Chapter 16
Heuristic Search

► Part I. Preliminaries
▼ Part II. Tightly Coupled Multicore
    Chapter 6. Parallel Loops
    Chapter 7. Parallel Loop Schedules
    Chapter 8. Parallel Reduction
    Chapter 9. Reduction Variables
    Chapter 10. Load Balancing
    Chapter 11. Overlapping
    Chapter 12. Sequential Dependencies
    Chapter 13. Strong Scaling
    Chapter 14. Weak Scaling
    Chapter 15. Exhaustive Search
Chapter 16. Heuristic Search
    Chapter 17. Parallel Work Queues
► Part III. Loosely Coupled Cluster
► Part IV. GPU Acceleration
► Part V. Map-Reduce
► Appendices
The exhaustive search program for the minimum vertex cover problem in Chapter 15 is guaranteed to find a minimum cover. But because the problem size is an exponential function of the number of vertices—\( N = 2^V \)—the program’s running time is too long to be practical for larger graphs, even on a parallel computer. If I want to tackle larger problem sizes, I’m going to need a different approach.

An alternative to exhaustive search is **heuristic search**. Rather than looking at every possible candidate solution, a heuristic search program looks at only a selected number of candidate solutions. The candidates are chosen by some rule, or *heuristic*. The heuristic attempts to select candidates that have a higher likelihood of solving the problem. Of the selected candidates, the program chooses the one that yields the best solution. Now the problem size \( N \) is the number of candidates selected rather than an exponential function. You can specify the number of candidates to yield a running time that you are willing to tolerate.

There are several general approaches for heuristic search programs aimed at solving problems with exponentially large problem spaces, such as simulated annealing, genetic algorithms, and tabu search. I’ve worked with all of these approaches. In my experience, they tend to require rather intricate data structures and algorithms. They also tend to depend on parameters that have to be tuned to obtain good solutions, and it’s often not clear what the optimum parameter settings are. Furthermore, these are inherently sequential algorithms, and it’s difficult to parallelize them.

To solve exponentially hard problems on parallel computers, I prefer a different approach, which I call *massively parallel randomized approximation (MPRA)*. An MPRA program generates a very large (but not exponentially large) number of candidate solutions, chosen at random using a simple heuristic. The program then evaluates all the candidates and reports the one that yields the best solution.

Because the candidates can all be generated and evaluated independently of each other, an MPRA program is trivial to parallelize. It is just a parallel loop to generate all the candidates, combined with a parallel reduction to choose the best candidate. An MPRA program also exhibits near-ideal weak scaling. If you increase the number of cores by some factor, you can increase the number of candidates by the same factor while keeping the running time the same. Increasing the number of candidates examined might increase the likelihood of finding a better solution.

There’s a catch, though. Because an MPRA program searches only some of the candidate solutions, not all possible candidate solutions, an MPRA program is not *guaranteed* to find the absolutely best solution. However, an MPRA program might be able to find an *approximate* solution that’s only a little worse than the absolutely best solution. This approximate solution
might still be useful for practical purposes—and it can be found in a practical amount of time.

Let’s apply these considerations to the minimum vertex cover problem. I need a heuristic for generating a random candidate solution, namely a random vertex cover, that I hope will be close to a minimum vertex cover. Many such heuristics could be envisioned. Here’s one that’s particularly simple: Start with an empty subset of vertices; repeatedly add a vertex chosen at random from those not yet in the subset; stop as soon as the subset is a cover; and use that as the candidate solution. Generate a large number of such candidates in parallel, and keep the one with the fewest vertices. (If multiple candidates are tied for the fewest vertices, any of them will do.) This heuristic doesn’t guarantee that a candidate cover will be a true minimum cover; depending on the order in which vertices are added, the candidate might end up with more vertices than necessary to cover all the edges. Still, by stopping as soon as the subset becomes a cover, the hope is that the candidate will be close to a minimum cover, if not actually a minimum cover.

How effective is this heuristic at finding a minimum vertex cover? It’s difficult to say in general. I wrote a program to study the question for smaller graphs (class edu.rit.pj2.example.MinVerCovDist in the Parallel Java 2 Library). The program generated a given number of random graphs; I used 100 random graphs in my study. Each random graph had a given number of vertices $V$; I used 20, 22, 24, 26, 28, 30, and 32 vertices. Each random graph had a given number of edges $E$; I chose $E$ to yield graphs with densities of 0.2, 0.4, 0.6, and 0.8. A graph’s density $D$ is the ratio of the actual number of edges to the total possible number of edges between $V$ vertices,

$$D = \frac{E}{V(V-1)/2} .$$  \hspace{1cm} (16.1)

For each random graph, the program did an exhaustive search to find the size of a minimum vertex cover; counted the total number of covers and the number of minimum covers; and computed the minimum cover fraction $F$,

$$F = \frac{\text{Number of minimum covers}}{\text{Total number of covers}} .$$  \hspace{1cm} (16.2)

Figure 16.1 plots the median $F$ over all 100 random graphs for each value of $V$ and $D$.

Why look at the minimum cover fraction $F$? Assume that the heuristic procedure yields a cover chosen at random from all the possible covers. Then roughly speaking, $F$ is the probability that the heuristic procedure will land on a minimum cover. This in turn tells me that the expected number of trials before landing on a minimum cover is $1/F$. So if I set the number of candidates at $10/F$, or $100/F$, there ought to be a reasonable chance that the MPRA program will find an actual minimum cover.
Unfortunately, because the program I used for my study does an exhaustive search, I was not able to study graphs with more than 32 vertices. Still, some trends are apparent in Figure 16.1. When I increased the number of vertices by 12 (from 20 to 32), $F$ went down by about a factor of 10. This means that $F$ is proportional to $10^{-V/12}$. I can then extrapolate the curves to get a rough $F$ value for larger $V$ values. For example, consider the $D = 0.4$ curve. Going from $V = 20$ to $V = 50$ multiplies $F = 1 \times 10^{-2}$ by a factor of $10^{-(50-20)/12}$, yielding $F = 3.16 \times 10^{-5}$. So doing $100/F = \text{about three million}$ candidates, the MPRA program ought to find a vertex cover pretty close to a minimum vertex cover for a 50-vertex random graph of density 0.4. Three million candidates is a lot fewer than the $2^{50}$, or about one quadrillion, candidates the exhaustive search program would have to examine.

However, this trend also shows that the MPRA program’s running time still increases exponentially—if I want a decent chance of it finding an actual minimum vertex cover. Each time the number of vertices increases by 12, the number of candidates has to increase by a factor of 10. At some point I’ll have to stop increasing the number of candidates to keep the running time reasonable; but then the program might not be able to find a true minimum vertex cover. How close can it come? There’s no way to know for sure, other

---

**Figure 16.1.** Minimum cover fraction versus vertices and density
package edu.rit.pj2.example;
import edu.rit.pj2.LongLoop;
import edu.rit.pj2.Task;
import edu.rit.pj2.vbl.BitSetVbl;
import edu.rit.util.BitSet;
import edu.rit.util.GraphSpec;
import edu.rit.util.Instance;
import edu.rit.util.IntAction;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
public class MinVerCovSmp3
extends Task
{
    // Graph being analyzed.
    AMGraph graph;
    int V;

    // Minimum vertex cover.
    BitSetVbl minCover;

    // Main program.
    public void main
    (String[] args)
    throws Exception
    {
        // Parse command line arguments.
        if (args.length != 3) usage();
        final String ctor = args[0];
        final long seed = Long.parseLong (args[1]);
        final long N = Long.parseLong (args[2]);

        // Construct graph spec, set up adjacency matrix.
        graph = new AMGraph ((GraphSpec) Instance.newInstance (ctor));
        V = graph.V();

        // Check N randomly chosen candidate covers.
        minCover = new BitSetVbl.MinSize (new BitSet (V));
        parallelFor (0L, N - 1) .exec (new LongLoop()
        {
            BitSetVbl thrMinCover;
            BitSet candidate;
            Random prng;
            RandomSubset rsg;
            public void start()
            {
                thrMinCover = threadLocal (minCover);
                candidate = new BitSet (V);
                prng = new Random (seed + rank());
                rsg = new RandomSubset (prng, V, true);
            }
            public void run (long i)
            {
                candidate.clear();
                rsg.restart();
                while (! graph.isVertexCover (candidate))
                    candidate.add (rsg.next());
        });
    }
}
than by comparing the MPRA program’s results to the exhaustive search program’s results.

Setting aside questions about how close the program can come to finding a minimum vertex cover, let’s examine the code for the minimum vertex cover MPRA program, class edu.rit.pj2.example.MinVerCovSmp3 (Listing 16.1).

Like the programs in Chapter 15, I need a class to represent a vertex set. This time, however, I don’t want to be limited to at most 63 vertices, because I’m no longer doing an exhaustive search. Instead, I want a vertex set that can support an arbitrary number of vertices. I still want to use a bitset data structure. Instead of using class edu.rit.util.BitSet64, which can only hold 64 elements, I’ll use class edu.rit.util.BitSet, which can accommodate an arbitrary number of elements. For doing parallel reduction, I’ll use class edu.rit.pj2.vbl.BitSetVbl.

To represent the graph itself, I’ll use class edu.rit.pj2.example.AMGraph. This is the same as class AMGraph64 from Chapter 15, except class AMGraph uses class BitSet for its internal adjacency matrix (array of vertex set objects).

The MinVerCovSmp3 main program’s command line arguments are the graph spec constructor expression, the pseudorandom number generator seed, and \( N \), the number of random candidate covers to generate. The program starts by creating the graph spec object and setting up the graph’s adjacency matrix, as in the previous programs. This time the parallel loop iterates over the \( N \) candidates (line 39). As in the previous programs, each parallel team thread has a per-thread minimum vertex cover variable (line 41) that is linked to the global minimum vertex cover reduction variable (line 47). Taking to heart the lesson from Chapter 15, each team thread also has a candidate vertex cover variable (line 42) that the program will reuse on each parallel loop iteration.

Each parallel team thread needs to generate its own series of random candidate covers. So each thread gets its own per-thread pseudorandom number generator, seeded differently in each thread (line 49). To generate a random candidate, each team thread needs to generate a series of vertices, chosen \( \textit{without replacement} \) from the set of all vertices 0 through \( V - 1 \)—that is, a \( \textit{random subset} \) of the set of all vertices. To do so, the program uses a \( \textit{random subset generator} \), an instance of class edu.rit.util.RandomSubset in the Parallel Java 2 Library, layered on top of the per-thread pseudorandom number generator (line 50).

Each parallel loop iteration (lines 54–59) performs the heuristic procedure for generating a random cover: Clear the candidate back to an empty set; restart the random subset generator to obtain a new random subset; as long as the candidate is not a cover, get a random vertex from the random subset generator and add the vertex to the candidate. As soon as the candidate
becomes a cover, stop; and if the candidate is smaller than the per-thread
minimum vertex cover variable, copy the candidate there, thus retaining
the smallest cover seen. When the parallel loop finishes, the per-thread minimum
covers are automatically reduced under the hood into the global minimum
cover, which the program prints.

I ran the exhaustive search MinVerCovSmp2 program on 12 cores of a
tardis node on a graph with 40 vertices and 312 edges (density 0.4). I ran
the heuristic search MinVerCovSmp3 program on 12 cores of a tardis node
on the same graph, generating 10 million random candidate covers. Here is
what the programs printed:

$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSmp2 \
"edu.rit.util.RandomGraph(40,312,348576)"
Cover = 0 1 2 3 4 5 7 8 9 10 11 12 13 14 17 18 19 21 22 23 24
25 26 27 31 32 33 36 37 38 39
Size = 31
Job 1 makespan 957095 msec
The heuristic search program found a cover that was almost, but not quite, a true minimum vertex cover; 32 vertices instead of 31. But the heuristic search program’s running time was just 2.5 seconds rather than over 15 minutes. These results are typical of heuristic search programs; the running times are much less—small enough to be practical, especially for larger problem sizes—but the solutions are only approximate, although close to the true solutions.

To see if the heuristic search program could find a true minimum cover, I increased the number of candidates from 10 million to 100 million. Here is what the program printed:

$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSmp3 \
"edu.rit.util.RandomGraph(40,312,348576)" 23576879 100000000 
Cover = 0 1 2 3 4 5 7 8 9 10 11 12 13 14 17 18 19 21 22 23 24 
25 26 27 31 32 33 36 37 38 39 
Size = 31 
Job 1 makespan 23928 msec

This time the heuristic search program found a true minimum cover in about 24 seconds.

To study the heuristic search program’s performance under weak scaling, I ran the sequential MinVerCovSeq3 program on one core and the multicore parallel MinVerCovSmp3 program on one to 12 cores of a tardis node and measured the running times. I ran the programs on five random graphs, with $V = 50, 100, 150, 200, \text{ and } 250$ and a density of 0.2. The number of candidate covers was 25 million on one core, 50 million on two cores, and so on, up to 300 million on 12 cores. Figure 16.2 plots the program’s running times, sizeups, and efficiencies. The program exhibits good weak scaling, with nearly constant running times and with efficiencies of 0.9 or better out to 12 cores.

An exhaustive search program would require $2^{50}$ to $2^{250}$ iterations to find the exact minimum vertex covers for these 50- to 250-vertex graphs—far too long to be practical. The heuristic search MinVerCovSmp3 program found approximate solutions with about one to three minutes of computation, depending on the graph’s size.

How close were the approximate solutions to the exact solutions? There’s no way to know without doing an exhaustive search. However, in general, the size of the minimum vertex cover the heuristic search finds does go down a bit as the number of iterations increases. The table below shows the sizes of
Figure 16.2. MinVerCovSmp3 weak scaling performance metrics
the solutions the MinVerCovClu3 program found for each graph after doing 25 million iterations and after doing 300 million iterations:

<table>
<thead>
<tr>
<th>Graph size (vertices)</th>
<th>Cover size, 25 million iterations</th>
<th>Cover size, 300 million iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>38</td>
<td>37</td>
</tr>
<tr>
<td>100</td>
<td>88</td>
<td>87</td>
</tr>
<tr>
<td>150</td>
<td>137</td>
<td>137</td>
</tr>
<tr>
<td>200</td>
<td>188</td>
<td>187</td>
</tr>
<tr>
<td>250</td>
<td>238</td>
<td>237</td>
</tr>
</tbody>
</table>

For four out of five of these graphs, by doing weak scaling from one core to 12 cores, the program was able to shave one vertex off the size of the (approximate) minimum vertex cover. In other words, the parallel program got a more accurate solution in roughly the same amount of time as the sequential program. As mentioned in Chapter 14, this is one of the primary rationales for weak scaling.

**Under the Hood**

Class edu.rit.util.BitSet32 uses a single value of type int (a private field) to hold the bitmap. Class BitSet64 uses a single value of type long to hold the bitmap. To accommodate an arbitrary number of elements, class BitSet uses an array of one or more ints to hold the bitmap. The required number of set elements is specified as an argument to the BitSet constructor, which allocates an int array large enough to hold that many bits. Class BitSet’s methods are implemented in the same way as class BitSet32 and BitSet64, except the methods loop over all the bitmap array elements.

The minimum vertex cover program needs to create a random subset of the set of vertices in the graph. This amounts to creating a random subset of the integers 0 through \( V - 1 \). To do so, the program uses class RandomSubset in package edu.rit.util in the Parallel Java 2 Library. To generate a random subset, this class has to pick integers at random, with equal probability, from the set \{0, 1, 2, \ldots, V - 1\}, without replacement. It’s the “without replacement” requirement that makes this process a bit tricky.

Here’s the wrong way to generate a random subset with \( k \) elements:

1. Subset ← \{\}
2. Repeat \( k \) times:
   3. Repeat:
      4. \( n \leftarrow \) Random integer in the range 0 through \( V - 1 \)
      5. Until \( n \) is not in the subset
      6. Add \( n \) to the subset
This is the wrong way to do it because of the open-ended loop on lines 3–5. That loop might have to iterate several times before finding a random integer that is not in the subset; and the more elements there are in the subset, the more iterations the loop might have to do. This wastes CPU time, which is especially problematic in a program that has to generate an enormous number of random subsets.

Here’s the right way to generate a random subset with \( k \) elements. The algorithm revolves around an array \( S \) with \( V \) elements.

1. For \( i = 0 \) to \( V - 1 \):
   2. \( S[i] \leftarrow i \)
3. For \( i = 0 \) to \( k - 1 \):
   4. \( j \leftarrow \) Random integer in the range 0 through \( V - 1 - i \)
   5. Swap \( S[i] \) and \( S[i + j] \)
   6. Subset \( \leftarrow S[0] \) through \( S[k - 1] \)

Why does this procedure work? The \( S \) array starts out containing each possible integer from 0 through \( V - 1 \) (lines 1–2). At each iteration of the loop on lines 3–5, the next element of the array is swapped with one of the available integers from the remainder of the array, chosen at random. Each available integer is chosen with equal probability. After \( k \) iterations, the first \( k \) elements of the array end up containing a randomly chosen subset of the integers 0 through \( V - 1 \).

Here’s an example of choosing a random four-element subset of the integers 0 through 9:

\[
S = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] \quad i = 0 \quad j = 3 \\
S = [3, 1, 2, 0, 4, 5, 6, 7, 8, 9] \quad i = 1 \quad j = 5 \\
S = [3, 6, 2, 0, 4, 5, 1, 7, 8, 9] \quad i = 2 \quad j = 0 \\
S = [3, 6, 2, 0, 4, 5, 1, 7, 8, 9] \quad i = 3 \quad j = 2 \\
S = [3, 6, 2, 5, 4, 0, 1, 7, 8, 9]
\]

The random subset is \{3, 6, 2, 5\}, or \{2, 3, 5, 6\}. (For a set, the order of the elements doesn’t matter.)

If you perform this procedure and iterate through the entire array (by setting \( k \) equal to \( V \)), the array elements—taken in order—comprise a random permutation of the integers 0 through \( V - 1 \). So class RandomSubset is useful both for generating random subsets and for generating random permutations.

Why is this procedure the right way to generate a random subset or a random permutation? Because the loop body (lines 4–5) is executed precisely once for each of the \( k \) elements in the subset or permutation. The procedure no longer has an open-ended loop like the previous version. This minimizes the CPU time required.

Class RandomSubset implements the above procedure. The constructor initializes the internal \( S \) array (lines 1–2) and initializes \( i \) to 0. The \texttt{next()}
method performs the loop body (lines 4–5) and returns \( S[i] \). (You write the enclosing loop yourself.) The \texttt{restart()} method reinitializes \( S \) and \( i \); this lets you generate multiple random subsets from the same \texttt{RandomSubset} object.

**Points to Remember**

- A heuristic search solves a problem by looking at a limited number of candidate solutions, generating using a heuristic, and keeping the best solution.
- The heuristic attempts to generate solutions that are close to the optimum solution.
- A massively parallel randomized approximation (MPRA) program generates and evaluates a large number of random candidate solutions, in parallel, using a heuristic.
- Because it does not consider all possible solutions, a heuristic search is not guaranteed to find a true optimum solution. However, it might find an approximate solution that is close enough to the true optimum solution for practical purposes.
- An MPRA program is easy to write, is trivial to parallelize, and exhibits good weak scaling behavior.
- Scaling an MPRA program up to more cores, thereby examining more candidate solutions, increases the likelihood of finding a better solution.

- Use class \texttt{edu.rit.util.RandomSubset} in the Parallel Java 2 Library to generate a random subset or a random permutation of a set of integers.