CSCE 686 Homework 6 - SCP Heuristics

Jon Knapp and Justin Fletcher

April 26, 2016

1 SCP Heuristic Design and Development [a, b]

In this section, we apply the disciplined design process advocated in [1] to construct the algorithms fundamental to the Set Covering Problem (SCP). Additionally, we introduce three heuristics to the algorithm with the goal of reducing the search number of states which must be visited.

1.1 Problem Domain Description: SCP

Given a set of elements: $E = \{e_1, ..., e_n\}$, a family of subsets, $\{S_1, ..., S_m\} \subseteq 2^E$, and weights $w_j \geq 0$ for each $j \in \{1, ..., m\}$, the SCP is defined by the following formula:

$$I \subseteq \{1, ..., m\} \min \sum_{J \in I} w_j$$
$$s.t. \bigcup_{J \in I} S_j = E$$

The input domain of the problem, which is a set of elements, E, can be specified by a set. The family of sets, S, is a set of sets, where each subset contains elements E' and cost c. Formally:

- Input Domain D_i: A set of elements, E, and a set of sets.
  - E: Elements

- S(E', c): Set of sets
    - E': Set of elements where $E' \in E$
    - c: Cost of set

- Output Domain D_o: A Set of sets, B, such that $B \in S$, and B satisfies set minimum covering property.

- Input Function $I(E, S)$: Determines if the input conditions on the input E and S are satisfied. The required input conditions for this domain is that, for all $S(E', c)$, $E' \in E$.

- Output Function: $O(B)$: Determines if the output conditions are met. The conditions are given met when:

$$I \subseteq \{1, ..., m\} \min \sum_{J \in I} w_j$$
$$s.t. \bigcup_{J \in I} S_j = E$$

1.2 Problem Domain and Algorithm Domain Integration

The algorithm domain to which the SCP problem will be mapped in this work is the global search via depth-first search with backtracking (GS/DFS/BT) algorithm. Thus, the algorithm domain used in this work is that of GS/DF-S/BT, which is described in [1]. In order to use this algorithm to solve the SCP problem, the SCP domain must be mapped to the GS/DFS/BT domain. This integration is accomplished as follows:

- GS/DFS/BT Basic Search Constructs
    - **initialization(D_i)**: Initializes T, where T is a tableau. T consists of M blocks of columns, one for each e_k of E. The kth block will consist of the sets of S that cover e_k, but do not contain lower numbered elements $e_1, ..., e_{k-1}$.
    - **next-state-generator(D_i)**: Returns B, where $B \in S$.
    - **selection(D_i)**: Returns b where $b \in S$. Generally, the specific set chosen is selected from a block of T in such a way as to maximize

the coverage of the elements E. In the case of SCP, we require that all elements E are covered and that the solution is the minimum cover for E.

  - **feasibility**: Returns a Boolean, which is true if, for some $b \subset S$, E is covered and is false otherwise.
  - **solution(B, z)**: Returns a Boolean which is true if S covers E and is the minimum cover of E, i.e. z is minimized.
  - **objective**:Returns a set D_o, which in this algorithm is a minimum cover of E.

- Delay Termination: Because the previous GS/DFS/BT search only finds a minimal set cover, rather than a minimum set cover, we must prevent the algorithm from terminating until all covering sets have been either implicitly or explicitly examined.
  - In some as yet undefined loop, iterate finding all minimal set covers possible in the problem instance.
  - Repeat the GS/DFS/BT search for SCP, avoiding duplication where possible.

1.3 Algorithm Domain Specification Refinement

In order to further refine this design, in pursuit of executable code, we must disambiguate several operations. We define a candidate solution set to be $B \subseteq S$. We first specify the next state generation and selection functions. The next state should be a state which has not been previously selected.

The algorithm initializes a tableau, T. The tableau consists of M blocks of columns, one for each e_k of E. The kth block will consist of the sets of S that cover e_k, but do not contain lower numbered elements $e_1, ..., e_{k-1}$. In each block, a variable g_i is initialized to the first position in each block i. This will keep track of the current set within the blocks.

The algorithm will define a partial solution D_0, a partial solution metric Z, a current best solution $\hat{B}$, and a current best metric $\hat{Z}$.

Three heuristics were created for the SCP. The first heuristic, which we call Heuristic 1, implements the dominance test as presented in [Chr. ref]. During the operation of the SCP algorithm, we will keep track of partial solutions in a set $L(E_{ck}, z_k)$, where E_{ck} is the set coverage at step k, and z_k is the associated cost at step k. If at some step $r > k$, $E_{cr} \subseteq E_{ck}$ and $z_r \geq z_k$, then we know that a previous solution was better than the solution at step r. We can thus abandon this branch of the search. One of the disadvantages of this technique is that maintaining L can require significant memory, and the time required, which is polynomial, to search the list can potentially diminish the advantage of pruning branches of the tree. While not implemented, there are possibly ways to improve Heuristic 1. If, rather than a set of lists, L was represented as a tree, so that searching L could be performed in greater than $O(m)$, there could be significant time reduction. This is a possible future avenue for exploration.

Heuristic 2 adjusts the sorted order of the tableau to prune additional branches of the search. In the original implementation, the sets within a Block n are first sorted by lexicographical order and then by cost. In this way, cost ties are sorted in lexicographical order. Heuristic 2 first sorts the sets by $|S|$, or the number of elements that are covered by the set, and then by cost. This will guarantee that sets with higher coverages will be examined first. The strategy for this sort order is that searching the sets with higher coverage first will eliminate a larger portion of the search tree early.

Heuristic 3 takes advantage of the fact that we do not need to include blocks in the tableau that only have one set. For each element $e_k \in E$, the number of covering sets in calculated, which we will call α_k.

$$\alpha_k = \sum S \; s.t. \; S \; covers \; e_k$$

If $\alpha_k = 1$, or there is only one set $S\prime$ in a block k, then that block contains a set that is the only cover for some element e_k. Rather than include this block in the search, S' is automatically included in the result set B. Additionally, the set coverage, E_c, is initialized to include those elements covered by S'. In problems that have high coverages, this heuristic will not yield an improvement. However, if there exists any singly covered elements e_k, large portions of the search space will be eliminated before the search is even started, since that element already exists with E_c. The selection phase of the algorithm will never need to select any element from that block, because it will already have been covered. If $\alpha_k = 0$ for some element e_k, then e_k does not have a valid cover, and there is no solution possible.

- D_i - (E, S), ci
  - E - Set of elements e_k, where $k = 1, ..., m$

  - D_o - Set of sets, S_r, where $r = 1, ..., n$ Each set S_r contains elements $e \in E$
  - ci_r - Cost for each set S_r.

- D_p - Partial solution set of sets, B, where each element $B_i \in S$ is a set cover.

- *Creative data sets/selection:*

  - E_c - Set of elements, $e_c \in E$
  - $L(ci)$ - List of E_c sets covered with cost ci. When a node is visited at step k, the current cover E_c and the current cost Z are added to L. When a subsequent node is visited at step $k+1$, it will search this list to see if some set $\{E_{c1}, ...E_{ck-1}\}$ covers the a subset of the elements in E_c, but at a lower cost ci_i. If so, then we do not need to explore this branch of the tree. Without adding additional heuristics to L, a linear search is performed each time it is accessed. This increase complexity O(n) for each node. Although not implemented in this application, a beneficial future enhancement would be to reduce the search time required for L.

- Tableau: The tableau consists of M blocks of columns, one for each e_k of E. The kth block will consist of the sets of S that cover e_k, but do not contain lower numbered elements $e_1, ..., e_{k-1}$. In each block, a variable l is initialized to the first position in each block. This will keep track of the current set within the blocks. The sets within each block are sorted by cost per element covered, ci_r. A merge sort was selected for this operation, which is O(n log n) [1]. The merge operation for each block i is O(($n_i m$ log($n_i m$). The $\sum_{n=1}^{m}$ O(($n_i m$ log($n_i m$) = O(n log n). Heuristic 2 adds an additional merge sort, this time by the $|S\prime| \in$ block i. Because merge sort preserves underlying sort order, we can use two consecutive sorts, the first by the $|S\prime|$, then then by the cost of the $S\prime$. The additional sort increases the initial sort complexity to O(2 * n log(n)). The idea behind this heuristic to reduce the search space by selecting sets that have a higher coverage first. By selecting sets with higher coverage, we should potentially reduce the number of sets that are visited later in the search, where pruning will take place with either Z or the L set.

Imports: *ADT set, array, sequence, Boolean, Integer*
Operations:

- Initialization: Initial (E, S), ci, T, and L
- Next State Generator: The algorithm will keep track of the current block, t in the Tableau T, as well as the current set within the block, g_i. Returns a set $b \in S_k$, where S_k is at the g_i position of block k.
- Feasibility: (e, s) âĂŞ Determine if $e \in E$ is covered by $s \in S$ <Add symbolic logic>
- Solution: (S, Z): Determines if all elements $e \in E$ have been covered by B at cost Z.
- Objective: Minimize Z to cover $e \in E$.
- Feasibility: I(x): x = (element, set) = (e_i, s_j), such that e_i should be an element in E in D_i and s_i should be a set in S in the input D_i O(z, B): z is a cost for step k, and $B \in D_o$

1.4 Algorithm Domain Design Continuing Refinement

Operations:

- Initialization: Initial (E, S), ci, T, and L
  - *setupTableau()*: Sets up the Tableau T
- Next State Generator: The algorithm will keep track of the current block, t in the Tableau T, as well as the current set within the block, g_i. Returns a set $b \in S_k$, where S_k is at the g_i position of block k.
  - *getMin()*: Gets the next block from the first uncovered element in E
- Feasibility: (e, s) - Determine if $e \in E$ is covered by $s \in S$

- *solutionPossible()*: Determines if a solution is possible. A solution is possible if:

$$I \subseteq \{1, ..., m\} \min \sum_{J \in I} w_j$$
$$s.t. \bigcup_{J \in I} S_j = E$$

- Solution: (S, Z) - Determines if all elements $e \in E$ have been covered by B at cost Z.
  - *step5()*: Corresponds directly to Christofides step 5, *Test for new solution* in section 4.3: A tree search algorithm for the SPP
- Objective: Minimize Z to cover $e \in E$.
- Backtrack: Determines when backtracking should occur in the algorithm. This is primarily where the heuristics should prune the search space.
  - *ste4()*: Corresponds directly to Christofides step 4, *Backtrack* in section 4.3: A tree search algorithm for the SPP
  - Heuristics:
    * If there exists e_i in E and e_i no in s_j for all j, then no solution exists
    * Heuristic 3: If there exists e_i in E with E_i in s_k and e_i not in s_j for all $j = k$, then s_k is in all solutions. Initialize $B\prime$ to include these sets. After the algorithm is completed, $\hat{B} = \hat{B} \cup B\prime$. Additionally, the vector of covered sets, E_c is initialized to include all those elements $e \in B\prime$. This initialization will ensure that these blocks in the Tableau will not be selected and thus not searched.
    * Heuristic 1: Determine if there is a set that covers a subset of the current elements at some step k at a lower cost.

2 SCP Heuristic Testing and Evaluation [c]

In this section, we evaluate the performance of the AFIT SCP Solver implemented with additional heuristics as described in the preceding section. A design of experiments which methodically evaluates the performance of the AFIT SCP Solver on a large variety of SCP instances is constructed. This experimental design is applied to both the unmodified and modified versions of the AFIT SCP solver, and the results are analyzed.

2.1 Problem Selection [c.1]

The USAF RIF problem, described in [2], is real-world problem upon which all randomly constructed instances of the SCP in this work are based. A complete description of the problem is found in [2]. Briefly, the problem is that of selecting a subset of UAV pilots such that the maximum number of aircraft can be flown simultaneously for the minimum personnel cost. The details of this problem are such that the density of the corresponding SCP instances turns out to be approximately 30%. Because the RIF must be implemented at organizations of all sizes, it is reasonable to evaluate this problem over a large range of instance dimensions.

2.2 Test Suite Description [c.2]

The testing suite used in the this work is of similar construction to that which is used in [2]. This software suite, written in the Julia technical computing language [3], and included as Appendix B, constructs random SCP instances, and applies the AFIT SCP Solver to those instances. Given the desired dimensionality of the SPC instance, which is the number of sets, elements, and the density of the instance, the suite produces an instance conforming to that request. An input file suitable for the AFIT SCP Solver is produced from

2.3 Results [c.3]

2.4 Search Tree [c.4]

A sample search tree is drawn below in figure 2.4. The input elements are $\{12345\}$, and the sets, defined in $(\{e_1, ..., e_n\}, cost)$ format, are $(\{1,3\},1)$, $(\{2,3\},2)$, $(\{1,4,5\},3)$, $(\{2,3,5\},4)$, $(\{4,5\},2)$.

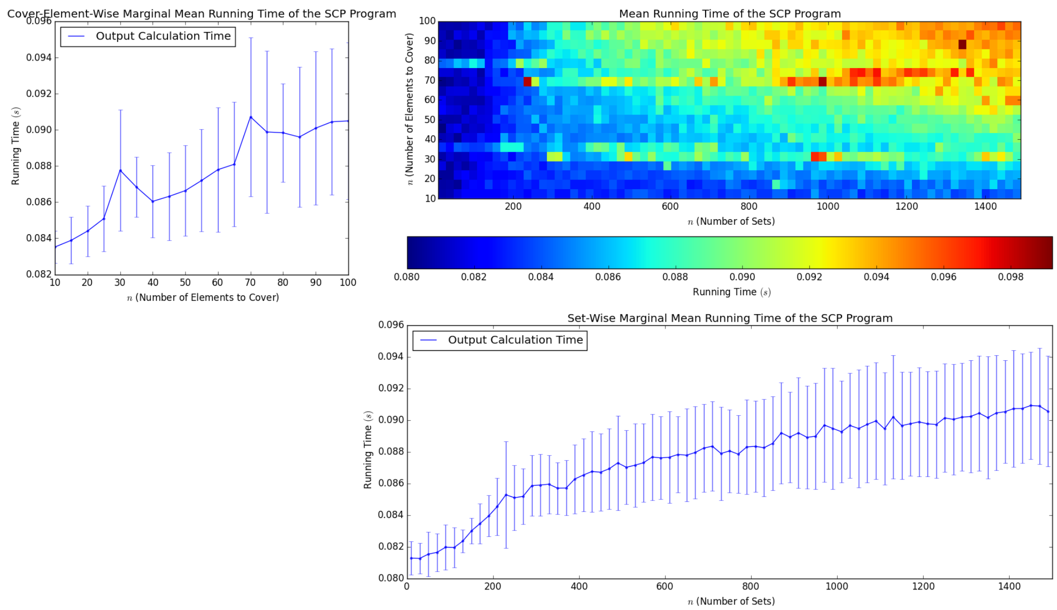

Figure 1: This figure displays the average running time performance of the original AFIT SCP Solver, over 20 runs, for various instance configurations from the problem domain. All instances in this figure have a density of approximately 0.3.

3 AFIT SCP Program Software Engineering Practices

3.1 SCP Implementation Discussion

Does the AFIT SCP Solver employ good software engineering practices? Due to issues with both supplied code packages, we decided to write our own SCP solver from scratch. The decision to do this was based on the limited time available to solve the compilation issues, and a conscious decision to further understand the problem algorithm by implementing it. We did study the C++ implementation prior to creating our own solution, and it does follow many good software engineering principles, although from personal experi-

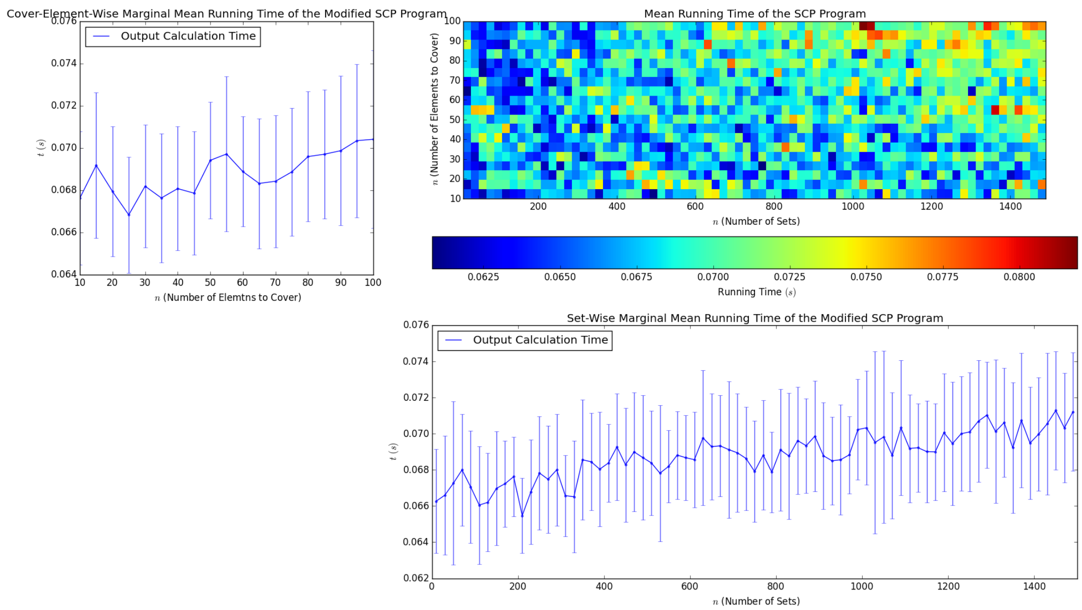

Figure 2: This figure displays the average running time performance of the modified AFIT SCP Solver, over 20 runs, for various instance configurations from the problem domain. All instances in this figure have a density of approximately 0.3.

ence this evaluation can be subject to opinion. The code subscribed to valid object oriented design practices, was well documented, did not use unnecessarily complicated structures, although we could not identify an underlying design pattern. By dividing the work into small pieces, such as the list and block objects, it was simplifying the problem by the "divide and conquer" paradigm [4]. We identified several instances of using functions to return multiple items with pointer references, which complicate the code somewhat. These variables probably should have been well labeled globals and used with caution. Because the code did not compile, I did examine the include structure of the files, and, as interpreted by the both Eclipse and Visual Studio, it appeared as if the software suffered from a "chicken or the egg" scenario where there was something of a circular dependency. This may have been

	1	3	2	4	1	2	4	5	3	5	3	4
1	1	1	0	0	1	0	0	0	1	0	1	0
2	0	0	1	1	0	1	1	0	0	0	0	1
3	1	0	1	1	1	1	1	0	0	0	0	1
4	0	1	0	0	0	0	0	1	1	1	1	0
5	0	1	0	1	0	0	1	1	1	1	1	1
	1	3	2	4	1	2	4	2	3	2	3	4

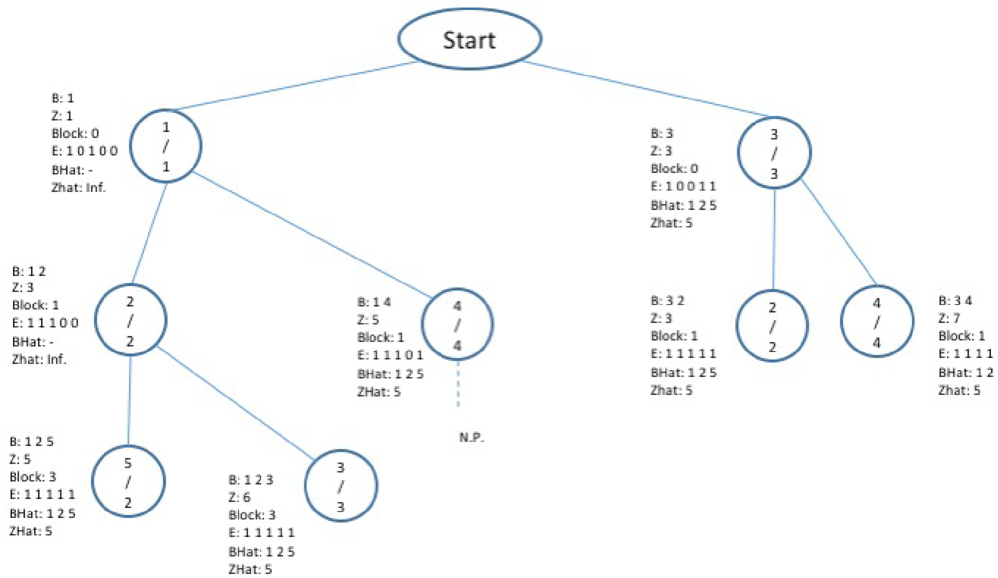

Figure 3: This figure displays a sample graph created using the supplied input data. The initial Tableau is shown above the graph.

eliminated with a more straight-forward order of object inheritance.

Our own implementation was written in Java, using Eclipse IDE and Java 7, and there is no connection to the SCP solver 2006 Java code. We did not examine that software package. Instead, the algorithm was implemented as it was presented in by Christofides in section 4.3, *A tree search algorithm for the SSP*, (modified according to Section 4.4). An advantage to this technique is that we obtained a greater understanding of the Christofides algorithm itself. We feel that this allowed us to better implement the heuristics.

Discuss ease of understanding code in the AFIT SCP Solver. Discuss ease of standard interfaces.

The interfaces in the AFIT SCP C++ program were straight forward to understand. The set_cover implements the main control structure of the algorithm, encapsulating steps 1 through 5 of the Christofides algorithm. Our implementation uses the searchSCP() function, which is very similar in control structure. Reductions are performed in reduction_2_1 and reduction_2_2, which are similarly performed in searchPossible, reduction 1 from section 4.2, and Heuristic 2, which is commented in SCPAlpha.java and implements reduction 2 from section 4.2. The C++ has several helper objects, such as List, opair, set, and block, with correspond to our own Block.java, Element.java, Tableau.java, and Set.java. We have an additional helper object to perform the sort, MergeSort.java. We believe that our own implementation has closer adherence to Object Oriented principles, however. Each of our objects directly correspond to a proposed data set in the design, and each is a simple extension of a standard ADT. The Block object is defined as t, the Tableau object is T, sets S are defined in the Set Object, and an element $e \in E$ is implemented by the Element object. Additionally, our implementation uses the same data files as the C++ SCP implementation.

4 AFIT SCP Program Complexity

[Justin] Describe observed complexity.

5 Integration

A Code Listing

Listing 1: Main SCP Search Class

```java
/**
 * ---------------------------------------------------------------------------
 * Classification: UNCLASSIFIED
 * ---------------------------------------------------------------------------
 *
 * Class: SCPAlpha
 * Program: SCP/SPP program
 *
 * DESCRIPTION: This program solves the set-covering problem and the
 *   set-partitioning problem using the algorithm in Christofides
 */
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class SCPAlpha
{

  public static final String SCP_H1 = "-H1";
  public static final String SCP_H2 = "-H2";
  public static final String SCP_H3 = "-H3";
  public static final String SCP_OUT = "-out";
  public static final String SCP_OUTALL = "-noout";

  ArrayList<Element> ogElements = null;
  ArrayList<Set> ogSetArray = null;
  ArrayList<Block> ogBlockArray = null;
  ArrayList<Integer> ogCB = null;
  ArrayList<LItem> ogL = null;
  ArrayList<Set> ogSingleSets = null;
  int ogSetID = 1;
  int[] ogE = null;
  ArrayList<Set> ogB = null;
  ArrayList<Set> ogBHat = null;
```

```
  int ogZ = 0;
  int ogZHat = 0;
  boolean ogDone = false;
  boolean ogUseH1 = true;
  boolean ogUseH2 = true;
  boolean ogUseH3 = true;
  boolean ogUseOut = false;
  boolean ogNoOut = false;

  int ogP = -1;


  public SCPAlpha(String[] args)
  {
    long time = System.currentTimeMillis();
    try
    {
      for (String m : args)
      {
        if (m.toUpperCase().equals(SCP_H1))
        {
          ogUseH1 = false;
        }
        else if (m.toUpperCase().equals(SCP_H2))
        {
          ogUseH2 = false;
        }
        else if (m.toUpperCase().equals(SCP_H3))
        {
          ogUseH3 = false;
        }
        else if (m.toLowerCase().equals(SCP_OUT))
        {
          ogUseOut = true;
        }
        else if (m.toLowerCase().equals(SCP_OUTALL))
        {
          ogNoOut = true;
        }
      }
      File temp = new File(args[0]);
      parseFile(temp);

      // *****   Reduction 2.1
      if (searchPossible())
```

```java
      {
        // Initialization Phase
        // Heuristic 2
        // The original algorithm sorts the sets according to cost, but ties are sorted by lexicographical order.
        // Heutistic two sorts ties using number of covered items per set. Sets with greater coverage are tried
        // first, because they eliminate a larger portion of the search space.

        // *** Set of candidates *** generation
        // Generates the tree in the form of a tableau.
        setupTableau(ogUseH2);

        if ((ogUseOut) && (!ogNoOut))
        {
          System.out.println(getTableauString());
        }

        searchSCP();

        if (!ogNoOut)
        {
          System.out.println("Best Solution:");
          printResultState();
        }
      }
      else if (!ogNoOut)
      {
        System.out.println("Search Not Possible");
      }
    } catch (Exception e)
    {
      e.printStackTrace();
    }
    if ((!ogNoOut) && (ogUseOut))
    {
      System.out
            .println("Time: " + (System.currentTimeMillis() - time) + " ms");
    }
  } // SCPAlpha

```

```java
  public void searchSCP()
  {
    // Initialization
    ogZ = 0;
    ogZHat = Integer.MAX_VALUE; //infinity

    // ogB is the partial solution Do
    ogB = new ArrayList<Set>();

    // ogBHat is the current best solution
    ogBHat = new ArrayList<Set>();
    ogBHat.clear();

    // Initialize the E variable to keep track of covered rows.
    ogE = new int[ogElements.size()];

    // Heuristic 3
    // If there are rows that are only covered by one set, then that set must be included.
    // Add the cover to E, and this will eliminate the block from consideration
    if ((ogUseH3) && (ogSingleSets != null) && (ogSingleSets.size() > 0))
    {
      for (Set m : ogSingleSets)
      {
        addCoveredElementsFromSet(m, ogE, true);

      }
    }

    // Initialize an array of current blocks. This array is used as a queue to keep
    // track of the current block and the history of selected blocks. This is also
    // an aid to backtracking.
    ogCB = new ArrayList<Integer>();

    // Heuristic 1
    // Initialize L Array. This will keep track of all intermediate solutions and the
    // associated values. If a cover is encountered that is within a previous cover, but
    // has a greater cost, then there is no reason to consider this branch.
```

```
    ogL = new ArrayList<LItem>();

    ogDone = false;

    // Check to make sure that, after adding sets from heruistic 3, we don't already have a solution.
    if (eCoversR())
    {
      ogBHat.addAll(ogSingleSets);
      ogZHat = 0;
      for (Set m : ogSingleSets)
      {
        ogZHat = ogZHat + m.ogCost;
      }
    }
    else
    {
      // Set up initial solution. Add first block to block queue.
      if (!ogUseH3)
      {
        ogCB.add(new Integer(0));
      }
      else
      {
        // Get first selectable block
        int j = 0;
        while ((j < ogBlockArray.size())
                && (ogBlockArray.get(j).ogSets.size() <= 1))
        {
          j++;
        }
        ogCB.add(new Integer(j));
      }

      Set currentSet = ogBlockArray.get(currentBlock()).ogSets
              .get(ogBlockArray.get(currentBlock()).ogCurrentPos);
      addCoveredElementsFromSet(currentSet, ogE, true);
      ogB.add(currentSet);
      ogZ = ogZ + currentSet.ogCost;

      // *** Next state/Feasibility *** Finds the next state from the tableau. The feasibility function
      // is implied, as only valid solutions are considered.
      addMin(getMin(currentBlock()));
```

```java

    // Check to make sure all sets have not been covered.
    if (currentBlock() == ogBlockArray.size())
    {
      ogDone = true;
    }

    while (!ogDone)
    {
      // Maintain partial solution
      currentSet = ogBlockArray.get(currentBlock()).ogSets
                .get(ogBlockArray.get(currentBlock()).ogCurrentPos);
      //System.out.println("Test1: " + currentSet.ogID + "/" + currentSet.ogCost + ", " + ogZ + ", " +
      //          (ogZ + currentSet.ogCost) + ", " + ogZHat + ", Block: " + ogBlockArray.get(currentBlock()).ogID);

      ogB.add(currentSet);
      ogZ = ogZ + currentSet.ogCost;
      addCoveredElementsFromSet(currentSet, ogE, true);
      //printCB();
      //printCP();
      //printState();

      // Heuristic 1
      // If there are covered sets that are greater than the current cover,
      // but have less cost, then there is no reason to search this branch.
      if ((ogUseH1) && (solutionInL(ogE, ogZ)))
      {
        //printState();
        step4();
      }
      else if (ogZ < ogZHat)
      {
        // Add current Do to L
        if (ogUseH1)
        {
          ogL.add(new LItem(ogE, ogZ));
        }

        // *** Solution ***
```

```java
          // Determines if a partial solution is a valid solution to the SCP.
          step5();
        }
        else
        {
          // *** Backtracking *** step
          // If we encounter a node that is higher in cost than the current
          // solution, then exploring this branch would
          // not be fruitful.
          step4();
        }
        // *** Next state/Feasibility *** Finds the next state from the tableau. The feasibility function
        // is implied, as only valid solutions are considered.
        addMin(getMin(currentBlock()));
      }

      // Add all sets from heuristic 3 and add to ZHat
      if ((ogSingleSets != null) && (ogSingleSets.size() > 0))
      {
        ogBHat.addAll(ogSingleSets);
        for (Set m : ogSingleSets)
        {
          ogZHat = ogZHat + m.ogCost;
        }
      }
    }
    //printState();

  } // searchSCP


  /**  *** Backtrack *** Step */
  public void step4()
  {
    if (ogB.isEmpty())
    {
      ogDone = true;
    }
    else
    {
      // For current block, increase position of current set with the block
```

```java
      Block currentBlock = ogBlockArray.get(currentBlock());
      currentBlock.ogCurrentPos++;
      // Remove the current cover from our cover array
      addCoveredElementsFromSet(ogB.get(ogB.size() - 1), ogE, false);
      // Remove the most recent Z value
      ogZ = ogZ - ogB.get(ogB.size() - 1).ogCost;
      // Go back to previous block using ogCB queue
      removeLastBlock();
      // Remove latest solution from partial solution B
      removeLastB();
      //printState();
      // If we are at the end of the first block, then we are finished.
      if ((currentBlock.ogCurrentPos >= currentBlock.ogSets.size())
              && (currentBlock.ogID == 0))
      {
        ogDone = true;
      }
      else if (currentBlock.ogCurrentPos >= currentBlock.ogSets.size())
      {
        // We have exhausted the block, so backtrack. Reset set position for the current block
        currentBlock.ogCurrentPos = 0;
        // Since we cannot continue in this block, we must backtrack
        step4();
      }
    }
  } // step4


  /** *** Solution *** step */
  public void step5()
  {
    // if all sets are covered, then we have a new best solution
    if (eCoversR())
    {
      // Set BHat latest solution
      ogBHat.clear();
      ogBHat.addAll(ogB);
      // Set Z to best solution
      ogZHat = ogZ;
```

```java
      if ((ogUseOut) && (!ogNoOut))
      {
        System.out.println("Paritial Solution:");
        printResultState();
      }
      // Backtrack from current position to continue exploring the blocks
      step4();
    }
    //printState();
  } // step5


  /**
   * **** Next state generator *****
   *
   * Gets the next block from the first uncovered element in E
   *
   * @param currentBlock - Current block position
   * @return
   */
  public int getMin(int currentBlock)
  {
    int result = -1;
    // Get next block from minimum non-covered item in E
    int i = 0;
    while ((i < ogE.length) && (result == -1))
    {
      if (ogE[i] == 0)
      {
        result = i;
      }
      else
      {
        i++;
      }
    }
    if (result == -1)
    {
      result = ogE.length;
    }
    return result;
  } // getMin


```

```java
/** Adds latest minimum block to queue of blocks. */
public void addMin(int min)
{
  if (ogCB.get(ogCB.size() - 1) != min)
  {
    ogCB.add(min);
  }
} // addMin


/**
 * Heuristic 1
 *
 * Determines if some partial solution E exists within the the already visited solutions.
 * If so, and it has a lower Z value, then we should stop exploring this branch as a
 * better solution has already been found.
 *
 * @param e - Current partial solution E
 * @param z - Current partial solution Z
 * @return - true if we should backtrack, false if we should not
 */
public boolean solutionInL(int[] e, int z)
{
  boolean result = false;

  int i = 0;
  while ((i < ogL.size()) && (!result))
  {
    if ((ogL.get(i).isWithin(e)) && (z > ogL.get(i).ogCost))
    {
      result = true;
    }
    i++;
  }

  return result;
} // solutionInL


/**
 * *** Initialization ***
 *
```

```java
     * Heuristic 2
     *   The original algorithm sorts the sets according to cost,
        ↪ but ties are sorted by lexicographical order.
     *   Heuristic two sorts ties using number of covered items per
        ↪  set. Sets with greater coverage are tried
     *   first, because they eliminate a larger portion of the
        ↪ search space.
     *
     * Sets up the Tableau
     *
     * @param useH2 - Determines if Heuristic 2 should be used.
     */
    public void setupTableau(boolean useH2)
    {
      ogBlockArray = new ArrayList<Block>();
      for (int i = 0; i < ogElements.size(); i++)
      {
        Block b = new Block(i);
        ArrayList<Set> sets = getAllCoveringSets(ogElements.get(i).
            ↪ ogName);
        b.ogSets = sets;

        // Heuristic 2
        // The original algorithm sorts the sets according to cost,
            ↪  but ties are sorted by lexicographical order.
        // Heuristic two sorts ties using number of covered items
            ↪ per set. Sets with greater coverage are tried
        // first, because they eliminate a larger portion of the
            ↪ search space.
        b.sortSets(useH2);
        ogBlockArray.add(b);
      }
    } // setupTableau


    /**
     * Gets all covering sets for the some item name
     *
     * @param name - name of item
     * @return - All sets that cover name
     */
    public ArrayList<Set> getAllCoveringSets(int name)
    {
      ArrayList<Set> result = new ArrayList<Set>();

```

```java
    for (Set m : ogSetArray)
    {
      if (m.ogSet[name - 1])
      {
        result.add(m);
      }
    }

    return result;
  } // getAllCoveringSets


  /**
   * Determines if E covered R, or all covered
   *
   * @return true if all items have been covered, false if not
   */
  public boolean eCoversR()
  {
    boolean result = true;
    for (int i = 0; i < ogElements.size(); i++)
    {
      if (ogE[ogElements.get(i).ogName - 1] == 0)
      {
        result = false;
        break;
      }
    }

    return result;
  } // eCoversR


  /**
   * *** Next State Generator ***
   * Adds the current set to E
   *
   * E is a an integer array. If multiple sets cover a single item,
   * its coverage will be reflected as a value in E.
   *
   * @param m Set to add to E
   * @param e Current E
   * @param add - True if the set is added, or false if the set is subtracted
```

```
 */
public void addCoveredElementsFromSet(Set m, int[] e, boolean add)
{
  for (int i = 0; i < m.ogSet.length; i++)
  {
    if (m.ogSet[i])
    {
      e[i] = e[i] + (add ? 1 : -1);
    }
  }
} // addCoveredElementsFromSet


public String getTableauString()
{
  String result = "";

  System.out.print(formatStringLength(" ", 5, " ", false) + "|");
  for (Block b : ogBlockArray)
  {
    for (Set m : b.ogSets)
    {
      System.out
              .print(" " + formatStringLength((m.ogID) + "", 2, " ", false));
    }
    System.out.print("|");
  }
  System.out.println();

  for (int i = 0; i < ogElements.size(); i++)
  {
    System.out.print(
            formatStringLength(ogElements.get(i).ogName + "", 5, " ", false)
                + "|");
    for (Block b : ogBlockArray)
    {
      for (Set m : b.ogSets)
      {
        System.out.print(" " + print(m.ogSet[i]));
      }
      System.out.print("|");
```

```
      }
      System.out.println();
    }

    System.out.print(formatStringLength(" ", 5, " ", false) + "|");
    for (Block b : ogBlockArray)
    {
      for (Set m : b.ogSets)
      {
        System.out.print(
                " " + formatStringLength((m.ogCost) + "", 2, " ", false));
      }
      System.out.print("|");
    }
    System.out.println();

    return result;
  } // getTableauString


  public static String print(boolean b)
  {
    return formatStringLength((b ? "1" : "0"), 2, " ", false);
  }


  public void printState()
  {
    System.out.print("E: ");
    for (int r : ogE)
    {
      System.out.print(r + " ");
    }
    System.out.println();
    System.out.print("B: ");
    for (Set r : ogB)
    {
      System.out.print(r.ogID + " ");
    }
    System.out.println("");
    printResultState();
    System.out.println("\n");

```

```java
  } // printState


  public void printResultState()
  {
    System.out.println("ZHat: " + ogZHat);
    System.out.print("BHat: ");
    for (Set r : ogBHat)
    {
      System.out.print(r.ogID + " ");
    }
    System.out.println("\n");
  } // printState


  public void printCP()
  {
    for (int i = 0; i < ogBlockArray.size(); i++)
    {
      System.out.print(i + ": " + ogBlockArray.get(i).
          ↪ ogCurrentPos + ", ");
    }
    System.out.println();
  }


  public int currentBlock()
  {
    if (ogCB.size() == 0)
    {
      ogCB.add(new Integer(0));
    }
    return ogCB.get(ogCB.size() - 1);
  } // currentBlock


  public void removeLastBlock()
  {
    ogCB.remove(ogCB.size() - 1);
  } // removeLastB


  public void removeLastB()
  {
    ogB.remove(ogB.size() - 1);
```

```java
  } // removeLastB


  public void printCB()
  {
    System.out.print("CB: ");
    for (int m : ogCB)
    {
      System.out.print(m + " ");
    }
    System.out.println();
  } // printCB


  public void parseFile(File theFile) throws Exception
  {
    ogElements = new ArrayList<Element>();
    ogSetArray = new ArrayList<Set>();
    String inFile = loadStringFromFile(theFile, true);
    // Get elements
    int pos1 = inFile.indexOf("{");
    int pos2 = inFile.indexOf("}");
    String temp = inFile.substring(pos1 + 1, pos2);
    String[] temp2 = temp.split(" ");
    int highest = 0;
    for (String temp3 : temp2)
    {
      if (!temp3.trim().equals(""))
      {
        ogElements.add(new Element(Integer.parseInt(temp3.trim())));
        int r = Integer.parseInt(temp3.trim());
        if (r > highest)
        {
          highest = r;
        }
      }
    }
    // Get sets
    pos1 = inFile.indexOf("{", pos2);
    pos2 = inFile.lastIndexOf("}");
    String sets = inFile.substring(pos1 + 1, pos2);
    //System.out.println("Test1: " + sets);
    String[] set = sets.split("\n");
    for (String setLine : set)
```

```
    {
      if ((!setLine.trim().equals("")) && (!setLine.equals("\n")))
      {
        //System.out.println("Test3: " + setLine);
        pos1 = setLine.indexOf("{");
        pos2 = setLine.indexOf("}");
        temp = setLine.substring(pos1 + 1, pos2);
        temp2 = temp.split(" ");
        Set newSet = new Set(highest);
        for (String temp3 : temp2)
        {
          if (!temp3.trim().equals(""))
          {
            int pos = Integer.parseInt(temp3.trim());
            newSet.ogSet[pos - 1] = true;
          }
        }
        pos1 = setLine.indexOf(",");
        pos2 = pos1;
        while ((setLine.charAt(pos2) != ')') && (pos2 < setLine.length()))
        {
          pos2++;
        }
        temp = setLine.substring(pos1 + 1, pos2);
        newSet.ogCost = Integer.parseInt(temp.trim());
        newSet.ogID = ogSetID++;
        ogSetArray.add(newSet);
      }
    }
  } // parseFile


  /**
   * searchPossible: reduction_2_1
   *
   * This function performs the first half of reduction 2 as mentioned in
   * Christofides.  This reduction removes from R and all sets in F, any
   * element that appears in every set--since this element would be covered
   * regardless of the solution.
```

```
   * the element is not added back to R so this functin is no longer called
   *
   * Heuristic 3
   * Collects all sets that are the only cover for a single item. These
   * Must be included in the final solution set.
   *
   * Determines if all sets can be covered. If not, then no solution is possible.
   *
   * @return - true if a solution is possible, or false if it is not.
   */
  public boolean searchPossible()
  {
    boolean result = true;
    ogSingleSets = new ArrayList();
    for (int i = 0; i < ogSetArray.size(); i++)
    {
      for (int j = 0; j < ogSetArray.get(i).ogSet.length; j++)
      {
        if (ogSetArray.get(i).ogSet[j])
        {
          addToElement(j + 1, ogSetArray.get(i));
        }
      }
    }
    int i = 0;
    while (i < ogElements.size())
    {
      if (ogElements.get(i).ogCoveringSets == 0)
      {
        result = false;
      }
      else if ((ogElements.get(i).ogCoveringSets == 1) && (ogUseH3))
      {
        // These sets must be included in the final result
        ogSingleSets.add(ogElements.get(i).ogCoveringSetArray.get(0));
      }

      i++;
    }
```

```

    if (ogUseH3)
    {
      if (ogSingleSets.size() != 0)
      {
        // Remove all single sets from the list of sets
        for (Set singleSet : ogSingleSets)
        {
          int j = 0;
          boolean found = false;
          while ((j < ogSetArray.size()) && (!found))
          {
            if (singleSet == ogSetArray.get(j))
            {
              ogSetArray.remove(j);
              found = true;
            }
            j++;
          }
        }
      }
    }

    return result;
  } // searchPossible


  private void addToElement(int name, Set m)
  {
    int i = 0;
    boolean found = false;
    while ((i < ogElements.size()) && (!found))
    {
      if (ogElements.get(i).ogName == name)
      {
        ogElements.get(i).ogCoveringSets++;
        ogElements.get(i).ogCoveringSetArray.add(m);
        found = true;
      }
      i++;
    }

  } // addToElement

```

```
  /** Loads text from file.
   *
   * @param dirName
   *          The directory of the file the text is to be written to.
   * @param fileName
   *          The name of the file.
   * @param text
   *          The text being written to a file. */
  public static String loadStringFromFile(File theFile, boolean addNewLine)
  {
    StringBuffer result = null;
    try
    {
      if ((theFile == null) || (!theFile.exists()) || (!theFile.isFile()))
      {
        return null;
      }
      BufferedReader b = new BufferedReader(new FileReader(theFile));
      String temp = b.readLine();
      while (temp != null)
      {
        if (result == null)
        {
          result = new StringBuffer();
        }
        if (addNewLine)
        {
          result.append(temp + "\n");
        }
        else
        {
          result.append(temp);
        }
        temp = b.readLine();
      }
      b.close();
    } catch (IOException io)
    {
      System.out.println("Error reading the file " + io);
    }
    return result.toString();
```

```java
  } // loadStringFromFile


  /** Formats a String to a length, with prefix specified as a
      ↪ parameter
   *
   * @param m
   *          - String to format
   * @param length
   *          - Lenght of String
   * @param theBuffer
   *          - String to append to front or back of the String
   * @return String - Formatted String */
  public static String formatStringLength(String m, int length,
          String theBuffer, boolean after)
  {
    StringBuffer result = new StringBuffer();

    if (m.length() == length)
    {
      result.append(m);
    }
    else if ((m.length() > length) && (length > 0))
    {
      if (after)
      {
        result.append(m.substring(0, length));
      }
      else
      {
        result.append(m.substring(m.length() - length, m.length()
            ↪ ));
      }
    }
    else
    {
      if (after)
      {
        result.append(m);
      }
      for (int i = 0; i < (length - m.length()); i++)
      {
        result.append(theBuffer);
      }
      if (!after)
```

```
      {
        result.append(m);
      }
    }
    return result.toString();
  } // formatStringLength


  public static void main(String[] args)
  {
    SCPAlpha m = new SCPAlpha(args);
  }
}

class LItem
{
  int[] ogElements = null;
  int ogCost = 0;


  public LItem(int[] elementsSource, int cost)
  {
    ogElements = new int[elementsSource.length];
    for (int i = 0; i < ogElements.length; i++)
    {
      ogElements[i] = elementsSource[i];
    }
    ogCost = cost;
  }


  public boolean isWithin(int[] setArray)
  {
    boolean result = true;

    for (int i = 0; i < ogElements.length; i++)
    {
      if ((ogElements[i] == 0) && (setArray[i] > 0))
      {
        result = false;
      }
    }

    return result;
  } // isWithin
```

```java


  public String toString()
  {
    String result = "";

    for (int i = 0; i < ogElements.length; i++)
    {
      System.out.print(ogElements[i] + " ");
    }
    System.out.println(", Cost: " + ogCost);
    return result;
  }

} // LItem
```

Listing 2: Element Class

```java
/**
 *
     ↪ ------------------------------------------------------------------------
     ↪
 * Classification: UNCLASSIFIED
 *
     ↪ ------------------------------------------------------------------------
     ↪
 *
 * Class: Element
 * Program: SCP/SPP program
 *
 * DESCRIPTION: Implements an element
 */
import java.util.ArrayList;

public class Element
{
  public int ogName = 0;
  public int ogCoveringSets = 0;
  public ArrayList<Set> ogCoveringSetArray = new ArrayList();

  public Element(int name)
  {
    ogName = name;
  } // Element

  public String toString()
```

```java
  {
    return ogName + "";
  }
} // Element
```

Listing 3: Block Class

```java
/**
 *
     ↪ ------------------------------------------------------------
     ↪
 * Classification: UNCLASSIFIED
 *
     ↪ ------------------------------------------------------------
     ↪
 *
 * Class: Block
 * Program: SCP/SPP program
 *
 * DESCRIPTION: Implements a Block
 */
import java.util.ArrayList;
import java.util.Comparator;

public class Block
{
  public ArrayList<Set> ogSets = new ArrayList<Set>();
  int ogCurrentPos = 0;
  int ogID;


  public Block(int theID)
  {
    ogID = theID;
  } // Block


  public void sortSets(boolean useH2)
  {
    MergeSort sorter = new MergeSort();
    if (useH2)
    {
      sorter.sort(ogSets, new SetSortCoverHandler());
    }
    sorter.sort(ogSets, new SetSortHandler());
  } // sortSets
```

```java


  public String toString()
  {
    return ogID + "";
  }

}

class SetSortHandler implements Comparator
{

  @Override
  public int compare(Object o1, Object o2)
  {
    int result = 0;
    Set a = (Set) o1;
    Set b = (Set) o2;
    if (a.ogCost < b.ogCost)
    {
      result = -1;
    }
    else if (a.ogCost > b.ogCost)
    {
      result = 1;
    }

    return result;
  }
}

class SetSortCoverHandler implements Comparator
{

  @Override
  public int compare(Object o1, Object o2)
  {
    int result = 0;
    Set a = (Set) o1;
    Set b = (Set) o2;
    int aN = 0;
    int bN = 0;
    for (boolean r : a.ogSet)
    {
      if (r)
```

```
      {
        aN++;
      }
    }
    for (boolean r : b.ogSet)
    {
      if (r)
      {
        bN++;
      }
    }
    if (aN < bN)
    {
      result = 1;
    }
    else if (aN > bN)
    {
      result = -1;
    }

    return result;
  }
}
```

Listing 4: Tableau Class

```
/**
 * ----------------------------------------------------------------------------
 * Classification: UNCLASSIFIED
 * ----------------------------------------------------------------------------
 *
 * Class: Set
 * Program: SCP/SPP program
 *
 * DESCRIPTION: Implements a tableau
 */
import java.util.ArrayList;

public class Tableau
{
  public ArrayList<Block> ogBlocks = new ArrayList();

```

```
} // Tableau
```

Listing 5: Merge Sort Class Class

```
/**
 * ----------------------------------------------------------------------------
 * Classification: UNCLASSIFIED
 * ----------------------------------------------------------------------------
 *
 * Class: MergeSort
 * Program: Util
 *
 */

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

/**
 * The MergeSort is handled by the Colloections.sort function. If the Collections.sort
 * function is ever changed so that it isn't stable, a stable Merge sort will have to
 * be added.
 *
 * @author
 */
public class MergeSort
{

  public void sort(ArrayList array, Comparator sortHandler)
  {
    Collections.sort(array, sortHandler);
  }

} // MergeSort
```

References

[1] G. Lamont, "Advanced algorthim design class notes: Algorithm design."

[2] J. Fletcher and J. Knapp, “"csce 686 - homework 5".”

[3] The julia technical computing language. [Online]. Available: http://julialang.org/

[4] Software engineering principles. [Online]. Available: https://www.vikingcodeschool.com/software-engineering-basics/basic-principles-of-software-engineering