Lecture 5: Backtracking Algorithm

Analysis and Design of Computer Algorithms

GONG Xiu-Jun

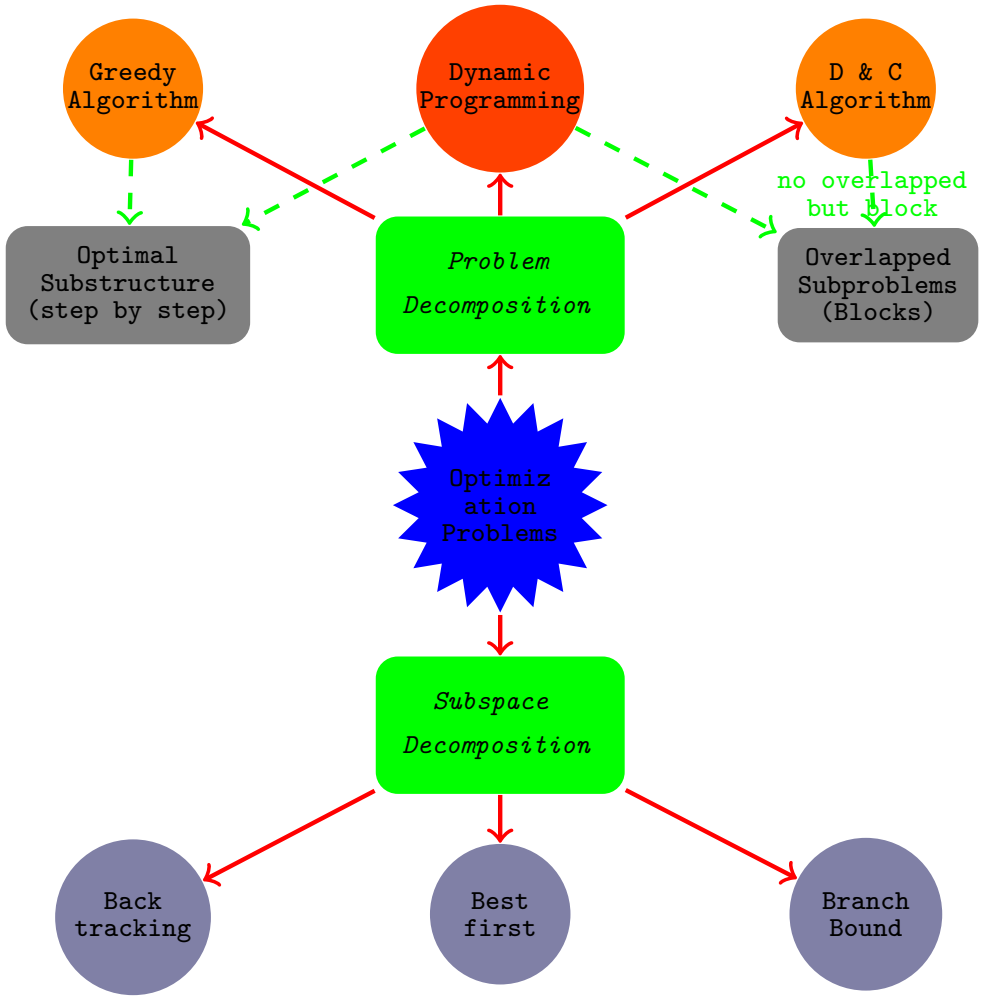
School of Computer Science and Technology, Tianjin University

Email: gongxj@tju.edu.cn
Website: http://cs.tju.edu.cn/faculties/gongxj/course/algorithm
Discussion : http://groups.google.com/group/algorithm-practice-team

May 13, 2014

Backtracking Algorithm

Motivations



Solution space

- A solution to a specific problem can be represented by a n tuple $(x_1, x_2, \cdots, x_n)$
  1. x_i comes from a limited set S_i.
  2. The dimensions of tuples for different solutions might vary depending on its definition, thus the representation can be classified:
  3. Fixed Length Tuple (FLT) representation: all tuples have same dimensions
  4. Varied Length Tuple (VLT) representation: tuples have different dimensions
- Solution space: A set consists of all enumerates in the form of solution representation.
  1. For FLT: $X = \{(x_1, x_2, \cdots, x_n) | x_i \in S_i\}$
  2. For VLT: $X = \{X^1, X^2, \cdots, X^k\}$ where $X^j = \{(x_{j1}, x_{j2}, \cdots, x_{jk})\}$

Solution space, cont.

- Not all $x \in X$ is the real solution
  1. Feasible solution : if x holds all constrain conditions $g_i(x) = true$ $(i \in [1..m])$, simply as $g(x)$
  2. Optimal solution : if x maximize/minimize target functions $f_i(x)$ $(i \in [1..k])$, simply as $f(x)$
- Our goal is to search for the feasible/optimal solutions from solution space
  1. Define solution representations
  2. Formulate solution space
  3. Search solution space

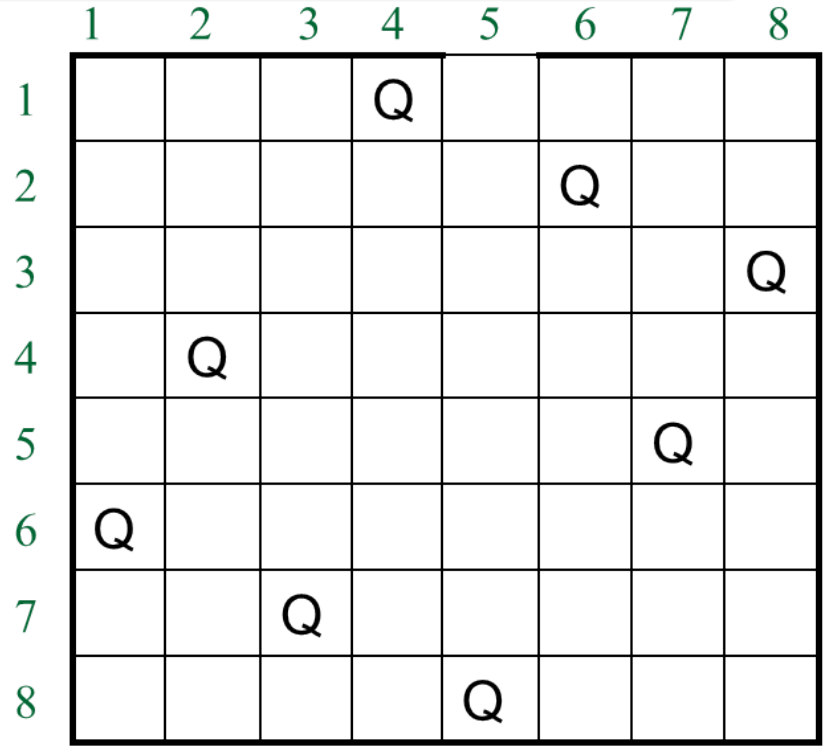
Define solution representations

Eight queens puzzle

Eight queens puzzle

Using a regular chess board, the challenge is to place eight queens on the board such that no queen is attacking any of the others.

- The solution can be represented by a 8- tuple $(x_1, \cdots, x_8)$ where x_i is the column number of i-th queen
- The size of solution space is 8^8
- Constrains: $x_i \neq x_j$ and $|x_i - x_j| \neq |j - i|$ for all i, j



Define solution representations

Subset sum problem

Subset sum problem

Finding what subset of a set of positive integers $S = \{w_1, w_2, \cdots, w_n\}$ has a given sum M,

For an example, M=31, n=4 and W=(11, 13, 24, 7), then 11+13+7=31 and 24+7=31

- The solution can be represented by

FLT $\mapsto$	$(x_1, x_2, \cdots, x_n)$ where $x_i = 1$ if it is chosen else 0	(1,1,0,1) (0,0,1,1)	$O(2^n)$
VLT$\mapsto$	$(j_1, j_2, \cdots, j_k)$ where j_i is the order number of i-th integer chosen in S and k is the total number chosen	(1,2,4) (3,4)	$O(2^n)$

Formulate solution space

- Problem state: a point in solution subspace. We can check if it reaches the goal
- Start state / (root): a problem state at which we start to search for the goal
- Solution state: a path from current (visiting) state to the root
- Answer/goal state: a point in solution space that is the goal

Definition

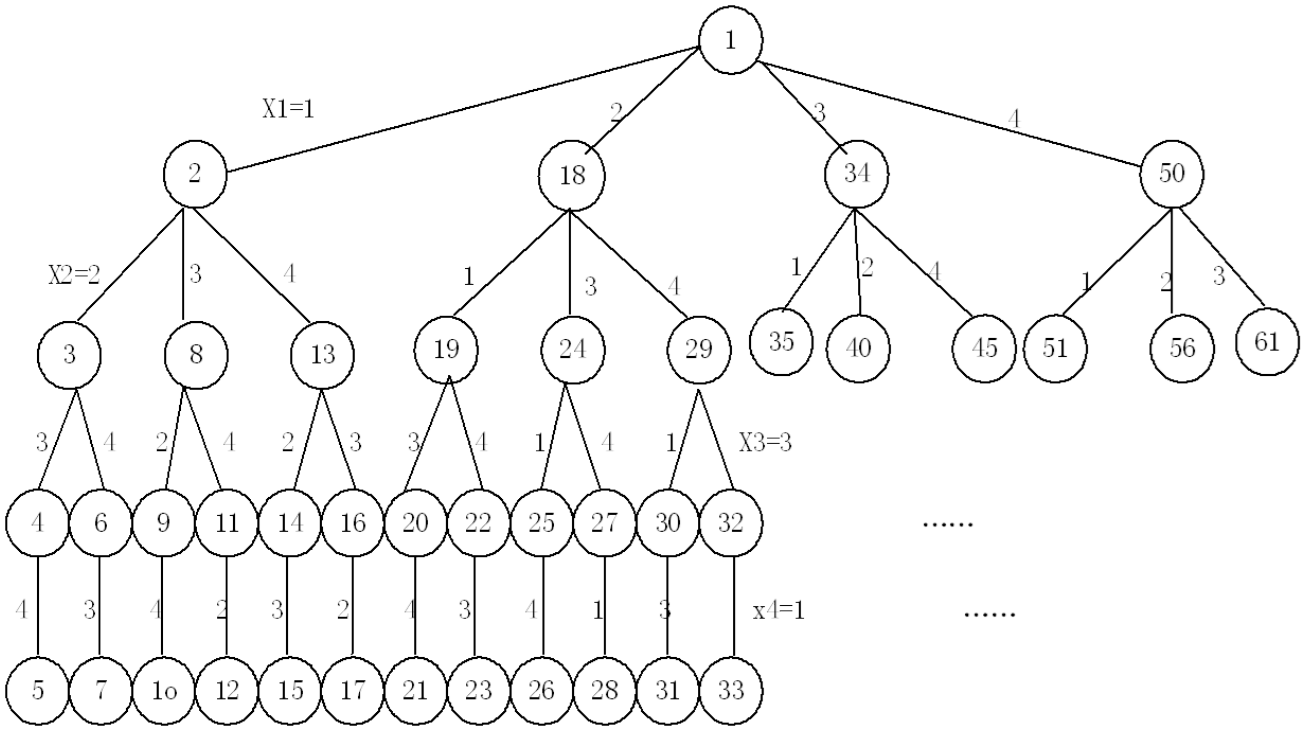
The state space of a problem is a 4-tuple (N, A, S, G) where:

- N is a set of problem states
- A is a set of arcs connecting the states
- S is a nonempty subset of N that contains start states
- G is a nonempty subset of N that contains the goal states.
- Our goal is to search solution states from S to G

State space tree

4-queens puzzle

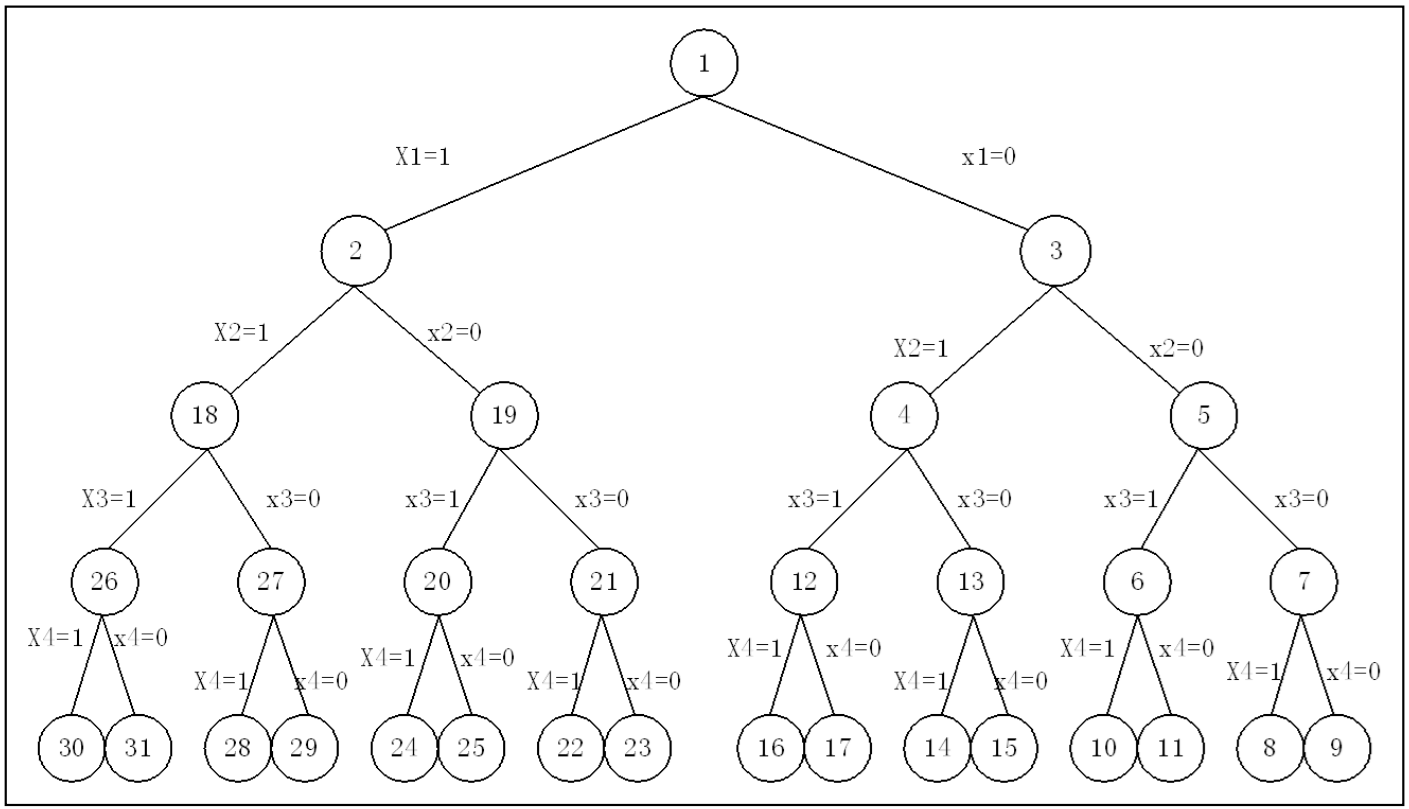
State space tree is the representation of state space in the form of tree structures



State space tree
Subset Sum

M=31, n=4, and W=(11,13,24,7)

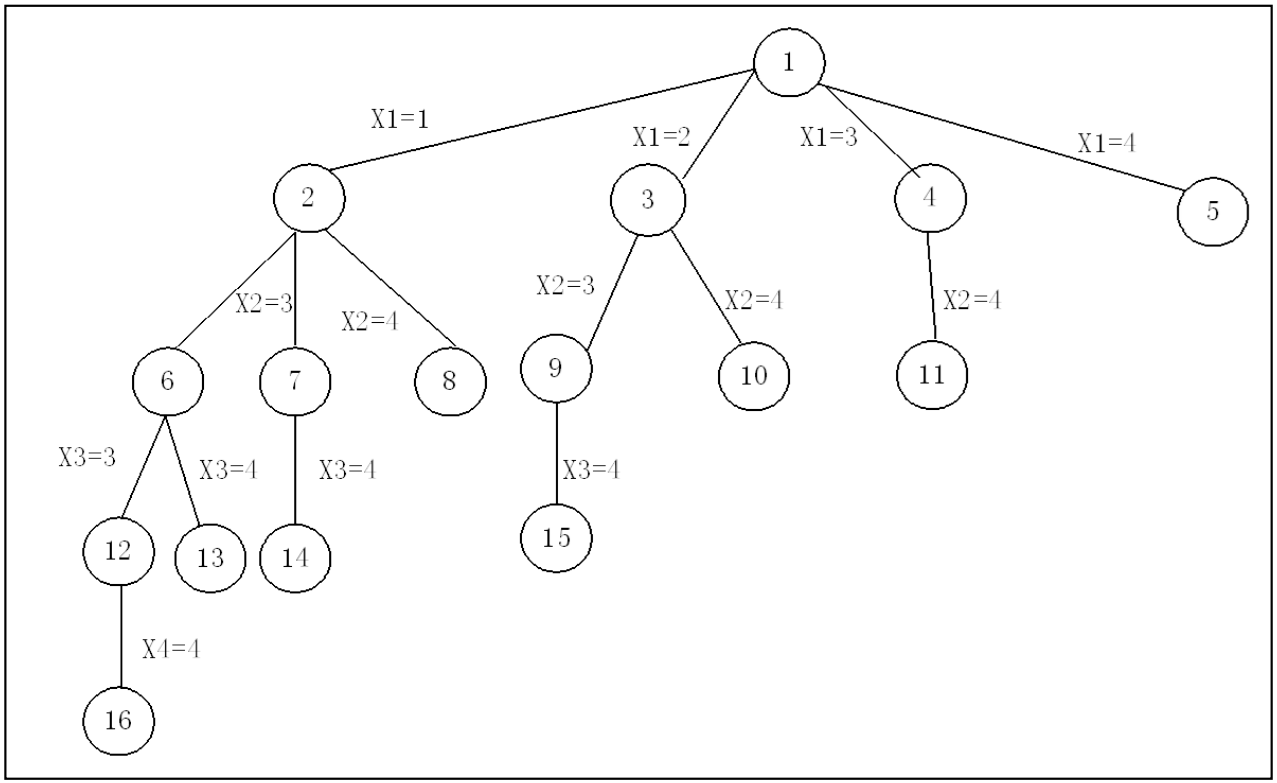
For fixed length tuple representation



State space tree
Subset Sum

M=31, n=4, and W=(11,13,24,7)

For varied length tuple representation



Searching solution space

- Searching solutions equals to traverse the state space tree
- Node has three states during expending a tree
  1. Live node: A node that has been reached, but not all of its children have been explored
  2. Died node: A node where all of its children have been explored
  3. E-Node (expansion node) A live node in which its children are currently being explored.
- Search strategy
  1. Backtrack: A variant version of depth first search with a bound function
  2. Branch-bound(Best first search): an enhancement of backtracking

Bounding is a boolean function to kill a live node

Backtrack algorithm

Algorithm 1: Backtrack Algorithm

```
1  Function backtrack(int n)
2      k=1;  while k > 0 do
3          forall the  x[k] ∈ T(X(1),···,X(k-1)) do
4              if  not B(X(1), ···, X(k)) then
5                  if (X(1), ···, X(k)) is an answer then
6                      print (X(1), ···, X(k)) ;
7                  k=k+1 /*loop next */
8              else
9                  k =k-1 /*backtrack */
```

- T(X(1),···,X(k-1)) is a set containing all possible values x(k), given X(1),···,X(k-1)
- B(X(1),···, X(k)) judge whether X(k) satisfies constrains
- Solution is store in X(1:n) , once it is decided, output it

Bounding function for
Subset sum problem

- Simple bounding: B(X(1),$\cdots$,X(k))=true iff

$$\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) < M \tag{1}$$

- Tighter bounding: B(X(1),$\cdots$,X(k))=true iff (1) and

$$\sum_{i=1}^{k} W(i)X(i) + W(k+1) > M \tag{2}$$

when sorting W(i) by non-decreasing order

Subset sum algorithm with bounding

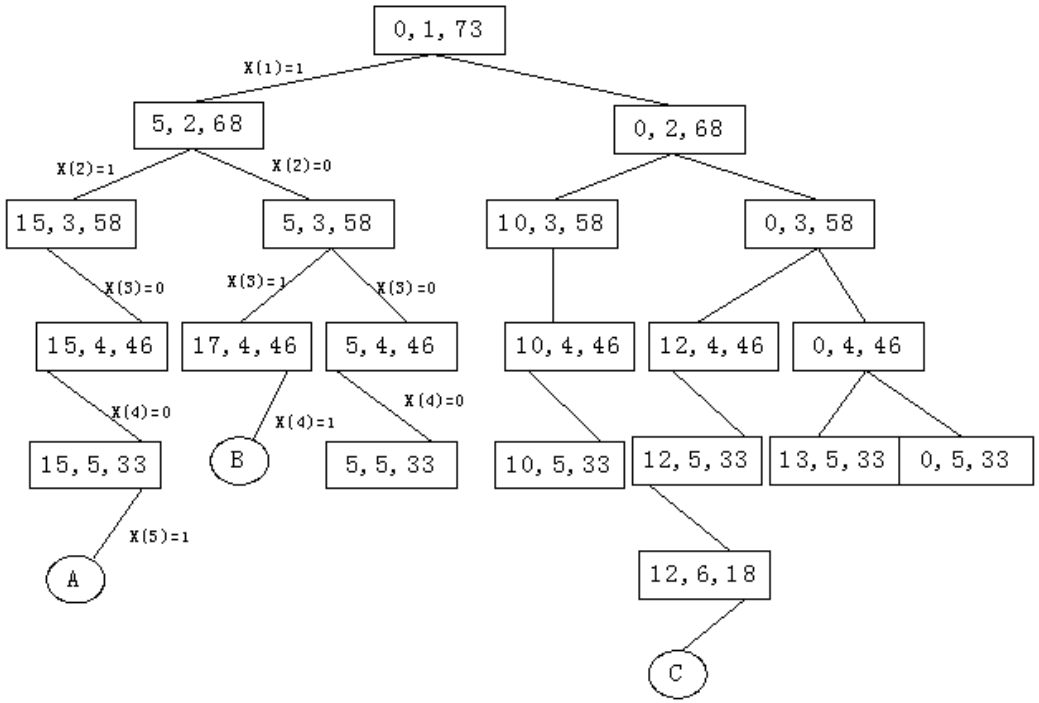
Algorithm 2: Subset sum problem: pseudo code

```
1  Let s=w(1)x(1)++w(k-1)x(k-1) ;
2      r=w(k)++w(n), assumed s+r ≥ M ;
3  Expanding left child node ;
4  if S + W(k) + W(k + 1) > M then
5  |   stop expanding ;
6  |   r=r-w(k) ;
7  |   Expanding right child node ;
8  else
9  |   x(k)=1 ;
10 |   s=s+w(k);
11 |   r=r-w(k) ;
12 |   let (x(1),,x(k)) be E-Node ;
13 Expanding right child node;
14 if s+r <M or s+w(k+1) >M then
15 |   stop expanding
16 else
17 |   x(k)=0
```

State space tree with bounding

Subset sum problem

M=30,n=6 and w=(5,10,12,13,15,18)



Numbers in a rectangle node corresponds to s, k and r values respectively
Circle nodes correspond to answer states
There are only 23 nodes, but 63 nodes without bounding

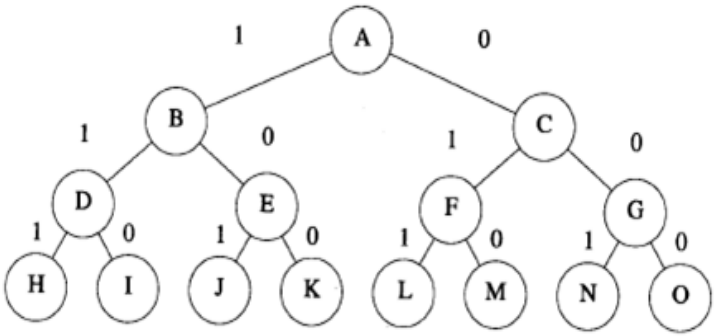
Problem statement

Container Loading Problem:CLP

- Given a ship has capacity c and n containers with weights $(w_1, \cdots, w_n)$ are available for loading.
- Aiming at loading as many containers as is possible without sinking the ship.

Using fixed length tuple $x = (x_1, \cdots, x_n)$ ($x_i = 1$, if container i is loaded) as solution space representation, an example of state space tree is shown below.

n= 4,w= [8 , 6 , 2 , 3], c1 = 12



Bound function

- Suppose node i-1 is the E-node, let

$$cw = \sum_{j=1}^{i-1} x_j w_j$$ total weight of containers loaded already

$$r = \sum_{j=i}^{n} w_j$$ total weight of unloaded containers

$bestw =$ the optimal total weight up to now

1. Bound1$(x_1, \cdots, x_i)$ =true if cw + $w_i > c$: Kill node i
2. Bound2$(x_1, \cdots, x_i)$ =true cw+r $\leq$ bestw : stop expanding node i.
3. Above two bounding functions can be used at same time

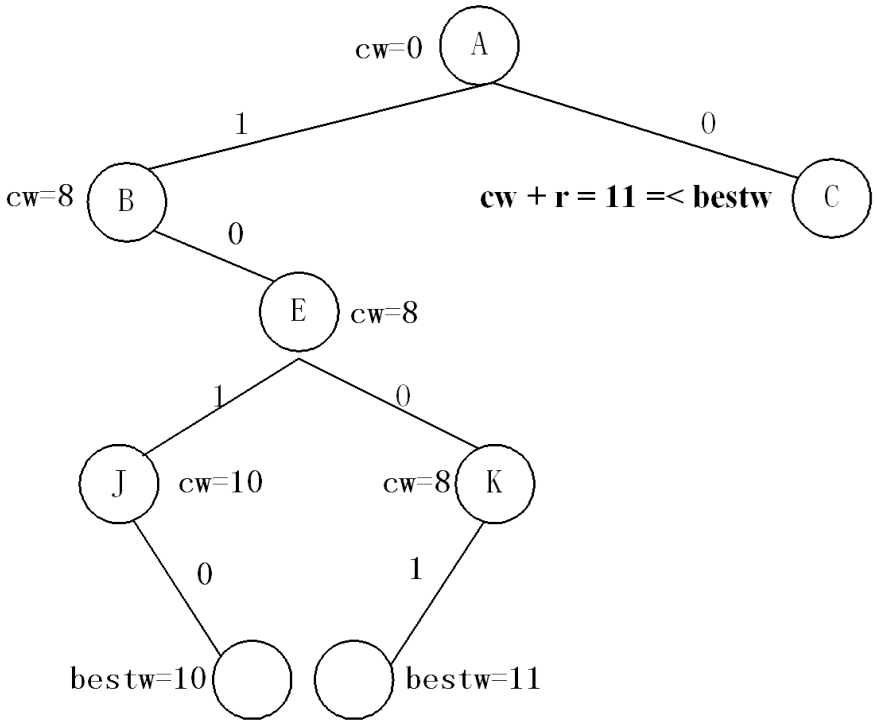
Backtrack algorithm for CLP

```
1   template<class T>
2   T MaxLoading(T w[], T c, int n, int
        bestx[])
3   {// Return best loading and its
        value.
4      Loading<T> X;
5      // initialize X
6      X.x = new int [n+1];
7      X.w = w;
8      X.c = c;
9      X.n = n;
10     X.bestx = bestx;
11     X.bestw = 0;
12     X.cw = 0;
13     // initial r is sum of all
           weights
14     X.r = 0;
15     for (int i = 1; i <= n; i++)
16        X.r += w[i];
17     X.maxLoading(1);
18     delete [] X.x;
19     return X.bestw;
20  }
```

```
1   template<class T>
2   void Loading<T>::maxLoading(int i)
3   {// Search from level i node.
4      if (i > n) {// at a leaf
5         for (int j = 1; j <= n; j++)
6            bestx[j] = x[j];
7         bestw = cw; return;}
8      // check subtrees
9      r -= w[i];
10     if (cw + w[i] <= c) {// try x[i]
           = 1
11        x[i] = 1;
12        cw += w[i];
13        maxLoading(i+1);
14        cw -= w[i];}
15     if (cw + r > bestw) {// try x[i]
           = 0
16        x[i] = 0;
17        maxLoading(i+1);}
18     r += w[i];
19  }
```

State space tree with bounding

n= 4, w= [8 , 6 , 2 , 3], c = 12



Bound function for 0/1 Knapsack

- Suppose that items are sorted in non-decreasing miner of p/w and node k-1 is the E-node, let

$$cp = \sum_{j=1}^{k-1} x_j p_j \qquad \text{profit of current packing}$$

$$rp = \sum_{j=k}^{n} p_j \qquad \text{total profit of remain items}$$

$$bestp = \qquad \text{max profit so far}$$

- Bound(x) =true if cp+rp $\leq$ bestp

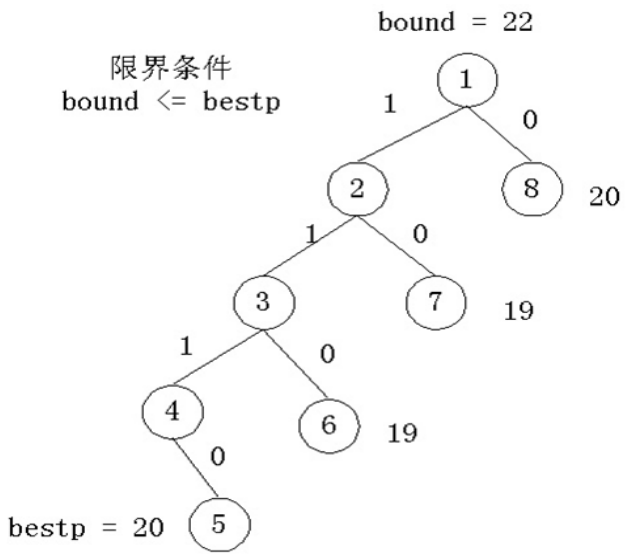
Backtrack algorithm for Knapsack

```
1   template<class Tw, class Tp>
2   void Knap<Tw, Tp>::Knapsack(int i)
3   {// Search from level i node.
4      if (i > n) {// at a leaf
5         bestp = cp;
6         return;}
7      // check subtrees
8      if (cw + w[i] <= c) {// try x[i]
           = 1
9         cw += w[i];
10        cp += p[i];
11        Knapsack(i+1);
12        cw -= w[i];
13        cp -= p[i];}
14     if (Bound(i+1) > bestp) // try x[
           i] = 0
15        Knapsack(i+1);
16  }
```

```
1   template<class Tw, class Tp>
2   Tp Knap<Tw, Tp>::Bound(int i)
3   {// Return upper bound on value of
4    // best leaf in subtree.
5      Tw cleft = c - cw;  // remaining
           capacity
6      Tp b = cp;          // profit
           bound
7      // fill remaining capacity
8      // in order of profit density
9      while (i <= n && w[i] <= cleft) {
10        cleft -= w[i];
11        b += p[i];
12        i++;
13        }
14
15     // take fraction of next object
16     if (i <= n) b += p[i]/w[i] *
           cleft;
17     return b;
18  }
```

State space tree with bounding

n= 4, c= 7, p= [9 , 10 , 7 , 4],w= [3 , 5 , 2 , 1]



Max Clique: problem statements

A subgraph $G' =< V', E' >$ is a complete subgraph of an undirected graph $G =< V, E >$, if and only if $V' \subset V$ and for $\forall u \in V', \forall v \in V'$, $(u, v) \in E' \subset E$.

A clique is a complete subgraph of G if no larger inclusion of other complete subgraphs.

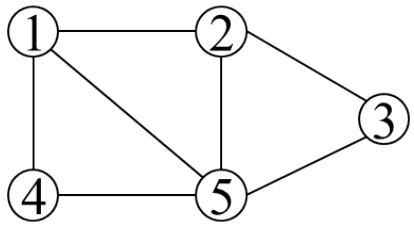
A max clique is a clique of the largest possible size in a given graph.

A indepedent vertex set is a subgraph of G with empty edges if no larger inclusion of other independent vertex sets.

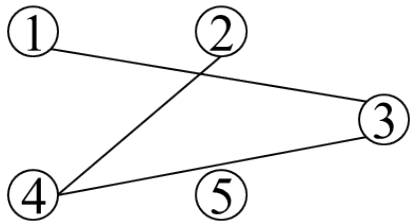
A max indepedent vertex set is a independent vertex set of the largest possible size in a given graph.

An example

{1,2} is a compete subgraph, but not a clique
{1,2,5} {1,4,5} {2,5,5} are max cliques
{2,4} is a max independent vertex set



{1,2} is a empty subgraph, but not a independent vertex set
{2,3} {1,2,5} are independent vertex sets
{1,2,5} is also a max independent vertex sets



Max clique: bounding

- Our goal is to find the max cliques of given graph G
- Using fixed length tuple X=x[1..n] (x[i]=1 if vertex i is included) to represent solution space
- Its state space tree is a subset tree
- Bounding
  1. B(x)=true if the vertexes from root to i can not form a complete subgraph
  2. B(x)=true if the number of vertexes from root to i plus remained vertexes is no larger than bestn

Backtrack algorithm for maxclique

```
1   int AdjacencyGraph::MaxClique(
          int v[])
2   {// Return size of largest
          clique.
3    // Return clique vertices in v
          [1:n].
4      // initialize for maxClique
5      x = new int [n+1];
6      cn = 0;
7      bestn = 0;
8      bestx = v;
9
10     // find max clique
11     maxClique(1);
12
13     delete [] x;
14     return bestn;
15  }
```

```
1   void AdjacencyGraph::maxClique(int i)
2   {// Backtracking code to compute largest cliq
3      if (i > n) {// at leaf
4         // found a larger clique, update
5         for (int j = 1; j <= n; j++)
6            bestx[j] = x[j];
7         bestn = cn;
8         return;}
9      // see if vertex i connected to others
10     // in current clique
11     int OK = 1;
12     for (int j = 1; j < i; j++)
13        if (x[j] && a[i][j] == NoEdge) {
14           // i not connected to j
15           OK = 0;
16           break;}
17
18     if (OK) {// try x[i] = 1
19        x[i] = 1;  // add i to clique
20        cn++;
21        maxClique(i+1);
22        x[i] = 0;
23        cn--;}
24
25     if (cn + n - i > bestn) {// try x[i] = 0
26        x[i] = 0;
27        maxClique(i+1);}
28  }
```

TSP

- Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?
- It can be modeled as an undirected weighted graph, such that cities are the graph's vertexes, paths are the graph's edges, and a path's distance is the edge's length.
- It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once.

Bounding

- Define x=x[1..n] (x_i is the order number of i-th vertex in the route) as the solution representation
- Its state space tree is a permutation tree
- Bounding
  1. B(i)=true if no edge connection between x[i] and x[i-1]
  2. B(i)=true if route distance from root to x[i] is larger than bestc (bestc is the shortest route distance so far)

Backtrack algorithm for maxclique

```
1   void AdjacencyWDigraph<T>::tSP(int i)
2   {// Backtracking code for traveling salesperson.
3      if (i == n) {// at parent of a leaf
4         // complete tour by adding last two edges
5         if (a[x[n-1]][x[n]] != NoEdge &&
6            a[x[n]][1] != NoEdge &&
7            (cc + a[x[n-1]][x[n]] + a[x[n]][1] <
                  bestc ||
8            bestc == NoEdge)) {// better tour found
9            for (int j = 1; j <= n; j++)
10              bestx[j] = x[j];
11           bestc = cc + a[x[n-1]][x[n]] + a[x[n
                  ]][1];}
12        }
13     else {// try out subtrees
14        for (int j = i; j <= n; j++)
15           // is move to subtree labeled x[j]
                  possible?
16           if (a[x[i-1]][x[j]] != NoEdge &&
17                 (cc + a[x[i-1]][x[i]] < bestc ||
18                  bestc == NoEdge)) {// yes
19              // search this subtree
20              Swap(x[i], x[j]);
21              cc += a[x[i-1]][x[i]];
22              tSP(i+1);
23              cc -= a[x[i-1]][x[i]];
24              Swap(x[i], x[j]);}
25        }
26  }
```

```
1   template<class T>
2   T AdjacencyWDigraph<T>::TSP(i
          v[])
3   {// Traveling salesperson by
          backtracking.
4    // Return cost of best tour,
          return tour in v[1:n].
5       // initialize for tSP
6       x = new int [n+1];
7       // x is identity permutati
8       for (int i = 1; i <= n; i+
9          x[i] = i;
10      bestc = NoEdge;
11      bestx = v;  // use array v
              store best tour
12      cc = 0;
13
14      // search permutations of
              [2:n]
15      tSP(2);
16
17      delete [] x;
18      return bestc;
19  }
```