ıdom instances of the
4.13 is applied with
D

thm 4.13	
	REDUCEBOUND
	18
	1,287
	53,486
	1,326,640

(n^2) by Algorithm 4.11.
$O(n^2)$. Algorithm 4.13
problem that incorpo-

MINEDGEBOUND and
ed random instances of
vertices. The edge costs
Table 4.3 we report the
4.13 is used with these

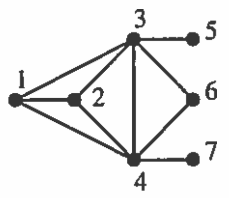

FIGURE 4.5
A graph with maximum clique $\{1, 2, 3, 4\}$.

4.6.3 The maximum clique problem

Recall that a maximum clique in a graph $\mathcal{G}$ 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 $\mathcal{G}$ 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.

Problem 4.5:	Maximum Clique
Instance:	A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Find:	a maximum clique of $\mathcal{G}$.

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 $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. 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_ℓ; 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 $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph, and $\mathcal{W} \subseteq \mathcal{V}$. The *induced subgraph* $\mathcal{G}[\mathcal{W}]$ has vertex set $\mathcal{W}$, and edge set

$$\{\{u, v\} \in \mathcal{E} : \{u, v\} \subseteq \mathcal{W}\}.$$

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 $\mathcal{G}[\mathcal{C}_\ell]$. Thus, we can obtain a bounding function by placing an upper bound on the size of a maximum clique in $\mathcal{G}[\mathcal{C}_\ell]$. If the size of a maximum clique in $\mathcal{G}[\mathcal{C}_\ell]$ is denoted by $\mathsf{mc}(\ell)$, and $\mathsf{mc}(\ell) \leq \mathsf{ub}(\ell)$, then

$$\mathsf{B}(X) = \ell + \mathsf{ub}(\ell)$$

is a bounding function.

Algorithm 4.14: MAXCLIQUE1 (ℓ)

```
global $A_\ell, B_\ell, \mathcal{C}_\ell \quad (\ell = 0, 1, \ldots, n-1)$
if $\ell > OptSize$
  then { $OptSize \leftarrow \ell + 1$
         $OptClique \leftarrow [x_0, \ldots, x_{\ell-1}]$
if $\ell = 0$
  then $\mathcal{C}_\ell \leftarrow \mathcal{V}$
  else $\mathcal{C}_\ell \leftarrow A_{x_{\ell-1}} \cap B_{x_{\ell-1}} \cap \mathcal{C}_{\ell-1}$
for each $x \in \mathcal{C}_\ell$
  do { $x_\ell \leftarrow x$
       MAXCLIQUE1($\ell + 1$)
main
  $OptSize \leftarrow 0$
  MAXCLIQUE1(0)
  output ($OptClique$)
```

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

$$\mathsf{mc}(\ell) \leq |\mathcal{C}_\ell|.$$

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

Algorithm 4.15: SIZEBOUND (X)

```
global $\mathcal{C}_\ell$
comment: $X = [x_0, \ldots, x_{\ell-1}]$
return ($\ell + |\mathcal{C}_\ell|$)
```

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

$$\mathsf{color} : \mathcal{V} \to \{0, \ldots, k-1\}$$

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

LEMMA 4.4 Let $\mathcal{G}$ be a gr
the maximum clique in $\mathcal{G}$ h

PROOF If vertices x and
the same clique.

Even though finding a ve
is not difficult to find k-colo
One easy way to do this is
greedy algorithms were int
vertices are processed in or
gorithm 4.16 presents such
vertex set is written as $\mathcal{V} =$
for some positive integer k
stored as a (global) array,
the algorithm constructs an
follows:

Color

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

Algorithm 4.16: GREED

```
global $color$
comment: $\mathcal{V} = \{0, \ldots,$
$k \leftarrow 0$
for $i \leftarrow 0$ to $n-1$
  do { $h \leftarrow 0$
       while $h < k$ and
       if $h = k$ then
       $ColorClass[h]$
       $\mathsf{color}[i] \leftarrow h$
return ($k$)
```

There are several ways
bounding function. One
before the backtracking a
color and it uses k colors
restricted to the vertices i
than k colors. The numl
bound on the size of a max
which we call the *samplin*

ique in $\mathcal{G}[C_\ell]$. If the size
nc(ℓ) $\leq$ ub(ℓ), then

ling functions. The sim-

lgorithm 4.15, which we

inding functions use the
if $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph
is a function

evance of vertex coloring
ving simple lemma.

LEMMA 4.4 *Let $\mathcal{G}$ be a graph, and suppose that $\mathcal{G}$ has a vertex k-coloring. Then the maximum clique in $\mathcal{G}$ 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 $\mathcal{V} = \{0, \ldots, n-1\}$. 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, *color*. In the process of constructing this coloring, the algorithm constructs an array of sets called *ColorClass*, which is defined as follows:

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

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

```
Algorithm 4.16: GREEDYCOLOR ($\mathcal{G} = (\mathcal{V}, \mathcal{E})$)

global color
comment: $\mathcal{V} = \{0, \ldots, n-1\}$
$k \leftarrow 0$
for $i \leftarrow 0$ to $n-1$
    do
        $h \leftarrow 0$
        while $h < k$ and $A_i \cap ColorClass[h] \neq \emptyset$ do $h \leftarrow h + 1$
        if $h = k$ then
            $k \leftarrow k + 1$
            $ColorClass[h] \leftarrow \emptyset$
        $ColorClass[h] \leftarrow ColorClass[h] \cup \{i\}$
        $color[i] \leftarrow h$
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 *color* and it uses k colors. For each induced subgraph $\mathcal{G}[C_\ell]$, the function *color*, restricted to the vertices in C_ℓ, defines a coloring of $\mathcal{G}[C_\ell]$ 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 $\mathcal{G}[C_\ell]$. The resulting bounding function, which we call the *sampling bound*, is presented in Algorithm 4.17.

```
Algorithm 4.17: SAMPLINGBOUND ($X$)

  global $\mathcal{C}_\ell$, $color$
  comment: $X = [x_0, \ldots, x_{\ell-1}]$
  return $(\ell + |\{color[x] : x \in \mathcal{C}_\ell\}|)$
```

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

```
Algorithm 4.18: GREEDYBOUND ($X$)

  external GREEDYCOLOR()
  global $\mathcal{C}_\ell$
  comment: $X = [x_0, \ldots, x_{\ell-1}]$
  $k \leftarrow$ GREEDYCOLOR($\mathcal{G}[\mathcal{C}_\ell]$)
  return $(\ell + k)$
```

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 $M \leq OptSize$ is true in every iteration of the loop. This is because the value of $OptSize$ 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: MAXCL

  external B()
  global $A_\ell, B_\ell, \mathcal{C}_\ell$   $(\ell =$
  if $\ell > OptSize$
    then { $OptSize \leftarrow \ell$
         { $OptClique \leftarrow [$
  if $\ell = 0$
    then $\mathcal{C}_\ell \leftarrow \mathcal{V}$
    else $\mathcal{C}_\ell \leftarrow A_{x_{\ell-1}} \cap B_{x_\ell}$
  $M \leftarrow$ B($[x_0, \ldots, x_{\ell-1}]$)
  for each $x \in \mathcal{C}_\ell$
    do { if $M \leq OptSize$
       {   then return
       { $x_\ell \leftarrow x$
       { MAXCLIQUE2($\ell$
  main
    $OptSize \leftarrow 0$
    MAXCLIQUE2(0)
    output ($OptClique$)
```

```
Algorithm 4.20: GENER

             { RandomInte
  external { SUBSETLEX
             { KSUBSETLE
  $r \leftarrow$ RandomInteger(0,
  $T \leftarrow$ SUBSETLEXUNRA
  $\mathcal{E} \leftarrow \emptyset$
  for each $j \in T$
    do { $\{x, y\} \leftarrow$ KSUBS
       { $\mathcal{E} \leftarrow \mathcal{E} \cup \{x - 1$
  return ($\mathcal{G} = (\{0, \ldots, n$
```

The only aspect of Alg line, where we add the ed KSUBSETLEXUNRANK r subset of $\{0, \ldots, n-1\}$. be included in $\mathcal{E}$.

Notice that the expectec ing was done were denot

ı a bounding function is
] every time we want to
ıg function is called the

ː (or any other bounding
acktracking algorithm as

ding functions, we check
eration of the loop. This
›rithm progresses, and so
ring to add a new node to

space tree, for graphs of
ˈerent bounding functions
l the size of the maximum
rated at random from the
vays to do this. One nice
loped in Chapter 2. Note
om integer in the interval
class $\mathcal{G}(n)$.

Algorithm 4.19: MAXCLIQUE2 (ℓ)

```
external B()
global $A_\ell, B_\ell, \mathcal{C}_\ell$   $(\ell = 0, 1, \ldots, n-1)$
if $\ell > OptSize$
  then { $OptSize \leftarrow \ell$
         $OptClique \leftarrow [x_0, \ldots, x_{\ell-1}]$
if $\ell = 0$
  then $\mathcal{C}_\ell \leftarrow \mathcal{V}$
  else $\mathcal{C}_\ell \leftarrow A_{x_{\ell-1}} \cap B_{x_{\ell-1}} \cap \mathcal{C}_{\ell-1}$
$M \leftarrow$ B$([x_0, \ldots, x_{\ell-1}])$
for each $x \in \mathcal{C}_\ell$
  do { if $M \leq OptSize$
         then return
       $x_\ell \leftarrow x$
       MAXCLIQUE2$(\ell + 1)$
main
 $OptSize \leftarrow 0$
 MAXCLIQUE2(0)
 output ($OptClique$)
```

Algorithm 4.20: GENERATERANDOMGRAPH (n)

```
external { RandomInteger()
           SUBSETLEXUNRANK()
           KSUBSETLEXUNRANK()
$r \leftarrow$ RandomInteger$(0, 2^{\binom{n}{2}} - 1)$
$T \leftarrow$ SUBSETLEXUNRANK$(\binom{n}{2}, r)$
$\mathcal{E} \leftarrow \emptyset$
for each $j \in T$
  do { $\{x, y\} \leftarrow$ KSUBSETLEXUNRANK$(j, 2, n)$
       $\mathcal{E} \leftarrow \mathcal{E} \cup \{x-1, y-1\}$
return $(\mathcal{G} = (\{0, \ldots, n-1\}, \mathcal{E}))$
```

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 $\mathcal{E}$. This is because the algorithm 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 $\mathcal{E}$.

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 $\overline{c}(n)$, and some values of $\overline{c}(n)$

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

number of vertices	50	100	150	200	250
number of edges	607	2535	5602	9925	15566
size of maximum clique	7	9	10	11	11
bounding function					
none	8687	257145	1659016	7588328	26182672
size bound	3204	57225	350310	1434006	5008767
sampling bound	2268	44072	266246	1182514	4093535
greedy bound	430	5734	22599	91671	290788

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

number of vertices	25	50	75	100	125
number of edges	236	959	2045	3720	5780
size of maximum clique	11	14	15	17	18
bounding function					
none	25570	2083770	12385596	186543706	1414266577
size bound	1840	91663	426279	5370268	35108264
sampling bound	794	37218	195567	2225982	15615755
greedy bound	91	2843	10476	70404	413421

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 $\binom{n}{2}$ (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 δ, $0 \leq \delta \leq 1$, Algorithm 4.21 can be used. In this algorithm the function 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.

```
Algorithm 4.21: GENERATERANDOMGRAPH2 ($n, \delta$)

external Random()
for $x \leftarrow 0$ to $n - 2$
  do for $y \leftarrow x + 1$ to $n - 1$
       do $r \leftarrow$ Random(0, 1)
          if $r \geq 1 - \delta$
            then $\mathcal{E} \leftarrow \mathcal{E} \cup \{x - 1, y - 1\}$
return ($\mathcal{G} = (\{0, \ldots, n - 1\}, \mathcal{E})$)
```

Tables 4.4 and 4.5 show
cantly as better bounding fu
for a bounding function dep
ing function, and on the an
duced. The relative compu
depend heavily on the impl
servations on the complexit
graph having n vertices, the
fore the greedy bound is cor
bound, on the other hand, c
rithms. Hence, there is a tr
a slower computation time.
algorithm for "large enougl
on the implementation and

4.7 Branch and bou

Another way in which we c
called *branch and bound*. T
ine each of the choices $x_\ell \in$
recursively for each choice.
termine the order in which
algorithm for a general max

We illustrate the branch a
problem. Suppose $X = [x_0$
of the Traveling Salesma
$1) - (\ell - 1) = n - \ell$ choi
corresponding to the partial
children of X in increasing
in this order.

phs with edge density .5

200	250
9925	15566
11	11
7588328	26182672
1434006	5008767
1182514	4093535
91671	290788

hs with edge density .75

100	125
3720	5780
17	18
186543706	1414266577
5370268	35108264
2225982	15615755
70404	413421

hese values to the exper-

f its edges to $\binom{n}{2}$ (which
phs generated by Algo-
obtain a random graph
can be used. In this algo-
al number in the interval
for randomly generated

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 $O(|\mathcal{C}_\ell|^2)$. The size bound and sampling bound, on the other hand, can be computed in time $O(|\mathcal{C}_\ell|)$ 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 $x_\ell \in \mathcal{C}_\ell$ 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 $X = [x_0, x_1, x_2, \ldots, x_{\ell-1}]$ 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_ℓ. 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_ℓ. There is no particular reason to proceed in this order.