1-1-2000

Two new algorithms for classical problems in computer science

John Gerard Howe
University of Nevada, Las Vegas

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

TWO NEW ALGORITHMS FOR CLASSICAL PROBLEMS IN COMPUTER SCIENCE

by

John Gerard Howe

Bachelor of Science
University of Nevada, Reno
1982

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science Degree
Department of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
December 2000

UNLV

Thesis Approval
The Graduate College
University of Nevada, Las Vegas

November 27, 2000

The Thesis prepared by

John G. Howe

Entitled

Two New Algorithms for Classical Problems in Computer Science

is approved in partial fulfillment of the requirements for the degree of

Master of Science

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

PR/1017-53/1-00

ABSTRACT

Two New Algorithms For Classical Problems in Computer Science

by

John G. Howe

Dr. Evangelos Yfantis, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

This thesis presents two algorithms dealing with problems in two classic algorithm areas in computer science. The first algorithm presents a simple solution to the selection problem. The sequential computing model form of this selection algorithm is presented first followed by a general parallel computing model version.

The second algorithm is a relatively simple bookkeeping approximation solution to the Steiner tree problem in graphs. The problem presented deals with determining the shortest tree connecting Steiner nodes in a graph that has no direct connections between the Steiner nodes. Both algorithms are described and analyzed in detail with an appropriate running example to illustrate the actions of the algorithms.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to thank Dr. Kia Makki, Ph.D., for his introduction and instruction into the fascinating world of algorithms. His depth of knowledge and ability to teach this material to his students in a manner so as to inspire them to further investigate this field was instrumental in my being able to produce this work. Furthermore, it was Dr. Makki who first proposed the ideas that were later developed into the two algorithms presented here. Without his inspiration and guidance I would never have though to look for alternative solutions to classical computer science problems, let alone gone on to develop two such algorithmic solutions myself. Thank you Dr. Makki, wherever you are.

I would also like to thank Dr. Evangelos Yfantis, Ph.D., for his unwavering support and encouragement during some very difficult times. Originally starting my work under Dr. Yfantis, he was gracious and supportive when I later changed to work under Dr. Makki, but his willingness to reassume the positions of advisor and thesis committee chairman was beyond value and the call of duty. Thank you just does not adequately express the gratitude I feel for such support.

Finally, I would like to thank everybody else who has supported me over the years I have been at UNLV. In particular, I would like to thank the members of my thesis committee for their forbearance during some difficult personal and professional times and for sticking with me while I got through all of this. And always, I would like to thank my parents for their support without which I would never have had the opportunity to start an advanced degree program, let alone to have finally finished one.

CHAPTER 1

BIN SELECTION ALGORITHM

A thorough knowledge of algorithm development and analysis is an important part of any computer science education. Depending upon the goals of the course any number of classical algorithms may be studied. One such algorithm is the Selection Problem. An O(N) solution to this problem has been available for many years [3] with a number of incremental improvements having been made since its introduction. It has been observed by many who have worked with this problem that further improvement of the time bound of the classic algorithm could probably be achieved.

Searching for such an improvement resulted in the creation of the Bin Selection Algorithm. While looking for ways to preprocess the data provided to the classic algorithm it was discovered that one such preprocessing operation, similar to the bucket sort, enabled the elimination of unneeded data items while ensuring that the desired element was not one of the items eliminated. Further examination of this 'preprocessing' step revealed that continued use of this method would, in fact, solve the selection problem. Once it was confirmed that this method would always generate a solution, a formal algorithm was generated replacing, in its entirety, the classic solution to the selection problem.

The classic solution to the selection problem is O(kN), where k is approximately 2.5. Under most circumstances, the BSA is a well-behaved algorithm whose time bound is also O(kN), where k usually lies between 1.0 to approximately 1.7. In the worst case, the unmodified behavior of this algorithm becomes $O(n^2)$. However, this case can be easily detected and subsequently handled so that the overall cost of the algorithm is still linear. Depending upon the method chosen for dealing with this situation, a final value of k will often lay between 2.0 to 3.0.

The *Bin Selection Algorithm* (BSA) is a substantially easier algorithm to teach, understand, and program. It was originally developed as an algorithm for parallel machines. However, it is such a simple algorithm that a sequential implementation is trivial, primarily involving the removal of steps unique to parallel computing systems.

In this chapter, the Selection Problem is formally introduced. This is followed by a brief description of the classical solution to the problem. The sequential version of the Bin Selection Algorithm is then presented and the overall algorithm and its behavior is described and analyzed in depth. The parallel version of the BSA is described next with attention being paid specifically to those issues unique to a parallel system implementation. Finally, this is followed by future work, a discussion of the various modifications, and additional uses of this algorithm.

1.1 The Selection Problem

The Selection Problem has a wide variety of applications in computer science and statistics. Of particular interest is the special case of finding the median. The Selection Problem, simply stated, is:

Given the sequence S *of* N *elements and an integer* K, *where* $1 \leq K \leq N$, *find the* K^{th} *largest or smallest element in* S.

The Selection problem has been well studied in the literature [13,15]. An optimal sequential divide-and-conquer algorithm exists for its solution which runs in O(n) time in the worst case [3]. Also, Floyd and Rivest [7] discuss the sampling approach that finds the median in 1.5n expected comparisons. Recently, a number of parallel algorithms have been devised for this problem [1,2,5,13,18]. Akl [1] was the first to come up with a cost optimal parallel version of the algorithm given in [3].

1.2 The Classic Algorithm Solution

The classic solution to the selection problem has been well studied and presented in many algorithms books. Therefore, only a cursory presentation will be made here. The reader of this work is encouraged to refer to virtually any work on computer algorithms should a more detailed presentation of the classic algorithm be desired.

In general, like the Bin Selection Algorithm, the classic solution seeks to partition the data set such that the desired K^{th} element can be found directly (rare but possible) or that some of the elements in the set can be eliminated from further consideration during the search for *K*. This is accomplished by creating a series of subsets containing an odd number of elements and then sorting the elements in each subset. Using the median values from all of the subsets, a 'median of medians' value is determined.

The median of medians value is critical to the partitioning process of the classic algorithm. By reordering the sorted subsets around this value it can be determined which

elements are known to be greater than the median of medians and which elements are known to be smaller. If the number of elements known to be greater is equal to the value $K-1$ then the median of medians element is the desired K^{th} element. If the number of elements known to be greater is less than K then these elements can be discarded since the K^{th} element being sought cannot possibly be in this subset.

In most implementations, the classic algorithm is repeated on the new, reduced data set once the elements mentioned above have been discarded. Modifications to the algorithm exist that allow for the removal of another group of elements known to be smaller than the median of medians under conditions similar to those described above for the larger elements. During each iteration, a similar number of elements are removed from further consideration until the K^{th} element is found directly or until the number of elements remaining is such that the desired element can be found by sorting the elements and picking it directly.

The number of elements eliminated during each iteration is, at best, approximately thirty percent (30%) in the case of discarding only the larger elements. If both the larger and smaller element subsets are eliminated then this value rises to approximately fifty percent (50%). The exact number of elements discarded during any single iteration of the classic algorithm is due to the location of the median of medians value in the data set and the size of the subsets used by the algorithm.

1.3 The Bin Selection Algorithm Solution

The Bin Selection Algorithm (BSA) allows for the selection of the K^{th} largest, or smallest, element in a list without sorting any of the elements in the list, S. Instead, the

range of possible element values is divided among a series of bins such that the subranges assigned to the bins is disjoint. Then it uniquely places each element in the bin whose subrange includes that particular element's value. A count of the number of elements in each bin is kept during the placement process. When all of the elements of S have been placed in the appropriate bin, the bin that contains the desired K^{th} element can be determined. If the indicated bin contains more than one element, this process can be repeated. If the count of elements in the bin is sufficiently low then the elements can be sorted and the K^{th} element picked trivially.

In general, the BSA operates by discarding all elements of S that have been determined, conclusively, to not be the K^{th} element. Thus each succeeding iteration of this algorithm operates on a subset of those members of S that were present during the preceding iteration. Dependent upon the distribution of the elements in S and the ranges assigned to the bins, an average element removal rate of 60% to 80% per iteration is not unreasonable and can be easily achieved.

This is accomplished, in part, because the BSA allows for the incorporation of any knowledge about the list that may be available. For example, simply knowing the actual minimum and maximum values in the list serves to restrict the number of bins that will be needed. If some knowledge of the distribution of the values is available, then the number of bins collecting a certain range of values can be increased or decreased accordingly. Both of these items make it possible to discard more elements per iteration than would be possible if this information were not known.

1.4 A Common Illustrating Example

To illustrate the performance of this algorithm during each step, the following example is provided. A set of integers S is given as shown below in Figure 1.1. The size of S, $n = |S|$ is 25.

3	15	49	17	79	62	31	95	26	51	99	7	37
75	33	57	1	82	11	43	22	87	55	91	14	

Figure 1.1 Initial list S of data items for the example illustrating the use of the *Bin Selection Algorithm*.

1.5 The Sequential Computing Model Version

The sequential version of the BSA is an uncomplicated and easy to implement algorithm. Since there is only one processor and memory resource available, there is no need for the processor synchronization pauses and memory conflict avoidance schemes that will be required in the parallel computing model version.

1.5.1 Resource Requirements

With the sequential version of the BSA the only resource that needs to be managed is memory. The maximum amount of memory needed by this algorithm assumes that the implementer desires to keep everything in memory during processing. In this case, enough memory to hold the entire list, S, bin storage for the members of S, and one accumulator per bin is required. Therefore, the maximum amount of memory

required by this algorithm is $(2 \times S) + (bins + 1)$. The extra accumulator is used during the summation process when the algorithm determines the location of the desired K^{th} element of S.

The minimum amount of memory required by the algorithm is the amount needed for the accumulators, $(bins + 1)$. In this event, it is assumed that the storage for the list, S, and as well as the contents of the individual bins is provided for elsewhere. The actual number of bins required for a solution using the BSA will be determined in the initialization step as described in Section 1.5.4.

1.5.2 Algorithm Description

The sequential BSA is an six step solution to the general selection problem. These steps can be summarized as follows:

Step 0 - Initial Assumptions

Step 1 - Initialization

Step 2 - Element Distribution

Step 3 - Locate the K^{th} Element Bin

Step 4 - Process/Terminate Logic

Step 5 - Result(s) Processing

Each step is described in detail below. Throughout this discussion the common example will be used to illustrate the operations performed during each step of the sequential version of the algorithm.

1.5.3 Step 0 - General Assumptions and Knowledge

For the purpose of this discussion, it is assumed that the following conditions exist:

- The list is finite.
- Every element in the list S is unique.
- The bounding values of the elements in S are either known or can be estimated with reasonable accuracy.
- All array or list indices start at zero.
- All of the memory requirements mentioned earlier have been met.

Only the first assumption is actually required by this algorithm. Obviously, if the list is not finite then no solution is possible. The remaining four assumptions have been made so that the description and discussion of the BSA in the following sections can be simplified. As will become clear from the rest of this section, none of the remaining assumptions must be met for the algorithm to function properly.

The assumptions about element uniqueness and bounds knowledge, while useful in simplifying this discussion, are not absolute requirements for the successful use of the BSA since initial knowledge of the actual contents of the list, S, is not required for a solution using this algorithm. The BSA can handle the lack of bounds knowledge at the expense of requiring more iterations to reach the solution. No additional memory or bins are needed.

Furthermore, the elements in the list, S, need not be unique. It only needs to be guaranteed that duplicate elements in the list, S, will be placed in the same bin. If it is necessary to determine which instance of a duplicated list member is the desired element

then the manner in which the elements are placed in their respective bins can be easily modified so that this information can be preserved during Step 2.

The last two assumptions stated above have been made simplify the mechanics of the algorithm. The array and list index start value reflects the method used to place the elements in their respective bins and may change based upon the placement algorithm used by the implementer. Storing all data and results in memory also simplifies this presentation. Appropriately modified, the BSA can handle the violation of any or all of these assumptions easily. Further discussion of the modifications mentioned in this section is presented in Section 1.9.

1.5.4 Step 1 - Initialization

During this stage of the BSA, the overall structure of the solution to the given problem is generated. Of importance to the efficient operation of the algorithm is the determination of the total number of bins needed. From this item the rest of the required initial data can be generated, namely, the bin content ranges and the element placement value. These items are discussed below and illustrated with the example in Figure 1.2.

1.5.4.1 Determining the Number of Bins Required

Based upon the bounds information derived from S in step 0, a number of bins will be generated. The minimum number of bins recommended is the base 2 logarithm of the maximum possible element value of S. This value was originally selected as a starting point for no particular reason but has been proven to be quite workable under normal circumstances. Actually, any number of bins greater than one will work if it can

be guaranteed that at least two or the bins contain two or more members of S during each pass through the algorithm.

In general, however, the more bins that can be used the less time is required to arrive at the desired solution since more of the elements in S can be eliminated during each pass through the algorithm. Thus, the maximum number of bins to use depends on several conditions. If sufficient memory is available then a number of bins equal to the range of potential member values of S is usually desirable. In this case, each bin represents a single value from the range and will contain only those members of S of that value. Finding the desired K^{th} element in this event is trivial since it will always be found during the first pass through the algorithm.

The size of the list, S, may also affect the number of bins needed. When the list is very large it may be necessary, if possible, to double the number of bins indicated by the above equation in order to maintain satisfactory performance of the algorithm. Obviously, the number of bins cannot exceed the range of potential member values as described above. Further simulation of the BSA is required before a definitive answer can be offered regarding the optimal minimum and maximum number of bins to use.

1.5.4.2 Element Placement Method and Bin Range Determination

As presented here, the BSA uses the integer division function to place the elements of S in the appropriate bin. In reality, any function that will allow every element of S to be uniquely placed in the bins will be satisfactory. This function will typically be driven by the type of data element (integer, real, etc.) that is contained in S.

To determine the divisor value required, we take the maximum possible element value and divide it by the number of bins that will be used by the algorithm to generate a solution for this particular list S. The ceiling of this result is the individual bin range size and is used to determine the range of elements, by value, that each bin may contain. This is shown in Figure 1.2 for the example illustrating the use of the BSA.

To ensure that a maximum valued element will not be placed in a nonexistent bin, the divisor value just obtained is multiplied by the number of bins and the result is compared to the maximum possible element value. If this result is greater than or equal to the maximum possible element value then the number of bins is sufficient. If the result is equal to the maximum possible element value then an extra bin must be generated. Only maximum valued elements will eventually be placed in this bin.

Let $S_{max} = 100$ and $S_{min} = 1$

Let $b = \lceil \log_2 S_{max} \rceil = \lceil \log_2 100 \rceil = 7$ bins required

Let $br = \lceil S_{max}/b \rceil = 15$

Thus, the bin ranges are:

Bin 0	0 – 14
Bin 1	15 – 29
Bin 2	30 – 44
Bin 3	45 – 59
Bin 4	60 – 74
Bin 5	75 – 89
Bin 6	90 - 105

Figure 1.2 Initial determination of the number of bins and the bin ranges for the example illustrating the use of the sequential version of the *Bin Selection Algorithm.*

1.5.4.3 Indirect Initialization

Should the situation occur where no information exists about the size of, or the range of values in, S then the implementer has two choices with respect to how the needed information is obtained. First, obviously, the implementer can make a guess as to the bin ranges and the number of bins to be used. Efficiency will definitely be sacrificed unless the guess is accurate but a solution will be generated if all of the elements in S can be placed in a bin (see the earlier discussion about maximum valued elements). If a guess is not possible, or desirable, then an initial pass through S can be made that will generate the needed information. Further discussion of this modification and others can be found in Section 1.9.

1.5.5 Step 2 - Element Distribution

Using the information generated in Step 1, the elements in S are placed into the appropriate bin and the corresponding accumulator is incremented. In the case of the illustrating example provided, integer division, using the value of br as shown in Figure 1.2 for the divisor, is used to place the members of S in the appropriate bins.

Table 1.1 Bin contents after the first pass through S has been completed.

Contents of Bin #						
0	1	2	3	4	5	6
3	15	31	49	62	79	95
7	17	37	51		75	99
1	26	33	57		82	91
11	22	43	55		87	
14						

For example, the first member of S, the value 3, is placed in bin 0 since $3/15 = 0$, where $br = 15$. The member 49 is placed in bin 3 because $49/15 = 3$. The member 87 is placed in bin 5 since $87/15 = 5$. Table 1.1 shows the contents of each bin once all of the members of S have been examined.

As each member of S is placed in a bin, the accumulator corresponding to that particular bin is incremented and reflects the total number of elements that have been placed in the bin. The accumulator value for each bin once all of the members of S have been examined is shown in Table 1.2.

Table 1.2 Final accumulator values for the first pass through S.

Bin	0	1	2	3	4	5	6
Total	5	4	4	4	1	4	3

1.5.6 Step 3 - Locate the K^{th} Element Bin

Once all of the members of S have been placed in a bin, we can determine which bin contains the desired K^{th} element. This is accomplished by starting at the appropriate end of the list of accumulator values and summing their values until the number of elements in the sum equals or exceeds the value of K. The bin that is associated with the last accumulator value added to the sum will be the bin that contains the K^{th} element.

If the K^{th} smallest element in S is desired then the summation of accumulator values starts with bin 0, the bin holding the lowest valued elements of S. The

accumulator value of each succeeding bin is then summed until the total equals or exceeds the value of *K*. In a similar manner, the K^{th} largest element can be found by starting with the bin holding the largest valued elements of *S*. When the summation process stops, the bin that contains the K^{th} element has been determined. The desired element is located in the last bin whose accumulator value was added to the sum.

For example, if we are searching for the 7^{th} largest element of *S*, then starting with bin 6 we sum the accumulator values, moving down the list of bins until the sum equals or exceeds 7. In this case, bin 6 only holds 3 elements so the $K = 7^{th}$ largest element cannot be in bin 6. Bin 5 holds 4 elements and the total number of elements encountered in our traversal of the bin accumulator values is now 3 + 4 = 7. This result equals the K^{th} value we are searching for, 7. Therefore, we have found the bin containing the desired element and can discard the contents of all of the other bins, a total of 21 elements or 84% of the members of *S*. Table1.3 illustrates the percentage elimination of elements of *S* from further consideration using the illustrating example for various values of *K*.

1.5.7 Step 4 - Process / Terminate Logic

With the location of the K^{th} element known, the elements that are stored in the other bins can now be discarded. The proof of this claim is given in Section 1.6. The value of K may need to be adjusted, if necessary, to reflect the removal of the discarded elements so that the correct element will be selected relative to the original list *S*.

Table 1.3 Range of eliminated elements on first pass by K value and the bin location of the desired K^{th} element.

K Value	Elements		
	Location	Remaining	Eliminated
$1 \le K \le 5$	Bin 0	5	20 (80%)
$6 \le K \le 9$	Bin 1	4	21 (84%)
$10 \le K \le 13$	Bin 2	4	21 (84%)
$14 \le K \le 17$	Bin 3	4	21 (84%)
$K = 18$	Bin 4	1	24 (96%)
$19 \le K \le 22$	Bin 5	4	21 (84%)
$23 \le K \le 25$	Bin 6	3	22 (88%)

At this point in the BSA only one of two situations can possibly exist. Obviously, the bin known to hold the desired K^{th} element has only one member of S contained therein. In this case, no further processing needs to be performed to find the K^{th} element and the algorithm can proceed directly to Step 5 (see section 1.5.8).

Otherwise, more than one element remains in the indicated bin. One of these elements is the K^{th} element being sought. Should the number of elements remaining in the bin be small, less than 10 or so, then the elements can be sorted and the desired K^{th} element determined trivially. The point at which the remaining elements should be sorted is left for the implementer to decide. The BSA will yield correct results without regard to the number of elements in S as will a simple sort and pick operation.

If the decision is made to run the remaining elements through the BSA again then several adjustments must be made before Steps 2 and 3 can be repeated. First, the original members of S are replaced with the members now residing in the bin known to

contain the K^{th} element. Next, the accumulators must be initialized to zero and the initialization and setup information determined in Step 1 will need to be recalculated. However, these calculations must reflect the altered problem that now exists. Typically, the number of bins needed will be reduced since a significant percentage of the original elements of S will have been eliminated. Also, the range of values of the remaining elements of S has likewise been reduced and may additionally limit the number of bins needed for the next round of the BSA.

1.5.8 Step 5 - Result(s) Processing

The processing done in this step is trivial if only one K^{th} element is to be found by the BSA. Only the value of the K^{th} element and any statistical information gathered by the implementer needs to be output. However, with minor modification, this algorithm is capable of returning multiple K^{th} largest or smallest element values and providing additional information about the members of the list, S. Further discussion of the modifications to the BSA required to implement this capability, as well as other algorithm modifications, are described in Section 1.9.

1.6 Basis and Correctness of the Bin Selection Algorithm

The basis for this algorithm rises from three simple facts.

1) When searching any list for the K^{th} largest element there must exist $K-1$ elements larger than the K element.
2) For the K^{th} largest element of a list of size X there must exist $X-K$ elements that are smaller than the K^{th} element.

3) Since the list is finite and each element in the list is unique we can determine, or make an informed guess about, the minimum and maximum possible values for any particular list.

The proofs of the first two facts are both obvious and trivial. Fact one is true because to be the K^{th} largest anything implies the existence of $K-1$ items that are larger than the K^{th} item. If fewer or more than $K-1$ items exist in S that are larger than the K^{th} element then the element selected cannot possibly be the K^{th} largest element of S.

Likewise, fact two is true because to be the K^{th} largest anything in a list of X elements implies the existence of $X-K$ items that are smaller than the K^{th} item. If fewer or more than $X-K$ items exist in S that are smaller than the K^{th} element then the element selected cannot possibly be the K^{th} largest element of S. Similar arguments can be made when searching for a K^{th} smallest element of S.

The third fact is derived from the basic assumptions that were made about the list S and its member elements. Obviously, if the list is not finite then a solution will never be found. If the implementer has any information about the nature of the elements in S, knowledge of the valid range of values to be expected from the elements in S is the most commonly available. Even if the implementer knows nothing about the valid range of values expected, a single pass through the members of S recording the maximum and minimum values encountered will provide the needed data for the BSA.

In either case, exact knowledge of the range values is not required for the BSA to operate properly. The implementer need only guarantee that every element in S will be placed in a bin during the first pass of the algorithm through S. So long as every member

of S is accounted for during each pass, the BSA will be able to select the correct element from S. The adjustments that are made in Step 5 before making another pass through the remaining elements of S will insure that this requirement is always met. Should it not be possible to make such a guarantee then the maximum and minimum values of the elements in S must be determined as mentioned above before using the BSA.

The method used by the Bin Selection Algorithm to locate and identify the desired K^{th} element makes use of all three facts above to partition S into three disjoint subsets, one of which is known to hold the K^{th} element. From the first two facts, it can be seen that if we can identify which elements belong to the $K-1$ and $X-K$ subsets of the list and discard only the elements of those subsets then we will be left with the K^{th} element. Sorting the list yields one such solution but it is costly and inefficient for all but the smallest lists.

The third fact provides the limits for the partitioning of the elements in S. Using the maximum and minimum element values, a series of contiguous, unique, and disjoint subranges can be created that will cover the entire range of possible values of the members of S. Direct assignment to a single subrange will allow the partial ordering of the elements of S without having to make any element to element comparisons. A 'bin' is generated for each of the subranges and each element of S is then placed into the bin whose subrange includes the value of that element. A count of the number of elements placed into the bin is maintained during the placement process.

Once the placement process is complete, the bin containing the K^{th} element can be determined. This is accomplished by simply summing the number of elements contained

in each bin until the sum equals or exceeds the value of K. The last bin whose count was added to the total contains the K^{th} largest element.

With the determination of the location of the K^{th} largest element the partitioning of S has been accomplished. By the virtue of the first two facts stated above and the knowledge of the location of the K^{th} largest element, the remaining bins can be safely assigned to the $K-1$ and the $X-K$ subsets. The contents of these bins can now be discarded because they do not contain the element that is being sought and will be of no benefit to the solution in future processing. By adjusting the value of K, if necessary, the BSA can be rerun on the remaining elements of S until the K^{th} element is found directly or until so few elements remain that sorting become feasible.

1.7 Performance Analysis

The Bin Selection Algorithm, as described in earlier sections, behaves in a linear fashion under most circumstances. However, its actual overall behavior is totally dependent upon the distribution of the elements in S and the distribution of the subranges assigned to the bins. Any distribution of elements and bin subranges that is uniform, or approximately uniform, will always result in linear algorithm behavior. The further the distribution of elements strays from the uniform, the worse the behavior of the algorithm.

With a uniform, or near uniform, element distribution in S and a uniform, or near uniform, subrange distribution among the bins, only a single pass will need to be made through all of the elements of S. Each succeeding pass of the BSA operates upon an ever decreasing subset of S until the K^{th} element is determined. In these cases, the time bound of the Bin Selection Algorithm is O(kN), where the value of k has been found to typically

lay between 1.0 and 1.7. These values correspond to an element discard rate of 60% or greater during each pass of the algorithm.

At its worst, the behavior of the Bin Selection Algorithm is $O(N^2)$. In this circumstance, only a single element of S is discarded during each pass and, at most, $N-1$ passes are required to locate the K^{th} element. This is an easily detectable situation. The modifications to the BSA required to account for this, and other, undesirable situations is discussed in Section 1.9.

While element distributions that can lead to $O(N^2)$ behavior are highly unlikely to occur in most instances, convincing arguments can be presented as to the actual occurrence in real-world data of such distributions. The simplest to visualize is an instance where most of the data collected lies within a narrow range of values but where a few values, valid or erroneous (depending upon the nature of the data collected), expand the range of data values enormously. In such a case, the majority of the uniformly generated subranges would be empty with only a few (or just one) containing a significant number of elements from S. Thus, the unmodified behavior of the BSA would approach $O(N^2)$.

1.8 The Parallel Computing Model Version

The parallel version of the Bin Selection Algorithm (pBSA) is a simple and adaptable parallel algorithm for the selection problem. The model of parallel computation used is the concurrent-read exclusive-write (CREW) parallel random access machine (PRAM). Memory conflicts are avoided without requiring the presence of specific hardware or software capabilities other than the ability to temporarily stop or

synchronize individual processors. The parallel form of this algorithm is optimal in the sense that its total cost is O(N) as is the sequential version of the BSA.

1.8.1 Resource Requirements

For the CREW parallel machine implementation being presented here, the resource requirements deal with processor capabilities and memory architecture. Obviously, the availability of at least two processors is required. With regards to the memory architecture, it is required that both local (to each processor) and shared memory resources be available. Details of these resource requirements are given below.

1.8.2 Processor Requirements

For the purpose of this discussion, each processor must:

1) be synchronizable.
2) be independent of all other processors.
3) have a unique id number and be able to identify itself using this id number.
4) have access to a common block of memory (shared memory).
5) have access to a private block of memory (local memory).

As presented, this version of the algorithm assumes that the five requirements mentioned above are available as stated. They greatly simplify various aspects of the pBSA, especially memory access conflict resolution during steps 3 and 5 of the algorithm (sections 1.8.9 and 1.8.11, respectively). However, the pBSA can be easily modified to

deal with machines and situations that do not meet some or all of these requirements. Such modifications will be discussed in Section 1.9.

The first three requirements listed above apply to processor capabilities needed by the parallel version of the BSA. At various steps in this algorithm, each processor will be required to copy data to or from the shared memory resource. To accomplish this it will be necessary to start or stop all of the processors at essentially the same time at various points during execution. This synchronizing operation ensures that no processor will race ahead of any other and potentially invalidate the results of the algorithm.

The last two requirements listed above are discussed below in detail with regards to overall algorithm behavior and requirements. All processors must have access to a section of memory that either is, or can be, dedicated exclusively to a single processor. This is needed so that the selection problem can be divided among the available processors without incurring unsolvable or time-wasting memory conflicts. Similarly, the processors must also have access to a common block of memory. This is needed so that the processors can share the results of their work without requiring a complex message passing scheme to accomplish this task.

As with the processors, certain assumptions have been made about the availability and amounts of certain forms of memory. In particular, the parallel version of the BSA, as presented here, requires shared and local memory resources. Each memory resource has a specific place in the operation of the algorithm and a specific required minimum amount for proper algorithm operation.

1.8.3 Shared Memory Requirements

The shared, or common, memory resource constitutes the main repository for the data set and the intermediate results information created during the solution process. It is assumed that enough sharable memory is available to hold the entire list S while processing occurs and a number of accumulators equal to the number of bins used to solve the problem. Therefore, the maximum amount of shared memory needed is $(|S| = bins)$, where *bins* is the number of bins required by the algorithm to solve the given problem, one accumulator per bin.

A reasonable estimate of the minimum amount of shared memory needed is enough memory for one accumulator for each bin required by the algorithm to solve the given problem. The actual minimum amount of shared memory needed will depend entirely upon any modifications made to the algorithm's implementation by the implementer. Some of these potential modifications will be mentioned in Section 1.9.

1.8.4 Local Memory Requirements

The local, or private, memory resource will be used to store the disjoint subset of S that is assigned to each processor and the local and global solution data generated by the pBSA as it executes. This is accomplished by using enough local memory to store a complete set of bins and a number of accumulators equal to twice the number of bins used by the pBSA to solve the given problem. Rather than store the members of the disjoint subset of S in the local memory resource and then place them in the appropriate bin, local memory usage is reduced by storing the elements at the time they are placed in

a bin so that only one copy of the subset is kept in local memory. Thus, the maximum amount of local memory required is $\lceil |S|/P \rceil + (2 \times bins)$.

As with the shared memory resource mentioned earlier, a reasonable estimate of the minimum amount of local memory needed is enough local memory for two sets accumulators $(2 \times bins)$. The actual minimum amount of shared memory needed will depend entirely upon any modifications made to the algorithm's implementation by the implementer. Some of these potential modifications will be mentioned in Section 1.9.

1.8.5 Algorithm Description

The parallel version of the Bin Selection Algorithm (pBSA) is an eight step solution to the general selection problem. This version differs from the sequential version previously discussed by adding steps to solve algorithmic difficulties unique to parallel computing, especially processor synchronization requirements and data communication. These steps can be summarized as follows:

Step 0 - Initial Assumptions

Step 1 - Initialization

Step 2 - Element Distribution

Step 3 - Determine Total Bin Counts

Step 4 - Locate the K^{th} Element Bin

Step 5 - Copy Selected Bin Contents to S

Step 6 - Process/Terminate Logic

Step 7 - Result(s) Processing

Each step is described in detail below. Throughout this discussion a running example will be used to illustrate the operations performed during each step of the algorithm.

1.8.6 Step 0 - General Assumptions and Knowledge

As for the sequential version of the BSA, for the purpose of easing the complexity of this discussion, it is assumed that certain conditions exist. The pBSA starts with the same assumptions that were presented during the discussion of the sequential version of the BSA and for the same reasons presented therein. For the discussion of the pBSA, the following assumptions are added to those described earlier:

- All of the processor and memory requirements described in Section 1.8.1 have been met.
- The number of processors used must be less than or equal to the total number of bins.

The assumption dealing with the number of processors and bins used to solve the given problem is necessary. Meeting it allows for the use of a very simple element write-back scheme later in this algorithm. Without it, it may not be possible to guarantee that all processors can access shared memory without conflict. This assumption is discussed in greater detail in Section 1.8.9.

The parallel systems implementation assumptions are required in general. In order for the algorithm to function properly on a parallel architecture machine, the capability to synchronize or to start and stop individual processors is essential in maintaining orderly processing. Otherwise it is possible that a processor with fewer list

elements to process or operates faster than its siblings will proceed to the next step of the algorithm before the current processing step is completed by all processors.

Should it not be possible to meet the synchronization requirement, it will be necessary to guarantee that each processor is operating on the same number of elements and that each processor takes the same amount of time to process the same number of elements. This situation will be discussed further in Section 1.9.

1.8.7 Step 1 - Initialization

During this stage of the pBSA, the overall structure of the solution to the given problem is generated. Of critical importance to the efficient operation of the algorithm is the determination of the number of processors to use and the total number of bins needed. From these two items the rest of the required initial data can be generated, namely, the bin content ranges and the element placement value. These items are discussed in greater detail below and illustrated with the example in Figure 1.3.

1.8.7.1 Determining Processor and Bin Requirements

The number of processors that should be used is related solely to the number of elements in the list, S. Using a number of processors equal to the base 2 logarithm of the size of S has yielded good results. The pBSA does not have any form of fixed requirement as to the number of processors it requires for valid processing. It will work equally well with one processor or with as many processors as are available. However, the number of processors used cannot be larger than the size of S.

Let $N = \lceil \log_2 |S| \rceil = \lceil \log_2 25 \rceil = 5$ processors needed, pids = 0 to 4.

Let $X = |S|/N = 5$ elements from S per processor.

Let $S_{max} = 100$ and $S_{min} = 1$.

Let $b = \lceil \log_2 S_{max} \rceil = \lceil \log_2 100 \rceil = 7$ bins required.

Let $br = \lceil S_{max}/b \rceil = 15$.

Thus, the bin ranges are:

bin 0	0 – 14
bin 1	15 – 29
bin 2	30 – 44
bin 3	45 – 59
bin 4	60 – 74
bin 5	75 – 89
bin 6	90 - 105

Figure 1.3 Initial determination of the number of processors and bins and the bin ranges for the example illustrating the use of the parallel version of the *Bin Selection Algorithm*.

The number of bins required by the parallel version of the BSA is the same as for the sequential version. The only difference is in the type of memory being used. In this instance, the memory being referenced is the local memory of each processor. See section 1.5.4 for further details.

1.8.7.2 Element Placement Method and Bin Range Determination

This is accomplished for the parallel version of the BSA using the same methods as the sequential version. See section 1.5.4 for further details.

1.8.7.3 Indirect Initialization

Likewise, this is accomplished for the parallel version of the BSA using the same methods as the sequential version. See section 1.5.4 for further details.

1.8.8 Step 2 - Element Distribution

Using the information generated in Step 1, the elements in S are divided among the processors, with each processor retrieving the elements assigned to it. Figure 1.3 shows the number of elements, X, each processor is to retrieve. Table 1.4 shows the location in S where each processor is to start copying its elements. Note that no processor will ever attempt to access the same memory location during this process.

As each element is copied from shared memory it is placed into the appropriate bin in local memory and the corresponding accumulator incremented. Since it is possible that not all processors will have the same number of elements of S to operate upon, no processor can be allowed to proceed until every processor has finished placing the elements of S assigned to them in their bins. Therefore, a pause for processor synchronization is needed before the next step in the pBSA can be executed.

Table 1.4 Positions in S where each processor is to start copying the elements of S to local memory.

Processor #	Q value
0	pid * X = 0 * 5 = 0
1	5
2	10
3	15
4	20

Table 1.5 Bin contents, by processor, after the first pass through S has been completed.

Processor	Contents of Bin #						
	0	1	2	3	4	5	6
0	3	15, 17		49		79	
1		26	31	51	62		95
2	7		37, 33			75	99
3	1, 11		43	57		82	
4	14	22		55		87	91

As each element is copied from shared memory it is placed into the appropriate bin in local memory and the corresponding accumulator incremented. Since it is possible that not all processors will have the same number of elements of S to operate upon, no processor can be allowed to proceed until every processor has finished placing the elements of S assigned to them in their bins. Therefore, a pause for processor synchronization is needed before the next step in the pBSA can be executed.

Table 1.5 shows the resulting bin placement of the elements of S by each processor. Table 1.6 shows the status of the processor and shared memory accumulators before the processors write their bin count (accumulator) values to their shared memory counterpart.

Table 1.6 Local memory bin element counts and shared memory accumulator values after the first pass element distribution has been completed.

Bin	Processor #													
	0		1		2		3		4		5		6	
	bc	f	bc	f	bc	f	bc	f	bc	f	bc	f	bc	f
0	1	0	2	0	0	0	1	0	0	0	1	0	0	0
1	0	0	1	0	1	0	1	0	1	0	0	0	1	0
2	1	0	0	0	2	0	0	0	0	0	1	0	1	0
3	2	0	0	0	1	0	1	0	0	0	1	0	0	0
4	1	0	1	0	0	0	1	0	0	0	1	0	1	0

where: bc - the number of elements in the bin (local memory).
f - accumulator value before bc is added (saved to local memory).

1.8.9 Step 3 - Determine Total Bin Counts

At this point all of the elements in S have been placed into the appropriate bin in the local memory of a processor and the total number of elements placed in each bin determined. Also, at the end of the previous step each processor was temporarily halted so that every processor can begin the execution of this step at the same time. This synchronization pause is required so that the processors can add their bin accumulator values to their shared memory counterparts without memory conflicts and to guarantee that every processor does, in fact, get the opportunity to do so for every bin accumulator.

The goal of this step in the pBSA is to reassemble the data generated by each processor from its disjoint subset of S so that the bin location of the K^{th} element can be

determined. In addition, the data required for the correct execution of Step 5 is generated during this process and must be preserved for later use.

Table 1.7 The order in which the processors write their bin counter values to the accumulators in shared memory

Time	Accumulator #										
	0	1	2	3	4	5	6	0	1	2	3
0	p0	p1	p2	p3	P4						
1		p0	p1	p2	p3	p4					
2			p0	p1	p2	p3	p4				
3				p0	p1	p2	p3	p4			
4					p0	p1	p2	p3	p4		
5						p0	p1	p2	p3	p4	
6							p0	p1	p2	p3	p4

Memory conflicts are avoided by using the processor id's to determine where in the accumulator list each processor should begin. Thus each processor is assigned to a unique starting position and continues up the accumulator list from that point until it has added all of its' bin counts to the appropriate accumulator. The write order for each processor is shown for the running example in Table 1.7.

It should be noted again that this step assumes that the number of processors used is less than or equal to the number of bins required by the BSA to solve the given problem. So long as this assumption holds then a simple write-back scheme, such as the

scheme illustrated in Table 1.7, will meet the needs of this step in the algorithm. The difficulty in generating a new write-back scheme to meet the requirements of this step is such that ensuring that the initial assumption holds is the only efficient and optimal approach to this problem.

The fifth step of the pBSA copies the contents of the bin known to contain the desired K^{th} element from each processors' local memory resource to the shared memory resource. It is necessary that each processor be able to write its data without overwriting the data placed in shared memory by the other processors. Furthermore, it is essential that all of the data written by the various processors to shared memory be written contiguously. Any gap that is left in shared memory during the write-back process will allow one or more of the previous elements of *S* to erroneously appear in the new list *S*. Table 1.8 shows the results of this step as each processor stores the current accumulator value before adding its local bin count to each accumulator in shared memory respectively.

Once every processor has finished adding their bin counts to the accumulators in shared memory, a second pass is made through the accumulator list in the same order. The final value of each accumulator is copied into the local memory of each processor and stored in the second set of accumulators contained therein. This data is needed by each processor in Step 4 so that the location of the K^{th} element can be determined. For the illustrating example the results of this last pass are shown in Table 1.9.

Table 1.8 Final results of steps 2 and 3 from the first pass through *S*.

Bin	Processor #													
	0		1		2		3		4		5		6	
	bc	f	bc	f	bc	f	bc	f	bc	f	bc	f	bc	f
0	1	0	2	1	0	3	1	2	0	1	1	3	0	3
1	0	5	1	0	1	2	1	1	1	0	0	3	1	2
2	1	4	0	4	2	0	0	1	0	0	1	2	1	1
3	2	2	0	4	1	3	1	0	0	0	1	1	0	1
4	1	1	1	3	0	3	1	3	0	0	1	0	1	0

where: bc - the number of elements in the bin (local memory).
f - accumulator value before bc is added (saved to local memory).

1.8.10 Step 4 - Locate the K^{th} Element Bin

Each processor, working with the final accumulator values that were copied to local memory during the previous step, now determines which bin contains the desired K^{th} element. This is accomplished in exactly the same manner as described earlier in Section 1.5.6 by each processor working on the given problem. Since all of the processors copied the same accumulator values from shared memory, each processor will reach the same result. What has not been determined, however, is which processor actually has the K^{th} element and its actual value.

Table 1.9 Final accumulator values for the first pass through *S*.

Bin	0	1	2	3	4	5	6
Total	5	4	4	4	1	4	3

Table 1.10 Range of eliminated elements on first pass by *K* value.

K Value	Elements		
	Location	Remaining	Eliminated
$1 \le K \le 5$	Bin 0	5	20 (80%)
$6 \le K \le 9$	Bin 1	4	21 (84%)
$10 \le K \le 13$	Bin 2	4	21 (84%)
$14 \le K \le 17$	Bin 3	4	21 (84%)
$K = 18$	Bin 4	1	24 (96%)
$19 \le K \le 22$	Bin 5	4	21 (84%)
$23 \le K \le 25$	Bin 6	3	22 (88%)

1.8.11 Step 5 - Copy Selected Bin Contents to *S*

With the completion of Step 4 of the pBSA, the bin containing the K^{th} element has been identified. In addition, those members of *S* that are of no further use in the search for the K^{th} element have been identified and can be discarded. In this step, the members of the targeted bin are preserved and the remaining members of *S* are discarded

by simply having each processor write the contents of the target bin back to shared memory overwriting the previous contents of S stored there.

Only those processors that have non-empty target bins perform the write back operation. Processors that have empty target bins make no attempt to write to the shared memory area and temporarily halt until all of the other processors have completed their write back operations. Since it cannot be predicted in advance how long this step will take, another processor synchronization step is required at this time.

Each processor copies the contents of the target bin back to the space occupied by S in shared memory independently of any other processor. Memory write conflicts are avoided because each processor starts writing the contents of its target bin at a unique position in the shared memory area that originally contained S. These unique starting positions were determined during Step 3 (see Section 1.8.9) and stored in the second set of accumulators kept in local memory for this purpose.

Two additional operations must be performed before moving to the next step of the pBSA. After the copy operation is complete, the size of S must be reset to equal the accumulator value of the targeted bin (i.e. the number of elements of S that the targeted bin contained). Also, since elements of S both larger and smaller than K may have been discarded, the value of K may itself have to be adjusted.

The adjustment to K is relatively simple. If the search is for the K^{th} largest element in S, then the sum of the number of elements of S in the bins above, but not including, the target bin must be subtracted from K. For the K^{th} smallest element in S, the sum of the number of elements of S in the bins below, but not including, the target bin must be subtracted from K.

Referring the illustrating example, if $K = 5$ smallest is the desired element, then from Table 1.9 it can be seen that this element must be located in bin 0. This is true because bin 0 holds the smallest elements of S and it currently contains five (5) elements. Thus, the contents of each processor's bin 0 must be copied back to shared memory.

Using the appropriate value of f, shown in Table 1.8, as the starting index into the shared memory space occupied by S, each processor copies the elements in the selected bin starting at that location. The size of S is set to the number of elements in bin 0. Since there are no bins below bin 0, no adjustment to the value of K is required. The final result of this step is illustrated in Figure 1.4.

Processor #	Starting at Position	Copies to Shared Memory
0	0	3
1	5	
2	4	7
3	2	1, 11
4	1	14

$S = (3,\ 14, 1, 11, 7)$ and $|S| = 5$

Figure 1.4 Copy the selected bin contents (bin 0) to shared memory.

1.8.12 Step 6 - Process / Terminate Logic

With the completion of Step 5, all of the processors can begin executing the process/terminate logic of the pBSA. At this point only one of two situations can

possibly exist. First, we can have $|S| = 1$ with the desired K^{th} element the only remaining member of S. If so, then just Step 7, result(s) processing, remains to be performed.

Otherwise, more than one element remains in S including the K^{th} element. Should the number of elements remaining in S be small, less than 10 or so, the remaining elements can be sorted and the desired K^{th} element determined trivially. The point at which the remaining elements should be sorted is left for the implementer to decide. The pBSA will yield correct results without regard to the number of elements in S as will a simple sort and pick operation.

If the decision is made to run the remaining elements of S through the pBSA once more then several adjustments must be made before Steps 2 through 6 can be repeated. Essentially, the initialization and setup information determined in Step 1 needs to be recalculated. These calculations will reflect the altered problem that now exists. Typically, the number of bins and processors needed will be greatly reduced since a significant percentage of the original elements of S will have been eliminated. The range of values of the remaining elements of S is much smaller than the original range.

1.8.13 Step 7 - Result(s) Processing

The processing done in this step is trivial if only one K^{th} element is to be found by the BSA. Only the value of the K^{th} element and any statistical information gathered by the implementer needs to be output. However, with minor modification, this algorithm is capable of returning multiple K^{th} largest or smallest element values and providing additional information about the members of the list, S. Further discussion of the

modifications to the BSA required to implement this capability, as well as other algorithm modifications, are described in Section 1.9.

1.9 Modifications, Other Uses, and Future Work

The adaptability of the BSA can be demonstrated by noting the ease with which it can be used by both sequential and parallel computing machines. This occurs because this selection method requires at least one processor and a minimum of two bins to function (trivial case). The use of additional processors, if available, decreases the time needed to determine the solution. The use of additional bins, should sufficient memory be available, allows for more elements to be discarded after each iteration thereby increasing the efficiency of the algorithm. Neither modification is required or needed, however, for the algorithm to function properly, effectively, or efficiently.

1.9.1 Non-Uniform Data and Bin Subrange Distributions

As originally designed and presented here, the Bin Selection Algorithm does not handle non-uniform data distributions well. As mentioned earlier, the worst case behavior for this algorithm is $O(N^2)$, occurring when only one element is eliminated during each pass through the BSA. When a non-uniform element distribution is detected, either before or during processing, the manner in which the bin subranges are assigned and the element placement method must be altered to reflect the expected distribution pattern as closely as is possible under the circumstances encountered.

One of the advantages of this algorithm over the classical solution is the simplicity and ease with which the data elements are placed in their respective bins. The

placement method used here, integer division, allows for a constant O(1) element placement cost. It is, obviously, very desirable to maintain this situation if at all possible. The following scheme suggests that it may indeed be possible to do so.

This alternative element placement scheme involves the use of a lookup table and, possibly, a hashing function. The lookup table would actually assign the element a bin location and would be accessed (indexed) in one of three ways. The first indexing method would be the simplest, direct indexing. Here, the table would have as many entries as the data range has values. For example, if the valid data range is from 0 to 100 then the table would have 100 entries. It is clearly obvious that in the case of a large range of possible values such a table would become unwieldy and, potentially, too large to store in memory (required if we are to maintain the 0(1) cost factor).

The second indexing method would involve the use of a hashing function to index the lookup table and determine the bin into which the element should be stored. Such a function would also maintain the desirable 0(1) cost factor for element placement. However, it is not possible to guarantee that a hashing function can be found for every data type and distribution situation that could be encountered when using the BSA.

The third indexing method involves the use of subranges and direct element value comparisons to assign elements to a particular bin. In this case, the element values are compared to a cutoff value. If they are less than or equal to the cutoff value then they are assigned to the corresponding storage bin. Otherwise, the element value must be compared to the next subrange cutoff value in the table. This process continues until the element is placed into a bin.

It is obvious that, as stated, this final indexing method is not 0(1) in cost but dependent upon the actual number of comparisons needed to place each element into a bin. Other methods of organizing the table for such a lookup exist but they all depend upon some level of knowledge of the actual element distribution pattern, knowledge which may not be available or easily obtained. The best overall solution to this problem, and one which reduces the overall cost of element placement in this case as much as possible, is to use a binary search to access the table. This would serve to keep the number of comparisons needed to place any element in the data set as close to a single constant factor as possible and yet retain the overall simplicity that the BSA, as a whole, exhibits.

1.9.2 Memory and Data Set Size Problems

It is entirely possible that the size of the data set being manipulated is too large to be kept entirely in memory or is large enough to disallow the possibility of keeping two copies of the set in memory as the original design of the Bin Selection Algorithm requires. In both cases, the algorithm can be easily adjusted to deal with both circumstances through the use of one or more files stored on a hard disk. It is assumed that enough memory exists to keep all of the necessary accumulators in memory and any bookkeeping information that the implementer may need when using disk files to handle the data set and bin element storage requirements.

The first situation mentioned above is a worst case situation when dealing with large data sets. In this case, the algorithm will have to be modified so that all reading and storage of elements will take place using disk files. In this event, some sort of lookup

table will be needed so that the correct disk file will be accessed during processing. Refer to the earlier discussion on non-uniform element distributions in Section 1.9.1 for further information on the creation and use of such tables. While the use of disk files for data set and bin element storage will greatly slow the overall speed of execution of this algorithm, a properly constructed method for handling the files and the placement of elements therein should not appreciably affect the overall cost of the algorithm.

The second situation mentioned above, where the size of the data set is such that two copies of the set will not fit in memory is the easier to the two situations to handle. Here, we do have enough memory to store one copy of the data set and the required accumulators. Therefore, the data set itself is kept on disk and is read from there while memory holds a copy of each element in the set based upon bin location. When the bin holding the K^{th} element is determined, the bin contents can be copied to a new data file on disk, or, if the implementer chooses, can be kept in memory if the number of elements has been sufficiently reduced so that it is now possible to maintain two copies of the reduced data set in memory.

1.9.3 Parallel System Implementation Problems

The number of different parallel system designs is such that no single parallel algorithm description is sufficient to deal with all of the differences between the various designs. The CREW PRAM design presented here is a general design intended to convey the overall functionality and simplicity of the Bin Selection Algorithm and the ease with which it can be modified for use on different computing machinery. Whether or not the BSA can be implemented on any particular machine as it is described in this work is

directly dependent upon the requirements of the algorithm as stated in its description presented earlier.

Regardless of the type of parallel machine being used, one factor must be maintained if the algorithm is to function as described. It must be guaranteed that no single processor can race ahead of any other processor during the execution of certain critical steps of the algorithm, such as when data is being written to the accumulators during step 3, Section 1.8.9. How this guarantee can be kept is up to the implementer since only this person can judge what works best for their particular machine.

This is accomplished in the BSA through the ability to stop any or all processors temporarily, to restart processors, and through the use of the read/write synchronization method illustrated in table 1.7. If the scheme presented will not work for the destination machine then the current methods for synchronizing the reading and writing of elements and data in memory must be modified such that all processors can perform all required operations without memory collisions, lockouts, or any other form of interference. Otherwise, it cannot be guaranteed that the algorithm will yield correct results or even be able to operate at all.

1.9.4 Other Uses

A major improvement afforded by this algorithm over the classic algorithm is that the Bin Selection Algorithm can be used to find more than one K^{th} element from a list S without having to search the entire list each time. By saving the bin contents and the accumulator values to permanent storage after the first element distribution pass through S, other K^{th} elements can be determined without repeating the expensive first pass. If

sufficient memory exists, the bin contents and accumulator values can be copied to another section of memory for later use.

In either event, the results of the first element distribution pass through S can be recalled and reused without incurring first pass costs. Step 3 of the BSA must be modified to perform the desired save operation since this is when the total bin counts are determined.

This algorithm can also be modified to yield more information about the elements in the list than just the single K^{th} element desired. The actual minimum and maximum values for each subrange as well as the entire list can be determined during execution. Element distribution information can be acquired as well as a count of the elements within specified subranges or those meeting other specified conditions.

1.9.5 Future Work

Further work is needed with regard to the method used to determine the optimal number of bins to be used. The integer division method works well and is simple to implement. However, no account is taken of element value distributions and the number of processors and memory available when the number of bins required is computed. It is believed that a method of calculating an optimal number of buckets required is possible and that such a calculation need not be complex or time-consuming.

1.10 Conclusion

As has been shown above, the Bin Selection Algorithm is a fairly well-behaved linear solution to the Selection problem. It is also a highly and easily modifiable

algorithm. Its very simplicity makes it a very easy to understand and implement on a wide variety of computer systems in any programming language. Designed for normal element distributions, with minor alterations this algorithm can be made to handle skewed element distributions efficiently. Even the worst case scenario for this algorithm, a sparse list with a large range, can be handled by relatively simple modifications to the basic algorithm.

CHAPTER 2

STEINER TREES

The general Steiner Tree problem in graphs is the problem of finding a tree connecting a given set of nodes, S, in a connected undirected distance graph $G=(V,E,d)$, where V is the set of vertices, $S \subseteq V$ is the set of Steiner vertices, and E is the set of edges in the graph. The minimum Steiner tree for G and S is a tree which spans S with a minimum total distance on its edges. For a graph $G=(V,E,d)$, where $d=1$ for all edges, the minimum Steiner tree for G and S is a tree which spans S that includes the minimum number of vertices not in S.

This problem has long played a prominent role in many physical design tasks. It has a wide variety of practical applications such as communication networks, layout design of printed circuit boards and integrated circuits. It also has application in the mechanical and electrical systems in buildings, distribution and transportation networks, the wire routing in physical VLSI design and in phylogeny (evolutionary trees.)

2.1 Previous Work

It has been shown that finding a minimum Steiner tree for any given G and S is NP-Complete [8]. In fact even when distance functions are restricted to a particular class, the problem is still NP-Complete [8]. This means that it is unlikely that an efficient algorithm can be found to compute the minimum Steiner tree for any given G and S.

Due to its importance, there has been a great deal of work on approximation algorithms for the Steiner tree problem [4, 9, 10, 11, 12, 16, 17, 19, 20, 21, 22, 23]. Takahashi and Matsuyama [19] present an $O(|S||V^2|)$ approximation algorithm for finding a Steiner tree in a given G and S. Moreover, it has been shown that $D(T_s)/D(T_{opt}) \leq 2(1-1/|S|)$ where $D(T_s)$ denotes the total distance on the edges of the Steiner tree generated by their algorithm and $D(T_{opt})$ denotes the total distance on the edges of the optimal Steiner tree [19].

Kou and Makki [9] later developed an $O(|E|+|V-S|\log|V-S|+n\log\beta(n,|S|))$ approximation algorithm where $n=\min(|E|,|S|(|S|-1)/2)$ and $\beta(n,|S|)=\min\{i,\log^i \leq n/|S|\}$. The ratio of the total distance on all the edges of a Steiner tree generated by the algorithm to that of the optimal tree is not greater than $2(1-1/L)$ where L represents the number of leaves in the optimal tree. Their bound has been further improved by Makki [11] to $O(|E|+|V|\log|V|)$.

Waxman and Imase [20] have shown that Rayward-Smith's approximation algorithm [16] is also bounded by $2(1-1/L)$ [20]. The bound $2(1-1/L)$ has been known to researchers in the field for a decade as the best worst case ratio. Recently Zelikovsky [23] has shown that this worst-case bound can be improved and his result has been further refined by Berman and Ramaiyer [4].

2.2 A New Steiner Tree Algorithm

This work considers a special case of the Steiner tree problem in graphs. For this problem it is assumed that the underlying graph G does not have any direct edge between the vertices in $S \subseteq V$ and that all edges in E are of unit length. The problem is to find a

tree in G which spans the vertices in S and uses a minimum number of vertices in $V - S$. Makki and Pissinou [14] were the first to formulate this problem and present an efficient approximation algorithm for it. Here, we present and analyze a new approximation algorithm for this problem.

2.3 General Description

This algorithm differs from the other solutions proposed for the Steiner Tree problem in that it does not follow normal computer science algorithm design practices. Rather than use prior art or an elegant application of various common algorithms and techniques, this algorithm uses a *basic engineering brute force approach.* This is a result of the basic philosophy used during the design of this algorithm that can be best described as the *'what do I know now and how do I take advantage of this fact'* philosophy. This question and the eventual answer guided and controlled the entire design process. Additionally, as each step in the solution process was determined and verified, every effort was made to keep the impact and implementation of the new solution step as simple as possible to avoid unnecessary complications to previous steps in the algorithm.

What finally emerged from this process is the relatively simple bookkeeping algorithm presented below. It makes use of five data structures to organize all of the original graph data and to track the solution process as the Steiner Tree is generated. As the status of the various nodes and edges of the graph G change, these data structures are modified accordingly. This process is the heart of the algorithm since the change in value of the various nodes and edges in G as the solution tree construction progresses determines the next step in the tree construction process.

In general, this algorithm operates by creating a series of subtrees that contain all of the Steiner vertices, S, and then connecting these subtrees until a single tree contains all of the Steiner vertices. First, all of the vertices in the $N-S$ subset that have edges leading to more than one Steiner vertex are selected. For each such vertex, a subtree consisting of this vertex, the Steiner vertices it is connected to by a single edge, and those edges is generated. If all of the Steiner vertices have been placed in subtrees during this step then a check is made to see if a solution has been obtained. If a single subtree contains all of the Steiner vertices then a solution has been found and the algorithm ends.

Should the first step not result in a solution then one of two possibilities exists. Either all of the Steiner vertices have not been placed into subtrees or there exists two or more disjoint subtrees holding all of the Steiner vertices. Should the first situation arise and not all of the Steiner vertices be members of the existing subtrees then each remaining Steiner vertex is placed in a subtree of its own along with one non-Steiner vertex and their connecting edge. All of the vertices selected during these two steps are considered *selected* and are so marked in the various data structures.

Now that all of the Steiner vertices are known to be in two or more disjoint subtrees it is necessary to connect these subtrees thereby generating the desired solution tree. It is possible that the addition of one or more edges will complete the connection process so the data structures are examined to locate such edges. Any such edge located is added to the subtree list as a subtree consisting of the two nodes and the desired new edge. All of the subtrees thus joined together by the addition of the new edge are marked with the status value *connected*. All vertices and edges so marked are considered to part

of the final solution tree. Should all of the Steiner vertices have the status *connected* at the end of this step then the solution tree has been generated and the algorithm halts.

If the addition of single edges, if possible, does not generate the solution tree then the disjoint subtrees are more than a single edge distant from each other. To join these subtrees it will be necessary to generate new subtrees from the list of available non-Steiner vertices until all of the currently disjoint subtrees can be connected together. Each new subtree will be composed of one new non-Steiner vertex and two edges, one of which leads to a Steiner or non-Steiner vertex that has been marked as *selected* or *connected.* This process continues until all of the Steiner vertices have been marked *connected*, thus indicating that the solution tree has been generated.

The key to this entire process is the record keeping required by the changing status and value of the various vertices in the original graph G. A vertex becomes more valuable and important to the eventual solution as the number of its edges leading to Steiner, *selected*, or *connected* vertices increases. Such vertices will be used before any other vertex is chosen for inclusion in the solution tree. When no such important vertices can be identified then a simple selection criterion is used to choose a vertex to add to the subtree list (and the eventual solution tree).

2.4 Basic Definitions

Given a connected undirected graph $G = (V, E, 1)$ where

- V is a set of vertices
- E is a set of pairs of vertices called *edges* that connect the vertices
- all edges are of unit length

We define

1. $S \subseteq V$ be the set of Steiner vertices (nodes)
2. $NS = V - S$ be the set of non-Steiner vertices
3. R to be the solution tree output by this algorithm
4. P_i to be the i^{th} subtree, not yet a part of R

In addition to being partitioned into one of the two above subsets of G, each vertex has two distinct attributes associated with it. These attributes are the select status and the connect status of the vertex. The select status is used to indicate if the vertex has been chosen as part of a subtree P_i in the subtree list. A '*selected*' vertex is informally considered to be a part of the solution tree R.

The connect status is used to indicate if the vertex has been formally included in the solution tree R. Connect status is also used to indicate which vertices and, therefore, which subtrees P_i have not yet been joined to the tree R. The algorithm stops when all of the Steiner vertices have been '*connected*'. A vertex cannot be '*connected*' to the tree R if it has not been '*selected*' by the algorithm. Thus the valid status values for any vertex are *unselected/unconnected*, *selected/unconnected*, and *selected/connected*.

2.5 An Example Problem

In order to demonstrate the operation of this algorithm, a sample graph G is provided for illustrative purposes. The example graph is shown in Figure 2.1.

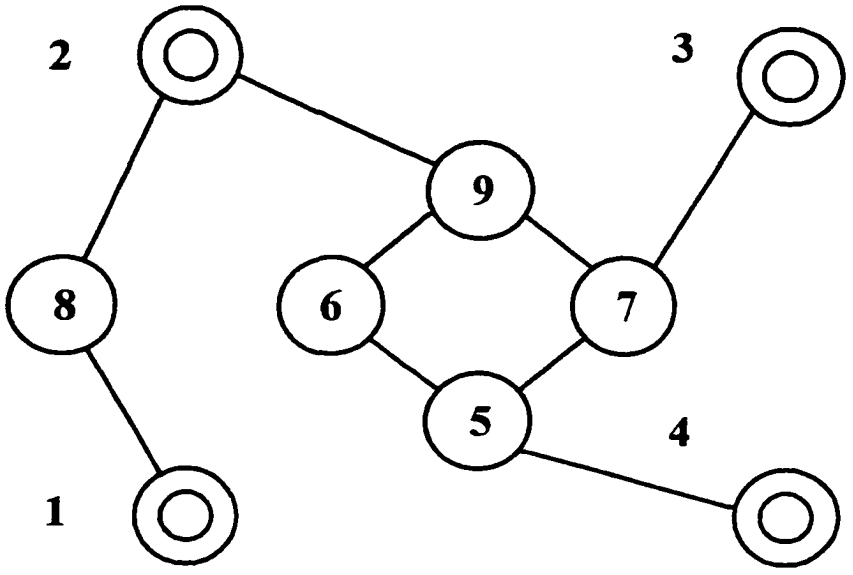


Figure 2.1 Example graph G with Steiner Nodes (donuts) used to illustrate the operation of this algorithm.

2.6 The Required Data Structures

Five data structures are used to organize and track the status of the Steiner and non-Steiner vertices in G as the solution tree R is generated.

2.6.1 The Steiner Node List (*SNL*)

This list contains vertex connection data from G pertaining to the vertices in S. This information is represented as a one-dimensional array the size of $|S|$. Each location in the array represents a single vertex in S, S_i, and has space reserved for a pointer to an ordered linked-list of vertices in S and *NS*, respectively. This list consists of those vertices that have an edge in common with S_i and, in the future, is referred to as the

'edge-list.' Each edge-list is ordered by vertex number. This array, representing the vertex set S, is the Steiner Node List, *SNL*.

Table 2.1 The Steiner Node List (*SNL*) after initialization.

S_i	Edge List
1	8 □
2	8 9 □
3	7 □
4	5 □

2.6.2 The Non-Steiner Node List (*NSNL*)

This list contains vertex connection data from *G* pertaining to the vertices in *NS*. This information is represented as a one-dimensional array the size of $|NS|$. Each location in the array represents a single vertex in *NS*, NS_i, and has space reserved for a pointer to an ordered linked-list of vertices in *S* and *NS*, respectively. This list consists of those vertices that have an edge in common with NS_i and, in the future, is referred to as the 'edge list.' Each edge list is ordered by vertex number. This array, representing the vertex set *NS* is the Non-Steiner Node List, *NSNL*.

Each location in the *NSNL* array also has space reserved for several vertex status values used by the algorithm to select or reject vertices for incorporation into the solution tree *R*. For each vertex in the *NSNL* four status values are kept. They are the number of Steiner nodes sharing an edge with the vertex, the total number of vertices sharing an edge with the vertex, the number of vertices that have '*selected*' status, and the number of

vertices that have '*connected*' status. Space is also reserved for each vertex in the *NSNL* for a pointer to the vertex's location in the *pick list*.

Table 2.2 The Non-Steiner Node List (*NSNL*) after initialization.

#S	#V	#sel	#con	NS_i	Edge List
1	3	0	0	5	4 6 7 □
0	2	0	0	6	5 9 □
1	3	0	0	7	3 5 9 □
2	2	0	0	8	1 2 □
1	3	0	0	9	2 6 7 □

2.6.3 The Pick List (*PL*)

The pick list, *PL*, is an ordered linked-list containing index values into the *NSNL*. Membership in the pick list changes twice during the algorithm's operation. Initially, the pick list contains indices to those *NSNL* vertices that have edges leading to multiple Steiner vertices. Later, the pick list contains those vertices in the *NSNL* that have the most edges leading to vertices in the subtrees P_i in the *SL*. Members of *PL* are ordered by vertex number.

Table 2.3 The first Pick List (*PL*) after initialization.

PL	8	5	7	9	□

2.6.4 The Subtree List (*SL*)

A data structure similar to the *SNL* is used to store the subtrees, P_i, generated by the algorithm that may eventually be part of the solution tree R. Each location in the array contains a pointer to a single subtree P_i. This array is the subtree list, *SL*.

Table 2.4 The Subtree List (*SL*) after initialization.

P_k	Vertex (edge) List
0	□
1	□

2.6.5 The Select/Connect Table (*SCT*)

A simple data table is used to track the solution status as the algorithm proceeds. For each vertex in G the following information is stored in the data table: the connection and selection status of the vertex, the number of subtrees in the subtree list, *SL*, that contain the vertex, and a linked list of *SL* array indices indicating the subtrees P_i in which the vertex has membership. This data table will be referred to as the Select/Connect Table, *SCT*.

Table 2.5 The Select/Connect Table (*SCT*) after initialization.

vertex #	1	2	3	4	5	6	7	8	9
selected									
connected									
# of subtrees	0	0	0	0	0	0	0	0	0
subtree list	-	-	-	-	-	-	-	-	-

2.7 The Algorithm

This approximation algorithm uses a simple bookkeeping method for determining the shortest tree that connects all of the Steiner vertices S in a given graph G. It consists of the following eight steps:

1. Graph Preprocessing
2. Initialization
3. Steiner Vertex Selection
4. Vertex Connection Evaluation
5. Subtree Traversal
6. Single Edge Connection
7. Non-Steiner Vertex Selection
8. Solution Tree Output

2.7.1 Step 0 - Graph Preprocessing

This step consists primarily of ordering the vertices in the graph G in a consistent manner so that the algorithm can operate efficiently. By the nature of the problem, the vertices in the graph G have already been separated into the two subsets S and $NS = V - S$, representing the Steiner and non-Steiner vertices respectively. Only a vertex numbering scheme is required.

The method used to number the vertices of the graph, G, is directly related to the efficiency of this algorithm as it is presented here. Besides serving as a way to order the vertices and provide a convenient manner of specifying the edges of G, it drives the

simple vertex selection methods that will be used at various stages during the execution of this algorithm. Therefore, care must be taken when the vertices of G are labeled.

This algorithm, as presented, favors the lower numbered vertices over those with higher numbered labels. Should the user have any special knowledge of the given graph, G, this can be taken into account as the vertices are labeled. With care, a potential, or desirable, solution path can be marked by the user by simply labeling those vertices the user would like to see included in the final tree with lower vertex numbers than the rest of the vertices in the graph.

Regardless of the numbering scheme used, the Steiner vertices, S, must be numbered before the non-Steiner vertices, NS. This is required so that the Steiner vertex subset can be easily identified and to separate them from the non-Steiner vertex subset. The vertex numbers finally assigned to these two vertex subsets must be done such that the two vertex subsets remain disjoint, otherwise this algorithm will not function properly. The order in which the Steiner vertices are numbered is not important although imposing some sort of criteria (clockwise, for example) is useful in organizing the vertices for later reference.

Unlike the Steiner vertices S, the order in which the non-Steiner vertices, NS, is numbered is important and will affect the size of the solution tree R produced by this algorithm. A regular and consistent numbering scheme is recommended, such as from the left to the right and from the bottom to the top of the graph G. Any special knowledge of the graph or other special preferred solution criteria is incorporated into the vertex numbering scheme at this time. A chaotic or random non-Steiner vertex numbering system should be avoided since it will cause the algorithm to output a

decidedly non-optimal solution which, in the worst case, can contain of all of the vertices and most of the edges in *G*. The numbering scheme chosen for the graph that will be used to illustrate this algorithm can be observed in Figure 2.1.

The data required to initialize the various data structures is also collected during this step. The total number of vertices in the graph *G*, |V|, and the number of Steiner vertices, |S|, are required for correct algorithm behavior and data structure creation and initialization. These values and others will be determined during the data entry process when the graph data is entered into the algorithm during the initialization step.

2.7.2 Step 1 - Initialization

Initialization of the *SNL* and the *NSNL* data structures is accomplished during the data entry process. Using the vertex numbering scheme imposed during the graph preprocessing step, each vertex is taken in numerical order. The ordered edge list for each vertex is also created in numerical order. For example, if vertex 3 has three edges leading to vertices 1, 5, and 6 respectively, then the edges will be entered in the following order 3-1, 3-5, and 3-6.

The vertex status information in the *NSNL* is also initialized during the data entry process. The values regarding the number of Steiner vertices and total number of vertices sharing an edge with the *NSNL* vertex being initialized are tabulated. The other status values are initialized to zero. No vertex status information is kept for vertices in the *SNL*.

After the *SNL* and *NSNL* arrays have been initialized, the ordered pick list *PL* can be created. Initially, the pick list contains an entry for each non-Steiner vertex *W* in the *NSNL* that has edges leading to two or more Steiner vertices in *S*. Each vertex *W* in the pick list is organized according to the number of edges from *W* that lead to Steiner

vertices and is further ordered by vertex number of *W* if necessary. Later in the algorithm the contents of the pick list will be replaced to reflect the changed nature of vertex selection as the solution tree, *R*, is generated. The remaining data structures, the *SL* and the *SCT* are initialized to null or zero where appropriate.

The Steiner Node List, *NSL*, generated by this step is shown in Table 2.1. The Non-Steiner Node List, *NSNL*, is shown in Table 2.2. The Pick List, *PL*, for the first portion of the algorithm that was generated during the initialization step is shown in Table 2.3. The Subtree List, *SL*, is shown in Table 2.4 and the Select/Connect Table, *SCT*, in Table 2.5.

2.7.3 Step 2 - Steiner Vertex Selection

The objective of this portion of the algorithm is to place all of the Steiner vertices S_i into one or more subtrees P_k and record these assignments in the subtree list *SL*, the select/connect table *SCT*, and the non-Steiner node list *NSNL*. It is very important that the appropriate data structures be correctly updated when a vertex is assigned to a subtree P_k. Any errors made in updating the various data structures and their components will cause this algorithm to generate a decidedly poor solution tree *R* since the vertices and edges added to it in a later step will be selected using erroneous criteria.

The placement of the Steiner vertices *S* is accomplished in two steps, if necessary. First, using the pick list *PL*, the non-Steiner vertices NS_i that possess multiple edges leading to Steiner vertices are selected first. For each such NS_i selected, a subtree P_k is generated using NS_i as the root and the Steiner vertices as leaves. The subtree P_k is stored in the *SL* and the appropriate entries in the *NSNL* and the *SCT* are then updated to

reflect the new solution situation. The selection process continues until the pick list is empty or until all of the Steiner vertices have been chosen.

It is possible to choose a vertex NS_i from the pick list that has edges leading to Steiner vertices that have already been included in a subtree P_k in the *SL*. If at least one edge leads to a Steiner vertex S_j that is not a member of an existing subtree then a new subtree P_k is generated. This subtree will have NS_i as the root and all S_j that exist and only one previously selected Steiner vertex as leaves. Otherwise, such vertices are rejected since they provide no advantage or new Steiner vertex connections to the solution process and are therefore redundant.

Second, using the same pick list, any Steiner vertices that have not been placed into a subtree in the subtree list *SL* are then added to the *SL* together with a single non-Steiner vertex and connecting edge. In this case, the pick list *PL* has been emptied of non-Steiner vertices with multiple edges leading to different Steiner vertices before all of the Steiner vertices were been placed into a subtree. This is a less desirable situation than the one described previously since it indicates that at least some of the Steiner vertices in the graph *G* are far apart (more that a single edge or one non-Steiner vertex and two edges distance) from each other.

For each missing Steiner vertex S_j, a new subtree P_k is created by taking from the *PL* the first non-Steiner vertex NS_i that has an edge leading to an unselected Steiner vertex S_j. Each entry into the *SL* will be the subtree P_k, whose root vertex is NS_i with one leaf vertex, S_j. Again, the appropriate entries in the *NSNL* and the *SCT* are then updated to reflect the new solution situation.

The following list describes the substeps, in order, that must be followed to correctly complete this portion of the algorithm.

1) If the pick list *PL* is empty then proceed to (8).
2) From the *PL*, remove the first vertex and make it NS_i.
3) Using the *SCT*, check the *select* status of every Steiner vertex S_j with an edge leading to NS_i.
    a) If any S_j is not marked *selected* then proceed to (4).
    b) If all S_j are marked *selected* then discard NS_i and proceed to (1).
4) Place the vertices NS_i and S_j into a new subtree P_k in the subtree list *SL* as follows: the root vertex NS_i followed by each S_j in numerical vertex order.
    a) If any of the vertices S_j are marked *selected* in the *SCT* then only the first such vertex S_j can be included in P_k. All other S_j that are marked *selected* are to be ignored and *not* included in P_k.
5) In the *SCT*, for each vertex NS_i and S_j in P_k:
    a) Increment the subtree membership count.
    b) Add *k* to the list of subtrees containing the vertex.
    c) Mark the vertex as *selected* if it is not already so marked.
6) In the *NSNL*, for NS_i,
    a) For each S_j added to P_k:
        i) Decrement the Steiner vertex count.
        ii) Delete S_j from the edge-list.
        iii) Decrement the total vertex count.
        iv) Increment the total *selected* count.

b) In the *SNL*, for each S_j added to P_k delete NS_i from the edge-list of S_j.

c) Using the *SNL*, for each NS_l in the edge-list of every S_j in the new P_k, where $NS_l \neq NS_i$, increment the total *selected* count.

d) For each remaining vertex NS_l in the edge-list of NS_i, where NS_l \not\in S_j, increment the total *selected* count.

7) Until all of the Steiner vertices S_j in the graph *G* are marked *selected* in the *SCT* proceed to (1).

8) If all of the Steiner vertices S_j have been marked *selected* in the *SCT* then Step 2 is finished. Proceed to Step 3, Section 2.7.4.

Table 2.6 shows the resulting data tables after completion of the first Steiner vertex selection step. Table 2.7 shows the resulting data tables after completion of the second Steiner vertex selection step. Table 2.8 shows the resulting tables after completion of the third and final Steiner vertex selection step.

With the completion of this step every Steiner vertex S_j is now be a member of some subtree, P_k, in the forest shown in the *SL*. This placement is reflected in the appropriate entries in the Non-Steiner Node List *NSNL* and the Select/Connect Table *SCT*. Next, the forest must be examined to determine if the solution tree *R* has been generated and, if not, then to determine how much of *R* was generated during this step and what must be done to complete the solution tree.

Table 2.6 The data structures after the first Steiner Node Selection step.

S_i	Edge List
1	□
2	9 □
3	7 □
4	5 □

#S	#V	#sel	#con	NS_i	Edge List
1	3	0	0	5	4 6 7 □
0	2	0	0	6	5 9 □
1	3	0	0	7	3 5 9 □
0	0	2	0	8	□
1	3	1	0	9	2 6 7 □

PL	5 7 9 □

P_k	Vertex (edge) List
0	8 1 2 □
1	□

vertex #	1	2	3	4	5	6	7	8	9
selected	✓	✓						✓	
connected									
# of subtrees	1	1	0	0	0	0	0	1	0
subtree list	0	0	-	-	-	-	-	0	-

Table 2.7 The data structures after the second Steiner Node Selection step.

S_i	Edge List
1	□
2	9 □
3	7 □
4	□

#S	#V	#sel	#con	NS_i	Edge List
0	2	1	0	5	6 7 □
0	2	1	0	6	5 9 □
1	3	1	0	7	3 5 9 □
0	0	2	0	8	□
1	3	1	0	9	2 6 7 □

PL	7 9 □

Pk	Vertex (edge) List
0	8 1 2 □
1	5 4 □

vertex #	1	2	3	4	5	6	7	8	9
selected	✓	✓		✓	✓			✓	
connected									
# of subtrees	1	1	0	1	1	0	0	1	0
subtree list	0	0	-	1	1	-	-	0	-

Table 2.8 The data structures after the third Steiner Node Selection step.

S_i	Edge List
1	□
2	9 □
3	□
4	□

#S	#V	#sel	#con	NS_i	Edge List
0	2	2	0	5	6 7 □
0	2	1	0	6	5 9 □
0	2	2	0	7	5 9 □
0	0	2	0	8	□
1	3	2	0	9	2 6 7 □

PL	9 □

Pk	Vertex (edge) List
0	8 1 2 □
1	5 4 □
2	7 3 □

vertex #	1	2	3	4	5	6	7	8	9
selected	✓	✓	✓	✓	✓		✓	✓	
connected									
# of subtrees	1	1	1	1	1	0	1	1	0
subtree list	0	0	2	1	1	-	2	0	-

2.7.4 Step 3 - Steiner Vertex Connection Evaluation

The objective of this step in the algorithm is to evaluate the forest generated previously. It must be determined if the subtrees as they exist in the subtree list *SL* constitute the solution tree *R* or if further work is required to generate *R*. Four situations are possible at this point in the algorithm. They are:

1. A single subtree, P_0, exists in the *SL* that contains all of the vertices in *S*.
2. Each Steiner vertex S_j is a member of only one subtree P_k in the *SL* (all of the subtrees are disjoint).
3. Some of the Steiner vertices S_j are members of more than one subtree P_k in the *SL* and that none of the subtrees are disjoint.
4. Some of the Steiner vertices S_j are members of more than one subtree P_k in the *SL* and that at least one of the subtrees is disjoint.

The first case, where the *SL* contains a single subtree P_0, is the ideal result. Here, a single non-Steiner vertex connects all of the Steiner vertices in the graph G. The subtree P_0 is the solution tree R for graph G. We can proceed directly to solution tree output step in Section 2.7.8.

The second case, where each Steiner vertex in G exists in only one subtree P_k in the *SL* and all of the subtrees are disjoint, is the worst case scenario for this algorithm. At best, it will require the addition of one or more single edges to connect the subtrees P_k in the *SL*. At worst, it will require the addition of more subtrees to the *SL* in order to connect the forest generated in the previous step.

This situation can be easily detected by examining the Steiner vertex entries in the *SCT* for membership in multiple subtrees in the *SL*. When the subtrees are disjoint, the

total subtree membership count for each Steiner vertex in the *SCT* will be one (1). The traversal of any subtree in the *SL* will yield no benefit or new information at this time. However, a starting point is needed for the remaining steps of this algorithm. Therefore, traverse the subtree P_0 in the *SL* using the method described in Section 2.7.5 and then proceed to the single edge connection step, Section 2.7.6.

The third and fourth cases involve subtrees in the subtree list *SL* that have Steiner vertices in common. The third case, like the first, is an ideal result since the solution tree *R* has been generated during the Steiner vertex selection step. However, it is not obvious that *R* has been found since multiple non-Steiner vertices have been included in the subtrees found in the *SL*.

The fourth case lies between the third and second in desirability. Here, at least two of the subtrees in the *SL* are not disjoint. It is not known which subtrees are disjoint and their number. However, once the subtrees that share a common vertex are found, and marked *connected* in the *SCT*, a good starting point for further processing will have been identified.

For the third and fourth cases, the existence of common Steiner vertices in the subtrees of the forest can be determined by examining the Steiner vertex data entries in the *SCT*. Any Steiner vertex that is a member of more than one subtree will have a value greater than one (1) in the total subtree membership count variable in the *SCT*. In both cases it will be necessary to traverse the subtrees P_k in the *SL* to determine their actual connection status and whether further processing is required to generate *R*.

2.7.5 Step 4 - Subtree Traversal

This step in the algorithm travels through the forest in the subtree list *SL* and, in essence, determines the connection status of the subtrees therein. As it proceeds through the *SL*, every vertex in each subtree P_k that is encountered is marked *connected* and the appropriate data in the *NSNL* and *SCT* is updated. Vertices that are marked *connected* are known to be part of the solution tree *R* and serve as starting points for further processing if it is determined that not all of the Steiner vertices are present in *R*. Once all of the Steiner vertices are so marked in the *SCT* further processing can be halted since the solution tree *R* will have been completed.

The traversal process can start with any subtree P_k in the *SL* and will do so depending on when this step is executed since this particular step in the algorithm can be called out of sequence when, and as, needed. Although the subtree that the traversal starts with is relatively unimportant, it is recommended that when this step is executed for the first time the starting subtree be the one with the Steiner vertex that appears in the most subtrees or with the lowest numbered Steiner vertex that appears in multiple subtrees. This will allow for the connection of the largest possible number of vertices early in the solution process and generate the best possible starting point for finishing the creation of the solution tree *R*.

To traverse the subtrees P_k in the *SL*, do the following:

1) Select a subtree P_x in the *SL* from which to start the traversal if P_x has not been previously specified.
    a) Create a temporary traversal list *TL* and place the subtree P_x in the *TL*.
2) For each vertex *W* in P_x, if *W* is not marked *connected* in the *SCT*:

a) Mark *W* as *connected* in the *SCT*.

b) If $W \in SNL$ then for every vertex NS_i in its edge-list, increment the total *connected* count variable.

c) If $W \in NSNL$ then for every vertex NS_i in its edge-list, increment the total *connected* count variable.

d) If *W* is a member of more than one subtree then add the subtree numbers *z*, where $z \neq x$, to the temporary traversal list *TL*.

i) If *z* currently exists in the *TL* or has been an earlier member of the list already traversed then do not add *z* to the *TL*.

3) While the *TL* is not empty, select a new P_x from the *TL* and repeat substep 2 above.

4) When the *TL* is empty examine the *connect* status of every Steiner vertex in the *SCT*. If all of the Steiner vertices are marked *connected* then the solution tree *R* has been generated. In this event, proceed to Step 7, Section 2.7.8, to complete the algorithm. Otherwise, proceed to Step 5, Section 2.7.6, for further processing.

By traversing this first subtree and all of the other subtrees subsequently encountered, the extent of the connection between the subtrees in the *SL* can be determined. If any of the subtrees in the *SL* are disjoint then at least one of the Steiner vertices in the *SCT* will not be marked *connected*. Should all of the Steiner vertices in the *SL* be marked *connected* then a solution tree *R* has been found. Only those vertices in the *SCT* that have been marked *selected* and *connected* and the subtrees P_k in the *SL* that contain these vertices make up the solution tree R.

If Steiner vertices remain in the *SCT* that are not marked as *connected* then at least one of the subtrees in the *SL* is disjoint. In this event a partial solution has been found and marked *connected* in the *SCT*. It is from this *connected* group of subtrees in the *SL* that the solution tree R will be built.

2.7.6 Step 5 - Single Edge Connection

When disjoint subtrees are found in the *SL* two possibilities exist for connecting such subtrees. First, the vertices making up the subtrees can be in close proximity to each other in the graph *G*. A single edge may be all that is needed to join such subtrees. Second, the vertices may be far apart in the graph *G* in which case one or more non-Steiner vertices and multiple edges will have to be added to join the disjoint subtrees.

In this step we search for those subtrees in the *SL* that can be joined by the addition of a single edge. This is accomplished by examining the edge-list of every non-Steiner vertex NS_i that is marked *selected* in the *SCT*. Should a member of this edge-list be found that is marked *selected* in the *SCT* then a single edge connecting two subtrees has been located. This single edge is added to the subtree list *SL* only when both vertices are not marked *connected* in the *SCT* and both vertices are not members of the same subtree. The following substeps will accomplish this task.

For every vertex NS_i in the *SCT* that is marked *selected* and every vertex NS_j in the edge-list of NS_i:

1) If NS_j is marked *selected* in the *SCT* then proceed to (2). Otherwise, discard NS_j and repeat this substep with another NS_j.
2) If NS_i and NS_j are both marked *connected* in the *SCT* then discard NS_j and start again at (1) with a new NS_j. Otherwise, proceed to (3).

3) Add NS_i and NS_j to the *SL* as a new subtree P_k with NS_i as the root vertex and NS_j as the only leaf.

4) In the *SCT*, for NS_i and NS_j:

   a) Increment the total subtree count

   b) Add *k* to the subtree list.

5) In the *NSNL*:

   a) For NS_i and NS_j, decrement the total vertex count.

   b) Delete NS_i from the edge-list of NS_j.

   c) Delete NS_j from the edge-list of NS_i.

6) Starting with the new subtree, P_k, use the traversal method detailed in Step 4, Section 2.7.5, to traverse P_k and all subsequent subtrees thereby marking their vertices *connected* in the *SCT*.

7) If *R* has been generated then proceed to Step 7, Section 2.7.8. Otherwise, continue the single edge connection step until all NS_i have been tested.

2.7.7 Step 6 - Non-Steiner Vertex Selection

This is the final vertex selection step of the algorithm. In this step we attempt to connect the disjoint subtrees in the *SL* by adding new subtrees to the *SL* that have at least one edge leading to an existing subtree whose members are marked *connected* in the *SCT*. This step is totally dependent upon the accuracy of the status information recorded during the earlier steps of the algorithm. Using this data, we will attempt to select those non-Steiner vertices that have the greatest value in connecting the disjoint subtrees and also provide the shortest path possible between those subtrees.

Table 2.9 The data structures after the Single Edge Connection step and a traversal of the subtrees.

S_i	Edge List
i	□
2	9 □
3	□
4	□

#S	#V	#sel	#con	NS_i	Edge List
0	1	2	0	5	6 □
0	2	1	1	6	5 9 □
0	1	2	0	7	9 □
0	0	2	0	8	□
1	3	2	1	9	2 6 7 □

Pk	Vertex (edge) List
0	8 1 2 □
1	5 4 □
2	7 3 □
3	5 7 □

vertex #	1	2	3	4	5	6	7	8	9
selected	✓	✓	✓	✓	✓		✓	✓	
connected			✓	✓	✓		✓		
# of subtrees	1	1	1	1	2	0	2	1	0
subtree list	0	0	2	1	1,3	-	2,3	0	-

Before beginning the final selection process, a new pick list of non-Steiner vertices from the *NSNL* must be generated. The pick list *PL* for this portion of the algorithm is created in a slightly different manner than before. The new pick list *PL* is generated using the following rules:

1) For every NS_i in the *NSNL* calculate (*total selected - total connected*) and store the result in a temporary variable t_i.

2) For every NS_i in the *NSNL*, add NS_i to the *PL* if:

   a) t_i is not zero or negative (if t_i is negative then the data recorded in the *NSNL* for that vertex is incorrect and the algorithm should be aborted and the partial solution discarded).

   b) NS_i is not marked *selected* in the *SCT*.

   c) The value of the *total connected* variable for NS_i is greater than zero.

   and order each entry into the *PL* such that:

   d) Those NS_i with the highest t_i values are in the front of the list.

   e) For those NS_i that have the same t_i value, order them by their vertex number, lowest to highest.

3) If the pick list *PL* is empty at this point then generate an entry for the *PL* by placing in the *PL* the lowest numbered NS_i whose *total connected* variable value is greater than zero and:

   a) NS_i is not marked *selected* in the *SCT*.

   b) NS_i does not have an edge leading to a Steiner vertex.

As before, vertices are chosen from the front of the pick list *PL*. For each vertex *W* selected from the *PL* a new subtree, P_k, is created in the *SL* with *W* as the root vertex. The new subtree P_k will consist of at least two vertices, including *W*, and one edge, depending upon the type of subtree that can be generated.

Each subtree P_k generated during this step in the algorithm is one of two valid types. The first valid type of subtree directly connects two or more disjoint subtrees P_i in the *SL*. Here, each leaf of P_k is a member of one of the subtrees P_i. Multiple vertices from a single subtree P_i are not allowed.

The second valid type of subtree has only one edge leading to an existing subtree in the *SL*. This subtree serves as a 'bridge-builder' to eventually connect two or more of the disjoint subtrees in the *SL*. It is the generation of this type of subtree that leads to less than optimal solutions. Poor selection criteria when generating these subtrees can result in adding many more non-Steiner vertices to the solution tree *R* than would otherwise occur.

The following steps will generate the two valid types of subtrees mentioned above.

1) If the pick list *PL* is empty then an error has occurred.
2) From the *PL*, remove the first vertex and make it NS_i.
3) Using the edge-list of NS_i in the *NSNL*:
    a) Locate the first vertex marked *connected* in the *SCT*.
    b) Make this vertex W_a and store its subtree number(s) in q_0. Note: q_0 should be able to hold more than one subtree number since any vertex can be a member of more than one subtree in the *SL*.

4) Place the vertices NS_i and W_a into a new subtree P_k in the subtree list *SL* as follows: the root vertex NS_i and W_a as a leaf.

5) Using the edge-list of NS_i in the *NSNL*, for every vertex W_j, $j \neq a$, add W_j to P_k if:

   a) W_j is marked *selected* but not *connected* in the *SCT* AND:

   b) W_j is not a member of any subtree in q_0.

6) If no W_j, $j \neq a$, was added to the subtree P_k in the previous step then proceed to (15).

7) In the *SCT*:

   a) For NS_i:

      i) Increment the subtree membership count.

      ii) Add *k* to the list of subtrees containing the vertex.

      iii) Mark the vertex as *selected* if it is not already so marked.

   b) For all W_j, j = a included, added to P_k:

      i) Increment the subtree membership count.

      ii) Add *k* to the list of subtrees containing the vertex.

8) For all W_j, j = a included, added to P_k:

   a) If W_j \in *SNL* then:

      i) Delete NS_i from the edge-list of W_j.

      ii) In the *NSNL* for NS_i:

         - Decrement the Steiner vertex count.
         - Decrement the total vertex count.
         - Delete W_j from the edge-list.

b) If W_j \in *NSNL* then:

i) Decrement the total vertex count of NS_i.

ii) Delete W_j from the edge-list of NS_i.

iii) Increment the total selected count of W_j.

iv) Decrement the total vertices count of W_j.

v) Delete NS_i from the edge-list of W_j.

9) For each remaining W_l in the edge-list of NS_i, increment the total selected count.

10) Traverse the subtree P_k using the method described in Step 4, Section 2.7.5.

11) If *R* has been generated then proceed to Step 7, Section 2.7.8.

12) Perform Step 5 - Single Edge Connection, Section 2.7.6.

13) Regenerate the pick list *PL* using the method described earlier.

14) Proceed to substep (1).

15) (The new subtree P_k consists of only two vertices, NS_i and W_a, and the edge that connects them. No traversal is required or needed in this case since two disjoint subtrees are not being joined.) The second valid subtree type has been generated.

a) In the *NSNL*:

i) For W_a:

- Decrement the total vertex count.
- Delete NS_i from the edge-list.
- Increment the total selected count.

ii) For NS_i:

- Decrement the total vertices count.
- Delete W_a from the edge-list.
- For each remaining W_l in the edge-list, increment the total selected count and the total connected count.

b) In the *SCT*:

i) For W_a:

- Increment the subtree membership count.
- Add *k* to the list of subtrees containing the vertex.

ii) For NS_i:

- Increment the subtree membership count.
- Add *k* to the list of subtrees containing the vertex.
- Mark the vertex as *selected.*
- Mark the vertex as *connected.*

16) Regenerate the pick list *PL* using the method described earlier.

17) Proceed to substep (1)

The data describing the solution tree *R* can be found in the *SCT* and the *SL*. The solution tree will be composed of those vertices in the *SCT* that are marked *connected.* The edges that connect these vertices will be listed in those subtrees in the subtree list *SL* that contain the aforementioned vertices. To generate the actual solution tree *R*, simply traverse the *SCT* and the *SL* and list the tree components as they are encountered.

Table 2.10 The data structures after the Non-Steiner Vertex Selection step and a traversal of the subtrees.

S_i	Edge List
1	□
2	9 □
3	□
4	□

#S	#V	#sel	#con	NS_i	Edge List
0	1	2	0	5	6 □
0	2	1	2	6	5 9 □
0	0	3	0	7	□
0	0	2	0	8	□
0	1	2	1	9	6 □

P_k	Vertex (edge) List
0	8 1 2 □
1	5 4 □
2	7 3 □
3	5 7 □
4	9 7 2 □

vertex #	1	2	3	4	5	6	7	8	9
selected	✓	✓	✓	✓	✓		✓	✓	✓
connected	✓	✓	✓	✓	✓		✓	✓	✓
# of subtrees	1	2	1	1	2	0	3	1	1
subtree list	0	0,4	2	1	1,3	-	2,3,4	0	4

2.7.8 Step 7 - Solution Tree Output

Before the components of *R* can be output, the solution must be examined for non-Steiner vertex leaves. Any such leaf that is found in the solution tree *R* can be discarded. These vertices can be discarded without damage to the solution tree *R* because the only vertex that can be a valid leaf in a tree created with this algorithm is a Steiner vertex.

This is due to the nature of the problem being solved, namely, that there be no edges connecting Steiner vertices to each other on the graph *G*. This requires that each pair of Steiner vertices be connected in *R* by at least one non-Steiner vertex, making such non-Steiner vertices the 'root' vertex of that particular subtree P_k when it is generated by this algorithm. These 'root' vertices are themselves connected to form the solution tree *R* through the addition of single edges and\or subtrees consisting of one new non-Steiner vertex and two edges leading to non-Steiner vertices previously *selected* as members of other subtrees in the *SL*.

Due to this process it can be seen that every Steiner vertex in the solution tree *R* will be a leaf. It can also be seen that every non-Steiner vertex in *R* will have at least two neighbors in the tree when these vertices actually serve to connect the Steiner vertices present. Therefore, any non-Steiner vertex in *R* that is a leaf does not connect any two subtrees in the *SL* nor does it connect a Steiner vertex to *R*. Thus, such vertices can be eliminated from *R* safely.

To check for, and delete, any non-Steiner vertex leaves in the solution tree *R*, perform the following substeps:

1) For every non-Steiner vertex NS_i marked *connected* in the *SCT*:

   a) If NS_i is a member of two or more subtrees then NS_i cannot be a leaf in the solution tree *R*.

   b) If NS_i is a member of only one subtree P_x then:

      i) If P_x is composed of more than two vertices and one edge then NS_i is not a leaf in *R*.

      ii) If P_x is composed of two vertices and one edge only then NS_i is a leaf L_f and can be listed for deletion from *R*.

   c) If any leaves L_f were found in the previous substep then delete them from the solution tree *R* as follows:

      i) In the *SCT*, for L_f:

         - From the subtree list, let P_x represent the subtree containing the leaf L_f to be deleted and let W_{lf} represent the second vertex in P_x.
         - Unmark the vertex as *selected*.
         - Unmark the vertex as *connected*.
         - Set the *number of subtrees* count to *zero*.
         - Delete the subtree list.

      ii) In the *SCT*, for W_{lf}:

         - Decrement the *number of subtrees* count.
         - Delete the subtree number *x* (representing the subtree P_x) from the subtree list.

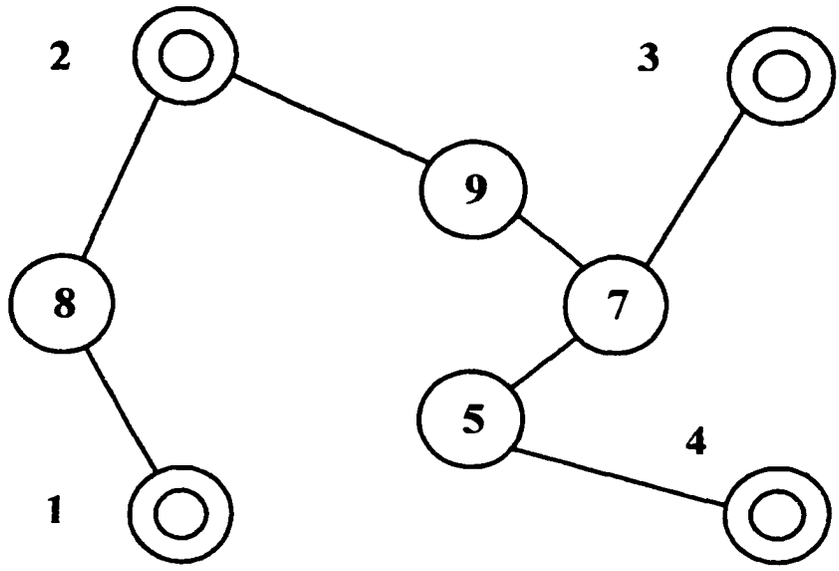
      iii) In the *SL*, delete P_x.

To generate the Solution Tree *R*, do the following:

1) Using the *SCT*:
    a) Every vertex marked *connected* is a member of *R*.
    b) For every vertex marked *connected*, add the numbers in its subtree list to a temporary subtree list *TSL* if they are not already members of *TSL*.
2) While the temporary subtree list *TSL* is not empty, generate the list of edges in *R* as follows:
    a) Get a subtree number *k* (for P_k) from the *TSL*.
    b) From P_k in the *SL*, designate the first vertex as *root* and the remaining vertices as W_j.
    c) For every W_j in P_k, generate the vertex pairs representing the edges in *R* as (*root*, W_j).

2.8 Performance Analysis

This section is under complete review. The original analysis showed this algorithm to be O(E) in the worst case. However, this algorithm has been modified since the original version was developed. With the corrections and alterations made to the original algorithm taken into account, it appears that the Steiner Tree Algorithm as presented in this work is O(VE) in the worst (and best) case.



Vertices	Edges
1,2,3,4,5,7,8,9	(8,1) (8,2) (5,4) (7,3) (5,7) (9,7) (9,2)

Figure 2.2 The solution tree R generated by the algorithm.

2.9 Future Work

Further testing is required to determine when an optimal solution can be generated as well as under what conditions and to determine the types of sub optimal solutions that can be generated including bounds on how far from optimal such solutions can be. Additional future work involves developing better methods of choosing the non-Steiner

vertex to accompany the single Steiner vertex chosen during the first step and the choosing the non-Steiner vertex and edges in the last step of the algorithm. Research to reduce the bookkeeping effort and overhead is also being considered for the future.

2.10 Conclusion

We have presented and analyzed a relatively simple bookkeeping algorithm for determining the shortest tree that connects the Steiner vertices in a given graph where the edges are of unit length. From the testing done, it has been found that this algorithm will often yield an near-optimal solution when care is taken in the labeling of the vertices. Variation from optimality has been found to occur primarily from poor vertex numbering schemes and from poor vertex selections made when adding individual Steiner vertices to the *SL* during the Steiner vertex selection step.

APPENDIX A

A SECOND WORKED EXAMPLE OF THE STEINER ALGORITHM

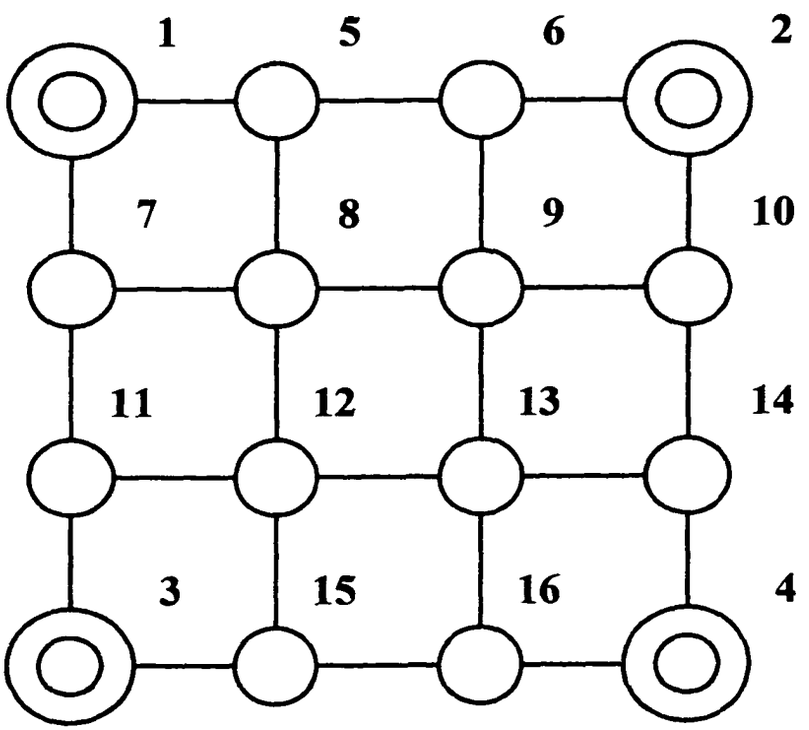


Figure A.1 The graph G for the appendix A example further illustrating the operation of the Steiner algorithm.

Table A.1 The Steiner Node List (SNL), the Non-Steiner Node List (NSNL), the Pick List (PL), and the Subtree List (SL) after the first Steiner Node selection step has been completed.

S_i	Edge List
1	7 □
2	6 10 □
3	11 15 □
4	14 16 □

#S	#V	#sel	#con	NS_i	Edge List
0	2	1	0	5	6 8 □
1	3	1	0	6	2 5 9 □
1	3	1	0	7	1 8 11 □
0	4	1	0	8	5 7 9 12 □
0	4	0	0	9	6 8 10 13 □
1	3	0	0	10	2 9 14 □
1	3	0	0	11	3 7 12 □
0	4	0	0	12	8 11 13 15 □
0	4	0	0	13	9 12 14 16 □
1	3	0	0	14	4 10 13 □
1	3	0	0	15	3 12 16 □
1	3	0	0	16	4 13 15 □

PL	6	7	10	11	14	15	16	□

P_k	Vertex (edge) List
0	5 1 □
1	□

Table A.2 The Select/Connect Table (SCT) after the first Steiner Node selection step has been completed.

vertex #	1	2	3	4	5	6	7	8	9	10
selected	✓				✓					
connected										
# of subtrees	1	0	0	0	1	0	0	0	0	0
subtree list	0	0	-	-	0	-	-	-	-	-

vertex #	1	2	3	4	11	12	13	14	15	16
selected	✓									
connected										
# of subtrees	1	0	0	0	0	0	0	0	0	0
subtree list	0	0	-	-	-	-	-	-	-	-

Table A.3 The data structures (SNL, NSNL, PL, and SL) after the second Steiner Node selection step has been completed.

S_i	Edge List
1	7 □
2	10 □
3	11 15 □
4	14 16 □

#S	#V	#sel	#con	NS_i	Edge List
0	2	2	0	5	6 8 □
0	2	2	0	6	5 9 □
1	3	1	0	7	1 8 11 □
0	4	1	0	8	5 7 9 12 □
0	4	1	0	9	6 8 10 13 □
1	3	1	0	10	2 9 14 □
1	3	0	0	11	3 7 12 □
0	4	0	0	12	8 11 13 15 □
0	4	0	0	13	9 12 14 16 □
1	3	0	0	14	4 10 13 □
1	3	0	0	15	3 12 16 □
1	3	0	0	16	4 13 15 □

PL	7 10 11 14 15 16 □

P_k	Vertex (edge) List
0	5 1 □
1	6 2 □

Table A.4 The Select/Connect Table (SCT) after the second Steiner Node selection step has been completed.

vertex #	1	2	3	4	5	6	7	8	9	10
selected	✓	✓			✓	✓				
connected										
# of subtrees	1	1	0	0	1	1	0	0	0	0
subtree list	0	1	-	-	0	1	-	-	-	-

vertex #	1	2	3	4	11	12	13	14	15	16
selected	✓	✓								
connected										
# of subtrees	1	1	0	0	0	0	0	0	0	0
subtree list	0	1	-	-	-	-	-	-	-	-

Table A.5 The data structures (SNL, NSNL, PL, and SL) after the third Steiner Node selection step has been completed.

S_i	Edge List
1	7 □
2	10 □
3	15 □
4	14 16 □

#S	#V	#sel	#con	NS_i	Edge List
0	2	2	0	5	6 8 □
0	2	2	0	6	5 9 □
1	3	2	0	7	1 8 11 □
0	4	1	0	8	5 7 9 12 □
0	4	1	0	9	6 8 10 13 □
1	3	1	0	10	2 9 14 □
0	2	1	0	11	7 12 □
0	4	1	0	12	8 11 13 15 □
0	4	0	0	13	9 12 14 16 □
1	3	0	0	14	4 10 13 □
1	3	1	0	15	3 12 16 □
1	3	0	0	16	4 13 15 □

PL	14 15 16 □

P_k	Vertex (edge) List
0	5 1 □
1	6 2 □
2	11 3 □

Table A.6 The Select/Connect Table (SCT) after the third Steiner Node selection step has been completed.

vertex #	1	2	3	4	5	6	7	8	9	10
selected	✓	✓	✓		✓	✓				
connected										
# of subtrees	1	1	1	0	1	1	0	0	0	0
subtree list	0	1	2	-	0	1	-	-	-	-

vertex #	1	2	3	4	11	12	13	14	15	16
selected	✓	✓	✓		✓					
connected										
# of subtrees	1	1	1	0	1	0	0	0	0	0
subtree list	0	1	2	-	2	-	-	-	-	-

Table A.7 The data structures (SNL, NSNL, PL, and SL) after the fourth Steiner Node selection step has been completed.

S_i	Edge List
1	7 □
2	10 □
3	15 □
4	16 □

#S	#V	#sel	#con	NS_i	Edge List
0	2	2	0	5	6 8 □
0	2	2	0	6	5 9 □
1	3	2	0	7	1 8 11 □
0	4	1	0	8	5 7 9 12 □
0	4	1	0	9	6 8 10 13 □
1	3	2	0	10	2 9 14 □
0	2	1	0	11	7 12 □
0	4	1	0	12	8 11 13 15 □
0	4	1	0	13	9 12 14 16 □
0	2	1	0	14	10 13 □
1	3	1	0	15	3 12 16 □
1	3	1	0	16	4 13 15 □

PL	15 16 □

P_k	Vertex (edge) List
0	5 1 □
1	6 2 □
2	11 3 □
3	14 4 □

Table A.8 The Select/Connect Table (SCT) after the fourth Steiner Node selection step has been completed.

vertex #	1	2	3	4	5	6	7	8	9	10
selected	✓	✓	✓	✓	✓	✓				
connected										
# of subtrees	1	1	1	1	1	1	0	0	0	0
subtree list	0	1	2	3	0	1	-	-	-	-

vertex #	1	2	3	4	11	12	13	14	15	16
selected					✓			✓		
connected										
# of subtrees					1	0	0	1	0	0
subtree list					2	-	-	3	-	-

Table A.9 The data structures (SNL, NSNL, PL, and SL) after the single edge connection step has been completed.

S_i	Edge List
1	7 □
2	10 □
3	15 □
4	16 □

#S	#V	#sel	#con	NS_i	Edge List
0	1	2	0	5	8 □
0	1	2	0	6	9 □
1	3	2	1	7	1 8 11 □
0	4	1	1	8	5 7 9 12 □
0	4	1	1	9	6 8 10 13 □
1	3	2	1	10	2 9 14 □
0	2	1	0	11	7 12 □
0	4	1	0	12	8 11 13 15 □
0	4	1	0	13	9 12 14 16 □
0	2	1	0	14	10 13 □
1	3	1	0	15	3 12 16 □
1	3	1	0	16	4 13 15 □

P_k	Vertex (edge) List
0	5 1 □
1	6 2 □
2	11 3 □
3	14 4 □
4	5 6 □

Table A.10 The Select/Connect Table (SCT) after the single edge connection step has been completed.

vertex #	1	2	3	4	5	6	7	8	9	10
selected	✓	✓	✓	✓	✓	✓				
connected										
# of subtrees	1	1	1	1	2	2	0	0	0	0
subtree list	0	1	2	3	0,4	1,4	-	-	-	-

vertex #	1	2	3	4	11	12	13	14	15	16
selected					✓			✓		
connected										
# of subtrees					1	0	0	1	0	0
subtree list					2	-	-	3	-	-

Table A.11 The data structures (SNL, NSNL, PL, and SL) after the first non-Steiner vertex selection step has been completed.

S_i	Edge List
1	□
2	10 □
3	15 □
4	16 □

#S	#V	#sel	#con	NS_i	Edge List
0	1	2	0	5	8 □
0	1	2	0	6	9 □
0	1	2	1	7	8 □
0	4	2	2	8	5 7 9 12 □
0	4	1	1	9	6 8 10 13 □
1	3	2	1	10	2 9 14 □
0	1	2	0	11	12 □
0	4	1	0	12	8 11 13 15 □
0	4	1	0	13	9 12 14 16 □
0	2	1	0	14	10 13 □
1	3	1	0	15	3 12 16 □
1	3	1	0	16	4 13 15 □

PL	7 10 □

P_k	Vertex (edge) List	P_k	Vertex (edge) List
0	5 1 □	4	5 6 □
1	6 2 □	5	7 1 11 □
2	11 3 □		
3	14 4 □		

Table A.12 The Select/Connect Table (SCT) after the first non-Steiner vertex selection step has been completed.

Vertex #	1	2	3	4	5	6	7	8	9	10
selected	✓	✓	✓	✓	✓	✓	✓			
connected	✓	✓	✓		✓	✓	✓			
# of subtrees	2	1	1	1	2	2	1	0	0	0
subtree list	0,5	1	2	3	0.4	1,4	5	-	-	-

vertex #	1	2	3	4	11	12	13	14	15	16
selected					✓			✓		
connected					✓					
# of subtrees					2	0	0	1	0	0
subtree list					2,5	-	-	3	-	-

Table A.13 The data structures (SNL, NSNL, PL, and SL) after the second non-Steiner vertex selection step has been completed.

S_i	Edge List
1	□
2	□
3	15 □
4	16 □

#S	#V	#sel	#con	NS_i	Edge List
0	1	2	0	5	8 □
0	1	2	0	6	9 □
0	1	2	1	7	8 □
0	4	2	2	8	5 7 9 12 □
0	4	2	2	9	6 8 10 13 □
0	1	2	1	10	9 □
0	1	2	0	11	12 □
0	4	1	0	12	8 11 13 15 □
0	4	1	0	13	9 12 14 16 □
0	1	2	0	14	13 □
1	3	1	0	15	3 12 16 □
1	3	1	0	16	4 13 15 □

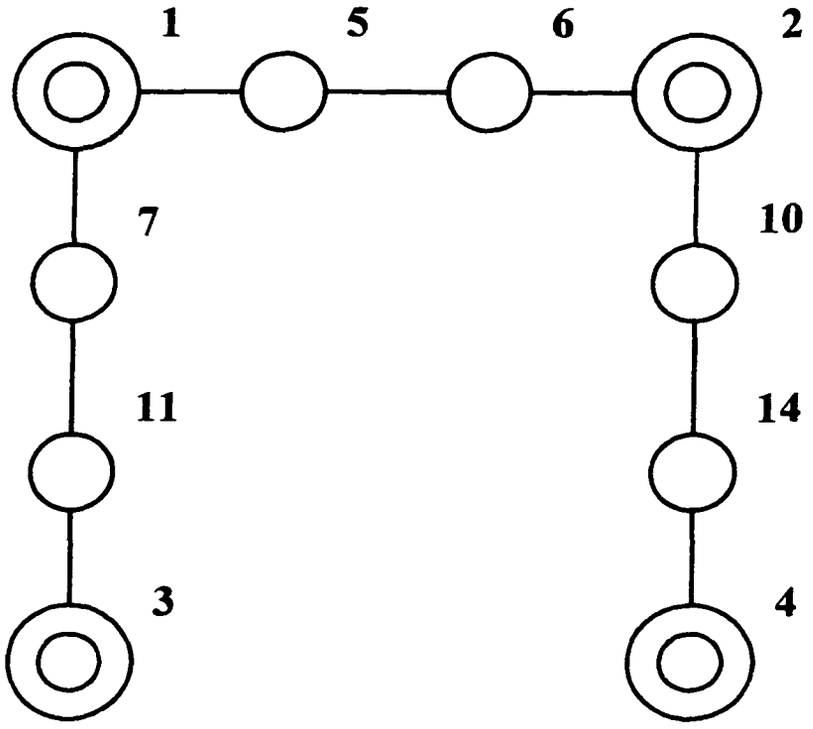
PL	10 □

P_k	Vertex (edge) List	P_k	Vertex (edge) List
0	5 1 □	4	5 6 □
1	6 2 □	5	7 1 11 □
2	11 3 □	6	10 2 14 □
3	14 4 □		

Table A.14 The Select/Connect Table (SCT) after the second non-Steiner vertex selection step has been completed.

vertex #	1	2	3	4	5	6	7	8	9	10
selected	✓	✓	✓	✓	✓	✓	✓			✓
connected	✓	✓	✓	✓	✓	✓	✓			✓
# of subtrees	2	2	1	1	2	2	1	0	0	1
subtree list	0,5	1,6	2	3	0,4	1,4	5	-	-	6

vertex #	1	2	3	4	11	12	13	14	15	16
selected					✓			✓		
connected					✓			✓		
# of subtrees					2	0	0	2	0	0
subtree list					2,5	-	-	3,6	-	-



Vertices	Edges
1,2,3,4, 5,6,7, 10,11,14	(7,1) (7,11) (11,3) (5,1) (5,6) (6,2) (10,2) (10,14) (14,4)

Figure A.2 The solution tree R generated by the algorithm.

BIBLIOGRAPHY

[1] S. G. Akl. An optimal algorithm for parallel selection. *Information Processing Letters*, 19:47-50, 1984.

[2] S. G. Akl. Parallel selection in O(log log n) time using O(n/log log n) processors. Technical Report 88-221, Queen's University, Department of Computing and Information Science, Kingston, Ontario, 1988.

[3] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer Science and System Sciences*, 7(4):448-461, 1973.

[4] P. Berman and V. Ramaiyer. Improved approximations for the Steiner tree problem. *Proceedings of the 3rd ACM/SIAM Symposium on the Discrete Algorithms, 1992.*

[5] R. J. Cole. An optimally efficient selection algorithm. *Information Processing Letters*, 26:295-299, 1988.

[6] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1, 1959.

[7] R. W. Floyd and R. L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18:165--172, 1975.

[8] M. R. Garey, R.L. Graham, and D.S. Johnson. Some NP-complete geometric problems. *Proceedings of the 8th Annual ACM Symposium on the Theory of Computing}, 1976.*

[9] R. Kou and K. Makki. An even faster approximation algorithm for the Steiner problem in graphs. *Congressus Numerantium*, 59, 1987.

[10] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for Steiner trees. *Acta Informatica*, 15, 1981.

[11] K. Makki. A more efficient approximation algorithm for the Steiner tree in graphs. Technical Report CSR90-055, University of Nevada, Las Vegas, 1990.

[12] K. Makki. A new approximation algorithm for the Steiner tree problem. *Congressus Numerantium*, 84, 1991.

[13] U. Manber. *Introduction to Algorithms, A Creative Approach.* Addison-Wesley Publishing Company Inc., 1989.

[14] K. Makki and N. Pissinou. The Steiner tree problem with minimum number of vertices in graphs. *Proceedings of the IEEE Second Great Lakes Symposium on VLSI}, February 1992.*

[15] E. M. Reingold, J. Nierergelt, and N. Deo. *Combinatorial Algorithms Theory and Practice.* Prentice-Hall Publishing Company Inc., 1977.

[16] V. J. Rayward-Smith. The computation of nearly minimal Steiner trees in graphs. *International Journal of Mathematics, Education, Science and Technology*, 14, 1983.

[17] V. J. Rayward-Smith and A. Clare. On finding Steiner vertices. *Networks*, 16, 1986.

[18] Q. F. Stout. Sorting, merging, selecting, and filtering on tree and pyramid machines. *Proceedings of the 1983 International Conference on Parallel Processing, IEEE Computer Society, 1983*, pages 214-221.

[19] H. Takahashi and A. Matsuyama. An approximation solution for the Steiner problem in graphs. *Math Japonica*, 24, 1980.

[20] B. Waxman and M. Imase. Worst-case performance on Rayward-Smith's Steiner tree heuristic. *Information Processing Letters*, 29, 1988.

[21] P. Winter. Steiner problem in networks: A survey. *Networks*, 17, 1987.

[22] Y. F. Wu, P. Widmayer, and C.K. Wong. A faster approximation algorithm for the Steiner problem in graphs. *Acta Informatica*, 23, 1986.

[23] A. Z. Zelikovsky. The 11/6 approximation algorithms for the Steiner problem on networks. *Information and Computation*, 1992.

[24] J. Howe and Kia Makki. A Simple and Adaptable Parallel Algorithm for the Selection Problem. (Unpublished) University of Nevada, Las Vegas, 1994.

[25] J. Howe and Kia Makki. An Algorithm for Determining the Shortest Tree Connecting Steiner Nodes with No Direct Connections. (Unpublished) University of Nevada, Las Vegas, 1996.

VITA

Graduate College
University of Nevada, Las Vegas

John Gerard Howe

Home Address:
4146 E. Harmon Ave.
Las Vegas, NV 89121

Degrees:
Bachelor of Science, Civil Engineering, 1982
University of Nevada, Reno

Publications:

J. Howe and Kia Makki. A Simple and Adaptable Parallel Algorithm for the Selection Problem. (Unpublished) University of Nevada, Las Vegas, 1994.

J. Howe and Kia Makki. An Algorithm for Determining the Shortest Tree Connecting Steiner Nodes with No Direct Connections. (Unpublished) University of Nevada, Las Vegas, 1996.

Thesis Title: Two New Algorithms For Classical Problems in Computer Science

Thesis Examination Committee:
Chairperson, Dr. Evangelos Yfantis, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Committee Member, Dr. Ajoy Datta, Ph.D.
Graduate Faculty Representative, Dr. Eugene McGaugh, Ph.D.