A Greedy Randomized Adaptive Search Procedure for the 2-partition Problem

APPROVED:

Supervisor: ________________________________
Thomas A. Feo

____
Olivier Goldschmidt

A Greedy Randomized Adaptive Search Procedure for the 2-partition Problem

by

Hal Connor Elrod, B.A.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

THE UNIVERSITY OF TEXAS AT AUSTIN

December 1989

Abstract

A Greedy Randomized Adaptive Search Procedure for the 2-partition Problem

by

Hal Connor Elrod, B.A.

Supervising Professor: Thomas A. Feo

A greedy randomized adaptive search procedure (GRASP) for the network 2-partition problem is presented. The heuristic is empirically compared with the Kernighan-Lin (K&L) method on a wide range of instances. The GRASP approach dominates K&L with respect to both CPU time and solution value on a large percentage of the instances tested.

Table of Contents

1. Introduction

1.1. Description of the partition problem

Consider a weighted graph; a collection of nodes connected by weighted edges. The 2-partition problem is to cluster the nodes into two equal sized sets such that the weight of the edges between the two sets in minimized.

Instance: A graph $G = (V,E)$ and a weight $w(e) \geq 0$ for each $e \in E$.

Question: What partition of V into disjoint sets A and B, $|A|=|B|$, has a minimum sum of the weights of the edges in E having one end point in A and one end point in B?

2-partition is a well known NP-hard problem [GAR79]. It is part of the larger class of k-partition problems that share the same objective, to cluster vertices in such a way as to minimize the interconnections between the k sets. In the 2-partition problem with n nodes, there are $\frac{1}{2}\binom{n}{n/2}$ feasible solutions. Given that an exhaustive search of such a solution space is intractable for even moderate size problems, heuristics are often used in practice to generate good solutions.

1.2. Importance and applications

The partition problem arises when there is a need to physically or logically separate clusters of objects such that the interactions between these clusters are minimized [KER70], [KHE87]. An example of clustering occurs in the design of VLSI circuits (very large scale integration). The components that make up a VLSI chip are defined as modules. These modules must be physically connected, yet the connections are relatively costly. The wires that make up the connections take up space on the VLSI chip. Increasing the size of a chip reduces the yield per wafer, that is, the number of error free VLSI chips that are produced on each silicon disk. In addition, the farther apart the clusters the greater the total wire length. This limits the speed and reliability of the chip. An efficient circuit design will place highly connected modules in close proximity to minimize the total area and wire length. The problem of finding a good placement contains within it a partitioning problem. Consider representing each circuit module as a vertex of a graph and each connection between modules as an edge. A partitioning method would divide the vertices into two or more sets with the objective of minimizing the edges between sets. Such a partition represents an intelligent grouping of modules within a VLSI chip [MAC78]. In practice, such a procedure is applied hierarchically, first on the modules and then on clusters of modules [CHE86]. This hierarchical procedure is then used to find good module placements.

An example of logical clustering occurs in program design for a paged memory computer system [KRA65],[MAC78]. A large computer program is often divided into subprograms that pass information between one another. The subprograms will be active at different stages of the execution of the main program. In a paged memory management system, a program is stored mostly on a disk and partially in core memory in fixed-size blocks or pages. Access between subprograms on the same page is usually rapid because the page in which the subprograms are located will be resident in the computer's core memory. On the other hand, access to subprograms across pages often requires moving data from disk to core — a relatively time-consuming task. An approach to improve the placement of subprograms is to represent each as a weighted vertex of a graph, and a transfer of control between subprograms as an edge in the graph. The weight of the vertex represents the size of the subprogram. A page is represented by a cluster of vertices. Each cluster is constrained to be of total vertex weight no more than the fixed page size. If the vertices are efficiently partitioned, the transfer of control between subprograms on different pages is minimized, and disk-to-memory swapping is reduced.

Consider a large job shop with many work stations. Individual orders take many different paths through such a shop, depending on the machines needed for processing. This arrangement can be efficient because of its flexibility; however, there is a tradeoff. Scheduling for a set of orders can be extremely difficult because the availability of the necessary machines at each stage of production depends on the status of all the jobs in the shop. The computation of even a reasonable machine schedule is difficult because of the large number of workstations that must be considered. A more manageable system is to partition the shop into distinct production cells, with each cell scheduled individually. The variability of time to complete jobs may then be reduced and management is greatly simplified. One method of forming these production cells is to duplicate all the machinery so that every cell has at least one of every type of equipment. A less costly method is to group the available machinery into cells while minimizing or eliminating the transfer of orders between cells. The path an order takes through various workstations across the shop may be represented as a series of edges in a graph, with each workstation type represented by a vertex. A partition of such a graph into a fixed number of clusters (cells) that minimizes the number of edges (orders) between clusters represents an intelligent grouping of equipment on the shop floor.

1.3. Objectives of this thesis

This thesis has two objectives. The first is to describe the greedy randomized adaptive search procedure (GRASP) in general terms and to outline its applicability to NP-hard problems. The second is to apply it specifically to the 2-partition problem and demonstrate its performance.

1.4. Literature review

Kernighan-Lin heuristic

Since its publication almost two decades ago, the Kernighan-Lin heuristic [KER70] has stood as the dominating procedure for finding good solutions to network 2-partitioning. The heuristic starts with an arbitrary initial partition, sets A and B, then selects groups of nodes in the two sets to be exchanged. Consider the gain δ, resulting from moving one node from set A to set B. Given a node $a \in A$, the gain for placing a in B is equal to the sum of the weights of the edges between a and all $b \in B$, minus the sum of the weights of all edges between a and all $a' \in (A/a)$. In 2-partition, $|A|$ must be equal to $|B|$ so exchanges are made in pairs. The total gain from an exchange of two nodes, $a \in A$ and $b \in B$, is $\delta_a + \delta_b - 2c_{ab}$ where c_{ab} is the weight of edge (a,b). We now examine K&L in detail.

Kernighan-Lin Heuristic

Input: An edge weighted graph $G = (V,E)$, arbitrarily partitioned into two sets A and B, $|A| = |B|$

Output: A minimal weight 2-partition of G

```
done ← false  {initialize flag variable}
while not done do  {while not a local minimum do}
    $A_1 \leftarrow A$, $B_1 \leftarrow B$  {temporary sets $A_1$ and $B_1$ receive starting partition}

    for m = 1 to n/2 do  {for all n/2 pairs of vertices}

    select $a_i \in A_m$, $b_j \in B_m$ such that the gain $g_m$ from the exchange of a and b
    is maximum,  where $g_m = \delta_a + \delta_b - 2c_{ab}$

    $a'_m \leftarrow a_i$, $b'_m \leftarrow b_j$  {record maximum gain pair of vertices}
    $A_{m+1} \leftarrow A_m - a_i$, $B_{m+1} \leftarrow B_m - b_j$  {update temporary sets}
    endfor
    choose k to maximize $g_{max} = \sum_{j=1}^{k} g_j$  {maximize partial sum}
    if $g_{max} > 0$ then  {if gain is positive}
        move $a'_1 \cdots a'_k$ to B and $b'_1 \cdots b'_k$ to A  {perform exchange}
    else
        done ← true  {no positive exchanges, procedure finished}
    endif
endwhile
```

The procedure starts with a copy of the initial partition sets, A_m and B_m (m starts as 1). It then looks through all vertices and selects a pair of them, one from set A_m and one from B_m, such that the exchange of those two vertices will have maximum gain g_m. Those two vertices are recorded and removed from the temporary partition sets A_m and B_m. The for-loop then repeats, again selecting the maximum gain exchange given the exchange of the first pair. This goes on until all the vertices have been selected (and the temporary sets A_m and B_m are empty). The result is a list of exchange gains g and two corresponding lists of vertices a' and b'. Now k is chosen to maximize the partial sum of the first k elements in g. This sum, g_{max}, is the improvement in the objective for one iteration of the while-do loop. There will be both positive and negative elements in g, as the sum of all elements in g is zero (the gain from exchanging all nodes in the two sets is zero). If g_{max} is positive, then the first k pairs of vertices corresponding to $a'_{1..k}$, $b'_{1..k}$, are exchanged, the sets A and B are updated, and the procedure repeats. If g_{max} is not positive, there are no positive gain exchanges left and the procedure terminates. Because the gain, g_{max}, is maximized at each iteration of the while loop, the heuristic moves quickly to a local optimum. In practice, the heuristic typically performs two or three while–loop iterations.

During each iteration of the while–loop, n/2 pairs of vertices are selected and to find the maximum gain pair at each selection requires, in the worst case, $O(n^2)$ time. In the worst case, the running time per while–loop iteration is $O(n^3)$. If the δ_j's are sorted each iteration, this reduces the complexity to $O(n^2 \log n)$ [KER70]. Fiduccia and Mattheyses report an $O(|V| + |E|)$ running time for an alternative heuristic based on K&L [FID82]. This heuristic considers unbalanced partitions, with single vertex exchanges. To guarantee a balanced partition, it either alternates between the two sets or a penalty function for an unbalanced partition is used. The K&L version implemented in the experimentation of this paper uses a heap data structure to order the δ_j's, the gain from moving node j from one set to the other. At each iteration of the inner for-loop, the program looks at the three largest gains in each set when selecting the pair a_i and b_j to be exchanged. This reduces the time spent searching through the list at the cost of a slight decrease in power and is an extension described in the K&L paper. The running time per iteration of the K&L heuristic *in practice*, when implemented with the proper data structures, is also O(|V| + |E|) [JOH89].

Simulated annealing

The simulated annealing approach has recently been applied to the 2-partition problem by David Johnson et. al. [JOH89]. This approach also attempts to find a good solution by starting with an arbitrary initial partition and improving it through exchanges. Exchange procedures are typically greedy, that is, they swap the vertices that provide a positive, if not maximum gain. However, a pure greedy approach can converge to a poor local optima. Simulated annealing attempts to avoid such dead ends by strategically allowing exchanges to be made that temporarily result in a negative gain. Such perturbations allow for a better local optima to be found. The probability of accepting a negative gain exchange decreases as the procedure is run, and is controlled by a variable t, refered to as the temperature. A negative gain exchange in accepted with probability $\exp\{-\Delta/t\}$, where Δ is the change in the objective function resulting from the exchange.

Simulated Annealing

Input: An edge weighted graph G = (V,E), arbitrarily partitioned into two sets A and B, |A| = |B|
r: the rate of reduction in temperature
l: the temperature length; the iterations spent at any one temperature

Output: A minimal weight partition of G

```
set initial temperature $t > 0$  {choose starting temperature}
while $t > 0$ do  {while not frozen}
    for $j = 1$ to $l$ do  {perform l iterations where l is the temperature length}
        choose a pair of nodes at random; $a \in A$, $b \in B$
        $\Delta \leftarrow$ the change in the objective from exchanging a and b
        if $\Delta \leq 0$ then
            swap a and b  {if gain is positive then always perform exchange}
        endif
        if $\Delta > 0$ then
            $p \leftarrow$ random(1) {p drawn from U(0,1) distribution}
            if $p < e^{-\Delta/t}$ then
                swap a and b {if negative gain exchange accepted}
            endif
        endif
    endfor
    $t \leftarrow r\,t$  {decrease temperature according to rate r}
endwhile
```

The procedure executes while the temperature, t, is positive. At each temperature, l possible exchanges are evaluated. The pair of vertices, one from each set in the partition, is selected at random. If the gain from exchanging that pair of vertices is positive or zero, they are exchanged. If the gain is negative, then a sample is drawn from a uniform [0,1] distribution. If this is less than the value $\exp\{-\Delta/t\}$ then the exchange is performed. As $t \varnothing 0$, negative gain exchanges are no longer accepted and the procedure is "frozen". After l pairs have been considered, the temperature is reduced (at rate r) and the for–loop is repeated.

With the proper parameters, simulated annealing can guarantee an optimal solution, but only by taking an exponential amount of time [MIT86],[SAS88],[JOH89]. Empirically, simulated annealing often generates better partitions than K&L, but the running time is extremely long in comparison [JOH89]. Because of this, we compare our method to K&L and not simulated annealing in this paper.

Bounded algorithm

Feo, Goldschmidt, and Khellaf [FEO88] give a 1/2-approximation algorithm for the 2-partition problem.

2-Partition Heuristic

Input: An edge weighted graph G = (V,E)

Output: A minimal weight partition of G.

```
Find a minimum weight perfect matching in G.  Construct m pairs of vertices
$P_1,P_2,...,P_{n/2}$ from each pair of nodes incident to a matched edge.

$A \leftarrow \{\varnothing\}$, $B \leftarrow \{\varnothing\}$   {initial 2-partition (empty)}
for j = 1 to n/2 do  {for all pairs in the perfect matching}
    Assign one of the nodes of $P_j$ to set A and the other to set B such that the
    weight of all inner edges in the current sets A and B is maximized.
endfor
```

The first step, finding a minimum weight perfect matching, provides a lower bound on the weight of the 2-partition. In a minimum weight perfect matching the graph is divided into exactly n/2 pairs of vertices, with the sum of the weights of the edges within each pair minimum. Finding this requires $O(|V|^3)$ time. The matched pairs of vertices are then added one pair at a time, one vertex to set A and one to B, such that the weight of the inner edges in the sets is maximized. Maximizing the inner weight of the sets is equivalent to minimizing the weight of the partition.

This heuristic guarantees a partition of at least half the weight of the optimal. That is, the weight of the inner edges, those edges inside the sets, is at least half the total inner weight of the optimal partition. Note that maximizing the weight of the inner edges is equivalent to minimizing the weight of the partition since each edge is either an inner edge or in the partition. In practice, the worst case bound is not observed and the heuristic often generates good solutions. With respect to solution value, it is observed to produce results comparable to the K&L heuristic, though it runs slower because of the $O(|V|^3)$ time required to find a minimum weight perfect matching. A more efficient and randomized version of this heuristic is the objective of this paper.

1.5. The GRASP approach

Consider being asked to construct a minimum or near minimum weight 2-partition of the ten node graph in Figure 1.

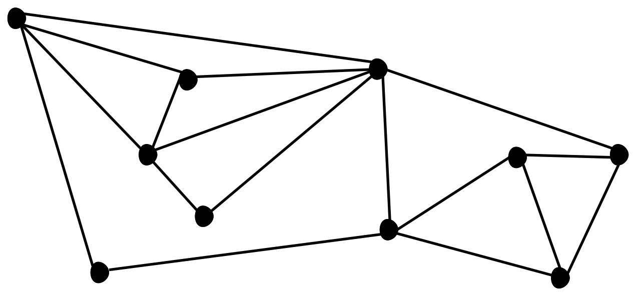


Figure 1. Ten node instance

A reasonable approach might be the greedy construction of an initial partition. Start by choosing one node at random and making it the first node in set A. Given that choice, look for a non-adjacent node to put into set B; if a non-adjacent node does not exist select any one of the remaining nodes. Then choose a node for A again. Given the two previous choices, look to minimize the additional weight of the partition. Continue adding nodes until they are all assigned to one of the two sets. Now examine sets A and B to see if a simple swap of two nodes might lower the weight of the partition. Keep swapping nodes until there are no swaps left that will improve the solution. If a lower weight or perhaps different solution is desired, start over. To avoid generating the same solution, choose a different starting node or, on those occasions when there is more than one node that adds a minimum weight to the partition, choose among those nodes in an iterative or random fashion. The name given to such a process is greedy randomized adaptive search procedure (GRASP).

Each GRASP iteration consists of constructing an initial solution and then improving that solution to find a local optimum. The construction phase is iterative, greedy, and adaptive. Iterative because the initial solution is built one element (i.e. node or pair of nodes) at a time. Greedy because the addition of each element is guided by a greedy function (i.e. minimize the augmented weight of the partition). Adaptive because the element chosen at any iteration is a function of those previously chosen (i.e. a function of the adjacencies of previously chosen nodes).

The probabilistic component of GRASP is applied to the selection process of elements during the construction phase. After choosing the first node for set A, all non-adjacent nodes are of equal quality with respect to the given greedy function.

If one of those nodes is chosen by some lexicographic rule, then the method is deterministic, and every GRASP iteration will produce the same result. In such stages where there are multiple best choices, choosing any one of them will not compromise the greedy approach, yet each will often lead to a very different solution. To generalize upon this strategy, consider forming a *candidate list* at each stage of the construction consisting of high quality elements according to the adaptive greedy function. For instance, the candidate list could be comprised of the top k (not necessarily best) nodes, where k is user specified. This is categorized by Hart and Shogan as a cardinality-based semi-greedy heuristic [HAR87]. It is no longer a pure greedy method, yet the perturbation introduced by such at strategy will lead to solutions different and possibly better than the strict greedy approach.

The solution generated by a greedy randomized adaptive construction can generally be improved upon by the application of an exchange postprocessor. That is, the construction does not guarantee a local minimum with respect to simple exchange neighborhoods. There is a tradeoff between the CPU time spent constructing a solution and the time spent improving each solution. An intelligent construction requires fewer exchanges in the postprocessor to reach a local minimum. This is important because the time spent in the postprocessor typically dominates the time required in construction. Intelligent construction results in a net reduction in the total CPU time spent per GRASP iteration [FEO89e]. Furthermore, the initial neighborhood to be searched by the exchange postprocessor is better. We observe that the local minimum found by the postprocessor is better using an intelligent construction than an arbitrary one.

Performing multiple GRASP iterations will generate a range of solutions. Most importantly, the solution values will be from a distribution, the mean of which may not be better than the value of the deterministic construction, but the best over all trials is observed to dominate the deterministic solution. See [BAR89], [FEO89a,b,c,d,e,f] for further discussions relating to such observations. The advantage of such a randomized heuristic is that increasing the number of GRASP iterations may lead to better solutions. Also, the method is robust in the sense that it is difficult to find or devise pathological instances for which the method will perform arbitrarily bad. Imagine an adversary attempting to devise such an instance; because the random component of GRASP, there is very low probability that he or she will correctly guess exactly what output the procedure will generate for a given input.

1.6. Outline of thesis

The remainder of this thesis proposal presents a greedy randomized adaptive search procedure for the 2-partition problem. Section 2 describes the developed heuristic. Section 3 contains the results of an extensive empirical study comparing the GRASP approach to the K&L method. Finally, Section 4 offers extensions to the 2-partition problem and other types of partitioning problems.

2. Heuristic

2.1. GRASP implementation

A number of objective functions and postprocessors were developed and coded before selecting the one used for the empirical testing presented in this proposal. In addition to refining our approach for the 2-partition problem, the relative performance of the various methods gave insight into the workings of GRASP.

As mentioned previously, the first heuristic is based on the 1/2-approximation algorithm described in [FEO88]. A Fortran minimum weight perfect matching code by Ball and Bodin [BAL83] is used to generate the matchings. The vertices are added one matched pairs at a time to the partition sets. The order in which the matched pairs are added is random, but their orientation follows the 1/2-approximation algorithm; the inner weight of the partition is maximized with the addition of each pair.

Empirical tests verify the strength of this heuristic method with regards to solution value, but the $O(|V|^3)$ time required each iteration makes it much slower than the K&L method. A better approach is to use a fast heuristic to generate a low-weight perfect matching, then add the matched pairs in the proper orientation. On relatively sparse 0-1 graphs it is very easy to generate a low-weight, but not necessarily minimum weight, matching. Several variations of the matching heuristic were tested. These were dense-dense, sparse-sparse, sparse-dense, and random. Dense-dense attempts to match 'dense' nodes together; that is, nodes with many adjacencies. The procedure orders the nodes by number of adjacencies and starts by matching the most dense node with the highest density non-adjacent node. The idea is to add high density pairs of nodes at the beginning stages of the construction of the partition, so they determine the orientation of the other nodes. Sparse-sparse and sparse-dense are variations on this theme. Random matching attempts to find a low weight matching without any particular order with regards to number of adjacencies. All four methods perform well, but not as well as the approach finally chosen. We found the best method was not to find a matching but to construct a partition directly by adding the vertices one at a time. At each stage of the construction the vertex adding the lowest (or near-lowest) additional weight is chosen. This has be advantage of being faster than the two stages of matching and construction. Empirically, it also gave results that were, on average, better than the matching method.

2.2. Formal Description of Heuristic and Postprocessors

Having described in general terms the different possible approaches to the 2-partition, we here formally detail the actual heuristic chosen.

Variable definitions:

c_{ij}	= cost of arc (i,j)
a_i,b_i	= the ith vertex in clusters A and B respectively
α_i	= the sum of the weights of all the arcs from node i to the vertices in the set A minus the sum of the weights of all the arcs from node i to the vertices in set B
U	= the set of vertices not currently in the partition
swapcost	= the cost of switching the current pair of vertices
lastswap	= the number of iterations since the last swap was performed

GRASP heuristic

```
Input:
    $c_{ij}\ \forall\ (i,j) \in E$
    $\alpha_i = 0\ \forall\ i \in V$
    $A = B = \{\emptyset\}$
    $K = \{\emptyset\}$  {the candidate list of size k}
Output:
    $a_i, b_i\ \forall\ i = 1, \ldots, \frac{|V|}{2}$
    $\alpha_i\ \forall\ i \in V$  {output cost vector for postprocessor}
  for $j = 1$ to $n/2$ do  {construction loop}
      $K \leftarrow$ k nodes of largest $\alpha_i$, $i \in U$  {form candidate list of top k nodes}
      randomly select $i \in K$  {randomly choose node from candidate list}
      $A_j \leftarrow i$  {add chosen node to set A}
      $\alpha_i \leftarrow \alpha_i + c_{ij}\ \forall\ i$  {adaptively update costs}
      $K \leftarrow$ k nodes of smallest $\alpha_i$, $i \in U$  {form candidate list of top k nodes}
      randomly select $i \in K$  {randomly choose node from candidate list}
      $B_j \leftarrow i$  {add chosen node to set B}
      $\alpha_i \leftarrow \alpha_i - c_{ij}\ \forall\ i$  {adaptively update costs}
  endfor
  call postprocessor  {improve solution with 2-exchange postprocessor}
  end
```

The α cost vector is an important improvement in the heuristic. As mentioned, α_i is the sum of the weights of all the arcs from node i to the vertices in set A minus the sum of the weights of all the arcs from node i to the vertices in set B. In other words, α_i represents the preference of node i for set A. A positive α_i indicates a positive gain from moving node i from set B to set A or a negative gain moving from set A to set B. Likewise, a negative α_i indicates a positive gain from moving node i from set A to set B. With each addition of a node, the costs are updated for all nodes adjacent to it.

The effectiveness of a GRASP for a fixed CPU time depends greatly on the number of iterations performed in that time, so it is imperative that each iteration be performed efficiently (to minimize the CPU time per iteration). For this reason the GRASP heuristic uses both an (n x n) incidence matrix and an adjacency list for each node. The incidence matrix allows the fastest lookup time when computing the net benefit of an exchange, whereas the adjacency list is fastest for updating the costs of adjacent nodes once an exchange is performed.

In forming a candidate list at each stage of the greedy adaptive construction, particularly for sparse graphs, a heap data structure is often used to order the costs [FEO89]. For an average node degree d, at each stage of the construction updating the heap will require O(d log n) time. This looks better than the O(n) time required for scanning the list of nodes, at least for sparse graphs. However, what is best in theory is not always best in practice and an efficient scanning approach empirically performed better for our heuristic. The costs of the unpartitioned nodes are kept in a linked list, which contracts as the procedure runs. The candidate nodes are chosen by performing one pass through the list, keeping the best k values along with pointers of their locations in the list. The node chosen randomly from the candidate list is then efficiently deleted from the list of unpartitioned nodes.

There are n/2 iterations in the greedy construction. Selecting a node requires O(n) time for a dense graph. If the nodes are ordered in a heap data structure for the sparse case, selecting a node is an O(1) operation. To adaptively update the costs takes O(n) operations for a dense graph; updating the heap requires an average O(d log n) a sparse graph, where d is the node degree. The overall complexity of the construction phase is $O(n^2)$ for a dense graph, $O(n \log n)$ for sparse. Analysis of the postprocessor will follow a discussion of the various methods.

The simplest and perhaps most intuitive postprocessor tried is 2-exchange (First Swap). All possible combinations of vertices in sets A and B are considered and a pair is exchanged if the exchange improves the objective. The procedure continues until no further improvement is possible.

First Swap

```
Input:
    $c_{ij}\ \forall\ (i,j) \in E$
    $a_i, b_i\ \forall\ i = 1,\ldots,\frac{|V|}{2}$
    $\alpha_i\ \forall\ i \in V$  {the cost vector $\alpha$ generated by the construction phase}
Output:
    $a_i, b_i\ \forall\ i = 1,\ldots,\frac{|V|}{2}$
    $\alpha_i\ \forall\ i \in V$
Initialize   lastswap $\leftarrow$ 0, i $\leftarrow$ 0  {initialization}
repeat
    lastswap $\leftarrow$ lastswap + 1  {increment counter lastswap}
    i $\leftarrow$ i + 1  {increment list pointer i}
    if i > $\frac{n}{2}$ then i $\leftarrow$ 1  {if at end of list move to beginning}
        for j = 1 to n/2 do  {inner loop}
            u $\leftarrow a_i$, v $\leftarrow b_j$  {u, v  the ith nodes in sets A and B}
            swapcost $\leftarrow \alpha_v - \alpha_u - 2c_{u,v}$  {calculate net benefit from exchange}
            if   swapcost > 0  {if net benefit is positive}
            then    $a_i \leftarrow$ v  {switch $a_i$ and $b_i$}
                    $b_j \leftarrow$ u
                    lastswap $\leftarrow$ 0  {reset counter lastswap}
                for m = 1 to n do {for all nodes in graph}
                    $\alpha_m \leftarrow \alpha_m - c_{m,v} + c_{m,u}$  {update cost}
                endfor  {m}
            endif
        endfor  {j}
    endif
until (lastswap = $\frac{n}{2}$ -1)  {until gone through whole list without any exchanges}
```

If the postprocessor could be made faster without compromising its strength then more GRASP iterations could be run in the same amount of cpu time, perhaps generating a better minimum cut partition. One idea is to search through all possible exchanges and choose the best possible exchange to make at each iteration. We call this Best Swap. In practice, this method is observed to perform poorly for 2-partition. Only a few exchanges are made, and poor solutions are obtained. A possible explanation for this result is that choosing large gain exchanges leads too quickly to a local optimum. Assuming this is the case, taking a small or even minimum gain at each exchange may lead to better solutions. Taking a near minimum gain exchange is called Slight Swap, choosing the absolute minimum gain exchange is refered to as Slightest Swap. With a 0-1 graph, the Slightest Swap will search all possible exchanges until it finds one with a minimum positive improvement. Thus, if it finds an exchange with a gain of 1 then the search ends. Slight Swap also searches for minimum positive improvement exchanges, but a gain of 1 or 2 ends the search. Since searching for the exchanges takes the largest part of the postprocessor time, the motivation behind Slight Swap is to trade off some of the power of choosing only minimum gain exchanges in return for an improvement in speed.

Slight and Slightest Swap

```
Input:
    $c_{ij}\ \forall\ (i,j) \in E$
    $a_i,b_i\ \forall\ i = 1,\ldots,\frac{|V|}{2}$
    $\alpha_i\ \forall\ i \in V$  {the cost vector $\alpha$ generated by the construction phase}
Output:
    $a_i,b_i\ \forall\ i =1,\ldots,\frac{|V|}{2}$
    $\alpha_i\ \forall\ i \in V$

Initialize  lastswap $\leftarrow$ 0
repeat
    u $\leftarrow$ 1
    found $\leftarrow$ false
    worst $\leftarrow$ false
    while (u $\le \frac{n}{2}$ and not worst) do  {not end of list and slight(est) swap not found}
        v $\leftarrow$ 1
        while (v $\le \frac{n}{2}$ and not worst) do  {not end of list and slight swap not found}
            swapcost $\leftarrow \alpha_v - \alpha_u - 2c_{u,v}$  {calculate net benefit}
            if   swapcost > 0  {if net benefit is positive}
            then    found $\leftarrow$ true  {positive benefit swap has been found}

            if  swapcost $\le$ 2  {swapcost = 1 for slightest swap}
            then    worst $\leftarrow$ true  {slight swap has been found}
                worstv $\leftarrow$ v
                worstu $\leftarrow$ u
            else    v $\leftarrow$ v+1  {else keep searching}
        endwhile
        u $\leftarrow$ u+1
    endwhile
    if found = true
    then update costs  {same as First Swap}
until (found = false)  {until no positive benefit exchanges}
```

Empirically, these methods have proven to be the best postprocessors, although they take slightly more time per iteration than First Swap. As expected, the Slightest Swap runs slower than Slight Swap because the number of possible exchanges to be compared until an appropriate exchange is found is greater than or equal to the number of such comparisons in Slight Swap. However, there is no significant difference in the solutions generated for a given input. Since the faster running time allows for more GRASP iterations, the Slight Swap is the method of choice.

For all the two-exchange postprocessors presented, there are $O(n^2)$ possible swaps to be compared at each iteration. When an exchange is made, there are $O(n)$ updates required for a dense graph. The overall complexity is $O(\kappa n^2)$ where κ is the maximum number of swaps performed. In the worst case, κ is exponential with respect to the input size of the general edge weighted instances. For for 0-1 edge weight instances, κ is $O(|E|)$. Empirically, κ is observed to be well behaved in either case.

The worst case complexity of the postprocessor may be improved to $O(|V| + |E|)$ per iteration. We distinguish two cases: non-weighted (0-1) graphs and edge-weighted graphs. For clarity consider two cost vectors α and β. Define α_ι as the sum of weights of the edges from node i in set A to the nodes in set B minus the sum of the weights of the edges between i and $A\backslash\{i\}$. Similarly define β_i for $i \in B$. For 0-1 graphs, the elements of α are bin sorted. The bin values range from the minimum α–value to the maximum α-value. Similarly, the vector β is bin sorted. Because the α-values (β-values) are bounded by the maximum degree of a vertex, the bin sorting takes $O(|E| + |V|)$ time. A beneficial exchange of vertices $i \in A$ and $j \in B$ is one where $\alpha_i + \beta_j \geq 1$ if i and j are non-adjacent, $\alpha_i + \beta_j \geq 3$ if they are adjacent. We now consider an improved version of the First Swap procedure for 0-1 instances. The first step is to calculate the α and β vectors by examining all edges. This requires $O(|E|)$ operations.

Improved First Swap

```
Input:
    $c_{ij}\ \forall\ (i,j) \in E$  {adjacency matrix}
    $L_e\ \forall\ e \in E$  {a list of edges with endpoints}
    A and B
    $a_i, b_i\ \forall\ i = 1,\ldots,\frac{|V|}{2}$
Output:
    $a_i, b_i\ \forall\ i = 1,\ldots,\frac{|V|}{2}$
    $\alpha_i, \beta_i\ \forall\ i = 1,\ldots,\frac{|V|}{2}$
found $\leftarrow$ true  {initialize flag}
$\alpha_i = 0\ \forall\ i \in A$  {initialize cost vectors}
$\beta_j = 0\ \forall\ j \in B$
```

```
for k = 1 to |E| do  {for each edge}
    let $L_k$ = (i,j)  {i and j are nodes incident to $L_k$}
    if i $\in$ A and j $\in$ A then  {if both nodes in set A}
        $\alpha_i \leftarrow \alpha_i - 1$, $\alpha_j \leftarrow \alpha_j - 1$
    else if i $\in$ A and j $\in$ B then  {if one node in A and one in B}
        $\alpha_i \leftarrow \alpha_i + 1$, $\beta_j \leftarrow \beta_j + 1$
    else if i $\in$ B and j $\in$ A then
        $\alpha_j \leftarrow \alpha_j + 1$, $\beta_i \leftarrow \beta_i + 1$
    else
        $\beta_i \leftarrow \beta_i - 1$, $\beta_j \leftarrow \beta_j - 1$  {otherwise, both nodes are in set B}
    endif
endfor
while found do  {while positive gain exchanges have been found}
    binsort $\alpha$, $\beta$
    k $\leftarrow$ 1
    found $\leftarrow$ false  {set flag to false}
    while not found and k $\leq$ |E| do  {while not found examine all edges}
        if one endpoint i of edge $L_k \in$ A and endpoint j $\in$ B and $\alpha_i + \beta_j \geq 3$ then
            found $\leftarrow$ true  {positive gain exchange found}
            u $\leftarrow$ i
            v $\leftarrow$ j
        endif
        k $\leftarrow$ k + 1  {go to next edge}
    endwhile
    for each vertex i in a non–empty $\alpha$ bin of value k and while not found do
        for each vertex j in a non–empty $\beta$ bin of value $\geq$ 1-k and while not found do
            if there is no edge (i,j) then
                found $\leftarrow$ true  {positive gain exchange found}
                u $\leftarrow$ i
                v $\leftarrow$ j
            endif
        endfor
    endfor
    if found then
        exchange u,v
        update $\alpha$ and $\beta$
    endif
endwhile
```

The inner while–loop examines all adjacent pair of vertices to find a positive gain exchange. This requires $O(|E|)$ time. The second set of loops is slightly more complicated. If one vertex has an α value of k and another a β value ≥ 1-k, then exchanging them gives a positive gain if they are not adjacent. If they are adjacent, the exchange cannot have positive gain because exchanges of adjacent vertices have already been examined. Therefore, the greatest number of vertices that can be compared that do not have a positive gain exchange is $O(|E|)$.

The same improvements may be applied to Slight and Slightest Swap. A slightest swap of nodes i and j is one where $\alpha_i + \beta_j = 1$ for non-adjacent vertices and 3 for adjacent nodes. If such a swap is found the search iteration ends and the nodes are exchanged. Otherwise the procedure selects the smallest possible positive swap. Both Slight and Slightest Swap can be implemented in $O(|V| + |E|)$ time.

For the case of a weighted graph, bin sorting is not applicable because the number of bins can be exponentially large. However, consider a similar procedure that calculates the α and β vectors and examines all edges for positive gain exchanges as in the Improved First Swap. Rather than sorting the α and β cost vectors in a bin structure, select the $\lceil \sqrt{|E|} + 1 \rceil$ largest values in each vector, an $O(|V|)$ time operation [AHO83]. Comparing all possible exchanges of these two sets of vertices requires $\lceil \sqrt{|E|} + 1 \rceil^2$ or $O(|E|)$ operations. Note that at least one pair of vertices will be non-adjacent, and the exchange of greatest benefit will be among the $\lceil \sqrt{|E|} + 1 \rceil^2$ pairs considered. If no non-adjacent vertices offer a positive gain exchange then the procedure stops. As before, the complexity for the First Swap in the edge weight case is O(|V| + |E|) per search iteration. However, to assure a minimum or near minimum positive gain as prescribed in Slightest or Slight Swap requires that the α and β values be sorted. With sorting, the complexity for Slight and Slightest Swap becomes O(|V|log|V| + |E|) per search iteration.

The argument could be made that the partitions generated could just be the result of repeated application of the postprocessor. To test this, we have run a heuristic where the nodes are randomly placed into two sets with no regards to the weight of the partition, then exchanged by the post-processor. Although this process may generate good solutions, it is quite slow when compared to the GRASP approach. This is due to two reasons. First, the postprocessor requires a great deal more time because many more swaps are performed. Thus, each iteration is slower than a GRASP iteration. Second, to obtain a given quality of solution, more overall iterations are required with a random start than with a GRASP construction start.

3. Empirical Experimentation

3.1. Implementation environment

For empirical comparison, a C implementation of the K&L heuristic was used. The code was provided courtesy of the Electronics Research Laboratory (ERL) at U.C. Berkeley. The running times of this implementation on problems between 124 and 1000 nodes and of various edge densities are comparable to those reported for the K&L implementation given in [JOH89] in terms of both absolute CPU time and its growth rate.

The GRASP code was also written in C. The initial testing was run under the MS-DOS operation system on an IBM PS/2 Model 80 personal computer. The comparisons presented in the Results section of this paper were performed on an IBM 3081 computer running under the VMS/SP operating system.

3.2. Experiment Design

3.2.1 Test problems

The K&L and GRASP methods were compared first on 50 randomly generated 0-1 instances each of 16, 32, 64, and 128 nodes with edge densities of 20, 40, and 80 percent. These tests were used to refine the GRASP program and determine the candidate list size. The random instances are of type $G_{n,p}$ and are generated in the following fashion.

Generate Random Graphs

```
Input: p = density, n= number of nodes  {G_{n,p}}
Output: A 0-1 graph G
    for i = 1 to n do {outer loop}
        for j = i +1 to n do {inner loop}
            if random(1) < p then $c_{ij} \leftarrow 1$  {p = probability of an edge}
        endfor
    endfor
```

The smaller problems proved to be less interesting (both methods tend to generate the same solutions), so the GRASP heuristic was refined on the larger problems (64 and 128 nodes). New sets of test problems were randomly generated to eliminate any bias that might result from reworking the same problems. In a randomized technique the choice of a seed value and candidate list size can dramatically change the final solution, and if the same problem is run many times the parameter settings may be unconsciously biased towards solving that particular problem. In the extreme, a seed could be chosen that would guarantee the optimal solution on the first iteration for any given problem which would prove little about the value of the technique in general.

Once the GRASP heuristic was selected, the two methods were compared on larger, sparse graphs as described in [JOH89]. These proved to be challenging problems as the K&L implementation runs particularly fast on many of these sparse graphs. Further experiments were run on a class of graphs known a geometric graphs, which are generated by the following procedure.

Generate Geometric Graphs

```
Input: d = minimum Euclidian distance, n = number of nodes
Output: A 0-1 geometric graph G
for w = 1 to n do  {Initialize n pairs of coordinates}
    $x_w \leftarrow$ random(1)  {x drawn from U(0,1)}
    $y_w \leftarrow$ random(1)  {y drawn from U(0,1)}
endfor
for i = 1 to n do {outer loop}
    for j = i+1 to n do {inner loop}
        $m \leftarrow \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$   {m = Euclidean distance}
        if m < d then $c_{ij} \leftarrow 1$  {if distance small enough}
    endfor
endfor
```

The initial for–loop randomly generates coordinates for a set of n points in the unit square. The next for–loops evaluates all pairs of points. If the distance between the points is less than the parameter d, then an edge exists between the corresponding vertices. These geometric graphs tend to have a natural clustering typical of real-world problems that makes them a good test for partitioning.

3.2.2 Statistics collected

The two most important statistics collected were the running time and the solution value generated. There was one solution and one running time collected for each instance applied to the deterministic K&L code. For the GRASP heuristic, the average solution, the time and number of iterations to the best solution found, and the total number of iterations were collected. Note that the iteration at which the GRASP heuristic solution ties or beats K&L gives a picture of how its performance improves over time.

3.2.3 Setting the candidate list restriction

Once the heuristic was coded, the only parameter to choose is the candidate list size. A candidate list of size 1 generates a deterministic solution, while size n generates a random start for the postprocessor. In between these two extremes, we found a candidate list of between 2 and 4 allows the necessary variability without overly compromising the greedy objective; a size of 2 was finally chosen. As the candidate list size gets above 4 performance degrades as the initial solutions passed to the postprocessor become worse.

3.3. Results

The results of the 300 initial instances demonstrated the strength of GRASP when compared empirically with the K&L heuristic. Further experiments were performed on larger problems using an IBM 3081 computer. The following tables illustrate the relative performance in terms of the best solution found for each of the 100 instances tested in each size and density. In Table 1, the GRASP heuristic was run for the same amount of time taken by the K&L method implemented. The number of instances on which the GRASP method found the best solution is compared with the number of the 100 instances K&L found the best solution. The expected average degree is equal to (n x p), where p is the probability of an edge. When the two numbers do not total 100, both methods found the same solution on some instances. Also presented is the CPU time in seconds and the average number of GRASP iterations performed during that time.

	Expected average degree of vertices						
N	**2.5**	**5**	**10**	**20**	**40**	**80**	**160**
124	**71/19**	**58/28**	**58/33**	**63/30**	**81/11**	**82/13**	—
cpu sec	.10	.14	.15	.27	.50	.90	—
# iter	4	3	5	7.9	11.9	16.6	—
250	**78/13**	**57/38**	**45/54**	**58/33**	**69/28**	**76/22**	**77/22**
cpu sec	.24	.32	.45	.63	1.15	2.17	4.42
# iter	3	3	4	5.4	8.7	13	19.6
500	**91/8**	**61/37**	**45/53**	**49/50**	**54/46**	**81/19**	**90/9**
cpu sec	.5	.62	.89	1.51	2.84	5.38	10.58
# iter	2	2	3	4	6	9.7	15.2
1000	**85/11**	**43/53**	**32/65**	**44/55**	**49/51**	**68/30**	**67/33**
cpu sec	1.01	1.49	2.1	3.69	6.98	14.6	21.24
# iter	1	1	2	3	4	~7	~8

Table 1. GRASP wins / K&L wins at K&L running time

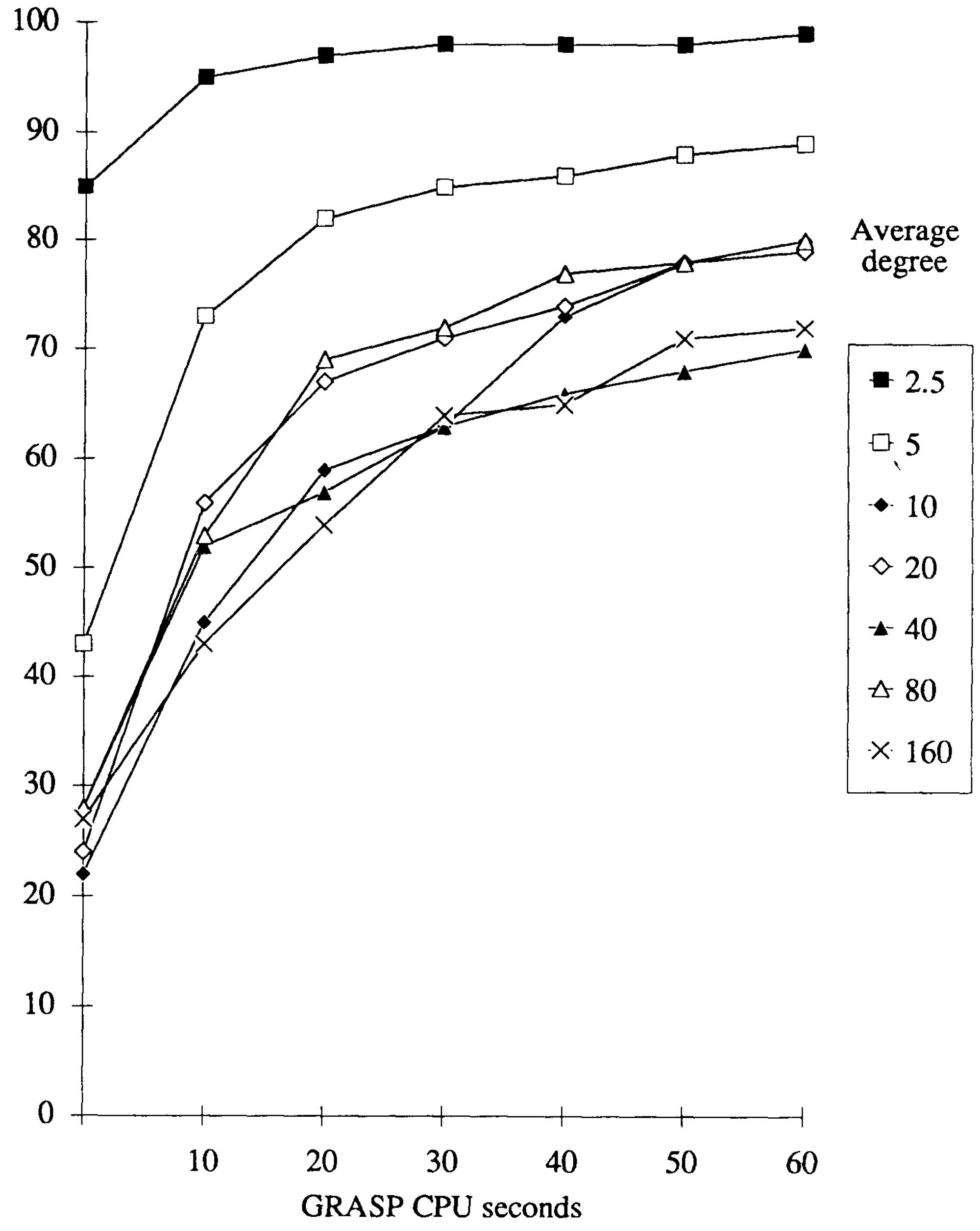
For most of the 2700 instances represented in Table 1 GRASP dominates, yet there is a pattern. While GRASP performs better overall, on the larger graphs around the column of degree 10, K&L performs better. When running for the CPU time of the K&L heuristic, particularly on the 1000 node problems, there was only time for as few as 1 or 2 GRASP iterations. A major advantage of a randomized method such as GRASP is that better solutions can be generated simply by increasing the running time. As shown in Table 2, the same instances were run on the GRASP for a constant amount of time, to a maximum of 60 seconds for the largest problems.

time (sec)	**n**	Expected average degree of vertices **2.5**	**5**	**10**	**20**	**40**	**80**	**160**
7.5	**124**	**94/0**	**92/2**	**91/0**	**85/1**	**93/0**	**90/0**	—
# iter		250	240	224	201	171	134	—
15	**250**	**98/1**	**96/2**	**87/10**	**94/4**	**90/6**	**94/4**	**90/8**
# iter		138	133	126	117	105	86.9	68.8
30	**500**	**96/4**	**93/5**	**85/15**	**87/12**	**84/16**	**92/7**	**95/4**
# iter		73.8	72.1	69.9	65.8	61.3	52.8	42.1
60	**1000**	**99/0**	**89/9**	**80/20**	**79/19**	**70/30**	**80/17**	**79/20**
# iter		37.7	37.1	36.6	35.0	33.0	30.0	24.3

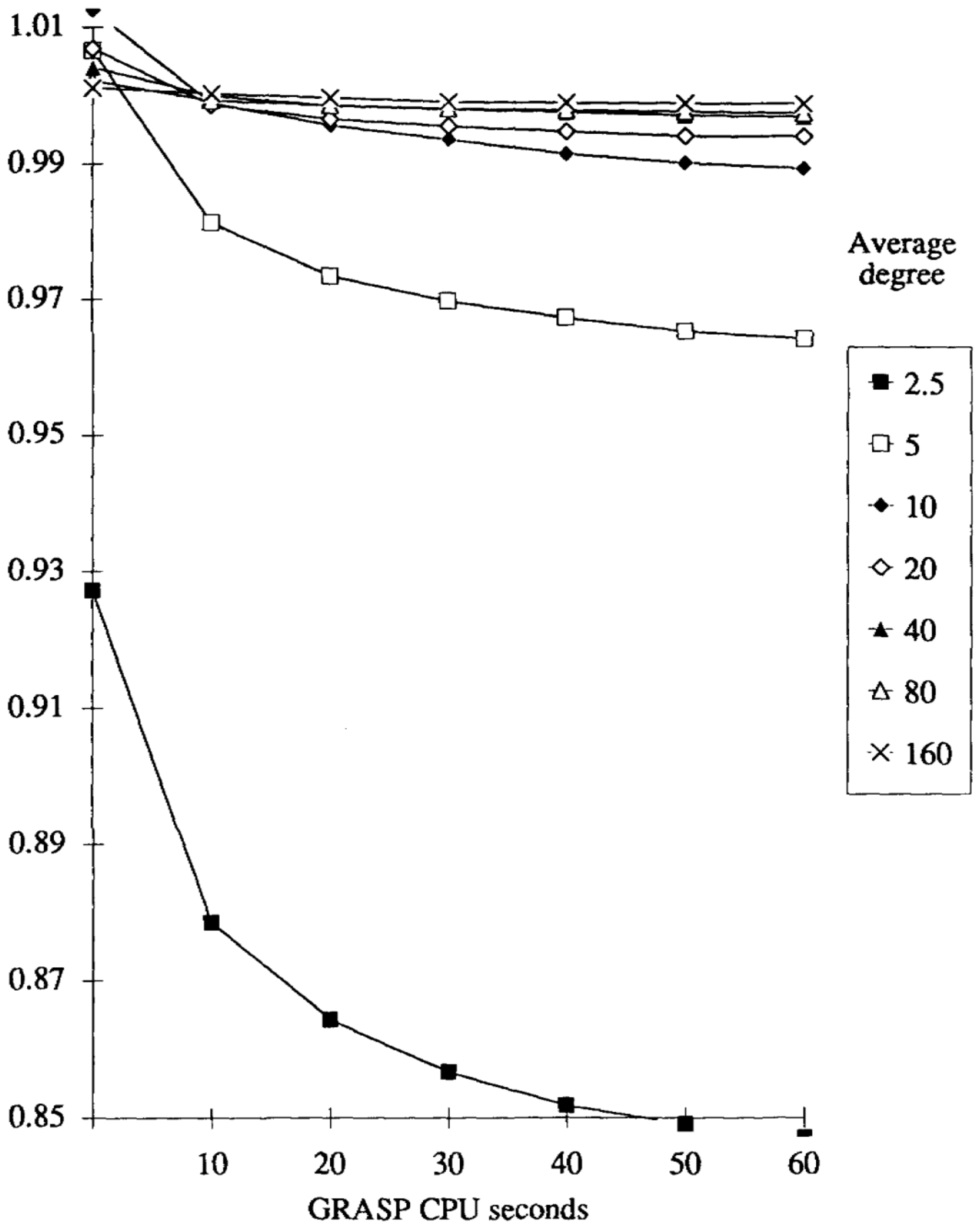
Table 2. GRASP wins / K&L wins at constant running time

Again, clearly GRASP dominates, yet in Table 2 it does so more consistently. On the "worst" instances, 1000 nodes with an expected average degree of 10, the number of wins increased from 32 to 80. This improvement for an increased running time is, as mentioned, an advantage of the GRASP method, and is illustrated in the following graphs. In Graph 1, the improvement in "wins" on the 1000 node instances is presented. Graph 2 represents the solution value found by the GRASP method divided by that found by K&L; at 1.0 the solutions are the same.

Graph 1. GRASP wins over 100 random graph instances



Graph 2. GRASP solution / K&L solution over 100 random graphs



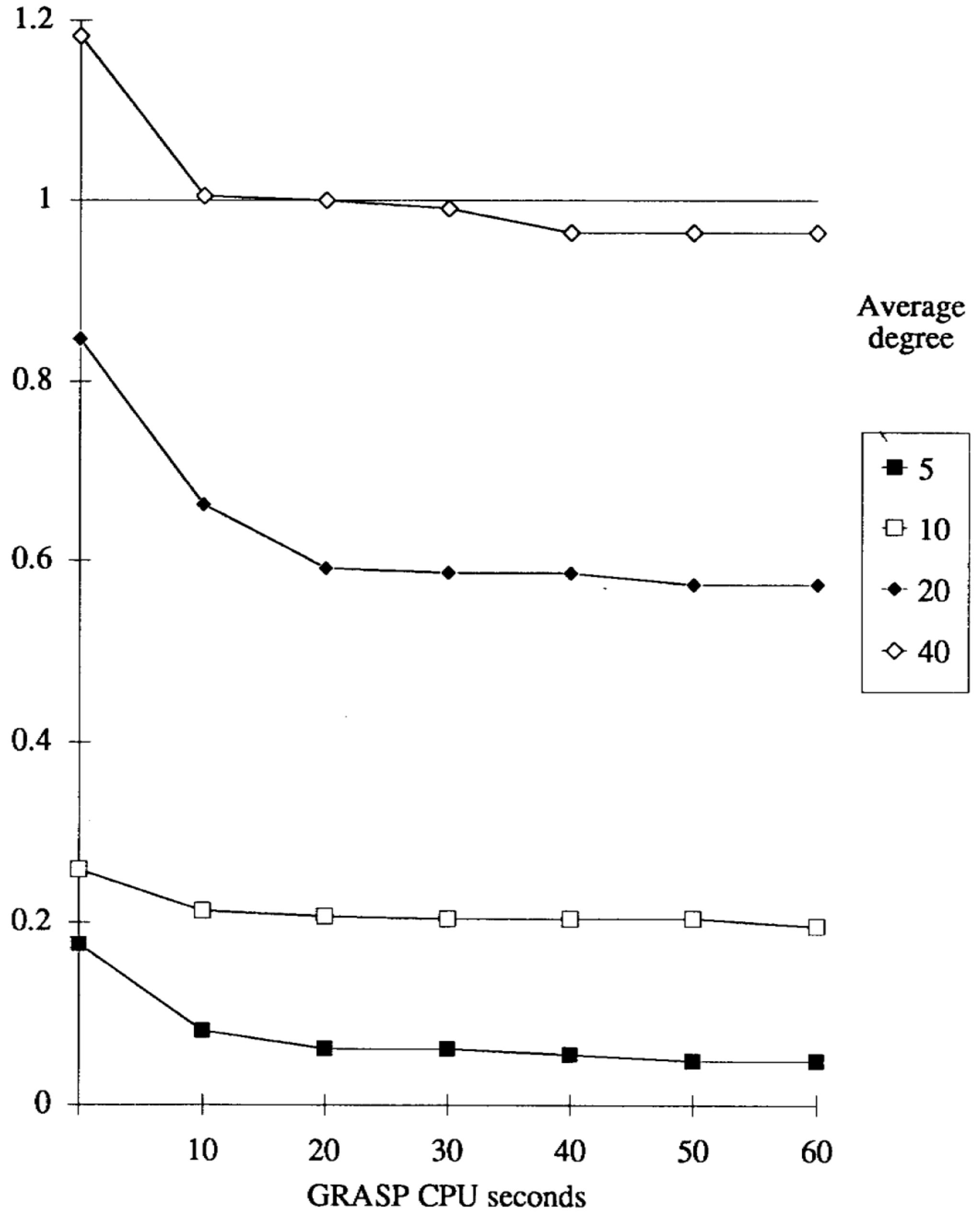
Finally, 40 instances of geometric graphs were tested. As mentioned previously, these are often particularly challenging because of their natural clustering. In Table 3, $n\pi d^2$ corresponds roughly to the expected average degree of the vertices.

time (sec)	**n**	**$n\pi d^2$ 5**	**10**	**20**	**40**
30	**500**	**5/0**	**5/0**	**4/0**	**0/0**
# iter		70.4	66.4	60.8	53
60	**1000**	**5/0**	**5/0**	**5/0**	**2/0**
# iter		36.6	35.0	33.4	29.0

Table 3. GRASP wins / K&L wins at K&L running time on geometric graphs

Besides the fact the GRASP dominates all geometric instances tested at a constant running time, there are patterns that emerge from the results. This can be seen from Graph 3, which compares the GRASP and K&L solution values found on the 1000 node graphs over the 60 seconds of CPU time spent. On all but the densest instances GRASP found solutions that were significantly better that those found by K&L. In [JOH89] it was observed that the solution space for a geometric graph differs greatly from that of a random graph because local optima may be far away from each other in terms of a neighbor exchange. Perhaps because of this, simulated annealing performed relatively poorly on geometric instances. It seems likely that the randomized construction in GRASP enables it to exploit the clustered nature of these graphs in finding better local optima by visiting many separate regions of the solution space.

Graph 3. GRASP solution / K&L solution on geometric instances



4. Conclusion

4.1. Evaluation of the GRASP approach

The results for the 2-partition problem are an example of the strength of the GRASP approach. Graphs 1-3 clearly demonstrates the improvement in solutions over time. Tables 1-3 show the competitiveness of the GRASP approach with the established K&L method. Not apparent from the tables is the methods relative simplicity. While there are many details specific to the 2-partition, the general approach is readily understandable. The actual programming was fairly straightforward. The only parameter to select is the candidate list size, and once chosen it works well for a broad range of problems. Yet underlying this simplicity, there is demonstrated robustness. The performance of GRASP has been excellent for all the combinations of size and density tested. The method is surprisingly strong with respect to geometric graphs. As mentioned previously, one advantage of a randomized heuristic is that it is difficult to find or contrive pathological instances that cause it to perform poorly. More generally, such a heuristic performs consistently across a broad range of problems. It is almost paradoxical that randomness breeds such consistency.

4.2. Extensions

The GRASP heuristic may be easily extended to different k-partition problems. The adjustments include incorporating k sets in the randomized objective function and extending the exchange postprocessor to evaluate swaps between all the sets. The α vector becomes a matrix, where $\alpha_{i,j}$ is the weight of the arcs from node i to the vertices in set j, minus the weight of its adjacencies to all other vertices.

k-partition heuristic

```
Input:
    $c_{ij}$ $\forall$ (i,j) $\in$ E
    $\alpha_{i,s} = 0$ $\forall$ i $\in$ V, s $\in$ 1..k
    $S_{1,2..k} = \{\emptyset\}$

Output:
    $\alpha_i$ $\forall$ i $\in$V
    $S_{1,2..k}$

    for j = 1 to n/k do  {construction loop}
        for m = 1 to k do
            find largest $\alpha_{i,m}$, i $\in$ U  {greedily choose best node}
            $S_j \leftarrow i$  {add chosen node list}
            $\alpha_{i,m} \leftarrow \alpha_i + c_{ij}$ $\forall$ i  {adaptively update costs}
        endfor
    endfor
    call postprocessor  {improve solution with 2-exchange postprocessor}
    end
```

The idea behind the construction is the same, however, there are now k sets to construct. In each iteration of the outer for–loop the procedure adds one node to each of the k sets in the inner for-loop. The problem is now considerably more complex than 2-partition, yet a randomized heuristic may perform well.

5. Appendix

```
/* .....................................................................
    HPART.C: A GRASP approach to the 2-partition problem

    GRASP == Greedy Randomized Adaptive Search Procedure
    Builds a low weight partition of 0-1 graph by greedily adding
    pairs of nodes from a candidate list of those nodes the maximize
    the current partition, then the weight of the partition is reduced
    by exchanging pairs of nodes when the exchange will increase the
    weight of the inner edges.

    Hal Elrod   --  Operations Research Group
              Department of Mechanical Engineering
              The University of Texas at Austin
              Austin, TX  78712

    Spring Break, March 1989
    last modified 6/30/89
  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

  Input: number of nodes, number of edges
         node, node, weight
          .      .     .
          .   .     .
         node, node, weight

  Use:        HPART <inputfile> <modea> <modeb> <c-list> <run-time>
  Where: <modea> = "1", heap greedy unmatched partition
              = "2", greedy unmatched partition
         <modeb> = "1", first swap -- generic 2-exchange
              = "2", slight swap
              = "3", slightest swap
              = "4", compact slight swap

  For MS-DOS systems, HPART should be compiled under COMPACT or HUGE models.
  ...................................................................... */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

#define ind(i,j,nn) (i*nn+j)
#define random(num) (rand() % (num))
#define hnode back
#define alpha next
#define apoint back
#define bpoint next

typedef struct anode{
     int node;
     struct anode *next;
     }nodez;

typedef nodez *nodep;
typedef struct {
     int back;
     int next;
     } indextype;

/* Function prototypes  */
void getgraph(int argc,char *argv[],
           int  **igraph,int **a, int **b,indextype **sindex,
           int *nn,int *ne,nodez **alist);
void readgraph(int ne,int nn,int igraph[],nodez alist[]);
void greedypart(int costa[],int nn,int ma[],
          int mb[],indextype sindex[],nodez alist[]);
void upaheap(indextype heap[],int i,indextype index[]);
void upbheap(indextype heap[],int i,indextype index[]);
void downaheap(indextype heap[],int i,int nn,indextype index[]);
void downbheap(indextype heap[],int i,int nn,indextype index[]);
void heappart(int costa[],int nn,int ma[],int mb[],nodez alist[]);
float periodt(void);
void remem(int **pa,int nn);
int finda_weight(nodez alist[],int node1,int node2);
void aslightswap(int ma[],int mb[],int costa[],int nn,int *cval,nodez alist[]);
void slightswap(int igraph[],int ma[],int mb[],int costa[],
          int nn,int *cval,nodez alist[]);
void slightestswap(int igraph[],int ma[],int mb[],int costa[],
             int nn,int *cval,nodez alist[]);
void hswap(int igraph[],int ma[],int mb[],int costa[],int nn,
        int *cval,nodez alist[]);

static FILE *f_in;
clock_t oldmtime;
int cand_list_size,big_flag = 1;

void main (int argc,char *argv[])
{
double sigx = 0.0 ,sigx2 = 0.0, sig;
int nn,ne,cval = 0,num_attemp = 0,mincval = 32600;
int *igraph, *ma,*mb,*costa,oneflag = 0;  /* ,*check,x; */
nodez *alist;
indextype *sindex;
clock_t startt;
float matcht = 0,partt= 0,swapt = 0,bestt,ttime,curtime,run_time;

     cand_list_size = atoi(argv[4]);
     if(argv[3][0] == '4')
          big_flag = 0;
     run_time = atof(argv[5]);
     periodt();
     startt = clock();
     srand(32063);
     getgraph(argc,argv,&igraph,&ma,&mb,&sindex,
           &nn,&ne,&alist);
     readgraph(ne,nn,igraph,alist);
     curtime = clock()/CLK_TCK;
     while (curtime < run_time * 2)
          {
          num_attemp++;
          remem(&costa,nn);
          switch (argv[2][0])
               {
               case '1': heappart(costa,nn,ma,mb,alist);
```

```
                partt += periodt();
                break;
          case '2': greedypart(costa,nn,ma,mb,sindex,alist);
                partt += periodt();
                break;
          default : exit(0);
          }
     switch (argv[3][0])
          {
          case '1': hswap(igraph,ma,mb,costa,nn,&cval,alist);
                break;
          case '2': slightswap(igraph,ma,mb,costa,nn,&cval,alist);
                break;
          case '3': slightestswap(igraph,ma,mb,costa,nn,&cval,alist);
                break;
          case '4': aslightswap(ma,mb,costa,nn,&cval,alist);
                break;
          default : hswap(igraph,ma,mb,costa,nn,&cval,alist);
          }
     swapt += periodt();
     free(costa);
     sig = cval;
     sigx += sig;
     sigx2 += (sig * sig);
     if (cval < mincval)
          {
          bestt = ((clock() - startt)/CLK_TCK);
          mincval = cval;
     /*   check = (int *) calloc (nn + 1,sizeof(int));
          for(x = 1; x<= nn / 2; x++)
               {
               check[ma[x]]++;
               check[mb[x]]++;
               }
          for (x = 1; x<= nn; x++)
               printf("\ncheck %d:  %d",x,check[x]);
          printf("\n\n");
          free(check);    */
          }
     cval = 0;
     curtime = clock()/CLK_TCK;
     if (!oneflag && curtime >= run_time)
          {
          oneflag++;
          ttime = ((clock() - startt)/CLK_TCK);
          printf("%d\n",mincval);
          printf("%f\n",ttime);
          printf("%f\n",bestt);
          printf("%f\n",matcht);
          printf("%f\n",partt);
          printf("%f\n",swapt);
          printf("%lg\n",sigx);
          printf("%lg\n",sigx2);
          printf("%d\n",num_attemp);
          }
     }    /* while */
   ttime = ((clock() - startt)/CLK_TCK);
printf("%d\n",mincval);
printf("%f\n",ttime);
printf("%f\n",bestt);
```

```
     printf("%f\n",matcht);
     printf("%f\n",partt);
     printf("%f\n",swapt);
     printf("%lg\n",sigx);
     printf("%lg\n",sigx2);
     printf("%d\n",num_attemp);
     free (alist);
     if (big_flag) free(igraph);
     free (ma); free(mb);
     free (sindex);
}

/*  Inputs nn, ne, and allocates memory for matching.      */
void getgraph(int argc,char *argv[],
           int **igraph,int **ma, int **mb, indextype **sindex,
           int *nn,int *ne,nodez **alist)
{
char inputfile[12];
int nsqar,nn1,nnhalf;

     if (argc < 5)
          {
          puts("HPART: generates graph partition using RAGH");
          puts("Usage: HPART <inputfile> <modea> <modeb> <cl-size> <run-time>");
          puts("modea : 1    - heap greedy unmatched 2-partition");
          puts("        2    - greedy unmatched partition");
          puts("modeb   1    - first swap = 2-exchange");
          puts("        2    - slight swap (gain < 3)");
          puts("        3    - slightest swap (gain == 1)");
          puts("        4    - compact slight swap");
          exit(0);
          }
     strcpy(inputfile,argv[1]);
     f_in = fopen(inputfile,"r");
     if (f_in == NULL)
          {
          puts("Input file not found");
          exit(0);
          }
     fscanf(f_in,"%d,%d,%d",nn,ne);
        nn1 = (*nn)+1;
     nnhalf = (*nn) / 2;
     nsqar = (nn1) * (nn1);
     if(big_flag)
          {
          *igraph = (int *) calloc (nsqar,sizeof(int));
          if (*igraph == NULL)
               {
               puts("Couldn't make nxn array");
               exit(0);
               }
          }
     *ma = (int *) calloc (nnhalf + 1,sizeof(int));
     *mb = (int *) calloc (nnhalf + 1,sizeof(int));
     *sindex = (indextype *) calloc (nn1,sizeof(indextype));
     *alist = (nodez *) calloc (nn1,sizeof(nodez));
     if (*ma == NULL || *mb == NULL || *sindex == NULL || *alist == NULL)
          {
          puts("Couldn't allocate an array");
          exit(0);
          }
}

void readgraph(int ne,int nn,int igraph[],nodez alist[])
{
int x,i,j,weight;
nodez *newi,*newj;
nodep *head;

     head = (nodep *) calloc (nn+1,sizeof(nodep));
     if(head == NULL)
          {
          puts("Couldn't allocate array in readgraph");
          exit(0);
          }
     for (x = 1; x<= nn; x++)
          {
          alist[x].next = NULL;
          head[x] = &alist[x];
          alist[x].node = 0;
          }

     for (x=1;x < ne + 1;x++)
          {
          fscanf(f_in,"%d,%d,%d",&i,&j,&weight);
          /* igraph[i]++; */
          /* igraph[j]++; */
          if (big_flag)
               {
               igraph[ind(i,j,nn)] = weight;
               igraph[ind(j,i,nn)] = weight;
               }
          alist[i].node++;
          alist[j].node++;
          newi = (nodez *) malloc (sizeof(nodez));
          newj = (nodez *) malloc (sizeof(nodez));
          if (newi == NULL || newj == NULL)
               {
               puts("Couldn't allocate node in readgraph");
               exit(0);
               }
          head[i]->next = newi;
          head[j]->next = newj;
          newi->node = j;
          newj->node = i;
          newi->next = NULL;
          newj->next = NULL;
          head[i] = newi;
          head[j] = newj;
          }
     free(head);
}

float periodt()
{
 clock_t marktime,temp;

     temp = oldmtime;
     marktime = clock();
     oldmtime = marktime;
     return    ((marktime-temp)/CLK_TCK);
}

void remem(int **pa,int nn)
{
     *pa = (int *) calloc (nn+1,sizeof(int));
     if (*pa == NULL)
          {
          puts("Couldn't allocate an array");
          exit(0);
          }
}

void greedypart(int costa[],int nn,int ma[],
          int mb[],indextype sindex[],nodez alist[])
{
int nnhalf = nn/2, candid,*i,*j,z,head,point,numcand;
indextype *ind;
nodez *lptr;
register int x,y;
int cand[9],candc[9],lowmax,lowmaxnode,highmin,highminnode;

     for (x = 0,ind = &sindex[1];x <=nn;ind++)
          {
          ind -> back = x;
          ind -> next = ++x + 1;
          }
     sindex[nn].next = 0;
     head = 1;
     for(z=0;z< cand_list_size;z++)
          {
          cand[z] = random(nn)+1;
          candc[z] = 0;
          }
     numcand = cand_list_size;
     for (x = 1,i = &ma[1],j = &mb[1];x <= nnhalf; x++,i++,j++)
          {
     /* Put node from candidate list into set A and update cost */
                if (numcand > cand_list_size)
               numcand = cand_list_size;
          candid = *i = cand[random(numcand)];
          if(candid == head)
               head = sindex[candid].next;
          else
               sindex[sindex[candid].back].next = sindex[candid].next;
          if (sindex[candid].next)
               sindex[sindex[candid].next].back = sindex[candid].back;
          for (y=1,z=alist[candid].node,lptr=alist[candid].next;y<=z;y++)
               {
               costa[lptr ->node]++;
               lptr = lptr ->next;
               }
          numcand = 0;
          for (z = 0; z<cand_list_size; z++)
               candc[z] = 9999;
          highmin = 9999;
          highminnode = 0;
     /* Form candidate list for set B */
          point = head;
          while (point)
```

```
                {
                if(costa[point] < highmin)
                     {
                     numcand++;
                     cand[highminnode] = point;
                     candc[highminnode] = costa[point];
                     highmin = -9999;
                     for (z = 0;z< cand_list_size;z++)
                          if (candc[z] > highmin)
                               {
                               highmin = candc[z];
                               highminnode = z;
                               }
                     }
                point = sindex[point].next;
                }
      /* Put node from candidate list into set B */
           if (numcand > cand_list_size)
                numcand = cand_list_size;
           candid = *j = cand[random(numcand)];
           if(candid == head)
                head = sindex[candid].next;
           else
                sindex[sindex[candid].back].next = sindex[candid].next;
           if(sindex[candid].next)
                sindex[sindex[candid].next].back = sindex[candid].back;
           for (y=1,z=alist[candid].node,lptr=alist[candid].next;y<=z;y++)
                {
                costa[lptr ->node]--;
                lptr = lptr ->next;
                }
           for (z = 0; z < cand_list_size; z++)
                candc[z] = -9999;
           lowmax = -9999;
           lowmaxnode = 0;
           numcand = 0;
      /* Form candidate list for set A */
           point = head;
           while (point)
                {
                if(costa[point] > lowmax)
                     {
                     numcand++;
                     cand[lowmaxnode] = point;
                     candc[lowmaxnode] = costa[point];
                     lowmax = 9999;
                     for (z = 0;z< cand_list_size;z++)
                          if (candc[z] < lowmax)
                               {
                               lowmax = candc[z];
                               lowmaxnode = z;
                               }
                     }
                point = sindex[point].next;
                }
           }
       /*  GREEDY PART */
/*     for(x = 1;x<=nnhalf;x++)
           printf("%d  %d\n",ma[x],mb[x]);
       printf("\n\n"); */
```

```
}

/* The following for functions are used in maintaining the "heap o' gains"
   for the greedy partition. Heapa (the nodes for set A) are sorted
   largest on top and heapb smallest on top, the two pairs of functions are
   only slightly different */
void upaheap(indextype heap[],int i,indextype index[])
{
int tempn,tempa,parent;

     while (i > 1)
          {
          parent = i/2;
          if (heap[i].alpha > heap[parent].alpha)
               {
               index[heap[parent].hnode].apoint = i;
               index[heap[i].hnode].apoint = parent;
               tempn = heap[parent].hnode;
               tempa = heap[parent].alpha;
               heap[parent].hnode = heap[i].hnode;
               heap[parent].alpha = heap[i].alpha;
               heap[i].hnode = tempn;
               heap[i].alpha = tempa;
               i = parent;
               }
          else
               i = 1;
          }
}

void downaheap(indextype heap[],int i,int nn,indextype index[])
{
int child,tempn,tempa;

     while (i < nn)
          {
          child = i << 1;
          if (child+1 <=nn && (heap[child+1].alpha > heap[child].alpha))
               child++;
          if(heap[i].alpha < heap[child].alpha && child <= nn)
               {
               index[heap[child].hnode].apoint = i;
               index[heap[i].hnode].apoint = child;
               tempn = heap[child].hnode;
               tempa = heap[child].alpha;
               heap[child].hnode = heap[i].hnode;
               heap[child].alpha = heap[i].alpha;
               heap[i].hnode = tempn;
               heap[i].alpha = tempa;
               i = child;
               }
          else
               i = nn;
          }
}

void upbheap(indextype heap[],int i,indextype index[])
{
int tempn,tempa,parent;
```

```
     while (i > 1)
          {
          parent = i/2;
          if (heap[i].alpha < heap[parent].alpha)
               {
               index[heap[parent].hnode].bpoint = i;
               index[heap[i].hnode].bpoint = parent;
               tempn = heap[parent].hnode;
               tempa = heap[parent].alpha;
               heap[parent].hnode = heap[i].hnode;
               heap[parent].alpha = heap[i].alpha;
               heap[i].hnode = tempn;
               heap[i].alpha = tempa;
               i = parent;
               }
          else
               i = 1;
          }
}

void downbheap(indextype heap[],int i,int nn,indextype index[])
{
int child,tempn,tempa;

     while (i < nn)
          {
          child = i << 1;
          if (child+1 <=nn && (heap[child+1].alpha < heap[child].alpha))
               child++;
          if(heap[i].alpha > heap[child].alpha && child <= nn)
               {
               index[heap[child].hnode].bpoint = i;
               index[heap[i].hnode].bpoint = child;
               tempn = heap[child].hnode;
               tempa = heap[child].alpha;
               heap[child].hnode = heap[i].hnode;
               heap[child].alpha = heap[i].alpha;
               heap[i].hnode = tempn;
               heap[i].alpha = tempa;
               i = child;
               }
          else
               i = nn;
          }
}

void heappart(int costa[],int nn,int ma[],int mb[],nodez alist[])
{
int nnhalf = nn/2,top,cand,maxa,c,temptop,*i,*j,temp,clist[10],temptr;
int z,y,candp;
register int x,csize;
indextype templist[30],*heapa,*heapb,*index,*ha,*hb,*ind,*tp,*tp2;
nodez *lptr;

     heapa = (indextype *) calloc (nn+1,sizeof(indextype));
     heapb = (indextype *) calloc (nn+1,sizeof(indextype));
     index = (indextype *) calloc (nn+1,sizeof(indextype));
     if (heapa == NULL || heapb == NULL || index == NULL)
          {
          puts("Couldn't allocate array in heappart");
          exit(0);
          }
     for(x = 1,ha = &heapa[1],hb = &heapb[1],ind = &index[1];
                    x <=nn; x++,ha++,hb++,ind++)
          {
          ha->hnode = hb->hnode = x;
          ind->apoint = ind->bpoint = x;
          }

     for (x = 1,i = &ma[1],j = &mb[1];x <= nnhalf; x++,i++,j++)
          {
          /* Find a candidate list from the top of the A heap */
          clist[0] = heapa[1].hnode;
          temptop = 1;
          csize = 0;
          tp = templist;
          for (c = 1;c < cand_list_size;c++)
               {
               if(temptop * 2 < nn)
                    {
                    ha = &heapa[temptop << 1];
                    if(ha->alpha > -9000)
                         {
                         tp->hnode = ha->hnode;
                         tp->alpha = ha->alpha;
                         tp++;
                         csize++;
                         }
                    ha++;
                    if(ha->alpha > -9000)
                         {
                         tp->hnode = ha->hnode;
                         tp->alpha = ha->alpha;
                         tp++;
                         csize++;
                         }
                    maxa = -9999;
                    for(temp = 0,tp2 = templist;temp<csize;
                                        temp++,tp2++)
                         if(tp2->alpha > maxa)
                              {
                              maxa = tp2->alpha;
                              top = tp2->hnode;
                              temptr = temp;
                              }
                    if (maxa == -9999)
                         break;
                    clist[c] = top;
                    temptop = index[top].apoint;
                    templist[temptr].alpha = -9999;
                    }
               else
                    break;
               }
          *i = cand = clist[random(c)];
          candp = index[cand].apoint;
          heapa[candp].alpha = -9999;
          downaheap(heapa,candp,nn,index);
          candp = index[cand].bpoint;
          heapb[candp].alpha = 9999;
          downbheap(heapb,candp,nn,index);
     /* Put node from candidate list into set A and update cost */
          for (y=1,z=alist[cand].node,lptr=alist[cand].next;y<=z;y++)
               {
               costa[lptr ->node]++;
               heapa[index[lptr->node].apoint].alpha++;
               heapb[index[lptr->node].bpoint].alpha++;
               upaheap(heapa,index[lptr->node].apoint,index);
               downbheap(heapb,index[lptr->node].bpoint,nn,index);
               lptr = lptr ->next;
               }
     /* Do the b side */
          clist[0] = heapb[1].hnode;
          temptop = 1;
          csize = 0;
                tp = templist;
          for (c = 1;c < cand_list_size;c++)
               {
               if (temptop * 2 < nn)
                    {
                    hb = &heapb[temptop << 1];
                    if(hb->alpha < 9000)
                         {
                         tp->hnode = hb->hnode;
                         tp->alpha = hb->alpha;
                         tp++;
                         csize++;
                         }
                    hb++;
                    if(hb->alpha < 9000)
                         {
                         tp->hnode = hb->hnode;
                         tp->alpha = hb->alpha;
                         tp++;
                         csize++;
                         }
                    maxa = 9999;
                    for(temp = 0,tp2 = templist;temp<csize;
                                        temp++,tp2++)
                         if(tp2->alpha < maxa)
                              {
                              maxa = tp2->alpha;
                              top = tp2->hnode;
                              temptr = temp;
                              }
                    if (maxa == 9999)
                         break;
                    clist[c] = top;
                    temptop = index[top].bpoint;
                    templist[temptr].alpha = 9999;
                    }
               else
                    break;
               }
          *j = cand = clist[random(c)];
          candp = index[cand].apoint;
          heapa[candp].alpha = -9999;
          downaheap(heapa,candp,nn,index);
          candp = index[cand].bpoint;
          heapb[candp].alpha = 9999;
```

```
          downbheap(heapb,candp,nn,index);
     /* Put node from candidate list into set A and update cost */
          for (y=1,z=alist[cand].node,lptr=alist[cand].next;y<=z;y++)
               {
               costa[lptr->node]--;
               heapa[index[lptr->node].apoint].alpha--;
               heapb[index[lptr->node].bpoint].alpha--;
               downaheap(heapa,index[lptr->node].apoint,nn,index);
               upbheap(heapb,index[lptr->node].bpoint,index);
               lptr = lptr->next;
               }
          }
          free (heapa);
          free (heapb);
          free (index);
      /*  HEAP GREEDY PART */
/*   for(x = 1;x<=nnhalf;x++)
          printf("%d  %d\n",ma[x],mb[x]);
      printf("\n\n"); */
}

/* Find out if two nodes are adjacent; assumes input was ordered */
int finda_weight(nodez alist[],int node1,int node2)
{
register int x,z;
nodez *lptr;

     for(x=1,z=alist[node1].node,lptr = alist[node1].next;
         x<=z && lptr->node <= node2;x++)
          {
          if(lptr->node == node2)
               return(1);
          lptr = lptr->next;
          }
     return(0);
}

/* Find a slight (gain < 3) positive swap and then perform it, using
   adjacency list */
void aslightswap(int ma[],int mb[],int costa[],int nn,int *cval,nodez alist[])
{
int nnhalf = nn / 2,worst,temp,found,gain,worstswap;
int *i, *j, *worstx, *worsty,z;
register int x,y;
nodez *lptr;

     do
         {
         x = 1;  i = &ma[1];
         worstswap =5000;
         found = 0;
         worst = 0;
         while (x <= nnhalf && !worst)
          {
          if (costa[*i] < 0)
               {
               y = 1; j = &mb[1];
               while (y <= nnhalf && !worst)
                    {
                    gain = costa[*j] - costa[*i]
```

```
                        -(finda_weight(alist,*i,*j)<<1);
                   if (gain > 0 && gain < worstswap)
                        {
                        if (gain < 3)
                             worst++;
                        found++;
                        worstswap = gain;
                        worstx = i;
                        worsty = j;
                        }
                   y++; j++;
                   }
              }
         x++; i++;
         }
        y = 1; j = &mb[1];
        while (y <= nnhalf && !worst)
         {
         if (costa[*j] > 0)
              {
              x = 1; i = &ma[1];
              while (x <= nnhalf && !worst)
                   {
                   gain = costa[*j] - costa[*i]
                        -(finda_weight(alist,*i,*j)<<1);
                   if (gain > 0 && gain< worstswap)
                        {
                        if (gain <3)
                             worst++;
                        found++;
                        worstswap = gain;
                        worstx = i;
                        worsty = j;
                        }
                   x++;
                   i++;
                   }
              }
         y++;
         j++;
         }
        if(found)
         {
         temp = *worstx;
         *worstx = *worsty;
         *worsty = temp;
         for(x=1,z=alist[*worstx].node,lptr=alist[*worstx].next;x<=z;x++)
              {
              costa[lptr ->node] += 2;
              lptr = lptr->next;
              }
         for(x=1,z=alist[*worsty].node,lptr=alist[*worsty].next;x<=z;x++)
              {
              costa[lptr ->node] -= 2;
              lptr = lptr->next;
              }
         }
        } while (found);
   *cval = 0;
   for (x = 1,i= &ma[1];x <= nnhalf; x++,i++)
```

```
     for (y = 1, j= &mb[1];y<= nnhalf; y++,j++)
          *cval += finda_weight(alist,*i,*j);
     /* printf("Cross value after SLIGHT swap is %d\n",*cval); */
}



/* New improved generic two-exchange, with more postprocessing power!
   This is a "first swap" postprocessor with some improvements -- it looks
   first at those exchanges that are most likely to have a postive gain.
   The idea is to avoid having to look throught the whole list (n^2) */
void hswap(int igraph[],int ma[],int mb[],
        int costa[],int nn,int *cval,nodez alist[])
{
int nnhalf = nn / 2,temp,found,gain;
register int x,y,z;
int *i, *j;
nodez *lptr;

     do
         {
         found = 0;
         x = 1; i = &ma[1];
         while (!found && (x <= nnhalf))
          {
          if (costa[*i] < 0)
               {
               y = 1; j = &mb[1];
               while (!found && (y <= nnhalf))
                    {
                    gain =  costa[*j] - costa[*i]
                         - ((igraph[ind(*i,*j,nn)]) << 1);
                    if (gain > 0)
                         found++;
                    else
                         {
                         y++;
                         j++;
                         }
                    }
               if (!found)
                    {
                    x++;
                    i++;
                    }
               }
          else
               {
               x++;
               i++;
               }
          }
         if (!found)
          {
          y = 1;
          j = &mb[1];
          }
         while (!found && (y <= nnhalf))
          {
          if (costa[*j] > 0)
```

```
                {
                x = 1; i = &ma[1];
                while (!found && (x <= nnhalf))
                     {
                     gain =  costa[*j] - costa[*i]
                          - ((igraph[ind(*i,*j,nn)]) << 1);
                     if (gain > 0)
                          found++;
                     else
                          {
                          x++;
                          i++;
                          }
                     }
                if (!found)
                     {
                     y++;
                     j++;
                     }
                }
           else
                {
                y++;
                j++;
                }
           }
          if(found)
           {
           temp = *i;
           *i = *j;
           *j = temp;
                 for(x=1,z=alist[*i].node,lptr = alist[*i].next;x<=z;x++)
                {
                costa[lptr ->node] += 2;
                lptr = lptr->next;
                }
           for(x=1,z=alist[*j].node,lptr=alist[*j].next;x<=z;x++)
                {
                costa[lptr ->node] -= 2;
                lptr = lptr->next;
                }
           }
          } while (found);
      /* printf("Cross value after HAL swap is %d\n",*cval); */
     *cval = 0;
     for (x = 1,i= &ma[1];x <= nnhalf; x++,i++)
      for (y = 1,j= &mb[1];y<= nnhalf; y++,j++)
           *cval += igraph[ind(*i,*j,nn)];

}

/* Find a slight (gain < 3) positive swap and then perform it */
void slightswap(int igraph[],int ma[],int mb[],int costa[],int nn,int *cval,nodez
alist[])
{
int nnhalf = nn / 2,worst,temp,found,gain,worstswap;
int *i, *j, *worstx, *worsty,z;
register int x,y;
nodez *lptr;
```

```
do
    {
    x = 1;  i = &ma[1];
    worstswap =5000;
    found = 0;
    worst = 0;
    while (x <= nnhalf && !worst)
     {
     if (costa[*i] < 0)
          {
          y = 1; j = &mb[1];
          while (y <= nnhalf && !worst)
               {
               gain = costa[*j] - costa[*i]
                    - ((igraph[ind(*i,*j,nn)])<<1);
               if (gain > 0 && gain < worstswap)
                    {
                    if (gain < 3)
                         worst++;
                    found++;
                    worstswap = gain;
                    worstx = i;
                    worsty = j;
                    }
               y++; j++;
               }
          }
     x++; i++;
     }
    y = 1; j = &mb[1];
    while (y <= nnhalf && !worst)
     {
     if (costa[*j] > 0)
          {
          x = 1; i = &ma[1];
          while (x <= nnhalf && !worst)
               {
               gain = costa[*j] - costa[*i]
                    - ((igraph[ind(*i,*j,nn)])<<1);
               if (gain > 0 && gain< worstswap)
                    {
                    if (gain <3)
                         worst++;
                    found++;
                    worstswap = gain;
                    worstx = i;
                    worsty = j;
                    }
               x++;
               i++;
               }
          }
     y++;
     j++;
     }
    if(found)
     {
     temp = *worstx;
     *worstx = *worsty;
     *worsty = temp;
```

```
          for(x=1,z=alist[*worstx].node,lptr=alist[*worstx].next;x<=z;x++)
               {
               costa[lptr ->node] += 2;
               lptr = lptr->next;
               }
          for(x=1,z=alist[*worsty].node,lptr=alist[*worsty].next;x<=z;x++)
               {
               costa[lptr ->node] -= 2;
               lptr = lptr->next;
               }
          }
         } while (found);
    *cval = 0;
    for (x = 1,i= &ma[1];x <= nnhalf; x++,i++)
     for (y = 1, j= &mb[1];y<= nnhalf; y++,j++)
          *cval += igraph[ind(*i,*j,nn)];
     /* printf("Cross value after SLIGHT swap is %d\n",*cval); */
}

/* Find the slightest (gain == 1) positive swap and then perform it */
void slightestswap(int igraph[],int ma[],int mb[],
             int costa[],int nn,int *cval,nodez alist[])
{
int nnhalf = nn / 2,worst,temp,found,gain,worstswap;
int *i, *j, *worstx, *worsty,z;
register int x,y;
nodez *lptr;

     do
         {
         x = 1;  i = &ma[1];
         worstswap =5000;
         found = 0;
         worst = 0;
         while (x <= nnhalf && !worst)
          {
          if (costa[*i] < 0)
               {
               y = 1; j = &mb[1];
               while (y <= nnhalf && !worst)
                    {
                    gain = costa[*j] - costa[*i]
                         - ((igraph[ind(*i,*j,nn)])<<1);
                    if (gain > 0 && gain < worstswap)
                         {
                         if (gain == 1)
                              worst++;
                         found++;
                         worstswap = gain;
                         worstx = i;
                         worsty = j;
                         }
                    y++; j++;
                    }
               }
          x++; i++;
          }
         y = 1; j = &mb[1];
         while (y <= nnhalf && !worst)
          {
```

```
          if (costa[*j] > 0)
               {
               x = 1; i = &ma[1];
               while (x <= nnhalf && !worst)
                    {
                    gain = costa[*j] - costa[*i]
                         - ((igraph[ind(*i,*j,nn)])<<1);
                    if (gain > 0 && gain< worstswap)
                         {
                         if (gain == 1)
                              worst++;
                         found++;
                         worstswap = gain;
                         worstx = i;
                         worsty = j;
                         }
                    x++;
                    i++;
                    }
               }
          y++;
          j++;
          }
         if(found)
          {
          temp = *worstx;
          *worstx = *worsty;
          *worsty = temp;
          for(x=1,z=alist[*worstx].node,lptr=alist[*worstx].next;x<=z;x++)
               {
               costa[lptr ->node] += 2;
               lptr = lptr->next;
               }
          for(x=1,z=alist[*worsty].node,lptr=alist[*worsty].next;x<=z;x++)
               {
               costa[lptr ->node] -= 2;
               lptr = lptr->next;
               }
          }
         } while (found);
    *cval = 0;
    for (x = 1,i= &ma[1];x <= nnhalf; x++,i++)
     for (y = 1, j= &mb[1];y<= nnhalf; y++,j++)
          *cval += igraph[ind(*i,*j,nn)];
     /* printf("Cross value after SLIGHTEST swap is %d\n",*cval); */
}
```

6. References

[AHO83] A. V. Aho, J. E. Hopcroft, J. D. Ullman, **Data Structures and Algorithms**, Addison-Wesley, Reading, MA, 1983.

[BAL83] M. Ball and L. Bodin, *A Matching Based Heuristic for Scheduling Mass Transit Crews and Vehicles*, **Transportation Science**, Vol. 17, 1985.

[BAR89] J. Bard and T. Feo, *Operations Sequencing in Discrete Parts Manufacturing*, **Management Science**, Vol. 35, 1989.

[CHE86] C. C. Chen, *Placement and Partitioning Methods for Integrated Circuit Layout*, Ph.D. Dissertation, Department of EECS, The University of California, Berkeley, 1986.

[FEO88] T. A. Feo, O. Goldschmidt, M. Khellaf, *1/2 - Approximation Algorithms for the k-Partition Problem* , Technical Report, Operations Research Group, Mechanical Engineering Department, University of Texas at Austin, 1988.

[FEO89a] T. Feo and J. Bard, *The Airline Scheduling and Maintenance Base Location Problem*, to appear in **Management Science**.

[FEO89b] T. Feo and J. McGahan, *Scheduling Jobs with Linear Delay Penalties and Sequence Dependent Setup Costs Using GRASP*, Technical Report, Operations Research Group, Mechanical Engineering Department, University of Texas at Austin, 1989.

[FEO89c] T. Feo and M. Resende, *A Probabilistic Heuristic for a Computationally Difficult Set Covering Problem*, **Operations Research Letters**, Vol. 8, 1989.

[FEO89d] T. Feo and S. Smith, *An Efficient Randomized Heuristic for Coloring Graphs*, Technical Report, Operations Research Group, Mechanical Engineering Department, University of Texas at Austin, 1989.

[FEO89e] T. Feo, M. Resende, and S. Smith, *A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set*, Technical Report, Operations Research Group, Mechanical Engineering Department, University of Texas at Austin, 1989.

[FEO89f] T. Feo, K. Venkatraman, J. Bard, *A GRASP on Single Machine Scheduling with Due Dates and Earliness Penalties*, Technical Report, Operations Research Group, Mechanical Engineering Department, University of Texas at Austin, 1989.

[FID82] C. M. Fiduccia and R. M. Mattheyses, *A Linear-Time Heuristic for Improving Network Partitions*. **Proceedings of the 19th Design Automation Conference**, Las Vegas, 1982.

[GAR79] M. R. Garey, D. S. Johnson, **Computers and Intractability: A Guide to the Theory of NP-Completeness**, W. H. Freeman, San Francisco, 1979.

[HAR87] J. Hart and A. Shogan, *Semi-Greedy Heuristics: An Empirical Study*, **Operations Research Letters**, Vol. 6, 1987.

[JOH89] D. S. Johnson, C. R. Aragon, L. A. McGeoch, C. Schevon, *Optimization by Simulated Annealing: An Experimental Evaluation (Part I)*, Technical Report, AT&T Bell Labs, Murray Hill, N.J., 1989.

[KHE87] M. Khellaf, *On the Partitioning of Graphs and Hypergraphs*, Ph.D. Dissertation, Engineering Science Department, The University of California, Berkeley, 1987.

[KER70] B. W. Kernighan,S. Lin, *An efficient heuristic procedure for partitioning graphs*, **Bell System Technical Journal**, Vol. 49, No. 2, 1970.

[KRA65] J. Kral, *To the problem of Segmentation of a Program*, **Information Processing Machines**, 1965.

[MAC78] R. M. MacGregor, *On Partitioning a graph: A heuristical and empirical study*, Memorandum No. UCB/ERL M78/14, Electronics Research Laboratory, University of California, Berkeley, 1978.

[MIT86] D. Mitra, R. Fabio, A. Sangiovanni-Vincentelli, *Convergence and Finite-time Behavior of Simulated Annealing*, **Advances in Applied Probability**, Vol 18, 1986.

[SAS88] G. H. Sasaki, B. Hajek, *The Time Complexity of Maximum Matching by Simulated Annealing*, **Journal of the Association for Computing Machinery**, Vol 35, 1988.

Vita

Hal Connor Elrod was born on November 22, 1965 in Portland, Oregon. Sometime later he graduated from Baylor University with a Bachelor of Arts in Economics and Computer Science in August of 1989. In that same month he was married. It is not surprising that in September of that year, confused and disoriented, he accidentally entered The Graduate School of The University of Texas. It was there he also was employed as a teaching assistant in engineering economics while at the same time employed as a software technician by Advanced Micro Devices.

Permanent address: 1842 Golden Way
Mountain View, California 94040

This thesis was typed by Hal Connor Elrod