3.8 Prove Theorem 3.12.
3.9 Find the rank of the partition \(\{1, 3, 5\}, \{2\}, \{4\}\).
3.10 Find the partition of \(\{1, \ldots, 6\}\) that has rank 153.
3.11 Prove Theorem 3.14.
3.12 Use Theorem 3.8 to prove that
\[
\sum_{n=0}^{m} S(m, n) x(x-1) \cdots (x-n+1) = x^m.
\]
3.13 Use Theorem 3.14 to prove that
\[
\sum_{n=0}^{m} s(m, n) x^n = x(x-1) \cdots (x-m+1).
\]
3.14 Use Exercises 3.12 and 3.13 to prove Theorem 3.15.
3.15 Find the tree on eight vertices having rank 126998.
3.16 Prove Lemma 3.18.
3.17 Prove that
\[
C_n = \sum_{i=0}^{n-1} C_i \cdot C_{n-i-1},
\]
where we define \(C_0 = 0\).
3.18 Prove that the number of ways of triangulating a polygon with \(n+2\) sides is equal to \(C_n\).
3.19 Compute the mountain range in \(C_6\) having rank 99.

4 Backtracking Algorithms

4.1 Introduction

A backtracking algorithm is a recursive method of building up feasible solutions to a combinatorial optimization problem one step at a time (recall that basic terminology relating to optimization problems was introduced in Section 1.3). A backtracking algorithm is an exhaustive search; that is, all feasible solutions are considered and it will thus always find the optimal solution. Pruning methods can be used to avoid considering some feasible solutions that are not optimal.

To illustrate the basic principles of backtracking, we consider the Knapsack (optimization) problem, which was presented as Problem 1.4. Recall that a problem instance consists of a list of profits, \(P = [p_0, \ldots, p_{n-1}]\); a list of weights, \(W = [w_0, \ldots, w_{n-1}]\); and a capacity, \(M\). It is required to find the maximum value of \(\sum p_i x_i\) subject to \(\sum w_i x_i \leq M\) and \(x_i \in \{0, 1\}\) for all \(i\). An \(n\)-tuple \([x_0, x_1, x_2, \ldots, x_{n-1}]\) of 0s and 1s is a feasible solution if \(\sum w_i x_i \leq M\).

One naive way to solve this problem is to try all \(2^n\) possible \(n\)-tuples of 0s and 1s. We can build up an \(n\)-tuple one coordinate at a time by first choosing a value for \(x_0\), then choosing a value for \(x_1\), etc. Backtracking provides a simple method for generating all possible \(n\)-tuples. After each \(n\)-tuple is generated it is checked for feasibility. If it is feasible, then its profit is compared to the current best solution found to that point. The current best solution is updated whenever a better feasible solution is found.

We will denote \(X = [x_0, x_1, \ldots, x_{n-1}]\) the current \(n\)-tuple being constructed, and \(CurP\) will denote its profit. \(OptX\) will denote the current optimal solution and \(OptP\) is its profit. A recursive backtracking algorithm for the Knapsack (optimization) problem is presented now.
Algorithm 4.1: KNAPSACK (ℓ)

```latex
\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{global} \(X, \text{OptP}, \text{OptX}\)
\If{\(\ell = n\)}
\State \If{\(\sum_{i=0}^{n-1} w_i x_i \leq M\)}
\State \If{\(\text{CurP} \leftarrow \sum_{i=0}^{n-1} p_i x_i\)}
\If{\(\text{CurP} > \text{OptP}\)}
\State \(\text{OptP} \leftarrow \text{CurP}\)
\State \(\text{OptX} \leftarrow [x_0, \ldots, x_{n-1}]\)
\EndIf
\EndIf
\State \(x_\ell \leftarrow 1\)
\State \text{KNAPSACK}(\ell + 1)
\Else
\State \text{KNAPSACK}(\ell + 1)
\EndIf
\EndIf
\end{algorithmic}
\end{algorithm}
```

Note that Algorithm 4.1 is invoked initially with \(\ell = 0\).

The recursive calls to Algorithm 4.1 produce a binary tree called the state space tree for the given problem instance. When \(n = 3\), this tree is given in Figure 4.1. A backtrack search performs a depth-first traversal of the state space tree.

![State space tree when \(n = 3\)](image)

Algorithm 4.1 generates the \(2^n\) binary \(n\)-tuples in reverse lexicographic order. It takes time \(\Theta(n)\) to check each solution, and so the asymptotic running time for this algorithm is \(\Theta(n2^n)\). Of course this approach is impractical for \(n > 40\), say. Notice that not all \(n\)-tuples of 0s and 1s are feasible, and a fairly simple modification to the backtracking algorithm would take this into account. This and other improvements will be considered in the remaining sections of this chapter.

### 4.2 A general backtrack algorithm

We now present a general backtrack algorithm. For many combinatorial optimization problems of interest, the (optimal) solution can be represented as a list \(X = [x_0, x_1, \ldots]\) in which each \(x_i\) is chosen from a finite possibility set, \(P_i\). The \(x_i\)s are defined one at a time, in order, as the state space tree is traversed. Hence, the backtrack algorithm considers all members of \(P_0 \times P_1 \times \cdots \times P_i\) for each \(i = 0, 1, 2, \ldots\). The length of the list \(X\) is the same as the depth of the corresponding node in the state space tree.

Given a partial solution \(X = [x_0, x_1, \ldots, x_{\ell-1}]\), the constraints for the optimization problem will restrict the possible values for \(x_\ell\) to a subset \(C_\ell \subseteq P_\ell\) that we call a choice set. The computation of the set \(C_\ell\) is referred to as pruning. If \(y \in P_\ell \setminus C_\ell\), then nodes in the subtree with root node \([x_0, x_1, \ldots, x_{\ell-1}, y]\) will not be considered by the backtracking algorithm. Thus we say that this subtree has been "pruned" from the original state space tree.

Algorithm 4.2: BACKTRACK (ℓ)

```latex
\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{global} \(X, C_\ell\) \(\ell = 0, 1, \ldots\)
\Comment: \(X = [x_0, x_1, \ldots]\)
\If{[\(x_0, x_1, \ldots, x_{\ell-1}\)] is a feasible solution}
\Comment: process it
\EndIf
\State \text{Compute} \(C_\ell\)
\For{each \(x \in C_\ell\)}
\Do{\(x_\ell \leftarrow x\)}
\State \text{BACKTRACK}(\ell + 1)
\EndDo
\EndFor
\end{algorithmic}
\end{algorithm}
```

The first step of the algorithm is to identify if the current partial solution, \(X\), is indeed a feasible solution. The operation "process it" could mean several things, e.g., save \(X\) for future use; print it out; or check to see if it is better than the best solution found so far (according to an optimality measure), as was done in Algorithm 4.1.

The second step is to construct the choice set \(C_\ell\) for the current value of \(X\). The third step is to assign every possible value in \(C_\ell\) in turn as the next coordinate, \(x_\ell\), calling the algorithm recursively each time an assignment is made.

In many problems, it may be the case that no feasible solution can possibly be extended. For example, in the Knapsack (optimization) problem, a feasible solution is an \(n\)-tuple, where \(n\) is specified in the problem instance. In this situation,
the choice set would be defined to be empty, so that no recursive calls would be made at that point. An alternative method, which accomplishes the same thing, would be to use an if-then-else construct, as was done in Algorithm 4.1.

In Algorithm 4.1, no pruning is performed. We could think of this algorithm as being an application of Algorithm 4.2 with $C_{\ell} = \{1,0\}$. In order for backtracking to have practical value, we need efficient ways of reducing the size of the choice sets $C_{\ell}$. That is, we would like to eliminate branches of the search tree that cannot lead to solutions, without actually traversing them.

For the Knapsack (optimization) problem, one simple method of pruning is to observe that we must have

$$\sum_{i=0}^{\ell-1} w_i x_i \leq M$$

for any partial solution $[x_0, x_1, \ldots, x_{\ell-1}]$. In other words, we can check partial solutions to see if the feasibility condition is satisfied. Consequently, if $\ell \leq n - 1$ and we set

$$\text{Cur}\ W = \sum_{i=0}^{\ell-1} w_i x_i,$$

then we have

$$C_{\ell} = \begin{cases} \{1,0\} & \text{if Cur}\ W + w_\ell \leq M; \\ \{0\} & \text{otherwise}. \end{cases}$$

Applying these ideas, using Algorithm 4.2 as a template, we obtain Algorithm 4.3, which is invoked with $\ell = \text{Cur}\ W = 0$.

### Algorithm 4.3: KNAPSACK2 ($\ell, \text{Cur}\ W$)

**global** $X, \text{Opt}\ X, \text{Opt}\ P, C_\ell$ ($\ell = 0, 1, \ldots$)

if $\ell = n$

if $\sum_{i=0}^{n-1} p_i x_i > \text{Opt}\ P$

then

$\text{Opt}\ P \leftarrow \sum_{i=0}^{n-1} p_i x_i$

$\text{Opt}\ X \leftarrow [x_0, \ldots, x_{n-1}]$

endif

endif

if $\ell = n$

then $C_\ell \leftarrow \emptyset$

elseif $\text{Cur}\ W + w_\ell \leq M$

then $C_\ell \leftarrow \{1,0\}$

else $C_\ell \leftarrow \{0\}$

endif

for each $x \in C_\ell$

$z_\ell \leftarrow x$

$\text{KNAPSACK2}(\ell + 1, \text{Cur}\ W + w_\ell z_\ell)$

endfor

4.3 Generating all cliques

Recall from Section 1.2 that a clique in an undirected graph $\mathcal{G} = (V, \mathcal{E})$ is a subset of vertices $S \subseteq V$ such that $\{x,y\} \in \mathcal{E}$ for all $x, y \in S$. We consider the empty set to be a clique, and $\{x\}$ is a clique for any $x \in V$. A clique is a maximal clique if it is not a subset of a larger clique. For example, in the graph given in Figure 4.2, the cliques are $\{\}$, $\{0,1\}$, $\{0,6\}$, $\{1,2\}$, $\{1,5\}$, $\{1,6\}$, $\{2,4\}$, $\{2,3\}$, $\{3,4\}$, $\{0,1,6\}$, $\{1,5,6\}$ and $\{1,3,4\}$. The maximal cliques are $\{0,1,6\}$, $\{2,4\}$, $\{1,5,6\}$ and $\{1,3,4\}$. As another example, the graph given in Figure 4.3 has 90 maximal cliques, one of which is $\{0,9,14\}$.

Many combinatorial search problems can be reduced to finding (maximal) cliques in an appropriately chosen graph. Here is an example to illustrate. There are $\binom{n}{2}$ unordered pairs that can be formed from a 6-element set $X = \{0,1,2,3,4,5\}$. Suppose we number them in lexicographic order, i.e., $0,1, \ldots, 14$. The graph given in Figure 4.3 contains an edge $xy$ if and only if the pairs corresponding to the vertices $x$ and $y$ are disjoint. A maximal clique in this graph is just a partition of $X$ into disjoint pairs. (This partition is in fact a perfect matching of the elements of $X$.)

The problem that we will study in this section is the generation of all the cliques, without repetition, in a given graph. See Problem 4.1.

#### Problem 4.1: All Cliques

**Instance:** a graph $\mathcal{G} = (V, \mathcal{E})$

**Find:** all the cliques of $\mathcal{G}$ without repetition.

To generate all of the cliques of a graph $\mathcal{G}$ by backtracking, we need to define what a partial solution is and give a method to compute the choice sets $C_\ell$. The first part is easy: a sequence $X = [x_0, x_1, \ldots, x_{\ell-1}]$ of vertices is a partial solution if and only if $\{x_0, x_1, \ldots, x_{\ell-1}\}$ is a clique. Now, denote $S_{\ell-1} = \{x_0, \ldots, x_{\ell-1}\}$ and $C_0 = V$. Then the choice set $C_\ell$ is given by

$$C_\ell = \{v \in V \setminus S_{\ell-1} : \{v,x\} \in \mathcal{E} \text{ for each } x \in S_{\ell-1}\}$$

FIGURE 4.2
A graph with four maximal cliques.
Unfortunately, if a clique has size $k$, then an algorithm based on the above choice function will generate it $k!$ times, once for each possible ordering of its vertices. To avoid this duplication of work, we arbitrarily place a total ordering "<" on the vertices $V$, and list them according to this ordering. That is, we think of $V$ as an ordered list, i.e., we write $V = [v_0, v_1, v_2, \ldots, v_{n-1}]$, where $v_0 < v_1 < \cdots < v_{n-1}$.

Then, we redefine $C_t$ as follows:

$$C_t = \{ v \in C_{t-1} : \{ v, x_{t-1} \} \in E \text{ and } v > x_{t-1} \}.$$ 

These choice functions can be more efficiently computed if we define, for each vertex $v \in V$, the auxiliary sets

$$A_v = \{ u \in V : \{ u, v \} \in E \}$$

and

$$B_v = \{ u \in V : u > v \}.$$ 

These can be precomputed before the backtracking algorithm begins. Now, we have

$$C_t = A_{x_{t-1}} \cap B_{x_{t-1}} \cap C_{t-1}.$$ 

(4.3)

Suppose we also define

$$N_t = N_{t-1} \cap A_{x_{t-1}},$$

where $N_0 = V$. Then $X = [x_0, \ldots, x_{t-1}]$ is a maximal clique if and only if $N_t = \emptyset$. The backtrack algorithm, Algorithm 4.4, generates each clique exactly once, and identifies the maximal cliques on the fly.

In Example 4.1, we illustrate the application of Algorithm 4.4 on a small graph. This graph contains five maximal cliques (which are marked with a star), and 21 cliques in total.

Example 4.1 Finding all the cliques in a graph
Algorithm 4.4: ALLCLIQUES(ℓ).

1. Global X, C_ℓ (ℓ = 0, 1, ..., n - 1)
2. Comment: every X generated by the algorithm is a clique
3. If ℓ = 0
   - Then output ([X])
   - Else output ([X_0, ..., X_{ℓ-1}])
4. If ℓ = 0
   - Then X_ℓ ← V
   - Else X_ℓ ← A_{X_{ℓ-1}} \cap N_{X_{ℓ-1}}
5. If N_ℓ = \emptyset
   - Then \{X_0, ..., X_{ℓ-1}\} is a maximal clique
6. Else
   - Then C_ℓ ← X_ℓ
   - Else C_ℓ ← C_{X_{ℓ-1}} \cap B_{X_{ℓ-1}} \cap C_{ℓ-1}
7. For each x ∈ C_ℓ
   - Do \{ALLCLIQUES(ℓ + 1)\}

4.3.1 Average-case analysis

In this section, we determine the average-case complexity of Algorithm 4.4.

Let n be a positive integer, and let G(n) denote the set of all graphs on vertex set V = {0, ..., n - 1}. There are \(2^\binom{n}{2}\) graphs in G(n), because any unordered pair \{(x, y) \in V\} can either be included as an edge or left out.

For a graph G ∈ G(n), define c(G) to be the number of cliques in the graph G. When Algorithm 4.4 is run on the graph G, the number of nodes in the state space tree is precisely c(G). The running time of Algorithm 4.4 is easily seen to be \(O(n c(G))\). Of course, this running time depends on the particular graph G, and so can vary greatly. For example, if G is the graph containing no edges, then c(G) = n + 1, because the only cliques have size zero or one. At the other extreme, c(K_n) = 2^n because any subset of vertices in a complete graph is a clique. It is because of this variation that we will do an average-case analysis.

We will be looking at the average value of c(G), where the average is computed over all graphs in G(n). Thus we define

\[
\bar{c}(n) = \frac{1}{2^\binom{n}{2}} \sum_{G \in G(n)} c(G).
\]

We now obtain an explicit formula for \(\bar{c}(n)\). Suppose G ∈ G(n) and \(W \subseteq V\). Define the indicator function

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

Then we have that

\[
c(G) = \sum_{W \subseteq V} \chi(G, W),
\]

and hence

\[
\bar{c}(n) = \frac{1}{2^\binom{n}{2}} \sum_{G \in G(n)} \sum_{W \subseteq V} \chi(G, W) = \frac{1}{2^\binom{n}{2}} \sum_{W \subseteq V} \sum_{G \in G(n)} \chi(G, W).
\]

For a given subset of vertices \(W \subseteq V\), W is a clique of a graph G if and only if the \(\binom{|W|}{2}\) pairs of vertices in W are all edges of G. There are \(\binom{n}{2} - \binom{|W|}{2}\) remaining possible edges, and so it follows that there are exactly

\[
2^\binom{n}{2} - \binom{|W|}{2}
\]

graphs G ∈ G(n) in which W is a clique. Hence we have that

\[
\sum_{G \in G(n)} \chi(G, W) = 2^\binom{n}{2} - \binom{|W|}{2}.
\]

Thus, we obtain the following expression for \(\bar{c}(n)\):

\[
\bar{c}(n) = \frac{1}{2^\binom{n}{2}} \sum_{W \subseteq V} 2^\binom{n}{2} - \binom{|W|}{2}.
\]

Now, for any integer k such that 0 ≤ k ≤ n, there are precisely \(\binom{n}{k}\) subsets of vertices \(W \subseteq V\) with |W| = k. Hence, we have the following formula.

\[
c(n) = \frac{1}{2^\binom{n}{2}} \sum_{k=0}^{n} \binom{n}{k} 2^\binom{n}{2} - \binom{k}{2} = \sum_{k=0}^{n} \binom{n}{k} 2^{-\binom{k}{2}}.
\]

We tabulate some values of \(\bar{c}(n)\) in Table 4.1.

Having derived an explicit formula for \(\bar{c}(n)\), it is natural to ask how \(\bar{c}(n)\) behaves as a function of n. Since \(\bar{c}(n)\) is expressed as a sum, this will require some further analysis.

Define

\[
t_k = \binom{n}{k} 2^{-\binom{k}{2}},
\]

then

\[
\bar{c}(n) = \sum_{k=0}^{n} t_k.
\]
TABLE 4.1
Average number of nodes in the state space tree for Algorithm 4.4

<table>
<thead>
<tr>
<th>n</th>
<th>(\bar{c}(n))</th>
<th>n</th>
<th>(\bar{c}(n))</th>
<th>n</th>
<th>(\bar{c}(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.0</td>
<td>10</td>
<td>52</td>
<td>110</td>
<td>321948</td>
</tr>
<tr>
<td>2</td>
<td>3.5</td>
<td>20</td>
<td>351</td>
<td>120</td>
<td>496385</td>
</tr>
<tr>
<td>3</td>
<td>5.6</td>
<td>30</td>
<td>1342</td>
<td>130</td>
<td>744800</td>
</tr>
<tr>
<td>4</td>
<td>8.5</td>
<td>40</td>
<td>3863</td>
<td>140</td>
<td>1091392</td>
</tr>
<tr>
<td>5</td>
<td>12.3</td>
<td>50</td>
<td>9316</td>
<td>150</td>
<td>1566330</td>
</tr>
<tr>
<td>6</td>
<td>17.2</td>
<td>60</td>
<td>19898</td>
<td>160</td>
<td>2206835</td>
</tr>
<tr>
<td>7</td>
<td>23.4</td>
<td>70</td>
<td>38876</td>
<td>170</td>
<td>3058400</td>
</tr>
<tr>
<td>8</td>
<td>31.1</td>
<td>80</td>
<td>70916</td>
<td>180</td>
<td>4176150</td>
</tr>
<tr>
<td>9</td>
<td>40.6</td>
<td>90</td>
<td>122485</td>
<td>190</td>
<td>5626373</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>202314</td>
<td>200</td>
<td>7488221</td>
</tr>
</tbody>
</table>

Consider the ratio of successive terms in the sequence \(t_0, t_1, \ldots, t_n\). Straightforward (but messy!) algebra shows that

\[
\frac{t_k}{t_{k-1}} = \frac{n - k + 1}{k^{2k-1}}.
\]

Hence, we see that \(t_k \geq t_{k-1}\) if and only if \(n \geq k - 1 + k^{2k-1}\). Since the function \(f(k) = k - 1 + k^{2k-1}\) is a strictly increasing function of \(k\), it follows that the sequence \([t_0, t_1, \ldots, t_n]\) is unimodular, i.e., there exists an index \(\ell\) such that

\[
t_0 \leq \ldots \leq t_{\ell-1} \leq t_{\ell} \geq t_{\ell+1} \geq \ldots \geq t_n.
\]

Then it is obvious that

\[
\bar{c}(n) \leq (n + 1)t_{\ell}.
\]

Now, we obtain an upper bound on the value \(\ell\). First, observe that

\[
f(\log_2 n) = \log_2 n - 1 + \frac{n \log_2 n}{2}.
\]

When \(n \geq 4\), we have that \(\log_2 n \geq 2\), and hence

\[
f(\log_2 n) \geq n + 1 > n.
\]

The inequality \(f(\log_2 n) > n\) implies that \(t_k < t_{k-1}\) when \(k \geq \log_2 n\). This establishes that \(\ell \leq \log_2 n\) if \(n \geq 4\).

Now, we consider the term \(t_{\ell}\):

\[
t_{\ell} = \left(\frac{n}{\ell}\right)^{2^{-\ell}} \leq \left(\frac{n \ell}{\ell}\right)^{2^{-\ell}} \leq n^{\log_2 n}.
\]

Thus we have shown that \(\bar{c}(n) \leq (n + 1)n^{\log_2 n}\) for \(n \geq 4\). From this, it follows that \(\bar{c}(n) = O(n^{\log_2 n + 1})\), and the average-case running time of Algorithm 4.4 is \(O(n^{\log_2 n + 2})\).

### 4.4 Estimating the size of a backtrack tree

In this section, we present an algorithmic method to estimate the number of nodes in a state space tree \(T\) for a backtracking algorithm. The algorithm will provide a quick way of getting a rough estimate on the number \(|T|\) of nodes in \(T\) without running the entire backtracking algorithm. It is therefore a useful method to predict how long a big backtrack search might take to finish.

To motivate the algorithm, we first consider a special case. Suppose \(n\) is a positive integer, and \(c_0, c_1, \ldots, c_{n-1}\) are also positive integers. Suppose that \(T\) is a tree of depth \(n\) in which every node of depth \(i\) has \(c_i\) children, for \(0 \leq i \leq n - 1\). Equivalently, for \(0 \leq i \leq n - 1\) and for any partial solution \(X = [x_0, \ldots, x_{i-1}]\), there are \(c_i\) choices for \(x_i\). The leaf nodes in \(T\) correspond to solutions \([x_0, \ldots, x_{n-1}]\). In this particular tree, it is easy to see that, for each \(i = 0, 1, \ldots, n\), the number of nodes at depth \(i\) is \(c_0 c_1 \cdots c_{i-1}\). Thus, the number of nodes in \(T\) is given by the following equation:

\[
|T| = 1 + c_0 + c_0 c_1 + c_0 c_1 c_2 + \cdots + c_0 c_1 c_2 \cdots c_{n-1}. \tag{4.4}
\]

In general, a state space tree \(T\) will not have such a regular structure. We will obtain an estimate of the number of nodes in \(T\) by probing a random path through \(T\) from the root node to a leaf node. As we follow this path, we compute a quantity analogous to Equation (4.4), in which the \(c_i\)'s are replaced by the number of choices available at the nodes in the given path. The algorithm is presented as Algorithm 4.5.

For any given state space tree \(T\), Algorithm 4.5 will return a value \(N = N(P)\) which depends on the path \(P\) that is probed. The value \(N(P)\) is an estimate of \(|T|\). It may be larger or smaller than \(|T|\). (We can increase the accuracy of the estimate by running the algorithm several times, and computing the average of the values \(N(P)\) over the various runs. In this way, we would expect to obtain a more accurate estimate of \(|T|\).) We will now show that the expected value of \(N\) is in fact equal to \(|T|\).

Define the following function on the nodes of \(T\):

\[
S([x_0, \ldots, x_{\ell-1}]) = \begin{cases} 1 & \text{if } \ell = 0 \\ |C_{\ell-1}([x_0, \ldots, x_{\ell-2}])| \cdot S([x_0, \ldots, x_{\ell-2}]) & \text{if } \ell \geq 1. \end{cases}
\]

Thus, if \(X\) is the root node, then \(S(X) = 1\). Further, if there are \(c\) choices available at a given node \(X\), then \(S(Y) = c S(X)\) for all children \(Y\) of \(X\). Observe that, if \(P\) is the path probed in Algorithm 4.5, then

\[
N(P) = \sum_{Y \in P} S(Y).
\]
Algorithm 4.5: ESTIMATEBACKTRACK()

global \( N, m, C_t \) \( (t = 0, 1, \ldots) \)

procedure PROBE(\( \ell \))

Compute \( C_t \) for the node \([x_0, \ldots, x_{\ell-1}]\)

c \( \leftarrow |C_t| \)

if \( c \neq 0 \)

then

\[
\begin{cases}
  m \leftarrow m \cdot c \\
  N \leftarrow N + m \\
  x_\ell \leftarrow \text{a random element of } C_t \\
  \text{PROBE}(\ell + 1)
\end{cases}
\]

main

\( N \leftarrow 1 \)

\( m \leftarrow 1 \)

\( \text{PROBE}(0) \)

return \( (N) \)

In Example 4.2, we present a small tree and the \( S \)-values at each node. In this
tree, there are six leaf nodes. The corresponding six paths from the root node to a
leaf node have values \( N(P) = 7, 15, 15, 9, 9 \text{ and } 9 \), and the paths are chosen with
probabilities \( 1/6, 1/8, 1/8, 1/6, 1/6 \text{ and } 1/6 \), respectively. Example 4.2 shows
that the expected value of \( N(P) \) is 10, which is equal to the number of nodes in the
tree.

**Example 4.2** A state space tree

![State space tree diagram]

\[
\mathcal{N} = \frac{1}{4} \cdot 7 + \frac{1}{8} \cdot 15 + \frac{1}{8} \cdot 15 + \frac{1}{6} \cdot 9 + \frac{1}{6} \cdot 9 + \frac{1}{6} \cdot 9 = 10
\]

We now state and prove the main theorem of this section.

**THEOREM 4.1** For any given state space tree \( T \), let \( P \) be the path probed by
Algorithm 4.5. If \( N = N(P) \) is the value returned by Algorithm 4.5, then the
expected value of \( N \) is \( |T| \).

**PROOF** For any leaf node \( X = [x_0, \ldots, x_{\ell-1}] \in T \), there is a unique path, say
\( P(X) \), from the root node to \( X \). We will denote the nodes in the path \( P(X) \)
by \( X_0 = [] \) (the root node), \( X_1 = [x_0], \ldots, X_\ell = [x_0, \ldots, x_{\ell-1}] \). The probability that \( X_1 \) is chosen in Algorithm 4.5 when \( \ell = 0 \) is \( 1/|C_0(X_0)| \). Then,
the probability that \( X_2 \) is chosen when \( \ell = 1 \) is \( 1/|C_1(X_1)| \). In general, the
probability that \( X_\ell \) is chosen, given that \( X_0, \ldots, X_{\ell-1} \) have already been chosen,
is \( 1/|C_{\ell-1}(X_{\ell-1})| \). Therefore, the probability that \( P(X) \) is the path chosen in
Algorithm 4.5 is

\[
\frac{1}{|C_0(X_0)|} \cdot \frac{1}{|C_1(X_1)|} \cdots \frac{1}{|C_{\ell-1}(X_{\ell-1})|} = \frac{1}{S(X)}.
\]

Now we can proceed to compute the expected value of \( N \). An expected value is
actually a weighted average. In this case, we are computing the weighted average of
the values of \( N(P(X)) \), over all leaf nodes \( X \), where the path \( P(X) \) is chosen
with probability \( 1/S(X) \). Let \( \mathcal{L}(T) \) denote the set of leaf nodes in the tree \( T \). The
desired weighted average, which we denote by \( \bar{N} \), is computed as follows:

\[
\bar{N} = \sum_{X \in \mathcal{L}(T)} \left( \text{prob}(P(X)) \cdot N(P(X)) \right) = \sum_{X \in \mathcal{L}(T)} \sum_{Y \in P(X)} \frac{1}{S(Y)} \sum_{Y \in P(X)} S(Y).
\]

We can interchange the order of the two summations in the above, obtaining

\[
\bar{N} = \sum_{Y \in T} \left( \sum_{\{X \in \mathcal{L}(T) : Y \in P(X)\}} \frac{S(Y)}{S(X)} \right).
\]

Now, for any \( Y \in T \), let's evaluate the inner sum,

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

For any non-leaf node \( Y \) in the tree, it is clear from the definition of the function
\( S \) that

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

Iterating this equation, we see that

\[
\frac{1}{S(Y)} = \sum_{\{X : X \text{ is a leaf node that is a descendant of } Y\}} \frac{1}{S(X)}.
\]
This is equivalent to saying that
\[ \sum_{(X \in \mathcal{P}(T): Y \in \mathcal{P}(X))} \frac{1}{S(X)} = \frac{1}{S(T)}. \]
Note that this equation also holds if \( Y \) is a leaf node. Hence, for any node \( Y \) in the tree, the sum in Equation (4.5) has the value 1. Thus
\[ \mathcal{N} = \sum_{Y \in T} 1 = |T|, \]
which is the result we wanted to prove.

### 4.5 Exact cover

In this section, we study a problem similar to the All Clique problem that is called Exact Cover.

<table>
<thead>
<tr>
<th>Problem 4.2: Exact Cover</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Instance:</strong></td>
</tr>
<tr>
<td><strong>Question:</strong></td>
</tr>
</tbody>
</table>

Let \( S = \{S_0, S_1, S_2, \ldots, S_{m - 1}\} \) be the collection of subsets of \( \mathcal{R} \) in an instance of Problem 4.2. Observe that a solution to this problem is a list of subsets from \( S \) whose members partition \( \mathcal{R} \). Instead of storing the chosen subsets, we will keep track of them using an array \( X = [x_0, x_1, \ldots, x_{\ell - 1}] \) of their indices. Thus, for example, \( x_3 = 22 \) will mean that the set \( S_{22} \in S \) was chosen as the third set in the (partial) solution represented by the array \( X \).

Let \( \mathcal{G} \) be the graph with vertex set \( V = \{0, 1, 2, \ldots, m - 1\} \) in which two vertices \( i \) and \( j \) are adjacent if and only if \( S_i \cap S_j = \emptyset \). Then the partial solutions for Exact Cover are precisely the cliques in the graph \( \mathcal{G} \). We can use Algorithm 4.4 to generate all the cliques in \( \mathcal{G} \). Furthermore, we can check each maximal clique to see if it corresponds to a partition of \( \mathcal{R} \), and thus solve Problem 4.2.

In order to use Algorithm 4.4, we first impose an ordering on the members of \( S \). Although any ordering can be used, it is beneficial to order the subsets of \( \mathcal{R} \) in decreasing lexicographic order, which we denote by \( > \). This ordering is chosen because it is useful for pruning.

We now describe how our choice sets can be determined. First, we define
\[ C_0 = \mathcal{V} \]
and
\[ C_{\ell} = A_{x_{\ell - 1}} \cap B_{x_{\ell - 1}} \cap C_{\ell - 1}, \]
where
\[ A_x = \{y \in \mathcal{V} : S_y \cap S_x = \emptyset\}, \]
and
\[ B_x = \{y \in \mathcal{V} : S_y \text{ lex } S_x\}, \]
for \( x = 0, 1, 2, \ldots, m - 1 \). If the sets \( C_{\ell} \) are used as choice sets, then the resulting backtracking algorithm is the same as Algorithm 4.4 for solving All Clique (see Equations 4.1, 4.2, and 4.3). However, we can take advantage of the structure of the set system \((\mathcal{R}, S)\) to further prune the search, as follows. Recall that we assumed that the sets in \( S \) are sorted in decreasing lexicographic order, so
\[ S_0 \geq S_1 \geq \ldots \geq S_{m - 1}. \]
Then the largest sets with respect to this ordering will all contain 0. The next bunch will not contain 0 but will contain 1; the next bunch will contain 2 but neither 1 nor 0, and so on. Thus, the ordering of these sets defines an ordered partition
\[ \mathcal{H} = [H_0, H_1, \ldots, H_{n - 1}] \]
of \( \{0, 1, 2, \ldots, m - 1\} \), in which we have
\[ H_i = \{x \in \mathcal{V} : S_x \cap \{0, \ldots, i\} = \{i\}\}, \]
for \( i = 0, 1, \ldots, n - 1 \).

Now, suppose \( X = [x_0, x_1, \ldots, x_{t - 1}] \) is a partial solution that we wish to extend, if possible. If \( X \) is not itself a solution, then the set
\[ U_t = \mathcal{R} \setminus \bigcup_{i=0}^{t - 1} S_{x_i} \]
is not the empty set. In this case, let \( r \) be the smallest integer in \( U_t \). If it is possible to extend \( X \) to a solution, say \([x_0; x_1, \ldots, x_{t - 1}, x_t, \ldots]\), then it must be the case that \( x_t \in H_r \). We can use this observation to facilitate further pruning, by defining the (improved) choice set \( C_{\ell} \) as follows:
\[ C_{\ell} = C_{\ell} \cap H_r. \]
In order to implement these modifications, we proceed as follows. First, before running the backtrack algorithm, we sort \( \mathcal{S} \) as described above. This can be done in \( O(m \log m) \) set operations. Then the sets \( A_x \) and \( B_x \), and the partition \( \mathcal{H} \), can be constructed easily with one pass through \( \mathcal{S} \). The time to construct them is \( O(m^2) \) set operations. The resulting algorithm is presented as Algorithm 4.6. An example, including the state space tree that results when this algorithm is run, is given in Example 4.3.

**Algorithm 4.6: EXACTCOVER \((n, \mathcal{S})\)**

```plaintext
global \( X, C_r \) \((\ell = 0, 1, \ldots)\)
procedure EXACTCOVERBT(\( \ell, r' \))
  if \( \ell = 0 \)
    then \( \{ U_0 \leftarrow \{0, \ldots, n-1\} \) 
    \( \{ r \leftarrow 0 \) 
    \( \{ U_{\ell} \leftarrow U_{\ell-1} \setminus \mathcal{S}_{x_{\ell-1}} \) 
    \( r \leftarrow r' \) 
  else \( \) while \( r \notin U_\ell \) and \( r < n \)
    do \( r \leftarrow r + 1 \)
  if \( r = n \)
    then output \( ([x_0, \ldots, x_{\ell-1}]) \)
  if \( \ell = 0 \)
    then \( C_0' \leftarrow \{0, 1, \ldots, m-1\} \)
  else \( C_\ell' \leftarrow A_{x_{\ell-1}} \cap B_{x_{\ell-1}} \cap C_\ell' \)
    \( C_\ell \leftarrow C_\ell' \cap \mathcal{H}_r \)
  for each \( x \in C_\ell \)
    do \( x_{\ell} \leftarrow x \)
    \{ EXACTCOVERBT(\( \ell + 1, r \) \)

main
\( m \leftarrow |S| \)
sort \( S \) in decreasing lexicographic order
for \( i \leftarrow 0 \) to \( m - 1 \)
  do \( A_i \leftarrow \{ j : S_i \cap S_j = \emptyset \} \)
for \( i \leftarrow 0 \) to \( m - 1 \)
  do \( B_i \leftarrow \{ i+1, i+2, \ldots, m-1 \} \)
for \( i \leftarrow 0 \) to \( n - 1 \)
  do \( H_i \leftarrow \{ j : S_j \cap \{0, \ldots, i\} = \{i\} \} \)
\( H_n \leftarrow \emptyset \)
EXACTCOVERBT(0, 0)
```

**Example 4.3** An instance of Exact Cover

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

| \( i \) | \( H_i \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 0 | \{0, 1, 2, 3, 4\} | \{5, 6, 7, 8, 9\} | \{10\} | \{11, 12\} | \emptyset | \emptyset | \emptyset |

![State space tree](image_url)
4.6 Bounding functions

A more sophisticated method of pruning is to use a bounding function. We require a few preliminary definitions, which will apply to any backtracking algorithm for a maximization problem. Let profit\( (X) \) denote the profit for any feasible solution \( X \). For a partial feasible solution, say \( X = [x_0, x_1, \ldots, x_{t-1}] \), define \( P(X) \) to be the maximum profit of any feasible solution which is a descendant of \( X \) in the state space tree. In other words, \( P(X) \) is the maximum value of profit\( (X') \) taken over all feasible solutions \( X' = [x_0', x_1', \ldots, x_{t-1}'] \) such that \( x_i = x_i' \) for \( 0 \leq i \leq \ell - 1 \). It follows from the definition that, if \( X = [\ ] \), then \( P(X) \) is the optimal profit of the given problem instance.

In general, \( P(X) \) can be computed exactly only by traversing the subtree with root node \( X \). In order to avoid doing this, if possible, we will employ a bounding function. A bounding function is a real-valued function \( B \), defined on the set of nodes in the state space tree, satisfying the following condition:

For any feasible partial solution \( X \), \( B(X) \geq P(X) \).

This property says that \( B(X) \) is an upper bound on the profit of any feasible solution that is a descendant of \( X \) in the state space tree. (For a minimization problem, the definition is the same, except that the inequality is reversed.)

Once a bounding function \( B(X) \) has been specified, it can be used to prune the state tree, as follows. Suppose that at some stage of the backtracking algorithm, we have a current partial solution \( X = [x_0, x_1, \ldots, x_{t-1}] \), and \( \text{Opt}(P) \) is the current optimal profit. If it happens that \( B(X) \leq \text{Opt}(P) \), then we have that

\[
P(X) \leq B(X) \leq \text{Opt}(P) \,.
\]

This means that no descendants of \( X \) in the state space tree can improve the current optimal profit. Hence we can prune this entire subtree, i.e., we can define \( C_\ell \) to be the empty set.

It is helpful to think of \( B(X) \) as an approximation to \( P(X) \). We want a bounding function to be:

1. easy to compute, and
2. close to \( P(X) \).

These two properties work against each other. For example, \( P(X) \) is itself a bounding function, but is too difficult to compute. Finding bounding functions which satisfy both of the above properties can be a challenging task, and we will describe some nice examples of useful bounding functions later in this chapter.

A general backtracking algorithm incorporating a bounding function is presented as Algorithm 4.7. In this algorithm, \( B \) is any specified bounding function, and the function \( \text{profit} \) computes the profit for a feasible solution \( X \).

---

**Algorithm 4.7: BOUNDING (** \( \ell \)**

**external** \( P(), B() \)

**global** \( X, \text{Opt}(P), \text{Opt}(X), C_\ell \) \((\ell = 0, 1, \ldots)\)

if \([x_0, \ldots, x_{t-1}]\) is a feasible solution

\[
\begin{align*}
P & \leftarrow \text{profit}([x_0, \ldots, x_{t-1}]) \\
\text{if} \ P & > \text{Opt}(P) \\
\text{then} \ & \{ \\
& \text{Opt}(P) \leftarrow P \\
& \text{Opt}(X) \leftarrow [x_0, \ldots, x_{t-1}] \\
\}
\end{align*}
\]

Compute \( C_\ell \)

\[
B \leftarrow B([x_0, \ldots, x_{t-1}])
\]

for each \( x \in C_\ell \)

\[
\begin{align*}
\text{if} \ B & \leq \text{Opt}(P) \ & \text{then} \text{return} \\
\end{align*}
\]

\[
\begin{align*}
& \{ \\
& x_t \leftarrow x \\
& \text{BOUNDING}(\ell + 1) \\
& \}
\end{align*}
\]

It is very important to note that we check to see if the pruning condition, \( B \leq \text{Opt}(P) \), is true every time we consider another element \( x \in C_\ell \). This is because the value of \( \text{Opt}(P) \) can increase as the algorithm progresses, and so we check to see if we can prune every time we are preparing to choose a new value for \( x_t \).

4.6.1 The knapsack problem

We now show how to define a useful bounding function for the Knapsack (optimization) problem. First, we consider a related problem called the Rational Knapsack problem, which we present as Problem 4.3.

**Problem 4.3: Rational Knapsack**

**Instance:** \( p_0, p_1, p_2, \ldots, p_{n-1}; \)

weights \( w_0, w_1, w_2, \ldots, w_{n-1}; \) and

**capacity** \( M \)

**Find:** the maximum value of

\[
\sum_{i=0}^{n-1} p_i x_i
\]

subject to

\[
\sum_{i=0}^{n-1} w_i x_i \leq M
\]

where \( x_i \) is rational and \( 0 \leq x_i \leq 1 \) for all \( i \).

It is straightforward to see that Algorithm 4.8, which uses a greedy strategy, returns the optimal profit for Rational Knapsack.
Algorithm 4.8: RKNAP \((p_0, p_1, \ldots, p_{n-1}, w_0, w_1, \ldots, w_{n-1}, M)\)

permute the indices so that \(\frac{p_0}{w_0} \geq \frac{p_1}{w_1} \geq \cdots \geq \frac{p_{n-1}}{w_{n-1}} \)

\(i \leftarrow 0\)
\(P \leftarrow 0\)
\(W \leftarrow 0\)

for \(j \leftarrow 0\) to \(n - 1\)

\(x_j \leftarrow 0\)

while \(W < M\) and \(i < n\)

if \(W + w_i \leq M\)

\[\begin{align*}
  x_i &\leftarrow 1 \\
  W &\leftarrow W + w_i \\
  P &\leftarrow P + p_i \\
  i &\leftarrow i + 1
\end{align*}\]

else

\[\begin{align*}
  x_i &\leftarrow \frac{M - W}{w_i} \\
  W &\leftarrow M \\
  P &\leftarrow P + x_i p_i \\
  i &\leftarrow i + 1
\end{align*}\]

end

return \((P)\)

We use RKNAP to define a bounding function for the Knapsack (optimization) problem as follows. Given a (feasible) partial solution \(X = [x_0, x_1, \ldots, x_{\ell-1}]\), define

\[
B(X) = \sum_{i=0}^{\ell-1} p_i x_i + \text{RKNAP}(p_\ell, \ldots, p_n, w_\ell, \ldots, w_n, M - \sum_{i=0}^{\ell-1} w_i x_i)
\]

Thus, \(B(X)\) is equal to the sum of:

1. the profit obtained from objects 0, 1, \ldots, \(\ell - 1\).

plus

2. the profit from the remaining objects, using the remaining capacity

\(M - \text{Cur} W\),

but allowing rational \(x_i\)s.

If we restricted each \(x_i\) to be 0 or 1 in part 2 above, then we would obtain \(P(X)\). Allowing the \(x_i\)s with \(\ell \leq i \leq n\) to be rational may yield a higher profit; so \(B(X) \geq P(X)\) and \(B\) is indeed a bounding function. It is also easy to compute, and thus may be useful for pruning.

Suppose we want to solve an instance of the Knapsack (optimization) problem. It will be useful to sort the objects in non-decreasing order of profit/weight ahead of time, before we begin the backtracking algorithm. Then, when we wish to evaluate our bounding function, the first step of Algorithm 4.8 will be unnecessary, and consequently RKNAP will run faster. Thus, we will assume that

\[
\frac{p_0}{w_0} \geq \frac{p_1}{w_1} \geq \cdots \geq \frac{p_{n-1}}{w_{n-1}}
\]

The improved algorithm is given as Algorithm 4.9.

Algorithm 4.9: KNAPSACK^3 (\(\ell, \text{Cur} W\))

external RKNAP()

global \(X, \text{Opt} X, \text{Opt} P, C_\ell\) \((\ell = 0, 1, \ldots)\)

if \(\ell = n\)

\[\begin{align*}
  &\text{if } \sum_{i=0}^{n-1} p_i x_i > \text{Opt} P \\
  &\text{then } \begin{cases} \text{Opt} P &\leftarrow \sum_{i=0}^{n-1} p_i x_i \\
  \text{Opt} X &\leftarrow [x_0, \ldots, x_{n-1}] \end{cases}
\end{align*}\]

if \(\ell = n\)

\[C_\ell \leftarrow \emptyset\]

\[\text{if } \text{Cur} W + w_\ell \leq M \]

else

\[C_\ell \leftarrow \{1, 0\}\]

else \(C_\ell \leftarrow \{0\}\)

\[
B \leftarrow \sum_{i=0}^{\ell-1} p_i x_i + \text{RKNAP}(p_\ell, \ldots, p_n, w_\ell, \ldots, w_n, M - \text{Cur} W)
\]

for each \(x \in C_\ell\)

\[\text{if } B \leq \text{Opt} P \text{ then return}
\]

\[
\begin{align*}
  x_\ell &\leftarrow x \\
  \text{KNAPSACK}^3(\ell + 1, \text{Cur} W + w_\ell x_\ell)
\end{align*}
\]

Example 4.4 An instance of the Knapsack (Optimization) problem

Suppose we have five objects, having weights 11, 12, 8, 7, 9; profits 23, 24, 15, 13 and 16 (respectively); and capacity \(M = 26\). Note that the objects are already arranged in decreasing order of profit/weight. We draw in Figure 4.4 the space tree traversed in the course of the backtracking algorithm KNAPSACK^3. At each node, we record the current values of \(X, B(X)\) and \(\text{Cur} W\).
Backtracking Algorithms

$X = []$
$B = 52.625$
$\text{CurW} = 0$

$X = [1]$
$B = 52.625$
$\text{CurW} = 23$

$X = [0]$
$B = 52.625$
$\text{CurW} = 23$

$X = [1,0]$
$B = 52.625$
$\text{CurW} = 23$

$X = [1,1,0,0,-]$
$B = 52.33$
$\text{CurW} = 23$

$X = [1,0,1,1,-]$
$B = 51$
$\text{CurW} = 26$

$X = [1,0,1,0,0,0]$
$P = 47, \text{so set } OptP = 47$
$\text{CurW} = 23$

$X = [1,0,1,1,0]$
$P = 51 > OptP,$
$\text{so set } OptP = 51$
$\text{CurW} = 26$

FIGURE 4.4
The state space tree traversed by KNAPSACK3.

To compare the amount of pruning provided by Algorithms 4.1, 4.3, and 4.9, we give experimental data in Table 4.2. For each $n = 8, 12, 16, 20,$ and 24, five instances of Problem 1.4 were generated by randomly selecting $n$ integer weights $w_0, w_1, \ldots, w_{n-1}$ between 0 and 1000000. In an attempt to generate "hard" instances of Problem 1.4, we defined, for each $i = 0, 1, \ldots, n-1$, the profit $p_i = 2 w_i \epsilon$, where $\epsilon$ was chosen at random in the interval $(0.9, 1.1)$. Hence the profit of each object is within 10% of twice its weight. The capacity $M$ was chosen to be half the sum of the weights.

Certainly the bounding function has a dramatic effect on the running time of the algorithm in these random problem instances.

4.6.2 The traveling salesman problem

In the traveling salesman problem, a salesman must visit $n$ cities and return home, doing so in such a way that the cost of the trip is minimized. More precisely, the Traveling Salesman problem is defined as follows.
Problem 4.4: Traveling Salesman

Instance: a complete graph on $n$ vertices, $G = (V, E)$; a cost function, $c : E \rightarrow \mathbb{Z}^+$

Find: a Hamiltonian circuit $X$ of $G$ such that
$$
cost(X) = \sum_{e \in E(X)} c(e)
$$
is minimized. (Recall that a Hamiltonian circuit in a graph $G$ is a circuit that passes through each vertex of $G$ exactly once.)

Let $V = \{0, 1, \ldots, n - 1\}$ be the vertices of the graph $G$. For convenience, we will define $c(x, y) = c(\{x, y\})$ if $x \neq y$, and $c(x, y) = \infty$ if $x = y$.

Any Hamiltonian circuit $X$ can be represented as a permutation of $V$, say $X = [x_0, \ldots, x_{n-1}]$. Without loss of generality, we can regard $X$ as starting and ending at vertex $0$; so, we can define $x_0 = 0$. For example, the circuit

$$25103462$$

would be represented by the 7-tuple

$$[0, 3, 4, 6, 2, 5, 1] \text{ or by } [0, 1, 5, 2, 6, 3, 4].$$

Algorithm 4.10 is a basic backtrack algorithm for the Traveling Salesman problem. To speed up Algorithm 4.10, we will construct some bounding functions. The Traveling Salesman problem is a minimization problem; so, a bounding function, $\text{Bound}(X)$, must provide a lower bound on the cost of any Hamiltonian circuit that is an extension of the partial solution $X$. Suppose $X = [x_0, x_1, \ldots, x_{\ell-1}]$ is a partial solution. Then $\ell \leq n - 1$, and $X$ represents the path $x_0 x_1 \cdots x_{\ell-1}$ of length $\ell - 1$. Define

$$Y = V \setminus \{x_0, x_1, \ldots, x_{\ell-1}\}.$$

Observe that, if $[x_0, \ldots, x_{n-1}]$ is a feasible solution that is an extension of $X$, then

$$Y = \{x_{\ell}, \ldots, x_{n-1}\}.$$

Algorithm 4.10: TSP1 ($\ell$)

```
global $C_\ell$ ($\ell = 0, 1, \ldots, n - 1$)

if $\ell = n$
    $C \leftarrow c([x_0, \ldots, x_{n-1}])$
    if $C < \text{Opt}\text{C}$
        then $\text{Opt}\text{C} \leftarrow C$
        $\text{Opt}\text{X} \leftarrow [x_0, \ldots, x_{n-1}]$
    if $\ell = 0$
        then $C_\ell \leftarrow \{0\}$
    else $C_\ell \leftarrow [1, \ldots, n - 1]$
    else $C_\ell \leftarrow C_{\ell - 1} \setminus \{x_{\ell - 1}\}$

for each $x \in C_\ell$
    do $x_\ell \leftarrow x$
    do $\text{TSP1}(\ell + 1)$
```

For $x \in V$ and $W \subseteq V$ (where $W \neq \emptyset$), define

$$b(x, W) = \min\{c(x, y) : y \in W\}.$$

We now prove an inequality that will lead to a bounding function.

THEOREM 4.2 Let $X' = [x_0, \ldots, x_{n-1}]$ be the minimum cost Hamiltonian circuit that extends $[x_0, x_1, \ldots, x_{\ell-1}]$, where $\ell \leq n - 1$. Then it holds that

$$\text{cost}(X') \geq \sum_{i=0}^{\ell-1} \text{cost}(x_i, x_{i+1}) + b(x_{\ell-1}, Y) + \sum_{y \in Y} b(y, Y \cup \{x_0\}).$$

PROOF Define $x_n = x_0$ for convenience; then we have

$$\text{cost}(X') = \sum_{i=0}^{n-1} \text{cost}(x_i, x_{i+1}).$$

First, the sum

$$\sum_{i=0}^{\ell-1} \text{cost}(x_i, x_{i+1})$$

represents the sum of the costs of the edges already chosen in $X$. Next, we have

$$\text{cost}(x_{\ell-1}, x_\ell) \geq b(x_{\ell-1}, Y),$$

because $x_\ell \in Y$. Finally, for $\ell \leq i \leq n - 1$, we have

$$\text{cost}(x_i, x_{i+1}) \geq b(x_i, Y \cup \{x_0\}),$$
because \( x_{i+1} \in \mathcal{Y} \cup \{x_0\} \) for \( 0 \leq i \leq n - 1 \). Since \( \mathcal{Y} = \{x_t, \ldots, x_{n-1}\} \), the result follows.

Let \( X = [x_0, x_1, \ldots, x_{n-1}] \). If \( 0 \leq n - 1 \), then define

\[
\text{MINCOSTBOUND}(X) = \sum_{i=0}^{t-1} \text{cost}(x_i, x_{i+1}) + b(x_{t-1}, \mathcal{Y}) + \sum_{y \in \mathcal{Y}} b(y, \mathcal{Y} \cup \{x_0\}),
\]

whereas if \( n = n \), then define

\[
\text{MINCOSTBOUND}(X) = \sum_{i=0}^{t-1} \text{cost}(x_i, x_{i+1}) + \text{cost}(x_{n-1}, x_0).
\]

Theorem 4.2 establishes that \( \text{MINCOSTBOUND} \) is a bounding function. It is straightforward to describe an algorithm to compute \( \text{MINCOSTBOUND} \) in \( O(n^2) \) time.

Another bounding function can be achieved using reduced matrices, which we will now study. A matrix \( M \) of integers is said to be reduced if the following three properties are satisfied:

1. all entries of \( M \) are non-negative;
2. every row of \( M \) contains at least one entry equal to 0;
3. every column of \( M \) contains at least one entry equal to 0.

Suppose \( M \) is an \( m \) by \( m \) matrix in which all the entries are non-negative. Algorithm 4.11 transforms \( M \) into a reduced matrix, and computes a quantity \( \text{val} = \text{REDUCE}(M) \) which we call the value of the matrix \( M \).

The following result shows the relevance of reduced matrices to the Traveling Salesman problem.

**THEOREM 4.3** Suppose cost is a cost function for the complete graph \( G \) on \( m \) vertices. Define \( M \) to be the \( m \) by \( m \) matrix in which \( M[i, j] = \text{cost}(i, j) \). Then any Hamiltonian circuit in \( G \) has cost at least \( \text{REDUCE}(M) \).

**Algorithm 4.11: \text{REDUCE}(M)**

\[
\begin{align*}
\text{comment: } M \text{ is an } m \text{ by } m \text{ matrix} \\
\text{val} & \leftarrow 0 \\
& \text{for } i \leftarrow 0 \text{ to } m - 1 \\
& \quad \text{for } j \leftarrow 0 \text{ to } m - 1 \\
& \quad \quad \text{if } M[i, j] < \text{min} \\
& \quad \quad \quad \text{then } \text{min} \leftarrow M[i, j] \\
& \quad \quad \text{for } j \leftarrow 0 \text{ to } m - 1 \\
& \quad \quad \text{do } M[i, j] \leftarrow M[i, j] - \text{min} \\
& \quad \quad \text{val} \leftarrow \text{val} + \text{min} \\
& \text{for } j \leftarrow 0 \text{ to } m - 1 \\
& \quad \text{min} \leftarrow M[0, j] \\
& \text{for } i \leftarrow 0 \text{ to } m - 1 \\
& \quad \text{if } M[i, j] < \text{min} \\
& \quad \quad \text{then } \text{min} \leftarrow M[i, j] \\
& \quad \text{for } i \leftarrow 0 \text{ to } m - 1 \\
& \quad \text{do } M[i, j] \leftarrow M[i, j] - \text{min} \\
& \text{val} \leftarrow \text{val} + \text{min} \\
& \text{return } (\text{val}) \\
\end{align*}
\]

Before giving a proof of this theorem, we present an example to illustrate it.

**Example 4.5 An illustration of Theorem 4.3**

Suppose the cost matrix for a graph \( G \) is:

\[
M = \begin{bmatrix}
\infty & 3 & 5 & 8 \\
3 & \infty & 2 & 7 \\
5 & 2 & \infty & 6 \\
8 & 7 & 6 & \infty
\end{bmatrix}
\]

is the cost matrix for a graph \( G \). Reducing \( M \), using Algorithm 4.11, we see that \( \text{REDUCE}(M) = 18 \). On the other hand, it is not difficult to check the costs of the three possible Hamiltonian circuits:

- the circuit 0 1 2 3 has cost \( 3 + 2 + 6 + 8 = 19 \)
- the circuit 0 1 3 2 has cost \( 3 + 7 + 6 + 5 = 21 \)
- the circuit 0 2 1 3 has cost \( 5 + 2 + 7 + 8 = 22 \)

Hence the minimum-cost Hamiltonian circuit has cost 19, and indeed,

\[
19 \geq \text{REDUCE}(M) = 18.
\]
This example shows that REDUCE($M$) is indeed only a lower bound on the cost of any Hamiltonian circuit; it need not give the exact value of the minimum-cost circuit.

**Proof** (of Theorem 4.3) Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be any Hamiltonian circuit of $G$. Define $x_n = x_0$. Then

$$\text{cost}(X) = M[x_0, x_1] + M[x_1, x_2] + \cdots + M[x_{n-1}, x_n].$$

This sum uses exactly one cell from each row and column of $M$. Define

$$r_i = \min \{M[i, j] : 0 \leq j \leq n-1 \}$$

and

$$c_j = \min \{M[i, j] - r_i : 0 \leq i \leq n-1 \},$$

for $0 \leq i \leq n-1$ and $0 \leq j \leq n-1$. Observe that

$$\text{REDUCE}(M) = \sum_{i=0}^{n-1} r_i + \sum_{j=0}^{n-1} c_j.$$

Clearly, we have

$$r_i + c_{x_{i+1}} \leq M[x_i, x_{i+1}],$$

for all $i$, $0 \leq i \leq n-1$. Hence, summing over $i$, we see that

$$\text{REDUCE}(M) \leq \text{cost}(X),$$

and the result follows.

We now use the idea of reduced matrices to define a bounding function. We need a way to determine a lower bound on the cost of any Hamiltonian circuit that is the completion of a given partial solution. Suppose we have a partial solution

$$X = [x_0, x_1, \ldots, x_{\ell-1}],$$

where $\ell \leq n-1$. As mentioned previously, $X$ represents the path $x_0 x_1 \ldots x_{\ell-1}$. Perform the following operations on the cost matrix $M$:

1. if $\ell < n$ then $M[x_{\ell-1}, 0] = \infty$;
2. delete rows $x_0, x_1, \ldots, x_{\ell-2}$ of $M$; and
3. delete columns $x_1, \ldots, x_{\ell-1}$ of $M$.

Call the resulting matrix $M'(X)$. Observe that $M'(X)$ is an $(n - \ell + 1) \times (n - \ell + 1)$ matrix. Now, we define our bounding function by the following formula:

$$\text{REDUCEBOUND}(X) = \text{REDUCE}(M'(X)) + M[x_0, x_1] + \cdots + M[x_{n-2}, x_{n-1}].$$

That is, $\text{REDUCEBOUND}(X)$ is the sum of the the costs of the edges in the partial solution $X$ and the value of the matrix $M'(X)$.

To show that this formula is indeed a bounding function, we need to prove that $\text{REDUCEBOUND}(M'(X))$ is a lower bound on the sum of the costs of any completion of $X$ to a Hamiltonian circuit. We argue informally that this is the case.

Consider the effects of operations 1, 2, and 3. Operation 1 rules out using the edge $\{x_{\ell-1}, x_0\}$ if $\ell < n$, because using this edge would close the circuit prematurely. Operation 2 rules out using edges leaving vertices $x_0, x_1, \ldots, x_{\ell-2}$ and operation 3 rules out using edges entering vertices $x_1, \ldots, x_{\ell-1}$. Any $n - \ell$ edges which provide a completion of $X$ to a Hamiltonian circuit will thus hit $M'(X)$ exactly once in every row and column. It can be shown that the sum of the costs of these edges must be no less than the value of $M'(X)$, in a manner similar to the proof of Theorem 4.3.

We now give a detailed description of the resulting bounding function.

**Algorithm 4.12: REDUCEBOUND (X)**

```java
external cost, REDUCE

global M, V

comment: X = [x_0, \ldots, x_{m-1}]
if m = n then return (cost(X))
M'[0, 0] = \infty
j = 1
for each y \in V \setminus \{x_0, x_1, \ldots, x_{m-1}\} do
M'[0, j] = M[x_{m-1}, y]
j += 1
i = 1
for each x \in V \setminus \{x_0, x_1, \ldots, x_{m-1}\} do
M'[i, 0] = M[x, x_0]
i += 1
for each x \in V \setminus \{x_0, x_1, \ldots, x_{m-1}\} do
\begin{align*}
&j = 1 \\
&\text{for each y \in V \setminus \{x_0, x_1, \ldots, x_{m-1}\} do}
&M'[i, j] = M[x, y] \\
&i += 1
\end{align*}
ans = REDUCE(M')
for i = 1 to m - 1
do ans = ans + M[x_{i-1}, x_i]
return (ans)
```
The value of an $n \times n$ matrix is computed in time $O(n^2)$ by Algorithm 4.11. Hence $\text{REDUCEBOUND}(X)$ can also be computed in time $O(n^2)$. Algorithm 4.13 is a backtracking algorithm for the Traveling Salesman problem that incorporates an arbitrary bounding function. To compare the effect of the two bounding functions $\text{MINEDGEBOUND}$ and $\text{REDUCEBOUND}$ described in this section, we generated random instances of the Traveling Salesman problem on 5, 10, 15, and 20 vertices. The edge costs were randomly chosen integers between 0 and 100. In Table 4.3 we report the number of nodes in the state space trees when Algorithm 4.13 is used with these two bounding functions.

**Algorithm 4.13:** TSP2($\ell$)

```plaintext
external B()
global $C_\ell$ ($\ell = 0, 1, \ldots, n - 1$)
if $\ell = n$
    $C \leftarrow \text{cost}([x_0, \ldots, x_{n-1}])$
    if $C < \text{OptC}$
        $\text{OptC} \leftarrow C$
        $\text{OptX} \leftarrow [x_0, \ldots, x_{n-1}]$
    if $\ell = 0$
        $C_\ell \leftarrow \{0\}$
    else if $\ell = 1$
        $C_\ell \leftarrow \{1, \ldots, n - 1\}$
    else $C_\ell \leftarrow C_{\ell-1} \setminus \{x_{\ell-1}\}$
    $B \leftarrow B([x_0, \ldots, x_{\ell-1}])$
for each $x \in C_\ell$
    if $B \geq \text{OptC}$
        then return
        $x_{\ell} \leftarrow x$
    TSP2($\ell + 1$)
```

**Problem 4.5:** Maximum Clique

**Instance:** A graph $G = (V, E)$

**Find:** a maximum clique of $G$.

This problem has been shown to be NP-complete, but in spite of its inherent difficulty, many algorithms have been developed that perform well in practice.

In Section 4.3 we developed Algorithm 4.4 for generating all the cliques in a graph $G = (V, E)$. This algorithm can easily be modified to find a maximum clique; see Algorithm 4.14. Note that we no longer need to maintain the sets $N_i$; we simply check to see if each clique constructed is larger than any previously constructed clique.

We now turn to the development of bounding functions for this problem. First we require a definition. Suppose $G = (V, E)$ is a graph, and $W \subseteq V$. The **induced subgraph** $G[W]$ has vertex set $W$, and edge set $\{\{u, v\} \in E : \{u, v\} \subseteq W\}$.

Now, at a typical point in Algorithm 4.14, we have the partial solution (i.e., clique) $X = [x_0, x_1, \ldots, x_{\ell-1}]$. Suppose $X' = [x_0, x_1, x_2, \ldots, x_j]$ is a clique which extends the partial solution $X$, where $j \geq \ell - 1$. Then $\{x_{\ell}, \ldots, x_j\}$ must be a clique in the induced subgraph $G[X]$.

**FIGURE 4.5**

A graph with maximum clique $\{1, 2, 3, 4\}$.

**4.6.3 The maximum clique problem**

Recall that a maximum clique in a graph $G$ is a clique of largest cardinality. For example, the maximal cliques in the graph in Figure 4.5 are $\{1, 2, 3, 4\}$, $\{3, 4, 6\}$, $\{3, 5\}$, and $\{4, 7\}$. The clique $\{1, 2, 3, 4\}$ is the only maximum clique. In general, a graph may have more than one maximum clique. The problem of finding a maximum clique in a graph $G$ is known as the **Maximum Clique** problem. A decision version of this problem was introduced in Section 1.6 as Problem 1.7. The optimization version of the problem is defined as follows.

**Algorithm 4.14:** TSP2($\ell$)

```plaintext
external B()
global $C_\ell$ ($\ell = 0, 1, \ldots, n - 1$)
if $\ell = n$
    $C \leftarrow \text{cost}([x_0, \ldots, x_{n-1}])$
    if $C < \text{OptC}$
        $\text{OptC} \leftarrow C$
        $\text{OptX} \leftarrow [x_0, \ldots, x_{n-1}]$
    if $\ell = 0$
        $C_\ell \leftarrow \{0\}$
    else if $\ell = 1$
        $C_\ell \leftarrow \{1, \ldots, n - 1\}$
    else $C_\ell \leftarrow C_{\ell-1} \setminus \{x_{\ell-1}\}$
    $B \leftarrow B([x_0, \ldots, x_{\ell-1}])$
for each $x \in C_\ell$
    if $B \geq \text{OptC}$
        then return
        $x_{\ell} \leftarrow x$
    TSP2($\ell + 1$)
```
by placing an upper bound on the size of a maximum clique in $G[C]$. If the size of a maximum clique in $G[C]$ is denoted by $mc(\ell)$, and $mc(\ell) \leq ub(\ell)$, then

$$B(X) = \ell + ub(\ell)$$

is a bounding function.

**Algorithm 4.14: MAXCLIQUE**

<table>
<thead>
<tr>
<th>global $A_t, B_t, C_t$ ($\ell = 0, 1, \ldots, n - 1$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>if $\ell &gt; \text{OptSize}$ then { $\text{OptSize} \leftarrow \ell + 1$ }</td>
</tr>
<tr>
<td>$\text{OptClique} \leftarrow [x_0, \ldots, x_{\ell-1}]$</td>
</tr>
<tr>
<td>if $\ell = 0$ then $C_t \leftarrow V$</td>
</tr>
<tr>
<td>else $C_t \leftarrow A_{x_{\ell-1}} \cap B_{x_{\ell-1}} \cap C_{\ell-1}$</td>
</tr>
<tr>
<td>for each $x \in C_t$ do { $\text{MAXCLIQUE}(\ell + 1)$ }</td>
</tr>
<tr>
<td>main $\text{OptSize} \leftarrow 0$</td>
</tr>
<tr>
<td>$\text{MAXCLIQUE}(0)$</td>
</tr>
<tr>
<td>output $(\text{OptClique})$</td>
</tr>
</tbody>
</table>

We can use this idea to obtain several different bounding functions. The simplest of them is to observe that

$$mc(\ell) \leq |C_{\ell}|.$$ 

This gives rise to the bounding function presented in Algorithm 4.15, which we call the *size bound*.

**Algorithm 4.15: SIZEBOUND**

| global $C_t$ |
| comment: $X = [x_0, \ldots, x_{\ell-1}]$ |
| return $(\ell + |C_{\ell}|)$ |

Other, more sophisticated, methods of obtaining bounding functions use the idea of *vertex coloring* (see Problem 1.5). Recall that, if $G = (V, E)$ is a graph and $k$ is a positive integer, then a (vertex) $k$-coloring of $G$ is a function

$$\text{color} : V \rightarrow \{0, \ldots, k - 1\}$$

such that $\text{color}(x) \neq \text{color}(y)$ for all $\{x, y\} \in E$. The relevance of vertex coloring to the Maximum Clique problem is stated in the following simple lemma.

**Lemma 4.4** Let $G$ be a graph, and suppose that $G$ has a vertex $k$-coloring. Then the maximum clique in $G$ has size at most $k$.

**Proof** If vertices $x$ and $y$ receive the same color, then they cannot both be in the same clique.

Even though finding a vertex $k$-coloring in which $k$ is minimized is NP-hard, it is not difficult to find $k$-colorings for values of $k$ that are larger than the minimum. One easy way to do this is to color the vertices by a greedy strategy (recall that greedy algorithms were introduced in Section 1.8.1). In a greedy algorithm, the vertices are processed in order, each vertex receiving the first available color. Algorithm 4.16 presents such an algorithm. In Algorithm 4.16, we assume that the vertex set is written as $V = \{0, \ldots, n - 1\}$. The algorithm constructs a $k$-coloring for some positive integer $k$, and returns that value of $k$. The actual $k$-coloring is stored as a (global) array, $\text{color}$. In the process of constructing this coloring, the algorithm constructs an array of sets called $\text{ColorClass}$, which is defined as follows:

$$\text{ColorClass}[h] = \{ i \in V : \text{color}[i] = h \}$$

for $0 \leq h \leq k - 1$.

**Algorithm 4.16: GREEDYCOLOR**

| global $\text{color}$ |
| comment: $V = \{0, \ldots, n - 1\}$ |
| $k \leftarrow 0$ |
| for $i \leftarrow 0$ to $n - 1$ do \{ $h \leftarrow 0$ \} |
| $\text{while } h < k \text{ and } A_i \cap \text{ColorClass}[h] \neq \emptyset$ do $h \leftarrow h + 1$ |
| $k \leftarrow k + 1$ |
| $\text{if } h = k$ then \{ $\text{ColorClass}[h] \leftarrow \emptyset$ \} |
| $\text{ColorClass}[h] \leftarrow \text{ColorClass}[h] \cup \{i\}$ |
| $\text{color}[i] \leftarrow h$ |
| return $(k)$ |

There are several ways in which Algorithm 4.16 can be incorporated into a bounding function. One way is to find an initial greedy coloring of the graph before the backtracking algorithm begins. Suppose that this coloring is denoted $\text{color}$ and it uses $k$ colors. For each induced subgraph $G[C_t]$, the function $\text{color}$, restricted to the vertices in $C_t$, defines a coloring of $G[C_t]$ which may use fewer than $k$ colors. The number of colors in this induced coloring yields an upper bound on the size of a maximum clique in $G[C_t]$. The resulting bounding function, which we call the *sampling bound*, is presented in Algorithm 4.17.
Algorithm 4.17: \textsc{SamplingBound} (X)

\begin{verbatim}
global $C_t$, color
comment: $X = [x_0, \ldots, x_{t-1}]$
return ($\ell + |\{\text{color}[x] : x \in C_t\}|$)
\end{verbatim}

Another way to use the greedy coloring algorithm in a bounding function is to apply Algorithm 4.16 to the induced subgraph $G[C_t]$ every time we want to compute the bounding function. The resulting bounding function is called the \textit{greedy bound} and it is presented in Algorithm 4.18.

Algorithm 4.18: \textsc{GreedyBound} (X)

\begin{verbatim}
external \textsc{GreedyColor()}
global $C_t$
comment: $X = [x_0, \ldots, x_{t-1}]$
\$k \leftarrow \textsc{GreedyColor}(G[C_t])$
return ($\ell + k$)
\end{verbatim}

Any of the three bounding functions discussed above (or any other bounding function, for that matter) can be incorporated into our backtracking algorithm as the function $B(X)$. Algorithm 4.19 is the result.

As was done in other algorithms incorporating bounding functions, we check to see if the condition $M \leq \text{OptSize}$ is true in every iteration of the loop. This is because the value of $\text{OptSize}$ can increase as the algorithm progresses, and so we check to see if we can prune every time we are preparing to add a new node to the clique being considered.

In Table 4.4 we list the number of nodes in the state space tree, for graphs of various sizes, when Algorithm 4.19 is run using the different bounding functions we have discussed. We also list the number of edges, and the size of the maximum cliques in these graphs. The graphs we used were generated at random from the class $\mathcal{G}(n)$ defined in Section 4.3.1. There are several ways to do this. One nice method uses ranking and unranking algorithms we developed in Chapter 2. Note that the function RandomInteger($a, b$) generates a random integer in the interval $[a, b]$. Algorithm 4.20 constructs a random graph in the class $\mathcal{G}(n)$.

Algorithm 4.19: \textsc{MaxClique2} ($\ell$)

\begin{verbatim}
external B()
global $A_\ell, B_\ell, C_\ell$ ($\ell = 0, 1, \ldots, n - 1$)
if $\ell > \text{OptSize}$
then
  \$\text{OptSize} \leftarrow \ell$
  \$\text{OptClique} \leftarrow [x_0, \ldots, x_{t-1}]$
if $\ell = 0$
then
  $C_t \leftarrow V$
else
  $C_t \leftarrow A_x \cap B_{\ell-1} \cap C_{\ell-1}$
$M \leftarrow B([x_0, \ldots, x_{t-1}])$
for each $x \in C_t$
  if $M \leq \text{OptSize}$
  then return
  do
    $x_t \leftarrow x$
    \$\text{MaxClique2}(\ell + 1)$
\end{verbatim}

main

$\text{OptSize} \leftarrow 0$
$\text{MaxClique2}(0)$
\text{output} (\text{OptClique})

Algorithm 4.20: \textsc{GenerateRandomGraph} ($n$)

\begin{verbatim}
\{RandomInteger()
external \{\textsc{SubsetLexUnrank()}
  \textsc{KSubsetLexUnrank()}
\}
$T \leftarrow \textsc{SubsetLexUnrank}([n], r)$
$E \leftarrow \emptyset$
for each $j \in T$
  do \{\$E \leftarrow E \cup \{x - 1, y - 1\}$
\}
return ($G = ([0, \ldots, n - 1], E)$)
\end{verbatim}

The only aspect of Algorithm 4.20 that might require explanation is the last line, where we add the edge $\{x - 1, y - 1\}$ to $E$. This is because the algorithm $\textsc{KSubsetLexUnrank}$ returns a 2-subset of $\{1, \ldots, n\}$, whereas we want a 2-subset of $\{0, \ldots, n - 1\}$. Thus we subtract one from $x$ and $y$ to create the edge to be included in $E$.

Notice that the expected (i.e., average) sizes of state space trees when no pruning was done were denoted in Section 4.3.1 by $\bar{\delta}(n)$, and some values of $\bar{\delta}(n)$.
were presented in Table 4.1. It is interesting to compare these values to the experimental results obtained in Table 4.4.

The edge density of a graph is the ratio of the number of its edges to \((\frac{n(n-1)}{2})\) (which is the total possible number of edges). The random graphs generated by Algorithm 4.20 will have edge density approximately .5. To obtain a random graph with a given edge density \(\delta\), \(0 \leq \delta \leq 1\), Algorithm 4.21 can be used. In this algorithm the function \(\text{Random}(a, b)\) generates a random real number in the interval \([a, b]\). Table 4.5 presents data similar to Table 4.4, but for randomly generated graphs with edge density approximately .75.

### Table 4.4
Size of state space trees for Algorithm 4.19 on random graphs with edge density .5

<table>
<thead>
<tr>
<th>number of vertices</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of edges</td>
<td>607</td>
<td>2535</td>
<td>5602</td>
<td>9925</td>
<td>15566</td>
</tr>
<tr>
<td>size of maximum clique</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>bounding function</td>
<td>none</td>
<td>size bound</td>
<td>sampling bound</td>
<td>greedy bound</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8687</td>
<td>16590.16</td>
<td>1434006</td>
<td>5008767</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3204</td>
<td>57225</td>
<td>1182514</td>
<td>4093535</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2268</td>
<td>44072</td>
<td>266246</td>
<td>2268</td>
<td></td>
</tr>
<tr>
<td></td>
<td>430</td>
<td>5734</td>
<td>22599</td>
<td>91671</td>
<td>290788</td>
</tr>
</tbody>
</table>

### Table 4.5
Size of state space trees for Algorithm 4.19 on random graphs with edge density .75

<table>
<thead>
<tr>
<th>number of vertices</th>
<th>25</th>
<th>50</th>
<th>75</th>
<th>100</th>
<th>125</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of edges</td>
<td>236</td>
<td>959</td>
<td>2045</td>
<td>3720</td>
<td>5780</td>
</tr>
<tr>
<td>size of maximum clique</td>
<td>11</td>
<td>14</td>
<td>15</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>bounding function</td>
<td>none</td>
<td>size bound</td>
<td>sampling bound</td>
<td>greedy bound</td>
<td></td>
</tr>
<tr>
<td></td>
<td>25570</td>
<td>2083770</td>
<td>12385596</td>
<td>186543706</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1840</td>
<td>91663</td>
<td>426279</td>
<td>5370268</td>
<td></td>
</tr>
<tr>
<td></td>
<td>794</td>
<td>37218</td>
<td>195567</td>
<td>2225982</td>
<td></td>
</tr>
<tr>
<td></td>
<td>91</td>
<td>2843</td>
<td>10476</td>
<td>70404</td>
<td>413421</td>
</tr>
</tbody>
</table>

Tables 4.4 and 4.5 show that the size of the state space tree decreases significantly as better bounding functions are employed. Of course, the optimal choice for a bounding function depends on both the time required to compute the bounding function, and on the amount by which the size of the state space tree is reduced. The relative computation times for the different bounding functions can depend heavily on the implementation. However, we can make a couple of observations on the complexity of these computations. First, when given as input a graph having \(n\) vertices, the greedy coloring algorithm takes time \(O(n^2)\). Therefore the greedy bound is computed in time \(O(|C_t|^2)\). The size bound and sampling bound, on the other hand, can be computed in time \(O(|C_t|)\) using standard algorithms. Hence, there is a tradeoff, because the more effective greedy bound has a slower computation time. In general, the greedy bound will result in a faster algorithm for “large enough” graphs. The crossover point, however, will depend on the implementation and is best determined by experimentation.

### 4.7 Branch and bound

Another way in which we can take advantage of a bounding function is a method called branch and bound. The usual implementation of backtracking is to examine each of the choices \(x_t \in C_t\) in some predetermined order, calling the algorithm recursively for each choice. A better strategy is to use a bounding function to determine the order in which the recursive calls are made. A branch and bound algorithm for a general maximization problem is presented as Algorithm 4.22.

We illustrate the branch and bound technique using the Traveling Salesman problem. Suppose \(X = [x_0, x_1, x_2, \ldots, x_{t-1}]\) is a partial solution for an instance of the Traveling Salesman problem, and \(\ell \leq n - 1\). Then there are \((n - 1) - (\ell - 1) = n - \ell\) choices for \(x_\ell\). Consider the node in the state space tree corresponding to the partial solution \(X\). Algorithm 4.13 would look at the \(n - \ell\) children of \(X\) in increasing order of \(x_\ell\). There is no particular reason to proceed in this order.

---

**Algorithm 4.21: GENERATERANDOMGRAPH2**

```
external Random()
for x ← 0 to n - 2
   do { for y ← x + 1 to n - 1
         do { r ← Random(0, 1)
                if r ≥ 1 - δ
                   then \(E' \leftarrow E \cup \{x - 1, y - 1\}\)
         return (G = (\{0, \ldots, n - 1\}, E'))
```
In a branch and bound algorithm, for the Traveling Salesman problem we will calculate \( B(X') \) for each of the \( n - \ell \) children \( X' \) of \( X \) before we make any recursive calls from this node. Then, we will make recursive calls in increasing order of the \( n - \ell \) values of \( B(X') \) that we computed (since it is a minimization problem). We hope that an optimal solution is most likely to be found in the branch of the state space tree where the bounding function is smallest. The remaining branches of the state space tree can then possibly be pruned without having to traverse them.

The resulting algorithm is presented as Algorithm 4.23. Other than the modification to the order in which recursive calls are made, Algorithm 4.23 is unchanged from the previous algorithm, Algorithm 4.13.

To evaluate the effect of the two bounding functions in a branch and bound algorithm, we used the same random instances as we did in Section 4.6.2. In Table 4.6 we report the number of nodes in the state space trees when Algorithms 4.13 and 4.23 are used, for both of the bounding functions.
4.8 Notes

Section 4.1
Backtracking algorithms are described in several textbooks and monographs, for example Brassard and Bratley [9], Goldberg [36], Horowitz and Sahni [43], Purdom and Brown [84], Reingold, Nievergelt and Deo [90] and Stinson [103].

Section 4.3.1
The average-case analysis of Algorithm 4.4 is due to Wilf; see [113, Section 5.6]. An example of an average-case analysis of a backtracking algorithm for a different problem (the Satisfiability problem) can be found in [84, Section 4.3].

Section 4.4
For a thorough treatment of the estimation of backtrack trees, see Purdom and Brown [84, Section 11.10].

Section 4.5
An earlier version of Algorithm 4.6 was developed and used by Frenz and Kreher in [30] to enumerate inequivalent cyclic Steiner triple systems. Wells discusses the Exact Cover problem in [111, Section 6.4]. He develops an algorithm which he then uses to construct Steiner triple systems. This is followed by a couple of additional refinements to the algorithm. If the subsets in $S$ are restricted to each having exactly 3 elements, then the problem is known as Exact Cover by 3-sets. Exact Cover by 3-sets was shown to be NP-complete by Karp in [51] (see also [31]).

Section 4.6.1
The bounding function we use for the Knapsack (optimization) problem is described in Horowitz and Sahni [43, Section 8.2]. The instances that are referred to in Table 4.2 can be found in the web pages at the following URL:

http://www.math.mtu.edu/~kreher/cages/Data.html

Section 4.6.2
An overview of bounding functions for the Traveling Salesman problem can be found in [63, Chapter 10]. The method of reduced matrices is described in several places, for example, in Horowitz and Sahni [43, Section 8.3]. The instances that are referred to in Tables 4.3 and 4.6 can be found in the web pages at the following URL:

http://www.math.mtu.edu/~kreher/cages/Data.html

Section 4.6.3
The monograph [49] edited by Johnson and Trick is a recent work devoted to three fundamental NP-hard problems, namely, the Maximum Clique problem, the Vertex Coloring problem, and the Satisfiability problem.

The 1973 branch and bound technique of Bron and Kerbosch [13], used in their algorithm CACM457, is the basis for most of the recent maximum clique algorithms. Among these, the most notable are the algorithms by Balas and Yu [5] and Babel [4]. The 1986 Balas-Yu Algorithm uses a greedy coloring and maximally triangulated induced subgraphs to achieve tighter bounds on the maximum clique size. This algorithm was one of the fastest until 1990, when Babel [4] introduced an algorithm that uses the DSAUT coloring method of Brelaz [10]. In 1998, Myrvold, Prsa and Walker [78] developed a promising method for testing maximum clique algorithms when the number of vertices becomes prohibitively large.

The instances of the Maximum Clique problem that are referred to in Tables 4.4 and 4.5 can be found in the web pages at the following URL:

http://www.math.mtu.edu/~kreher/cages/Data.html

Exercises

4.1 Define choice sets and describe backtracking algorithms for the following problems:
   (a) Find all ways of placing $n$ mutually non-attacking queens on an $n$ by $n$ chess board.
   (b) Find all self-avoiding walks of length $n$. (A self-avoiding walk is described by a sequence of edges in the Euclidean plane, beginning at the origin, such that each of the edges is a vertical or horizontal line segment of length one, and such that no point in the plane is visited more than once. There are precisely three such walks of length one, 12 walks of length two, and 36 walks of length three.)
   (c) Find all $k$-vertex colorings of a graph $G$.

4.2 Find a formula for the number of nodes in the state space tree that results when Algorithm 4.10 is run on an instance of the Traveling Salesman problem having $n$ vertices.

4.3 Determine the complexity of Algorithm 4.8, with and without the assumption that the objects are sorted according to their profit / weight ratios.

4.4 Use Algorithm 4.9 to solve the following instances of the Knapsack (optimization) problem.
4.5 In Algorithm 4.4 and Algorithm 4.14, the vertices are processed according to the prespecified ordering "<", always extending a clique with vertices that appear later in the ordering than the vertices already chosen. This trick allows each clique to be generated only once. However, the ordering defined on the vertices can greatly affect the point in the backtrack algorithm when the maximum clique(s) are discovered. Similarly, the speedup provided by pruning using a bounding function may depend strongly on the ordering of the vertices.

There are several natural ways to define the ordering on the vertices. In general, the best choice depends on the graph under consideration. Some of the possible orderings are as follows:

(a) random, in which the vertices are arbitrarily ordered;

(b) increasing, in which the vertices are sorted from lowest to highest degree;

(c) decreasing, in which the vertices are sorted from highest to lowest degree;

(d) induced, in which a minimum degree vertex is placed last in the list. This vertex is then deleted from the graph and the vertex of minimal degree in the new graph is placed next-to-last, and so on.

Investigate the effect of the vertex ordering on Algorithm 4.19, for each of the bounding functions described, using the graphs considered in Tables 4.4 and 4.5 as sample graphs.

4.6 Show that Algorithm 4.8 always produces an optimal solution for the Fractional Knapsack problem. Hint: Suppose that Algorithm 4.8 generates the solution \( X = \{x_0, \ldots, x_{n-1}\} \), with profit \( P = \sum_{i=0}^{n-1} p_i x_i \). Let \( Y = \{y_0, \ldots, y_{n-1}\} \) be any optimal solution with profit \( Q = \sum_{i=0}^{n-1} p_i y_i \). Since \( Y \) is optimal we must have \( P \leq Q \). Show that \( P = Q \).

4.7 Use Algorithm 4.13 to solve the instance of the Traveling Salesman problem on the vertex set \( V = \{0, 1, 2, \ldots, 9\} \) in which the cost of the edge \( \{x, y\} \) is given by the \( [x, y] \) entry of the matrix given below.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>22</td>
<td>72</td>
<td>56</td>
<td>17</td>
<td>57</td>
<td>13</td>
<td>38</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>22</td>
<td>0</td>
<td>95</td>
<td>29</td>
<td>84</td>
<td>75</td>
<td>39</td>
<td>19</td>
<td>26</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>95</td>
<td>0</td>
<td>87</td>
<td>78</td>
<td>70</td>
<td>39</td>
<td>99</td>
<td>21</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>72</td>
<td>29</td>
<td>87</td>
<td>0</td>
<td>95</td>
<td>90</td>
<td>82</td>
<td>33</td>
<td>60</td>
<td>76</td>
</tr>
<tr>
<td>4</td>
<td>56</td>
<td>84</td>
<td>78</td>
<td>95</td>
<td>0</td>
<td>39</td>
<td>93</td>
<td>35</td>
<td>2</td>
<td>59</td>
</tr>
<tr>
<td>5</td>
<td>17</td>
<td>75</td>
<td>70</td>
<td>90</td>
<td>39</td>
<td>0</td>
<td>36</td>
<td>81</td>
<td>98</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>57</td>
<td>39</td>
<td>39</td>
<td>82</td>
<td>93</td>
<td>36</td>
<td>0</td>
<td>28</td>
<td>47</td>
<td>88</td>
</tr>
<tr>
<td>7</td>
<td>13</td>
<td>19</td>
<td>99</td>
<td>33</td>
<td>35</td>
<td>81</td>
<td>28</td>
<td>0</td>
<td>78</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td>26</td>
<td>21</td>
<td>60</td>
<td>2</td>
<td>98</td>
<td>47</td>
<td>78</td>
<td>0</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>63</td>
<td>12</td>
<td>12</td>
<td>76</td>
<td>59</td>
<td>25</td>
<td>88</td>
<td>13</td>
<td>72</td>
<td>0</td>
</tr>
</tbody>
</table>

4.8 For the Traveling Salesman problem discussed in Section 4.6.2 a Hamiltonian circuit was represented as a permutation

\[ X = [0, x_1, x_2, \ldots, x_{n-1}] \]

of the vertices \( V \) starting at 0. This circuit is also represented by

\[ X' = [0, x_{n-1}, x_{n-2}, \ldots, x_1] \]

Thus Algorithms 4.10 and 4.13 will consider every Hamiltonian circuit twice. Develop a pruning method so that every Hamiltonian circuit is examined only once and incorporate it into an algorithm for solving the Traveling Salesman problem. Compare your new algorithm to Algorithms 4.10 and 4.13 by running it on the data given in Exercise 4.7 and computing the number of nodes in the corresponding state space trees.

4.9 Use Algorithm 4.19 to find a maximum clique in each of the graphs \( G = (V, E) \) where \( V = \{0, 1, 2, \ldots, 9, a, b, c, d\} \) and

(a) \[ E = \left( \begin{array}{cccccc}
0, 2, & 0, 7, & 0, a, & 0, b, & 1, 3, & 1, 8, & 1, 9, & 1, a, & 1, e, & 2, 3 \\
1, 2, & 2, d, & 2, e, & 3, 4, & 3, 6, & 4, 8, & 4, 5, & 5, a, & 5, e, & 6, c \\
6, e, & 7, 8, & 7, a, & 7, e, & 8, 6, & 9, a, & a, c, & c, d, & e, c, & e, f \end{array} \right) \]

(b) \[ E = \left( \begin{array}{cccccc}
0, 1, & 0, 2, & 0, 7, & 0, 8, & 0, 9, & 0, a, & 0, e, & 1, 4, & 1, 7, & 1, 8 \\
1, 9, & 1, a, & 1, b, & 1, c, & 1, e, & 2, 3, & 2, 5, & 6, 2, & 7, 7, & 2, 8 \\
2, 2, & 2, b, & 3, 4, & 3, 5, & 3, 6, & 3, a, & 3, f, & 4, 5, & 4, 6, & 4, 8 \\
4, 9, & 4, a, & 4, e, & 5, 6, & 5, 8, & 5, 9, & 5, b, & 5, d, & 5, e, & 5, f \\
6, f, & 7, 8, & 7, a, & 7, e, & 7, f, & 8, b, & 8, e, & 8, e, & 8, f \\
9, d, & 9, e, & a, c, & a, d, & a, e, & b, c, & 4, e, & 4, f, & c, d, & e, f \end{array} \right) \]

(c) \[ E = \left( \begin{array}{cccccc}
0, 2, & 0, 3, & 0, 4, & 0, 5, & 0, 6, & 0, 7, & 0, 8, & 0, 9, & 0, a, & 0, c \\
0, d, & 0, e, & 1, 5, & 1, 6, & 1, 7, & 1, 8, & 1, 9, & 1, a, & 1, b, & 1, c \\
1, d, & 1, e, & 1, f, & 2, 5, & 2, 6, & 2, 7, & 2, 8, & 2, a, & 2, b, & 2, e \\
2, f, & 3, 4, & 3, 5, & 3, 6, & 3, 7, & 3, 8, & 3, a, & 3, c, & 3, d, & 3, e \\
3, f, & 4, 6, & 4, 8, & 4, 9, & 4, b, & 4, c, & 4, e, & 4, f, & 5, 6, & 5, 7 \\
5, 9, & 5, a, & 5, b, & 5, c, & 5, e, & 5, f, & 6, 7, & 6, 8, & 6, 9, & 6, a \\
6, b, & 6, d, & 6, e, & 6, f, & 7, 8, & 7, 9, & 7, b, & 7, c, & 7, d, & 7, e \\
7, f, & 8, 9, & 8, a, & 8, b, & 8, f, & 9, b, & 9, c, & 9, d, & 9, f, & a, c \\
a, d, & 4, e, & a, f, & b, c, & b, d, & b, e, & b, f, & c, d, & c, e, & d, f \end{array} \right) \]
4.10 Given a graph $G$, define the **chromatic number** of $G$ to be

$$\chi(G) = \min\{k : G \text{ has a vertex } k\text{-coloring}\}$$

and define the **clique number** of $G$ to be

$$\omega(G) = \max\{k : G \text{ has a clique of size } k\}.$$  

Theorem 4.4 shows that $\omega(G) \leq \chi(G)$. 

(a) Show that strict inequality can hold in Theorem 4.4. That is, find a graph $G$ such that $\omega(G) < \chi(G)$.

(b) Show that for any integer $d \geq 0$ there is a graph $G$ with $\chi(G) - \omega(G) \geq d$.

4.11 An edge-decomposition of the complete graph $K_n$ into triangles is called a **Steiner triple system** of order $n$ (or, STS($n$)). More formally, an STS($n$) is a pair $(P, B)$ in which $P$ is an $n$-element set of points; $B$ is a collection of $n(n-1)/6$ 3-element subsets of $P$ called triples (or blocks); and every pair of points is contained in exactly one triple.

(a) Write a backtracking algorithm to find all STS($n$) (on a given set of $n$ vertices), and use your algorithm to determine the number of different STS(7).

(b) Define a graph $G = (V, E)$, where $V$ consists of the $\binom{n}{3}$ 3-subsets of an $n$-set, and two vertices are adjacent if and only if the intersection of the corresponding subsets has cardinality at most one. Show that an STS($n$) is equivalent to a (maximum) clique in $G$ having size $n(n-1)/6$.

(c) Using any of the clique-finding algorithms described in this chapter, determine the number of different STS(7).

4.12 If $x, y \in \{0, 1\}^n$, then recall that $\text{dist}(x, y)$ denotes the Hamming distance between $x$ and $y$. A **non-linear code** of length $n$ and minimum distance $d$ is a subset $C \subseteq \{0, 1\}^n$ such that $\text{dist}(x, y) \geq d$ for all $x, y \in C$. Denote by $A(n, d)$ the maximum number of $d$-tuples in length $n$ non-linear code of minimum distance $d$.

(a) Use a backtracking algorithm to compute $A(n, 4)$ for $n \leq 8$.

(b) Project: Determine the values of $A(9, 4)$ and $A(10, 4)$ (these values are more difficult to obtain; they are 20 and 40, respectively).

(c) Research problem: Determine the value of $A(11, 4)$ (the exact value of $A(11, 4)$ is unknown, but it is known that $72 \leq A(11, 4) \leq 79$).

4.13 A Latin square on the $n$-element set $\{1, 2, \ldots, n\}$ is said to be a **reduced Latin square** if the elements in the first row and in the first column occur in the natural order $1, 2, \ldots, n$. Write a backtracking program to determine the number of reduced Latin squares of order $n$. Run your algorithm for $n = 2, 3, 4$ and 5.

4.14 The **girth** of a graph is the size of the smallest circuit it contains. An $(r, g)$-**cage** is an $r$-regular graph of minimum order having girth $g$. Let $f(r, g)$ denote the number of vertices in an $(r, g)$-cage.

(a) Prove that

$$f(r, g) \geq \begin{cases} \frac{(r-1)^h - 2}{r-2} & \text{if } g = 2h + 1 \\ \frac{(r-1)^h - 2}{r-2} & \text{if } g = 2h. \end{cases}$$

Show that $f(3, 5) = 10$. 

**Hint:** the Petersen graph, presented in Exercise 1.13, is a $(3, 5)$-cage.

(b) Develop a backtracking algorithm to search for $(r, g)$-cages. Construct a $(3, 5)$-cage using your algorithm.

(c) Find some other examples of cages using your algorithm. (For example, it is known that $f(3, 6) = 14$ and $f(3, 7) = 24$.)