EXHAUSTIVE GENERATION AND
SEARCH: BACKTRACKING
Backtracking Algorithms

Knapsack (Optimization) Problem

Instance: Profits $p_0, p_1, \ldots, p_{n-1}$
Weights $w_0, w_1, \ldots, w_{n-1}$
Knapsack capacity $M$

Find: and $n$-tuple $[x_0, x_1, \ldots, x_{n-1}] \in \{0, 1\}^n$
such that $P = \sum_{i=0}^{n-1} p_i x_i$ is maximized,
subject to $\sum_{i=0}^{n-1} w_i x_i \leq M$.

Example:

<table>
<thead>
<tr>
<th>Objects:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight (lb)</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>profit</td>
<td>$500$</td>
<td>$1,000$</td>
<td>$300$</td>
<td>$210$</td>
</tr>
</tbody>
</table>

Knapsack capacity: $M = 10$ lb.

Two feasible solutions and their profit:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>$1,500$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>$1,510$</td>
</tr>
</tbody>
</table>

This problem is NP-hard.
Naive Backtracking Algorithm for Knapsack

Examine all $2^n$ tuples and keep the ones with maximum profit.


Algorithm **KNAPSACK1** ($l$)

if ($l = n$) then
  if $\sum_{i=0}^{n-1} w_ix_i \leq M$ then
    $CurP \leftarrow \sum_{i=0}^{n-1} p_ix_i$;
    if ($CurP > OptP$) then
      $OptP \leftarrow CurP$;
      $OptX \leftarrow [x_0, x_1, \ldots, x_{n-1}]$;
    else $x_l \leftarrow 1$;
    **KNAPSACK1** ($l + 1$);
  $x_l \leftarrow 0$;
  **KNAPSACK1** ($l + 1$);

First call: $OptP \leftarrow -1$; **KNAPSACK1** (0).

Running time: $2^n$ $n$-tuples are checked, and it takes $\Theta(n)$ to check each solution. The total running time is $\Theta(n2^n)$.

Note: not all $n$-tuples are feasible but the algorithm will test all (the whole search tree is examined).

We will improve this algorithm!!!
A General Backtracking Algorithm

- Represent a solution as a list: $X = [x_0, x_1, x_2, \ldots]$.
- Each $x_i \in P_i$ (possibility set)
- Given a partial solution: $X = [x_0, x_1, \ldots, x_{l-1}]$, we can use constraints of the problem to limit the choice of $x_l$ to $C_l \subseteq P_l$ (choice set).
- By computing $C_l$ we prune the search tree, since for all $y \in P_l \setminus C_l$ the subtree rooted on $[x_0, x_1, \ldots, x_{l-1}, y]$ is not considered.

Part of the search tree for the previous Knapsack example:

<table>
<thead>
<tr>
<th>$w_i$</th>
<th>8</th>
<th>1</th>
<th>5</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>$500$</td>
<td>$1,000$</td>
<td>$300$</td>
<td>$210$</td>
</tr>
</tbody>
</table>

$M = 10$. 

\[ \text{profit} = $1,500 \]
\[ \text{profit} = $500 \]

$\times$: pruning
General Backtracking Algorithm with Pruning

Global Variables \( X = [x_0, x_1, \ldots], \ C_l, \) for \( l = 0, 1, \ldots \).

Algorithm \textsc{Backtrack} \((l)\)

if \((X = [x_0, x_1, \ldots, x_{l-1}] \) is a feasible solution) then

“Process it”

Compute \( C_l; \)

for each \( x \in C_l \) do

\( x_l \leftarrow x; \)

\textsc{Backtrack}(l + 1);
Backtracking with Pruning for Knapsack


Algorithm $\text{Knapsack2} \ (l, CurW)$

if $(l = n)$ then
    if $(\sum_{i=0}^{n-1} p_i x_i > OptP)$ then
        $OptP \leftarrow \sum_{i=0}^{n-1} p_i x_i$;
        $OptX \leftarrow [x_0, x_1, \ldots, x_{n-1}]$;
    if $(l = n)$ then $C_l \leftarrow \emptyset$
else if $(CurW + w_i \leq M)$ then
    $C_l \leftarrow \{0, 1\}$;
else $C_l \leftarrow \{0\}$;
for each $x \in C_l$ do
    $x_i \leftarrow x$;
    $\text{Knapsack2} \ (l + 1, CurW + w_i x_i)$;

First call: $\text{Knapsack2} \ (0, 0)$.
Backtracking: Generating all Cliques

**Problem:** All Cliques

**Instance:** a graph \( G = (V, E) \).

**Find:** all cliques of \( G \) without repetition

Cliques (and maximal cliques): \( \emptyset, \{0\}, \{1\}, \ldots, \{6\}, \{0, 1\}, \{0, 6\}, \{1, 2\}, \{1, 5\}, \{1, 6\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, \{5, 6\}, \{0, 1, 6\}, \{1, 5, 6\}, \{2, 3, 4\} \).

**Definitions:**

Clique in \( G(V, E) \): \( C \subseteq V \) such that for all \( x, y \in C \), \( x \neq y \), \( \{x, y\} \in E \).

Maximal clique: a clique not properly contained into another clique.
Many combinatorial problems can be reduced to finding cliques (or the largest clique):

1. Largest independent set in $G$ (stable set): is the same as largest clique in $\overline{G}$.
2. Exact cover of sets by subsets: find clique with special property.
3. Find a Steiner triple system of order $v$: find a largest clique in a special graph.
4. Find all intersecting set systems: find all cliques in a special graph.
5. Etc.
In a Backtracking algorithm:

\[ X = [x_0, x_1, \ldots, x_{l-1}] \text{ is a partial solution} \]

\[ \iff \{x_0, x_1, \ldots, x_{l-1}\} \text{ is a clique.} \]

But we don’t want ot get the same \(k\)-clique \(k!\) times:

\([0, 1] \text{ extends to } [0, 1, 6] \]
\([0, 6] \text{ extends to } [0, 6, 1] \]

So we require partial solutions for be in sorted order:

\[ x_0 < x_1 < x_2 < \ldots < x_{l-1}. \]

Let \(S_{l-1} = \{x_0, x_1, \ldots, x_{l-1}\}\) for \(X = [x_0, x_1, \ldots, x_{l-1}]\).

The **choice set** of this point is:

if \(l = 0\) then \(C_0 = V\)

if \(l > 0\) then

\[ C_l = \{v \in V \setminus S_{l-1} : v > x_{l-1} \text{ and } \{v, x\} \in E \text{ for all } x \in S_{l-1}\} \]

\[ = \{v \in C_{l-1} \setminus \{x_{l-1}\} : \{v, x_{l-1}\} \in E \text{ and } v > x_{l-1}\} \]

To compute \(C_l\), define:

\(A_v = \{u \in V : \{u, v\} \in E\}\) (vertices adjacent to \(v\))

\(B_v = \{v + 1, v + 2, \ldots, n - 1\}\) (vertices larger than \(v\))

\(C_l = A_{x_{l-1}} \cap B_{x_{l-1}} \cap C_{l-1}.\)

To **detect if a clique is maximal** (set inclusionwise):

Calculate \(N_l\), the set of vertices that can extend \(S_{l-1}\):

\(N_0 = V\)

\(N_i = N_{i-1} \cap A_{x_{l-1}}.\)

\(S_{l-1}\) is maximal \(\iff N_i = \emptyset.\)
Algorithm \textbf{ALLCLICES}(l) 
Global: $X, C_l (l = 0, \ldots, n - 1), A_l, B_l$ pre-computed.

if ($l = 0$) then output ([ ]); 
else output ([ $x_0, x_1, \ldots, x_{l-1}$ ]); 
if ($l = 0$) then $N_l \leftarrow V$; 
else $N_l \leftarrow A_{x_{l-1}} \cap N_{l-1}$; 
if ($N_l = \emptyset$) then output (“maximal”); 
if ($l = 0$) then $C_l \leftarrow V$; 
else $C_l \leftarrow A_{x_{l-1}} \cap B_{x_{l-1}} \cap C_{l-1}$; 
for each ($x \in C_l$) do 
$x_i \leftarrow x$; 
\textbf{ALLCLICES}(l + 1); 

First call: \textbf{ALLCLICES}(0).
Average Case Analysis of \textbf{ALLCLIQUES}

Let $G$ be a graph with $n$ vertices and let $c(G')$ be the number of cliques in $G$.
The running time for \textbf{ALLCLIQUES} for $G$ is in $O(nc(G'))$, since $O(n)$ is an upper bound for the running time at a node, and $c(G)$ is the number of nodes visited.

Let $\mathcal{G}_n$ be the set of all graphs on $n$ vertices.
$|\mathcal{G}_n| = 2^{\binom{n}{2}}$
(bijection between $\mathcal{G}_n$ and all subsets of the set of unordered pairs of $\{1, 2, \ldots, n\}$).

Assume the graphs in $\mathcal{G}_n$ are equally likely inputs for the algorithm (that is, assume uniform probability distribution on $\mathcal{G}_n$).
Let $T(n)$ be the average running time of \textbf{ALLCLIQUES} for graphs in $\mathcal{G}_n$.
Let $\overline{c}(n)$ be the average number of cliques in a graph in $\mathcal{G}_n$.
Then, $T(n) \in O(n\overline{c}(n))$.

So, all we need to do is estimating $\overline{c}(n)$.

$$\overline{c}(n) = \frac{\sum_{G \in \mathcal{G}_n} c(G)}{|\mathcal{G}_n|} = \frac{1}{2^{\binom{n}{2}}} \sum_{G \in \mathcal{G}_n} c(G).$$

We will show that:

$$\overline{c}(n) \leq (n + 1)n^{\log_2 n}, \text{ for } n \geq 4.$$
**Sketch of the Proof:**

Define the indicator function, for each subset \( W \subseteq V \):

\[
\chi(G, W) = \begin{cases} 
  1, & \text{if } W \text{ is a clique of } G \\
  0, & \text{otherwise}
\end{cases}
\]

Then,

\[
\overline{c}(n) = \frac{1}{\binom{n}{2}} \sum_{G \in \mathcal{G}_n} c(G) \\
= \frac{1}{\binom{n}{2}} \sum_{G \in \mathcal{G}_n} \left( \sum_{W \subseteq V} \chi(G, W) \right) \\
= \frac{1}{\binom{n}{2}} \sum_{W \subseteq V} \sum_{G \in \mathcal{G}_n} \chi(G, W)
\]

Now, for fixed \( W \), \( \sum_{G \in \mathcal{G}_n} \chi(G, W) = 2^{\left(\binom{n}{2} - (\binom{|W|}{2})\right)} \).

(\text{Number of subsets of } \binom{V}{2} \text{ containing edges of } W)

\[
\overline{c}(n) = \frac{1}{\binom{n}{2}} \sum_{W \subseteq V} 2^{\left(\binom{n}{2} - (\binom{|W|}{2})\right)} \\
= \frac{1}{\binom{n}{2}} \sum_{k=0}^{n} \binom{n}{k} 2^{\left(\binom{n}{2} - (\binom{k}{2})\right)} \\
= \sum_{k=0}^{n} \frac{n!}{k!(n-k)!} 2^{\left(\binom{n}{2} - \binom{k}{2}\right)}
\]

So, \( \overline{c}(n) = \sum_{k=0}^{n} t_k \), where \( t_k = \frac{n!}{k!(n-k)!} 2^{\left(\binom{n}{2} - \binom{k}{2}\right)} \).

A technical part of the proof bounds \( t_k \) as follows: \( t_k \leq n^{\log_2 n} \)

(see the textbook for details).

So, \( \overline{c}(n) = \sum_{k=0}^{n} t_k \leq \sum_{k=0}^{n} n^{\log_2 n} = (n + 1)n^{\log_2 n} \in O(n^{\log_2 n + 1}) \).

Thus, \( T(n) \in O(n\overline{c}(n)) \subseteq O(n^{\log_2 n + 2}) \).
Estimating the size of a Backtrack tree

State Space Tree: tree size = 10

Probing path $P_1$:
Estimated tree size: $N(P_1) = 15$

Probing path $P_2$:
Estimated tree size: $N(P_2) = 9$
Game for choosing a path (probing):
At each node of the tree, pick a child node uniformly at random.

For each leaf $L$, calculate $P(L)$, the probability that $L$ is reached.

We will prove later that the expected value of $\mathcal{N}$ of $N(L)$ turns out to be the size of the space state tree. Of course,

$$\mathcal{N} = \sum_{L \text{ leaf}} P(L)N(L) \quad \text{(by definition)}$$

In the previous example, consider $T$:

The numbers besides the nodes represent the estimated number of nodes at this level of the tree if this node is in the path to the chosen leaf.

$P(L_1) = \frac{1}{4}$, $P(L_2) = P(L_3) = \frac{1}{8}$,

$P(L_4) = P(L_5) = P(L_6) = \frac{1}{6}$

$N(L_1) = 1 + 2 + 4 = 7$

$N(L_2) = N(L_3) = 1 + 2 + 4 + 8 = 15$

$N(L_4) = N(L_5) = N(L_6) = 1 + 2 + 6 = 9$

$$\mathcal{N} = \sum_{i=1}^{6} P(L_i)N(L_i) = \frac{1}{4} \times 7 + 2 \times \left(\frac{1}{8} \times 15\right) + 3 \times \left(\frac{1}{6} \times 9\right) = 10 = |T|$$
In practice, to estimate $N$, do $k$ probes $L_1, L_2, \ldots, L_k$, and calculate the average of $N(L_i)$:

$$N_{est} = \frac{\sum_{i=1}^{k} N(L_i)}{k}$$

Each probe is performed by running the following algorithm:

Algorithm **ESTIMATEBACKTRACKSIZE()**

```
s ← 1; N ← 1; l ← 0;
Compute $C_0$;
while $C_l \neq \emptyset$ do
    $c ← |C_l|$;
    $s ← c \times s$;
    $N ← N + s$;
    $x_l ←$ a random element of $C_l$;
    Compute $C_{l+1}$ for $[x_0, x_1, \ldots, x_l]$;
    $l ← l + 1$;
return $N$;
```
In the example below, doing only 2 probes:

we get:

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
P_1: & l & C_l & c & x_l & s & N \\
\hline
0 & b, c & 2 & b & 2 & 3 & 1 \\
1 & d, e & 2 & e & 4 & 7 & 1 \\
2 & i, j & 2 & i & 8 & 15 & 2 \\
3 & \emptyset & 2 & \emptyset & 8 & 9 & 2 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
P_1: & l & C_l & c & x_l & s & N \\
\hline
0 & b, c & 2 & b & 2 & 3 & 1 \\
1 & f, g, h & 3 & g & 6 & 9 & 2 \\
\hline
\end{array}
\]

Based on these 2 probes the estimated size of the tree is:

\[
N_{est} = \frac{15 + 9}{2} = 12.
\]
Theorem.
For a state space tree $T$, let $P$ be the path probed by the algorithm \textsc{EstimateBacktrackSize}.
If $N = N(P)$ is the value returned by the algorithm, then the expected value of $N$ is $|T|$.

Proof.
Define the following function on the nodes of $T$:

$$S([x_0, x_1, \ldots, x_{l-1}]) = \begin{cases} 1, & \text{if } l = 0 \\ |C_{l-1}| \times S([x_0, x_1, \ldots, x_{l-2}]) & \text{(s} \leftarrow c \ast s \text{ in the algorithm)} \end{cases}$$

The algorithm computes: $N(P) = \sum_{Y \in P} S(Y)$.
$P = P(X)$ is a path in $T$ from root to leaf $X$, say $X = [x_0, x_1, \ldots, x_{l-1}]$.
Call $X_i = [x_0, x_1, \ldots, x_i]$.
The probability that $P(X)$ is chosen is:

$$\frac{1}{|C_0(x_0)|} \times \frac{1}{|C_1(x_1)|} \times \ldots \times \frac{1}{|C_{l-1}(x_{l-1})|} = \frac{1}{S(X)}.$$

So,

$$\overline{N} = \sum_{X \in \mathcal{L}(T)} \text{prob}(P(X)) \times N(P(X))$$

$$= \sum_{X \in \mathcal{L}(T)} \frac{1}{S(X)} \sum_{Y \in P(X)} S(Y)$$

$$= \sum_{Y \in T} \frac{\sum_{\{X \in \mathcal{L}(T): Y \in P(X)\}} S(Y)}{S(X)}$$

$$= \sum_{Y \in T} S(Y) \frac{\sum_{\{X \in \mathcal{L}(T): Y \in P(X)\}} 1}{S(X)}$$
We claim that: \[ \sum_{X \in \mathcal{L}(T): Y \in \mathcal{P}(X)} \frac{1}{S(X)} = \frac{1}{S(Y)}. \]

**Proof of the claim:**
Let \( Y \) be a non-leaf. If \( Z \) is a child of \( Y \) and \( Y \) has \( c \) children, then \( S(Z) = c \times S(Y) \).

So,
\[
\sum_{\{Z: Z \text{ is a child of } Y\}} \frac{1}{S(Z)} = c \times \frac{1}{S(Y)} = \frac{1}{S(Y)}
\]

Iterating this equation until all \( Z \)'s are leafs:
\[
\frac{1}{S(Y)} = \sum_{\{X: X \text{ is a leaf descendant of } Y\}} \frac{1}{S(X)}
\]

So the claim is proved!

Thus,
\[
\overline{N} = \sum_{Y \in T} S(Y) \sum_{\{X \in \mathcal{L}(T): Y \in \mathcal{P}(X)\}} \frac{1}{S(X)}
\]
\[
= \sum_{Y \in T} S(Y) \frac{1}{S(Y)}
\]
\[
= \sum_{Y \in T} 1 = |T|.
\]

The theorem is thus proved!
**Exact Cover**

**Problem:** Exact Cover

**Instance:** a collection $\mathcal{S}$ of subsets of $\mathcal{R} = \{0, 1, \ldots, n - 1\}$.

**Question:** Does $\mathcal{S}$ contain an exact cover of $\mathcal{R}$?

Rephrasing the question:
Does there exist $\mathcal{S}' = \{S_{x_0}, S_{x_1}, \ldots, S_{x_{l-1}}\} \subseteq \mathcal{S}$ such that every element of $\mathcal{R}$ is contained in exactly one set of $\mathcal{S}'$?

**Transforming into a clique problem:**

$\mathcal{S} = \{S_0, S_1, \ldots, S_{m-1}\}$

Define: $G(V, E)$ in the following way: $V = \{0, 1, \ldots, m - 1\}$
\[
\{i, j\} \in E \iff S_i \cap S_j = \emptyset
\]

An exact cover of $\mathcal{R}$ is a clique of $G$ that covers $\mathcal{R}$.
Good ordering on $S$ for pruning:

$S$ sorted in decreasing lexicographical ordering.

Choice set:

\[
\begin{align*}
C'_0 &= V \\
C'_l &= A_{x_{l-1}} \cap B_{x_{l-1}} \cap C'_{l-1}, \text{ if } l > 0,
\end{align*}
\]

where

\[
\begin{align*}
A_x &= \{ y \in V : S_y \cap S_x = \emptyset \} \quad (\text{vertices adjacent to } x) \\
B_x &= \{ y \in V : S_x >_{\text{lex}} S_y \}
\end{align*}
\]

Further pruning will be used to reduce $C'_l$ by removing $H_r$’s, which will be defined later.
Example: (corrected from book page 121)

<table>
<thead>
<tr>
<th>$j$</th>
<th>$S_j$</th>
<th>$\text{rank}(S_j)$</th>
<th>$A_j \cap B_j$</th>
<th>corrected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0,1,3,</td>
<td>104</td>
<td>10</td>
<td>Y</td>
</tr>
<tr>
<td>1</td>
<td>0,1,5</td>
<td>98</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0,2,4</td>
<td>84</td>
<td>7,9</td>
<td>Y</td>
</tr>
<tr>
<td>3</td>
<td>0,2,5</td>
<td>82</td>
<td>8,9,12</td>
<td>Y</td>
</tr>
<tr>
<td>4</td>
<td>0,3,6</td>
<td>73</td>
<td>5,9</td>
<td>Y</td>
</tr>
<tr>
<td>5</td>
<td>1,2,4</td>
<td>52</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1,2,6</td>
<td>49</td>
<td>11</td>
<td>Y</td>
</tr>
<tr>
<td>7</td>
<td>1,3,5</td>
<td>42</td>
<td>$\emptyset$</td>
<td>Y</td>
</tr>
<tr>
<td>8</td>
<td>1,4,6</td>
<td>37</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>32</td>
<td>10,11,12</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>2,5,6</td>
<td>19</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3,4,5</td>
<td>14</td>
<td>$\emptyset$</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3,4,6</td>
<td>13</td>
<td>$\emptyset$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_i$</td>
<td>0,1,2,3,4</td>
<td>5,6,7,8,9</td>
<td>10</td>
<td>11,12</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

The solution is: $\{3,9,12\}$
**ExactCover** $(n, S)$

Global $X, C_l, l = (0, 1, \ldots)$

Procedure **ExactCoverBT**$(l, r')$

if $(l = 0)$ then $U_0 \leftarrow \{0, 1, \ldots, n - 1\}$;

$r \leftarrow 0$;

else $U_l \leftarrow U_{l-1} \setminus S_{x_{l-1}}$;

$r \leftarrow r'$;

while $(r \notin U_l)$ and $(r < n)$ do $r \leftarrow r + 1$;

if $(r = n)$ then output $([x_0, x_1, \ldots, x_{l-1}])$.

if $(l = 0)$ then $C'_0 \leftarrow \{0, 1, \ldots, m - 1\}$;

else $C'_l \leftarrow A_{x_{l-1}} \cap B_{x_{l-1}} \cap C'_{l-1}$;

$C_l \leftarrow C'_l \cap H_r$;

for each $(x \in C_l)$ do

$x_l \leftarrow x$;

**ExactCoverBT**$(l + 1, r)$;

Main

$m \leftarrow |S|$;

Sort $S$ in decreasing lexicographic order

for $i \leftarrow 0$ to $m - 1$ do

$A_i \leftarrow \{j : S_i \cap S_j = \emptyset\}$;

$B_i \leftarrow \{i + 1, i + 2, \ldots, m - 1\}$;

for $i \leftarrow 0$ to $n - 1$ do

$H_i \leftarrow \{j : S_j \cap \{0, 1, \ldots, i\} = \{i\}\}$;

$H_n \leftarrow \emptyset$;

**ExactCoverBT**$(0, 0)$;

( $U_i$ contains the uncovered elements at level $i$.  
$r$ is the smallest uncovered in $U_i$.)

Lucia Moura
BACKTRACKING WITH BOUNDING
Backtracking with bounding

Bounding functions:

When applying backtracking for an optimization problem, we use bounding for pruning the tree.

Let us consider a maximization problem.

Let profit(\(X\)) = profit for a feasible solution \(X\).

For a partial solution \(X = [x_0, x_1, \ldots, x_{i-1}]\), define

\[
P(X) = \max \{ \text{profit}(X') : \text{for all feasible solutions } X' = [x_0, x_1, \ldots, x_{i-1}, x'_i, \ldots, x'_{n-1}] \}.
\]

A bounding function \(B\) is a real valued function defined on the nodes of the space state tree, such that for any feasible solution \(X\), \(B(X) \geq P(X)\).

\(B(X)\) is an upper bound on the profit of any feasible solution that is descendant of \(X\) in the state space tree.

If the current best solution found has value \(OptP\), then we can prune nodes \(X\) with \(B(X) \leq OptP\), since \(P(X) \leq B(X) \leq OptP\), that is, no descendant of \(X\) will improve on the current best solution.
General Backtracking with Bounding

Algorithm **BOUNDING**(l)

Global $X$, $OptP$, $OptX$, $C_i$, $l = (0, 1, \ldots)$

if ([$x_0, x_1, \ldots, x_{i-1}$] is a feasible solution) then

$P \leftarrow \text{profit}([x_0, x_1, \ldots, x_{i-1}])$;

if ($P > OptP$) then

$OptP \leftarrow P$;

$OptX \leftarrow [x_0, x_1, \ldots, x_{i-1}]$;

Compute $C_i$;

$B \leftarrow B([x_0, x_1, \ldots, x_{i-1}])$;

for each ($x \in C_i$) do

if $B \leq OptP$ then return;

$x_i \leftarrow x$;

**BOUNDING**(l + 1)
Maximum Clique Problem

**Problem:** Maximum Clique (optimization)
**Instance:** a graph $G = (V, E)$.
**Find:** a maximum clique of $G$.

This problem is NP-complete.

Example:

![Graph](image)

Maximum cliques:

$\{2, 3, 4, 5\}, \{3, 4, 5, 6\}$
Modification of **ALLCLIQEUS** in order to find the maximum clique (no bounding).

**Boldface** adds **bounding** to this algorithm.

**Algorithm MAXCLIQUE**(l)

Global: X, C_l(l = 0, ..., n - 1), A_l, B_l pre-computed.

```plaintext
if (l > OptSize) then 
    OptSize ← l;
    OptClique ← [x_0, x_1, ..., x_{l-1}];
if (l = 0) then C_l ← V;
    else C_l ← A_{x_{l-1}} \cap B_{x_{l-1}} \cap C_{l-1};
M ← B([x_0, x_1, ..., x_{l-1}]);
for each (x ∈ C_l) do
    if (M ≤ OptSize) then return;
    x_l ← x;
    MAXCLIQUE(l + 1);

Main
    OptSize ← 0;
    MAXCLIQUE(0);
    output OptClique;
```
Bounding Functions for MAXCLIQUE

Definition. Induced Subgraph
Let $G = (V, E)$ and $W \subseteq V$. The subgraph induced by $W$, $G[W]$, has vertex set $W$ and edgset: $\{ \{u, v\} \in E : u, v \in W \}$.

If we have:
- partial solution: $X = [x_0, x_1, \ldots, x_{l-1}]$ with choice set $C_l$,
- extension solution $X = [x_0, x_1, \ldots, x_{l-1}, x_l, \ldots, x_j]$,
Then $\{x_i, \ldots, x_j\}$ must be a clique in $G[C_l]$.

Let $mc(l)$ denote the size of a maximum clique in $G[C_l]$, and let $ub(l)$ be an upper bound on $mc(l)$.

Then, a general bounding function is $B(X) = l + ub[l]$.

Bound based on size of subgraph
Since $mc(l) \leq |C_l|$, we derive the bound:

$$B_1(X) = l + |C_l|.$$
Bounds based on colouring

Definition. Vertex Colouring

Let $G = (V, E)$ and $k$ a positive integer. A (vertex) $k$-colouring of $G$ is a function

$$\text{COLOR}: V \to \{0, 1, \ldots, k - 1\}$$

such that, for all $\{x, y\} \in E$, COLOR$(x) \neq$COLOR$(y)$.

Example: a 3-colouring of a graph:

\begin{center}
\begin{tikzpicture}
\node [fill] (1) at (0,0) {1};
\node (2) at (-1,-1) [fill] {2};
\node (3) at (1,-1) [fill] {3};
\node (4) at (-1,-2) {};\node (5) at (1,-2) {};\node (6) at (0.5,-1.5) {}; \node [fill] (6) at (0.5,-1.5) {6};
\draw (1) -- (2);\draw (1) -- (3);\draw (1) -- (6);\draw (2) -- (3);\draw (2) -- (4);\draw (3) -- (5);\draw (4) -- (5);\draw (4) -- (6);\draw (5) -- (6);
\end{tikzpicture}
\end{center}

- \text{\textbullet{}}: colour 0
- \text{\textcircled{}}: colour 1
- \text{\textcircled{\textbullet{}}}: colour 2

Lemma. If $G$ has a $k$-colouring, then the maximum clique of $G$ has size at most $k$.

Proof. Let $C$ be a clique. Each $x \in C$ must have a distinct colour. So, $|C| \leq k$. This is true for any clique, in particular for the maximum clique.
Finding the minimum colouring gives the best upper bound, but it is a hard problem. We will use a greedy heuristic for finding a small colouring.

Define $\text{COLOURCLASS}[h] = \{i \in V : \text{COLOUR}[i] = h\}$.

\[
\text{GREEDYCOLOUR}(G = (V, E))
\]

Global COLOUR

$k \leftarrow 0$; // colours used so far

for $i \leftarrow 0$ to $n - 1$ do

$h \leftarrow 0$;

while $(h < k)$ and $(A_i \cap \text{COLOURCLASS}[h] \neq \emptyset)$ do

$h \leftarrow h + 1$;

if $(h = k)$ then $k \leftarrow k + 1$;

$\text{COLOURCLASS}[h] \leftarrow \emptyset$;

$\text{COLOURCLASS}[h] \leftarrow \text{COLOURCLASS}[h] \cup \{i\}$;

$\text{COLOUR}[i] = h$;

return $k$;
Sampling Bound:

Statically, beforehand, run GreedyColour\((G)\), determining \(k\) and COLOUR\([x]\) for all \(x \in V\).

\[
\text{SamplingBound}(X = [x_0, x_1, \ldots, x_{l-1}])
\]

Global \(C_l\), COLOUR
return \(l + |\{\text{COLOUR}[x] : x \in C_l\}|\);

Greedy Bound:

Call GreedyColour dynamically.

\[
\text{GreedyBound}(X = [x_0, x_1, \ldots, x_{l-1}])
\]

Global \(C_l\)
\(k \leftarrow \text{GreedyColour}(G[C_l])\);
return \(l + k\);
Here I discuss the performance for random graphs, comparing the 3 bounds seen.

Please, refer to Tables 4.4 and 4.5 in the textbook.
BRANCH-AND-BOUND
The book presents branch-and-bound as a variation of backtracking in which the choice set is tried in decreasing order of bounds. However, branch-and-bound is usually a more general scheme. It often involves keeping all active nodes in a priority queue, and processing nodes with higher priority first (priority is given by upper bound).

Here is the book’s version of branch-and-bound:

Algorithm **BranchAndBound**($l$)

```plaintext
external $B()$, $\text{PROFIT}()$;
global $C_l$ ($l = 0, 1, \ldots$)
if ($[x_0, x_1, \ldots, x_{l-1}]$ is a feasible solution) then
  $P \leftarrow \text{PROFIT}([x_0, x_1, \ldots, x_{l-1}])$
  if ($P > \text{OptP}$) then
    $\text{OptP} \leftarrow P$;
    $\text{OptX} \leftarrow [x_0, x_1, \ldots, x_{l-1}]$;
  Compute $C_i$;
  $count \leftarrow 0$;
  for each ($x \in C_i$) do
    $x_i \leftarrow x$;
    $\text{nextchoice}[count] \leftarrow x$;
    $\text{nextbound}[count] \leftarrow B([x_0, x_1, \ldots, x_{l-1}, x])$;
    $count \leftarrow count + 1$;
  Sort $\text{nextchoice}$ and $\text{nextbound}$ by decreasing order of $\text{nextbound}$;
  for $i \leftarrow 0$ to $count - 1$ do
    if ($\text{nextbound}[i] \leq \text{OptP}$) then return;
    $x_i \leftarrow \text{nextchoice}[i]$;
    $\text{BranchAndBound}(l + 1)$;
```

Lucia Moura