p_i x_i + RK(lev, M - \sum_{i=1}^{lev} w_i x_i) \\
= \sum_{i=1}^{lev} p_i x_i + RK(lev, M - CURRW).
\]
A Bounding Function based on RK solution

• Thus, $B(X)$ is equal to the sum of:
  1. the profit obtained from objects $1, \ldots, lev$, plus
  2. the profit from the remaining objects, using the remaining capacity $M - CURRW$, but allowing rational $x_i$’s.

• If, in (2), we restrict each $x_i$ to be 0 or 1, then we would obtain $C(X)$. Allowing the $x_i$’s ($lev + 1 \leq i \leq n$) to be rational may yield a higher profit, so $B(X) \geq C(X)$, and $B(X)$ is indeed a bounding function.

• $B(X) \geq C(X)$ must be proven (omitted here).
To effectively use this bounding function in a backtrack algorithm, we would first sort the objects in non-increasing order of profit/weight, in order to make the values $\frac{p_i}{w_i}$ easier to calculate. Hence, we assume that the objects have been renamed so that $\frac{p_1}{w_1} \geq \ldots \geq \frac{p_n}{w_n}$.
0-1 knapsack with bounding function

knap2(lev, CURRW, OPTP, OPTX) {
    double B;
    if (lev = n+1) { as before }
    else{
        B = \sum_{i=1}^{lev-1} p_i x_i + RK(lev-1, M - CURRW);
        if (B > OPTP){
            if (CURRW + w_{lev} <= M) {
                x_{lev} = 1;
                knap2(lev+1, CURRW+w_{lev}, OPTP, OPTX);
                x_{lev} = 0;
                knap2(lev+1, CURRW, OPTP, OPTX);
            }
        }
    }
}
0-1 knapsack with bounding function

• An instance of 0-1 knapsack: 5 objects, having weights 11, 12, 8, 7 and 9, and profits 23, 24, 15, 13 and 16 (respectively), and a knapsack capacity of 26.

• Note that the objects are already sorted in decreasing order of $p_i/w_i$. We draw below the portion of the state space tree traversed in the course of the backtrack algorithm knap2. At each node we record $X, B(X)$ and $CURRW$. 
0-1 knapsack with bounding function

X=(−,−,−,−,−) B=52.625 CURRW = 0

X=(1,−,−,−,−) B=52.625 CURRW = 11

X=(1,1,−,−,−) B=52.625 CURRW = 23

X=(1,1,0,−,−) B=52.57 CURRW = 23

X=(1,1,0,0,−) B=52.33 CURRW = 23

X=(1,1,0,0,0) P=47, so set OPTP=47 CURRW = 23

X=(0,1,−,−,−) B=51 CURRW = 20

X=(0,1,1,−,−) B=51 CURRW = 20

X=(0,1,1,1,−) B=51 CURRW = 27

X=(0,1,1,1,0) P=51 > OPTP, so set OPTP = 51 CURRW = 27
Next lecture

• Introduction to: P, NP, and NP completeness
• Recommended: Read up on Traveling Salesman Problem definition and solutions.