Chapter 12
Heuristic Search

- Part I. Preliminaries
- Part II. Tightly Coupled Multicore
  - Chapter 2. Parallel Loops
  - Chapter 3. Parallel Loop Schedules
  - Chapter 4. Parallel Reduction
  - Chapter 5. Reduction Variables
  - Chapter 6. Load Balancing
  - Chapter 7. Overlapping
  - Chapter 8. Sequential Dependencies
  - Chapter 9. Strong Scaling
  - Chapter 10. Weak Scaling
  - Chapter 11. Exhaustive Search
  - Chapter 12. Heuristic Search
- Part III. Loosely Coupled Cluster
- Part IV. GPU Acceleration
- Part V. Map-Reduce
The exhaustive search program for the minimum vertex cover problem in Chapter 11 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 ran-
dom 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 ran-
dom 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 can-
didates 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 heuris-
tic 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 Li-
brary). 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 ver-
tices \( V \); 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,

\[
D = \frac{E}{V(V-1)/2} .
\]  

(12.1)

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 num-
er of minimum covers; and computed the minimum cover fraction \( F \),

\[
F = \frac{\text{Number of minimum covers}}{\text{Total number of covers}} .
\]  

(12.2)

Figure 12.1 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 candi-
dates 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 exhaus-
tive search, I was not able to study graphs with more than 32 vertices. Still, some trends are apparent in Figure 12.1. 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 = 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 than by comparing the MPRA program’s results to the exhaustive search program’s results.

Figure 12.1. 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.Random;
import edu.rit.util.RandomSubset;
import java.io.File;
import java.util.Scanner;
public class MinVerCovSmp3
extends Task
{

// Number of vertices and edges.
  int V;
  int E;

  // The graph’s adjacency matrix. adjacent[i] is the set of
  // vertices adjacent to vertex i.
  BitSet[] adjacent;

  // Minimum vertex cover.
  BitSetVbl minCover;

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

    // Read input file, set up adjacency matrix.
    Scanner s = new Scanner (file);
    V = s.nextInt();
    E = s.nextInt();
    if (V < 1) usage ("V must be >= 1");
    adjacent = new BitSet [V];
    for (int i = 0; i < V; ++ i)
      adjacent[i] = new BitSet (V);

    // Check N randomly chosen candidate covers.
    minCover = new BitSetVbl.MinSize (V) .add (0, V);
    parallelFor (0L, N - 1) .exec (new LongLoop()
      {
        BitSetVbl thrMinCover;
        BitSet candidate;
        Random prng;
      });
  }
}

Listing 12.1. MinVerCovSmp3.java (part 1)
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 12.1).

Like the programs in Chapter 11, 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 its subclass edu.rit.pj2.vbl.BitSetVbl.

The MinVerCovSmp3 main program’s command line arguments are the graph file, the pseudorandom number generator seed, and $N$, the number of random candidate covers to generate. The program starts by reading the graph file and setting up the graph’s adjacency matrix, as in the previous programs. This time the parallel loop iterates over the $N$ candidates (line 54). As in the previous programs, each parallel team thread has a per-thread minimum vertex cover variable (line 56) that is linked to the global minimum vertex cover reduction variable (line 62). Taking to the lesson from Chapter 11, each team thread also has a candidate vertex cover variable (line 57) 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 64). To generate a random candidate, each team thread needs to generate a series of vertices, chosen without replacement from the set of all vertices 0 through $V–1$—that is, a random subset of the set of all vertices. To do so, the program uses a 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 65).

Each parallel loop iteration (lines 67–75) 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 four cores of a tardis node on a graph with 40 vertices and 312 edges (density 0.4). I ran
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 (! isCover (candidate))
        candidate.add (rsg.next());
    if (candidate.size() < thrMinCover.size())
        thrMinCover.copy (candidate);  
}  

// Print results.
System.out.printf ("Cover =");
for (int i = 0; i < V; ++ i)
    if (minCover.contains (i))
        System.out.printf (" %d", i);
System.out.println();
System.out.printf ("Size = %d\n", minCover.size());

// Returns true if the given candidate vertex set is a cover.
private boolean isCover
    (BitSet candidate)
{
    boolean covered = true;
    for (int i = 0; covered && i < V; ++ i)
        if (! candidate.contains (i))
            covered = adjacent[i].isSubsetOf (candidate);
    return covered;
}

// Print an error message and exit.
private static void usage
(String msg)
{
    System.err.printf ("MinVerCovSmp3: %s\n", msg);
    usage();
}

// Print a usage message and exit.
private static void usage()
{
    System.err.println ("Usage: java pj2 " +
            "edu.rit.pj2.example.MinVerCovSmp3 <file> <seed> <N>\n");
    System.err.println ("<file> = Graph file");
    System.err.println ("<seed> = Random seed");
    System.err.println ("<N> = Number of trials");
    throw new IllegalArgumentException();
}

Listing 12.1. MinVerCovSmp3.java (part 2)
the heuristic search MinVerCovSmp3 program on four cores of a tardis node on the same graph, generating 100 million random candidate covers. Here is what the programs printed:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSmp2 \ 
g40.txt
Cover = 0 2 3 4 6 8 9 11 12 13 14 16 18 19 20 21 22 23 24 25 26 27 30 31 32 33 34 36 37 38 39
Size = 31
Job 62 makespan 6113078 msec

$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSmp3 \ 
g40.txt 23576879 100000000
Cover = 0 2 3 4 5 6 8 9 10 11 12 13 14 16 19 20 21 22 23 24 25 26 27 30 31 32 33 34 36 37 38 39
Size = 32
Job 64 makespan 76399 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 a little over a minute rather than nearly two hours. These results are typical of heuristic search programs; the running times are more practical, 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 100 million to one billion. Here is what the program printed:

```
$ java pj2 debug=makespan edu.rit.pj2.example.MinVerCovSmp3 \ 
g40.txt 23576879 1000000000
Cover = 0 2 3 4 6 8 9 11 12 13 14 16 18 19 20 21 22 23 24 25 26 27 30 31 32 33 34 36 37 38 39
Size = 31
Job 63 makespan 771547 msec
```

This time the heuristic search program found a true minimum cover in about 13 minutes.

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 four cores of a tardis node and measured the running times. I ran the programs on five random graphs, with $V = 50, 100, 150, 200,$ and $250;$ and $E = 500, 1000, 1500, 2000,$ and $2500.$ I did 10 million candidate covers on one core, 20 million on two cores, 30 million on three cores, and 40 million on four cores. Figure 12.2 plots the program’s running times and efficiencies. Although slightly less than ideal, the efficiencies droop very little as the number of cores increases, evincing good weak scaling behavior.
Chapter 12. Heuristic Search

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.

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.

Figure 12.2. MinVerCovSmp3 weak scaling performance metrics
• 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.