Finding Optimal Solutions to the Graph Partitioning Problem with Heuristic Search

Ariel Felner
Department of Information Systems Engineering,
Ben-Gurion University of the Negev
Beer-Sheva, 85104, Israel
Email: felner@bgu.ac.il

Abstract

As search spaces become larger and as problems scale up, an efficient way to speed up the search is to use a more accurate heuristic function. A better heuristic function might be obtained by the following general idea. Many problems can be divided into a set of subproblems and subgoals that should be achieved. Interactions and conflicts between unsolved subgoals of the problem might provide useful knowledge which could be used to construct an informed heuristic function. In this paper we demonstrate this idea on the graph partitioning problem (GPP). We first show how to format GPP as a search problem and then introduce a sequence of admissible heuristic functions estimating the size of the optimal partition by looking into different interactions between vertices of the graph. We then optimally solve GPP with these heuristics. Experimental results show that our advanced heuristics achieve a speedup of up to a number of orders of magnitude. Finally, we experimentally compare our approach to other states of the art graph partitioning optimal solvers on a number of classes of graphs. The results obtained show that our algorithm outperforms them in many cases.

1 Introduction and overview

The A* algorithm [14] is an admissible best-first algorithm which uses a cost function of the form $f(n) = g(n) + h(n)$, where $g(n)$ is the actual distance from the initial state to the current state and $h(n)$ is a heuristic estimation of the cost from node n to the nearest goal. If $h(n)$ never overestimates the actual cost from node n to a goal then we say that $h(n)$ is *admissible*. When using an admissible heuristic $h(n)$, A* was proved to be admissible, complete and optimally effective [7][1] The main weakness

[1]We say that an algorithm is *admissible* if it is guarantied to return an optimal solution. An algorithm is *complete* if it is guarantied to return a solution if one exists. *Optimally effective* means that its time complexity is a lower bound.

of A* is its exponential memory complexity [24] - A* usually exhausts the available memory very quickly.

1.1 Linear space search algorithms

Iterative-deepening A* (IDA*) [24] is a variant of A* which solves this memory problem. The memory complexity of IDA* is linear in the maximum search depth, rather than exponential as in A*. IDA* is actually a sequence of iterations of DFS. In each iteration it expands all the nodes having a total cost not exceeding a given global threshold for that iteration. The threshold for the first iteration is the cost of the root of the search tree. The threshold for the next iteration is the lowest cost of a generated node from the current iteration that exceeded the current threshold. The algorithm halts when a goal node is chosen for expansion. When using an admissible heuristic, IDA* generates new nodes in a best-first order. In this paper we use this algorithm to solve the graph partitioning problem.

Another algorithm used in this paper is Depth-First Branch and Bound (DFBnB) [40]. DFBnB is effective when the depth of the solution in the search tree is known in advance. DFBnB performs a depth-first search of the tree but uses a global threshold for pruning nodes. This threshold is the best solution that has been found so far. At the beginning of the search, this threshold is set to infinity and DFBnB arrives at a solution in one dive into the tree. The threshold is set to the cost of that solution and the depth-first search process is continued until a better solution is found. When we arrive at a node where the cost function is greater than the current threshold we know that solutions under that node will not be better than the best solution and the subtree under that node can then be pruned. DFBnB begins with a solution and improves it until it attains the optimal solution. It then keeps searching the tree to verify that a better solution is not feasible. Thus, in that sense, DFBnB is an anytime algorithm.

It was proven that an admissible search algorithm must visit at least all the nodes visited by A* [7]. IDA* does not visit any other node but visits some nodes more than once. DFBnB never visits any node more than once but visits many nodes not visited by A*. See [40] for more comparisons between these two algorithms.

1.2 Other memory efficient search algorithms

After IDA* and DFBnB were introduced, memory constraints were no longer a problem since both algorithms use a depth-first search mechanism for traversing the search tree and therefore use memory which is only linear in the depth of the search. As computer memories became larger one approach taken by many researchers was to develop better search algorithms by taking advantage of the large memory available. MREC [37], MA* [4] and SMA* [35] are all based on A* but differ in the ways they proceed when the memory is full. They all continue to execute A* until the memory is almost full. When the memory is full, MREC begins to execute IDA* on the leaves of the search tree which is currently stored in memory. MA* and SMA*, on the other hand, delete bad nodes from the open-list thus freeing memory for new nodes. The strength of these approaches is that these algorithms are general and can be applied to almost any domain. Their weakness is that they are more complicated than IDA* and that their improvement over IDA* is somewhat modest.

1.3 Improving the heuristic function

A somewhat better approach to speed up the search is to look for a more accurate admissible heuristic function where the search would be guided faster towards the goal state and a smaller number of nodes will be generated. However, in order to obtain a more accurate heuristic function, a larger overhead will be caused by each node, resulting in a larger constant time per node. In general, the smaller number of generated nodes tends to compensate for the larger overhead resulting in a significant improvement in the running time.

Several methods were developed to attain better heuristic functions by storing large tables in memory. Both Perimeter Search (PS) [8] and BIDA* [28] are algorithms that store a large table in memory containing all the nodes surrounding the goal node up to a fixed depth. These nodes are called the *perimeter nodes.* The heuristic function used is the estimation of the distance between a node n and the closest node to it in the perimeter. Adding this amount to $g(n)$ and to the depth of the perimeter results in a more accurate heuristic in comparison to the regular heuristic to the goal.

Another method for intelligent usage of memory in order to attain a better heuristic is the *pattern database* method which was first used to solve the Fifteen Tile puzzle in [6] and Rubik's cube in [25]. This method was taken further in [27] and then in [26, 11] where a number of disjoint sets of pattern databases were used to solve the Fifteen and Twenty-Four Puzzles. Pattern databases have also made it possible to significantly shorten the length of solutions constructed using a macro-table [16] and have proven useful in heuristic-guided planning [9].

In a pattern database system we group together a number of subgoals such as individual tiles in the tile puzzles or individual cubies in Rubik's cube. We then calculate the costs of solving this set of subgoals for all their possible combinations in any possible state of the problem without looking at other subgoals. These values contain the costs of solving the specific set of subgoals while considering interactions and conflicts between them. The costs are stored in a lookup table consulted during the search. The cost of solving a set of subgoals is used as a lower bound estimation on the optimal distance to the goal of the complete state which also includes all the other subgoals. In general, experimental results of all these heuristics show an impressive reduction in the number of generated nodes and in the actual running time when comparing them to simple heuristics such as *Manhattan distance* in the tile puzzle. The reduction was sometimes a number of orders of magnitude. The weakness of these approaches is that they are domain dependent and also that a large amount of memory is needed to store these large databases.

An important discussion about new methods for improving search algorithms was presented in [20]. The authors conclude that using domain specific enhancements is usually superior in performance over domain independent enhancements. They show empirical evidence from the game of Sokoban to support their claims and indeed show many domain specific enhancements for that game which greatly reduce the search effort. The challenge would be to introduce search enhancements that are as general as possible and thus might be implemented in many domains. This generality does not necessarily mean that this method can be activated for every domain with no domain specific adjustments. It does mean that one has to follow this general idea when implementing it but must also consider domain-specific constraints and behavior in order to adjust this idea to the specific domain in question so as to achieve an effective enhancement.

In this work we show how general complicated heuristics can be applied to the graph-partitioning problem. A general idea for improving on simple heuristics given that there exist a number of subgoals to achieve in the search task is the following. While simple heuristics (e.g., the Manhattan distance for the tile puzzles) consider each individual subgoal alone, complicated heuristics look deeper into interactions and conflicts between a number of unsolved subgoals and use this knowledge to attain a more accurate heuristic function. This idea of calculating conflicts and interactions between unsolved subgoals is one of the advantages of the pattern database systems described above. We used this general idea when constructing our heuristics for the graph partitioning problem and we will show our work in this paper.

The main contribution in our approach is twofold. First, we present an optimal solver for the graph partitioning problem which outperforms the best existing solvers in many cases. Second, we show how to apply search techniques from Artificial Intelligence to successfully solve a problem from another area of computer science.

This paper is organized as follows. Section 2 introduces the graph partitioning problem and discusses related work. In sections 3, 4 and 5 we describe our search algorithm and our new heuristic functions. Section 6 presents experimental results that show the benefit of using our advanced heuristic functions. Section 7 experimentally compares our approach to other state of the art graph partitioning optimal solvers. Section 8 presents conclusions and future work.

2 The graph partitioning problem

In a graph partitioning problem (GPP), we are given a graph with an even number of vertices and must divide all the vertices of the graph into two equal-sized subsets, so that the number of edges from any vertex in one subset to one in the other is minimized. This problem has many practical applications such as compiler design, load balancing and the physical layout of VLSI circuits where the vertices are circuit elements and the edges are wires connecting them. In the simple form of this problem, we assume that all edges have a uniform weight of one.

To solve this problem using heuristic search, we form a search tree where each node corresponds to a partial partition of the vertices of the graph.[2] We order the vertices in the graph in some order, and then search a binary tree where each level of the tree corresponds to a different vertex. At each node of the search tree, the left branch would assign the corresponding graph vertex to one subset and the right branch would assign it to the other subset. If the number of vertices in the graph is n then this tree is a binary tree of depth $n-1$. While leaves of such a binary tree correspond to the 2^n different

[2] Throughout this work we will use the term *vertex* to denote a vertex of the input graph and the term *node* to denote a node of the search tree.

subsets of n, we are only interested in the leaves that partition the vertices into equal size subsets. Therefore, each subtree rooted by a node where one of the partitions has more than half of the vertices is pruned.

At any given node k of the search tree, corresponding to a partial solution, some of the vertices have already been assigned and some not. As a result, some edges of the graph already go from a vertex in one subset to a vertex in the other. We call the number of such edges $g(k)$, since they represent the cost already incurred by the partial solution represented by node k. In the following sections we present a series of admissible heuristics to get a lower-bound estimate on the remaining number of edges that must cross the partition, given the partial partition. We call this number $h(k)$. Each of these heuristics looks more deeply than its predecessor into interactions and conflicts between unassigned vertices and on edges that connect such vertices. We can then use any admissible search algorithm with the cost function of $f(k) = g(k) + h(k)$ in order to search the tree and find an optimal solution. We will show that even though the overhead in time per node for calculating more complicated heuristics is greater, we get a large improvement in the number of generated nodes, and thus the overall time needed to solve a problem is significantly reduced. We then compare our approach to other existing GPP optimal solvers and show that in many cases we outperform them.

2.1 Related work

2.1.1 Suboptimal solution

Since the GPP is known to be NP-hard [13], most of the algorithms that solve it were developed to find a sub-optimal solution to the problem. A large portion of these algorithms are based on local search. In a local search, we begin with any feasible solution dividing the vertices into two subsets. Then, we start swapping pairs of vertices between the subsets as long as the cost of the solution decreases. This local search may get caught up in a local optimum and therefore does not guarantee an optimal solution. Many algorithms were developed in order to improve this local search schema. These algorithms differ in the way they generate the neighbors of a given feasible partition.

Perhaps the most popular of these improvements is the KL algorithm [23]. It utilizes the fact that some vertices are more strongly connected than others and it swaps groups of vertices instead of just a pair of vertices between the two partitions. With this method there is a potential of escaping some local optimum.

Another well known improvement to the above local search schema was developed in [18]. This work introduces an algorithm for solving the GPP based on simulated annealing (SA). SA is a stochastic optimization algorithm which starts with a partitioning and improves it by using a probabilistic hill-climbing strategy. Several studies were conducted using the concept of genetic algorithms (GA). In these works there is a large population of different partitionings. New generations are evolved from the current population by using genetic algorithms techniques. The most recent approach for genetic algorithms is presented in [3] which also summarizes previous genetic algorithm approaches. An approach to solve the GPP with the help of a learning automata was presented in [29].

Another known algorithm is the Extended Local Search algorithm (XLS) [32].

While in most local searches a vertex can only be swapped once between the two partitions, XLS allows a vertex to be swapped more than once. In this way, previous bad decisions can be corrected and we can escape local optimum more effectively.

All the above algorithms are based on local search and their search space is actually all the different complete partitionings. In this sense, their search space is limited to the leaves of the tree that was presented above. They provide different methods of how to move between these leaves.

A somewhat different and new approach was presented in [34]. This is an efficient programming technique for solving the GPP, based on Lagrangian relaxation and subgradient search. This mathematical programming technique offers an advantage because it produces a lower bound on the solutions. This bound information is useful as we are able to evaluate the quality of any given solution at any time.

GPP is very important from a parallel computational point of view because a balanced partitioning minimizing the crossing edges corresponds to low communication overhead. Several research groups in mathematic and computer science have built entire systems for solving these problems. There are a number of competing tools such as Chaco [15], Jostle [39], Metis [22], Scotch [31] and PARTY [33]. Most of these solvers apply parallel search and lower bounds based on integer program relaxation as well as other heuristic approaches. We refer the reader to [36] which has a comprehensive survey on these methods and tools. This survey divides these heuristics into the following classes: Geometric Scheme, Spectral Methods, Combinatorial Schemes, Multilevel Schemes, Dynamic Repartitioners and Parallel Graph Partitioners.

These tools and algorithms are designed to find suboptimal solutions and are usually used to partition very large graphs. Because of the practical nature of these algorithms, many of them are tailored to partition special classes of graphs and their behavior will degrade on another class of graphs or on random graphs. Most of these tools are usually designed to work on parallel processors. For example, extremely large graphs (over 0.5 billion vertices) have been partitioned on machines consisting of thousands of processors. For more details see [36].

2.1.2 Optimal solutions

While the above algorithms and tools are somewhat related to our work we present a completely different approach as we want to find optimal solution to the GPP and thus can only handle graphs of small size since the problem is known to be NP-hard. In order to find an optimal solution to an NP-hard problem a systematic global search must be used. We use such a schema but develop methods to prune a large portion of the search space with the help of our new heuristics. Unlike some of the methods and tools described above, our algorithm is general and is not restricted to a specific type of graph.

A number of different approaches for obtaining optimal solutions to the GPP have been presented. An exact solution to this problem based on branch-and-cut and linear programming relaxation or on other methodologies such as column generation or bounding functions were proposed [2, 12, 19, 5]. However, all these exact solutions are usually limited to somewhat small graphs of size 60 or smaller.

Recently, more approaches that optimally solve this problem for larger graphs were presented. In 2000, Karisch et al. [21] presented a branch-and-cut method for the general graph partitioning problem. Their method is based on a cutting plane approach combining semidefinite and polyhedral relaxations. Their search tree is similar to our search tree described above. However, they cut irrelevant parts of the tree by using the cutting plane approach which identifies violating inequalities.

In 2001, Sensen [38] presented an algorithm for finding exact solutions to GPP using Multicommodity Flows. This approach first finds a lower bound for the problem. The lower bound is based on the well known lower bound of embedding a clique into the given graph with minimal congestion which is equivalent to a multicommodity flow problem where each vertex sends a commodity of size one to every other vertex. Then, this bound is used to find a feasible solution by a branch and bound algorithm.

Since these are the state of the art optimal GPP solvers, we compare their results to our approach and present experimental evidence to show that our approach is a strong competitor to these approaches. We will present experimental results showing that our method outperformed these methods on many graphs.

3 The different cost functions

We have already defined the search tree above in section 2. In each node of the search tree some of the vertices of the graph were already assigned to one of the sets of the partition and some were not. In order to find an optimal solution, we need an admissible cost function for that node. This cost function should be a lower bound on the number of edges that must cross the partition given the partial partition of the node. Below, we propose several methods that suggest such cost functions.

We also give a mathematical analysis of the time complexity of calculating the different heuristics. Note that for the search algorithms we use either IDA* or DFBnB described above. Both algorithms use a depth-first search mechanism. Thus, in each generated node we only need to make the necessary changes from its parent and undo these changes when backtracking from that node.

3.1 Definitions

For simplicity, throughout the discussion we will assume that the edges of the input graph have uniform weights of 1. We will then show how to apply the presented techniques to a weighted graph.

- Let n denote the number of vertices in the input graph.
- We will use x to denote a vertex in the graph and k to denote a node in the search tree.
- For each node of the search tree we call the vertices that were already assigned to one of the parts of the partition the *assigned* vertices. Vertices that are have not been assigned yet are called the *free* vertices.
- We denote the two sets of the partition that are being build up during the search A and B. Vertices in A and B are the assigned vertices.

- Within each node of the search tree some of the vertices are free. Each of the suggested heuristic functions below, divides the free vertices into two sets - A' and B'. These sets will respectively complete sets A and B of the assigned vertices in the final partitioning. In other words, $\{A \cup A'\}$ and $\{B \cup B'\}$ are the two sets of the final partitioning).
- For each node of the search tree we can divide the edges of the graph into four types:
  1. Type I: Edges within $\{A \cup A'\}$ or within $\{B \cup B'\}$. These edges do not cross the partition and therefore will not be counted by any heuristic.
  2. Type II: Edges from A to B. These edges are already crossing the partition.
  3. Type III: Edges from a free vertex to an assigned vertex of the opposite set, i.e., either edges from A' to B or from B' to A.
  4. Type IV: Edges between two free vertices from opposite sets, i.e., edges from A' to B'.

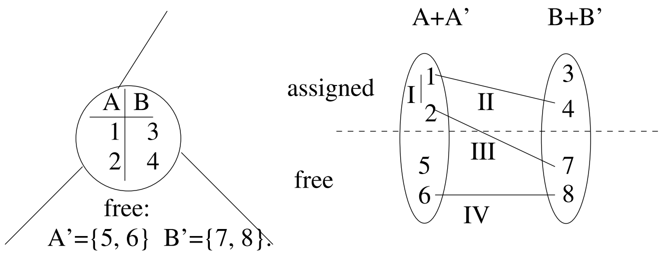

Figure 1: The partition and different edge types.

- For each free vertex x we define $d(x, A)$ and $d(x, B)$ to be the number of edges connected between x and vertices from A or B, respectively. If we place a free vertex x in A' then we know that $d(x, B)$ edges that connect x to vertices in B will cross the partition.

Figure 1 illustrates a node of the search tree. The search tree assigns vertices 1 and 2 to A and vertices 3 and 4 to B. The heuristic function then completes the partitioning by placing 5 and 6 in A' and putting 7 and 8 in B'. Therefore the edge $(1, 2)$ is of type I, $(1, 4)$ is of type II, $(2, 7)$ is of type III while edge $(6, 8)$ is of type IV. As the heuristics that we describe below become more complicated, they include more types of edges.

3.2 f_0: Uniform cost search.

The first cost function used, mainly for comparison reasons, is the trivial cost function. If k is a node in the search tree let $g(k)$ be the number of edges crossing the partial partitioning associated with that node. $g(k)$ is therefore the number of edges between vertices that are assigned to different parts of the partial partitioning or actually edges of type II. Thus we define:

$$f_0(k) = g(k).$$

This trivial cost function is usually called uniform cost search (UCS) since it only evaluates the steps that have already been taken for the current node. We call this cost function f_0 since it does not have any heuristic evaluation on any of the free vertices. f_0 is simple to compute but is also inefficient.

3.2.1 Computational complexity of f_0

For the purpose of the computational complexity analysis of the different cost functions we make the following definitions.

- We are generating a new node k where a free vertex z is assigned (without loss of generality)to the A subset of the partition.
- Let d be the degree of z, i.e., the total number of neighbors of z.
- Let r denote the number of free vertices at a given point. Note that during the course of the search r decreases from n to 0.
- Let d_r, d_A and d_B be the number of neighbors of z that are currently free, and already assign to A and B respectively. Note that $d_r + d_A + d_B = d$.

To calculate f_0 we need to maintain a counter that keeps the number of crossing edges at all times. For the root node, this number is initialized to be 0. The overhead for each new node k is as follows. We need to loop over all the d neighbors of z and increment the counter for each neighbor on the other side of the partition, B. We do this d_B times. If we save the value of d_B in that node then when backtracking from this node we can decrement the counter by d_B at a constant time c_m. Let c_q be the time taken to query each neighbor and let c_0 be the time taken to increment the counter. The complexity for f_0 is:

$$T(f_0) = c_q d + c_0 d_B + c_m = O(d)$$

3.3 f_1: Edges from free to assigned vertices.

For each free vertex x, we defined $d(x, A)$ and $d(x, B)$ as the number of edges from x to vertices already assigned to A or to B. If x is assigned to A then $d(x, B)$ more edges will cross the partition. If x is assigned to B then the number will be $d(x, A)$. An admissible heuristic for x would therefore be

$$h_1(x) = min(d(x, A), d(x, B)).$$

This number is a safe lower bound on the number of edges that will cross the partition due to x. Summing $h_1(x)$ for all free vertices will yield an admissible heuristic on the number of edges that must cross the partition. Therefore, for each node in the search tree, k, we define:

$$h_1(k) = \sum_{x \in free(k)} h_1(x).$$

Thus, the corresponding cost function that we use is

$$f_1(k) = g(k) + h_1(k).$$

h_1 assigns each free vertex either to A' or to B' by looking into potential edges of type III. The h_1 heuristic only looks on each free vertex alone and is not concerned with the interaction between any two free vertices. In this sense, h_1 can be associated with the *Manhattan distance* in the tile puzzle domain. The Manhattan distance heuristic, also, counts the number of positions that a tile is away from its goal position and does not look at interactions and conflicts between two tiles that are not in their goal positions.

3.3.1 Computational complexity of f_1

In order to calculate f_1 then for each free vertex x, we need to maintain $d(x, A)$, $d(x, B)$ and their maximum accurate at all times. At the root node they are all initialized to 0 since both A and B are empty. Then, at each new node we have the following overhead. We now need to loop over all the neighbors of z and for each free neighbor w, we need to increment $d(w, A)$ and then update $max(d(w, A), d(w, B))$. Assume the time taken to do this is c_1. When backtracking from this node, we again loop over all the neighbors of z, and for free neighbor, w, we need to decrement $d(w, A)$ and restore $max(d(w, A), d(w, B)$ to its previous value. Thus, we multiple this part of the overhead by 2. The complexity for f_1 is therefore,

$$T(f_1) = T(f_0) + 2c_1 d_r = c_q d + c_0 d_B + c_m + 2c_1 d_r = O(d)$$

3.4 f_2: Sorting the free vertices

Assume that G is a graph with n vertices and that G has a partial partition into sets A and B, with the cardinality of A and B being a and b respectively. Since in a balanced partition there must be $n/2$ vertices in each set then we know that $n/2 - a$ free vertices must be placed in A' so that in the end the number of vertices in $A \cup A'$ will be $n/2$. Similarly, $n/2 - b$ other vertices must be placed in B'. Thus, we have to partition the $n - a - b$ free vertices into two sets with cardinalities $n/2 - a$ and $n/2 - b$. With f_1, we do not look into this but rather place each free vertex in one of the subsets without considering where the other free vertices went. Thus, for f_1, an unbalanced partial partition that has most of the vertices on one side will tend to have a low heuristic value, since assigning most free vertices to the set with most of the assigned vertices will generate few crossing edges. We solve this problem as follows. For each vertex x from the $n - a - b$ free vertices, we define:

$$N_A(x) = d(x, A) - d(x, B)$$

$$N_B(x) = d(x, B) - d(x, A).$$

Vertex	$d(x, A)$	$d(x, B)$	$N_A(x)$	$N_B(x)$	part
a	3	1	2	-2	
b	2	1	1	-1	A'
c	1	1	0	0	
d	1	2	-1	1	
e	1	3	-2	2	B'
f	2	5	-3	3	

Table 1: Free vertices that are left for the h_2 heuristic.

$N_A(x)$ denotes the advantage of placing x in A' over placing it in B'. If, for example, $N_A(x) = 3$, it infers that it is preferable by three edges to place x in A' than to place it in B'. Note that $N_B(x) = -N_A(x)$. Now, we sort the $n - a - b$ unassigned vertices in decreasing order of $N_A(x)$. We take the first $n/2 - a$ vertices and place them in A' (the set of vertices that will complete A). The rest of the vertices (which are the $n/2 - b$ vertices that have the best $N_B(x)$ value) will be placed in B'. This is an admissible way to partition these vertices to A and B and then have both partitions with $n/2$ vertices. After constructing A' and B' we define:

$$h_2(x) = \left\{ \begin{array}{ll} d(x, B) & if(x \in \text{A'}) \\ d(x, A) & if(x \in B') \end{array} \right\}$$

Summing $h_2(x)$ for all x in the free graph will yield an admissible heuristic which is never worse but usually better than h_1 since in h_1 we took the minimum between $d(x, A)$ and $d(x, B)$ while in h_2 we sometimes take the maximum between them.

For example, suppose that we are left with six unassigned vertices as illustrated in Table 1. Suppose we need to take four vertices to A' and two to B'. The four vertices with the best $N_A(x)$ are a, b, c, and d (they are A'). The total h_2 heuristic for these vertices will be the sum of their $d(x, B)$ which is $1 + 1 + 1 + 2 = 5$. Vertices e and f will go to B' with a heuristic of $2+1 = 3$. The h_2 heuristic here is therefore $3+5 = 8$. Note that h_1 would assign vertex d to B' and take $d(d, A) = 1$ as its heuristic, while h_2 would assign it to A' and take $d(d, B) = 2$. h_2 is still admissible but in this case is larger than h_1. While f_1 divided the free vertices into two sets with arbitrary sizes, f_2 divides them so that the resulting sets are balanced. Thus we define:

$$f_2(k) = g(k) + h_2(k)$$

Where $h_2(k)$ is calculated by sorting the free vertices by decreasing order of $N_A(x)$, splitting the sorted list into A' and B' and then summing the number of edges (of type III) that cross the partition.

Note that the sorted list can be calculated once for the root node of the search tree and only incremental changes are made when traversing the tree. In fact, we used an incremental version of bucket sort. Each distinct value of $N_A(x)$ had a unique bucket (where n is an upper bound for the number of buckets). Now, we need to loop over all the free neighbors of z and move them to their new bucket.

3.4.1 Computational complexity of f_2

We now have to maintain all the free vertices sorted at all times by decreasing order of N_A so that we can cut them into A' and B'. We keep them sorted in buckets where each bucket corresponds to a distinct value of N_A. All the vertices in a given bucket are cross referenced in a linked list. At the root node the N_A value of all the free vertices is initialized to 0 since both A and B are empty. As in f_1 we need to loop over all the neighbors of z and for each free neighbor w we need to increment $d(w, A)$. Here, however, a change in $d(w, A)$ also increases its N_A and it might propagate in the sorted list to another bucket. For doing this, we need to remove w from its bucket and place it in the succeeding bucket while also updating the necessary pointers inside the two buckets. Let c_2 be the time it takes to do all this. Naturally, all these changes should be undone when backtracking from a node and similar to f_1 we multiple this part of the overhead by 2.

Thus the complexity for f_2 is:

$$T(f_2) = T(f_0) + 2c_2d_r = c_qd + c_0d_B + c_m + 2c_2d_r = O(d)$$

As will be shown below, it turns out that the number of nodes per second of f_1 is more than twice greater than that of f_2. This is because the overhead of maintaining the buckets of N_A and the pointers involved consumes much more time than just updating two variables.

3.5 f_3: Adding edges from two free vertices to f_2

Following the general idea of looking deeper into interactions between unsolved subgoals we now want to look closer at edges of type IV. These are edges between free vertices that will be assigned later to different components. For this purpose we define the *free graph*. Vertices of the *free graph* are the free vertices and the edges are edges from the input graph where the two vertices connected by them were assigned to the opposite component by h_2. Edges of the free graph are actually all the edges of type IV. The free graph is a bipartite graph where the two sets of the graph are A' and B'.

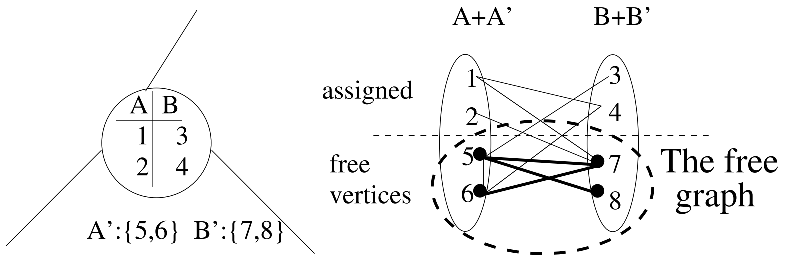

Figure 2: The free graph.

Figure 2 illustrates the free graph. The vertices of this graph are vertices 5, 6, 7 and 8. Edges of this graph are the bold edges and connect only edges between free vertices that were placed in different components. We would like to add as many edges from the free graph to h_2 without loosing admissibility.

For a reason that will become clear below, suppose that we want to move a vertex x from A' to B'. In that case $N_A(x) = d(x, A) - d(x, B)$ edges will be added to the partition. However, since we want both parts of the partition to be balanced, some other vertex y from B' should be swapped with x. In that case $N_B(y)$ or $(-N_A(y))$ more edges should also be added to the partition. Thus, if we are forced to move x from A' to B' we want to find a vertex y from B' with the best (smallest) $N_B(y)$, (or largest $N_A(y)$), since it will add the minimum number of edges of type III if it moves from B' to A'. Since for h_2, we already have all the vertices of the free graph sorted by their N_A value, it is very simple to spot a vertex in B' with the best N_A value. We call this vertex the *swappable* vertex and denote it $S_{B'}$. Once this $S_{B'}$ is spotted, the number of edges to be added when moving a vertex x from A' to B' will be at least $N_A(x) - N_A(S_B)$. We denote this number $N(x)$. In a symmetric manner, if we want to move a vertex x from B' to A' we should spot a *swappable* vertex $S_{A'}$ in A' with the best (smallest) N_A value. The number of edges of type III to be added in such a case will be at least $N_B(x) - N_B(S_{A'})$. Therefore, after these swappable vertices from B' and A', $S_{B'}$ and $S_{A'}$, are spotted we define:

$$N(x) = \left\{ \begin{array}{ll} N_A(x) - N_A(S_{B'}) & if(x \in A') \\ N_A(S_{A'}) - N_A(x) & if(x \in B') \end{array} \right\}$$

$N(x)$ stands for the "number of edges that are allowed for x". $N(x)$ is a lower bound on the number of edges of type III that will be added to h_2 if we move x from the component of the partition that was suggested by h_2 and swap it with some other vertex of the other component.

Suppose that x is placed in A' by h_2 and is connected to a number of vertices from B' (edges of type IV). $N(x)$ is therefore an upper bound on the number of such edges that can be added to h_2 without loosing admissibility. We can add as many edges connected to x of type IV to h_2 as long as the overall heuristic for x does not exceed the possibility of placing x in the other component which will add $N(x)$ edges to the partition. For each vertex x, $N(x)$ edges of type IV can be safely added, because if we move x to the other component we must add a number of edges of type III that are uniquely connected to this vertex (and the corresponding swappable vertex) from other assigned vertices and are not related whatsoever to the free graph. This number is at least as large as the number of edges of type IV that we added.

Consider again the example in Table 1. The swappable vertex in B' (the vertex with the best N_A value) is vertex e with $N_A(e) = -2$. Now, consider vertex a. $N(a) = N_A(a) - N_A(e) = 2 - -2 = 4$. As long as the number of edges from the free graph that we add to vertex a is not greater than 4 we do not lose admissibility. This is because only if it is greater than 4 it may be worthwhile exchanging a and e. The swappable vertex in A' is vertex d with $N_B(d) = 1$. If we take f, for example, then $N(f) = N_B(f) - N_B(d) = 3 - 1 = 2$. Only two edges are allowed for f. Otherwise we may want to swap f and d.

We want to take as many edges as possible from the free graph such that for each vertex x, we will not take more than $N(x)$ edges connected to x. This problem is a generalization of the maximal matching problem, since the simple matching problem is a special case where $N(x) = 1$ for all the vertices.

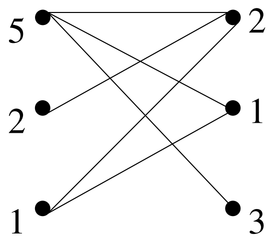

Figure 3: Generalized matching of the free graph.

Figure 3 illustrates the generalized matching problem of the free graph. Each vertex x, is associated with a number which is its $N(x)$. We want to take as many edges from the free graph as long as the number of edges that are connected to a vertex x, does not exceed $N(x)$. With this restriction we do not loose admissibility.

This generalized matching problem can be solved as a flow problem. We connect the source vertex S to each of the vertices of A', with each edge that connects S to x having a capacity of $N(x)$. Each of the edges of the bipartite free graph (edges from A' to B') will have a capacity of one. Then, we connect each of the free vertices from B' to the target vertex T. The capacity of an edge from y in B' to T will be $N(y)$. Now we are looking for a maximal flow from S to T. The size of this flow is the maximum of the generalized matching which can be added to the h_2 heuristic without losing admissibility. The formula for f_3 will therefore be

$$f_3(k) = g(k) + h_2(k) + h_3(k)$$

where the calculation of $f_3(k)$ is performed as follows:

1. Calculate N_A for each of the free vertices.
2. Sort the free vertices in a decreasing order of N_A
3. Construct the free graph, i.e., the sets A' and B' and the edges between them.
4. Calculate h_2 for each of the free vertices.
5. Identify the swappable vertices and for each vertex x in the free graph calculate $N(x)$.
6. calculate h_3 by solving the generalized matching problem for the free graph such that no vertex x will be connected to more than $N(x)$ neighbors.

Instead of optimally solving the generalized matching problem we can compromise on a brute force technique that is efficient but does not necessarily find the optimal matching. This technique loops on all the edges. When reaching an edge (x, y), if both $N(x) > 0$ and $N(y) > 0$ then it adds this edge to the overall heuristic and decreases

Algorithm	$Nodes/second$
f_0	2,412,132
f_1	1,280,940
f_2	445,553
f_3	194,191

Table 2: Nodes per second for graphs of size 50 with an average degree of 8.

both $N(x)$ and $N(y)$ by one. This method does not guarantee an optimal set of edges and there might be a larger set of edges which satisfies the constraints. Nevertheless, in the experiments that we will present below we used this brute force technique since it turned out to be more cost effective than finding the optimal set of edges.

3.5.1 Computational complexity of f_3

Here in addition to the work of f_2, we need to keep the two swappable nodes and the $N(x)$ value for each of the f free vertices. Note that initially in the root node, the $N(x)$ value is set to 0 for all the free vertices.

Since in f_2 we already spotted the cut in the sorted list, then it takes a constant time, denoted by c_{31}, to spot the two swappable vertices as they are adjacent to each other from the two sides of the cut. Since the swappable vertices have changed, we also need to loop over all the free vertices and change their $N(x)$ value at a constant time denoted by c_{32} for each free vertex.

We then need to solve the generalized matching problem. As described above, we loop over all the edges of the free graph at a constant time of c_{33} per edge. We define $d_{r'}$ to be the average number of neighboring vertices (for a free vertex) that are located in the other part of the bipartite free graph. Therefore the number of edges in the free graph is $r \cdot d_{r'}$. Thus the complexity for f_3 is:

$$T(f_3) = T(f_2) + 2 \times (c_{31} + c_{32} \cdot r + c_{33} \cdot r \cdot d_{r'} = O(r \cdot d_{r'})$$

Here, in the worst case $r = O(n)$ and $d_{r'} = O(d)$ and in this case $T(f_3) = O(n \cdot d)$.

3.6 Computational effort in practice

We have shown that the computational overhead complexity increases as the heuristics become more complicated. Table 2 shows the number of nodes per second for each of the algorithms on graphs of size 50 with an average degree of 8[3]. The search algorithm was DFBnB as described below. As could be expected, the constant time per node increases as the heuristic function is more complicated. The nodes per second rate decreases by a factor of around two for each heuristic in the series. We have observed that these nodes per second rates are rather stable and do not significantly change for all sizes and average degrees of graphs. However, as will be shown below, with the better

[3]The reason for choosing 8 here and later on, is because 8 is the smallest average degree where DFBnB was faster than IDA* for all the heuristics and all sizes of the graph.

heuristics the overall number of generated nodes decrease by a larger factor yielding a significant reduction in overall time needed for solving a given problem.

3.7 Generalizing to weighted graphs.

All our heuristics can be simply generalized to weighted graphs. We often made calculations with a number that corresponds to the number of edges in a given set. In a weighted graph we perform exactly the same calculations but use the sum of costs of the relevant edges instead of the number of edges. Apart from this adaptation, all our algorithms remain exactly the same.

In this way, for example, we define $d(x, A) = \sum_i w(e_i)$, where x is a vertex, A is a set of vertices (that are not free) and e_i is an edge connecting x to any vertex in A. In the same manner we define $N_A(x)$, N_x and so on.

4 The search algorithms

Now that we have defined the search tree and the heuristic functions we should describe the search algorithms used. Basically, we used two known search algorithms. The first algorithm is Iterative-deepening A* (IDA*) [24] which is a linear space version of A*. The other algorithm used is Depth-First Branch and Bound (DFBnB) [40]. Both algorithms were described in section 1. Note again that DFBnB is very effective when the depth of the solution in the search tree is known in advance. This is the case with the search tree provided above for GPP.

4.1 Enhancements to the search process

We implemented the following enhancements into the search in order to obtain a speedup. These enhancements are not directly related to GPP and might be effective in other domains as well.

4.1.1 Ordering the vertices of the graph

The main testbed for our algorithms were random graphs constructed as follows. First, we determine the values for the number of vertices in the graph, n, and the average degree, b. We then build an empty graph with n vertices. Then each edge is independently added to the graph with a probability of b/n. Therefore the graph obtained has an average degree of b but with some variance. Since the degree of vertices of the the graph is not uniform then before starting the search process we first sort the vertices of the graph by decreasing order of their degrees. In this manner, vertices with many neighbors will be treated sooner. Since these vertices have more edges connected to them they add more edges crossing the partial solution. The heuristic function in this case should also be greater and more accurate. We have found that this simple enhancement may speed the search process by a factor of more than ten in many cases. This idea is similar to the well known *most constraint variable* strategy widely used in constraint satisfaction problems.

4.1.2 Calculating related heuristics

Another improvement concerns the heuristic functions and can be applied whenever a more complicated heuristic is calculated by first calculating a simpler heuristic. This is a general idea and can also be applied to game tree searches. Indeed, this idea has been implemented in chess programs for many years, e.g., in Slate's CHESS 4.0 and in the Deep Blue program [17]. It works as follows. When generating a new node, both IDA* and DFBnB need to decide whether to expand that node or to prune it by checking whether the new node exceeds the current threshold. Instead of computing the complicated heuristic and then checking to see if a node should be expanded or pruned we should first check if the simpler heuristic already exceeds the threshold for pruning. If it does, we can prune the node without computing any further heuristics, which is the expensive part of the search. In our case of the GPP heuristics, h_3 is an addition to h_2. When using h_3, we can first calculate h_2. If for a given node, h_2 already exceeds the threshold, there is no need to calculate h_3 for that node. While applying this idea does not decrease the number of generated nodes it does decrease the time per node for many nodes. Applying this enhancement to GPP reduced the average constant time per node by more than 20% in many cases.

4.1.3 Node ordering of the search tree

A known enhancement to DFBnB is to order the descendants of a node by increasing order of their cost value and to visit them in this order. This method helps the search to first explore a subtree whose root seems to be more promising than its siblings. Nevertheless, this ordering of the nodes need not to be according to the same cost function that the search uses. We can take any heuristic to order these nodes. In fact, we have observed that the most cost effective combination we obtained was using h_3 for the heuristic of a node but ordering the descendants of a node with h_1. An explanation for using a simpler heuristic for the ordering of the nodes is that we want a very fast method for ordering these nodes because some of these nodes (which appear later in the order) will not be visited by the search process as DFBnB will prune them. Thus, we want to avoid spending time on these nodes. Ordering a node with a simple heuristic seems good enough to recognize such nodes. In all the experiments below we used h_1 to order the descendants of a node in the search tree for all the search algorithms except for f_0.

5 Experimental results

The first sets of experiments below were conducted on a 500MHZ pentium III PC. At first we experimented with graphs with 50 vertices and average degrees of 2, 4, ..., 20. For each type of graph we generated 30 random graphs and all the data numbers presented below are the average of these 30 instances.

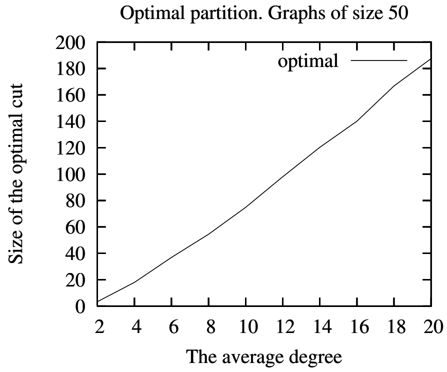

Figure 4: The optimal cut for graphs of size 50.

5.1 Graphs of size 50

Figure 4 illustrates the average size of the optimal partition for graphs of size 50 with different average degrees. The size of the optimal solution tends to increase linearly when increasing the average degree.

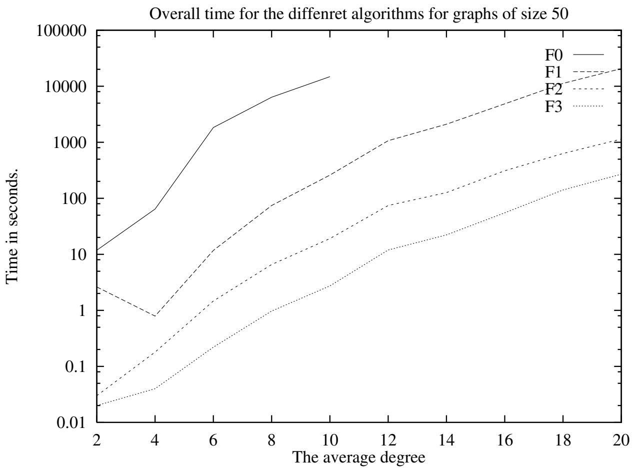

Figure 5: Time in seconds on graphs of size 50. DFBnB used.

Figure 5 shows the overall time of the different cost functions on graphs with 50 vertices. Each curve corresponds to a different cost function where each point is an average of 30 different graphs of that size and degree. The search in this figure was performed by DFBnB. The figure clearly shows that using a more complex heuristic function results in a much better overall time. The improvement factor between two succeeding algorithms is around 10.

Table 3 focuses on two points of this data, namely, on graphs with an average

Average degree	Alg	Solution	Nodes	Seconds
6	f_0	36.90	53,161,224.20	1,836.04
6	f_1	36.90	18,923,257.97	11.76
6	f_2	36.90	655,027.73	1.47
6	f_3	36.90	44,404.33	0.22
20	f_1	184.80	13,664,811,427.40	20,590.75
20	f_2	184.80	342,200,788.97	1,139.51
20	f_3	184.80	33,850,497.73	269.16

Table 3: Nodes visited and time elapsed.

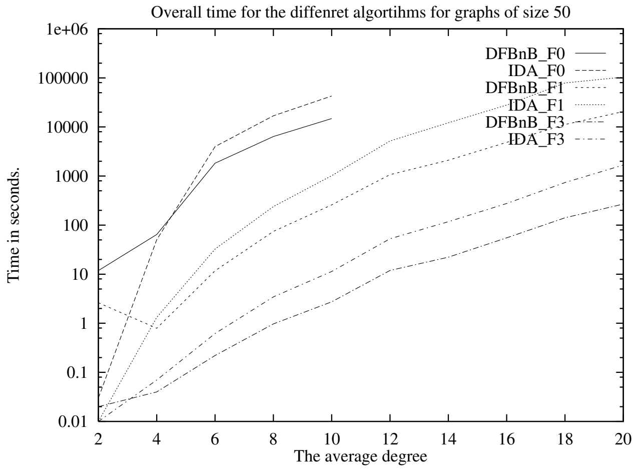

Figure 6: Comparing IDA* to DFBnB

degree of 6 and 20. We can see that the difference between the two extreme cases, f_0 and f_3, for graphs with a degree of 6 is almost 5 orders of magnitude in the number of generated nodes and 4 orders for the overall time. While a simple heuristic function based on f_0 needs about half an hour to solve such a problem, a complicated heuristic based on f_3 solves it in a fraction of a second. With a dense graph this difference becomes even greater and if we compare f_1 to f_3 we see that at an average degree of 6, f_3 outperforms f_1 by a factor of 50 in the overall time while for an average degree of 20 the improvement factor is more than 100.

Fig. 6 compares DFBnB to IDA* for f_0, f_1 and f_3. The figure show that at an average degree of 4 and smaller, IDA* outperforms DFBnB while for larger degrees DFBnB is much better. The explanation for that is that with 50 vertices in the graph, the search tree is fixed with $O(2^{50})$ nodes. Each node is given a heuristic value of the best possible cut associated with it. With larger degrees, the optimal cut is larger and therefore the number of distinct values in the search tree is larger. This number directly affects the number of iterations performed by IDA*. With more different distinct values there are more iterations of IDA* and many nodes are generated more than once. This

Size	Cut	f_1:seconds	f_2:seconds	f_3:seconds
20	27.07	0.04	0.02	0.01
30	34.67	0.15	0.04	0.03
40	47.07	3.89	0.59	0.15
50	54.43	80.88	7.53	1.06
60	63.13	1,482.14	59.70	4.87
70	69.87	30.922.20	1,085.62	55.10
80	83.59	502,238.58	4741.89	119.66

Table 4: Graphs with average degree of 8.

phenomenon does not seem to affect DFBnB in the same manner and therefore, in graphs with high average degree DFBnB outperforms IDA*. For example, for a graph with an average degree of 2 and an optimal cut of 4, IDA* using heuristic f_3 has 5 iterations with thresholds increasing from 0 to 4. On the other hand, DFBnB on the same graph first found a solution of 31 and then improved it until it found a solution of 4. If we can associate each improvement with an iteration, then DFBnB performs 27 iterations for that graph. However, for a graph with an average degree of 8 and an optimal cut of 51, IDA* performed 52 iterations while DFBnB reduced its solutions from 91 to 51 with only 41 iterations. A thorough analysis about the relation between the performance of IDA* and the different distinct values was presented in [30]. Since DFBnB is better on this domain then for the rest of the paper we concentrate on results obtained by this algorithm and omit results obtained by IDA*.

5.2 Graphs with a degree of 8

We conducted another set of experiments. In this set the average degree of the graphs was always set to 8 but the size of the graph varied from 20 to 80. The search was done with DFBnB. Again, each number in the following data is an average over 30 different graphs with the same size and average degree.

Table 4 shows data for that set of experiments. It also shows that the optimal cut grows linearly with the size of the graph while the overall time grows exponentially. We only report results for DFBnB which used f_1, f_2 and f_3. We can see that as the graphs grow larger the gap between these algorithms also grows. While for graphs of size 20, f_3 is twice as fast as f_1, for graphs of size 80 the improvement factor is almost 1000.

5.3 Graphs of size 100

Another domain studied was graphs of size 100. Here we only solved five graphs for each average degree and only with our best algorithm f_3. The results are shown in Table 5. We have also performed many experiments on other sizes of graphs and all the results showed the same tendency namely that a large speedup is obtained by the more complicated heuristic functions.

Average degree	Cut	f_3:Nodes	f_3:seconds
2	5.57	170,103.70	0.82
4	30.97	5,333,677.50	51.85
6	66.63	122,199,646.13	1,542.94
8	106.30	2,004,165,640.23	29,214.19
10	144.75	13,464,048,386.75	227,607.09

Table 5: Graph with size 100.

5.4 DFBnB as an anytime algorithm

A DFBnB search can be used as an anytime algorithm and return the best solution found so far. We have performed such experiments on a variety of graphs and found that bound for the DFBnB search tends to converge very fast. This is a known phenomenon in computer science where most of the effort of optimal algorithms is spent in improving a very good solution or in verifying that a given solution is optimal. This is especially true in our case where we try to optimally partition very small graphs compared to the graphs that were sub-optimally partitioned by other approaches in the literature and in the industry.

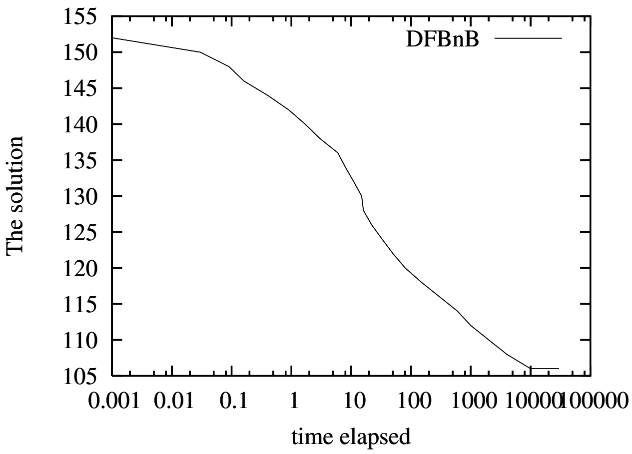

Figure 7: DFBnB as an anytime algorithm

Figure 7 presents the the DFBnB bound versus time elapsed for a graph of size 100 with an average degree of 8 (forth line of Table 5). The number of seconds for solving this version is 29,214. The initial bound was 152 and the table shows that the bound converges very fast to the optimal partition of 106. Note that after 30% of the running time (10,000 seconds) we already have an optimal solution in hand, but we do not know this for sure until the search verifies that a better solution does not exist and halts.

We have also tried to combine our global search with some of the other suboptimal algorithms mentioned above. We first run the suboptimal algorithm and receive a suboptimal solution from it. We then activate our DFBnB search with that solution as the initial threshold in order to either improve that solution or verify that it is an optimal solution. Doing this did not speedup the search significantly as the bound tends to converge very fast as shown in Figure 7.

6 Experimental comparison to other optimal GPP solvers.

As discussed in section 2.1, a number of optimal GPP solvers were developed. The best ones are the algorithms developed by Karisch et al. [21] and by Sensen [38]. In this section we optimally solve benchmark graphs that have been solved by these two approaches.

Since the time complexity of all the algorithms in question is exponential the only way to compare between them is to compare their actual running time on given problem instances. Indeed, Sensen [38] compared his work with that of Karisch et al. [21] by measuring the exact running time. We take the same approach and compare the running time of our algorithm to the running time presented by these works.

Comparing actual time has many flaws. First, results are affected by the specific architecture of the machine. Karisch et al. [21] performed their experiments on an HP 9000/735 machine, Sensen [38] performed his runs on a Sun Enterprise 450 Model 4400 with Sun UltraSPARC-II 400MHz processor while we performed our experiments below on a Pentium 4 PC with a 1.6MHz processor. Also, programs are affected by the efficiency of the exact implementation. Thus, one should be careful when comparing the exact actual times reported by all three approaches. If the reported times are within the same range, deriving accurate conclusion out of this data with regards to which is the fastest approach is misleading. However, if the times reported are in different ranges, for example, if one approach solves a problem in tenth of a second while the other approach needs a number of minutes then concluding that the first approach is faster is reliable.

To normalize our results with those of Karisch et al. [21] and Sensen [38] we ran a small sample of our experiments on a HP 9000/735 machine and on a Sun UltraSPARC-II 400MHz (the same machine used by Karisch et al. and by Sensen respectively). We followed the normalizing techniques provided in the TSP DIMACS challenges [1]. We found that our results obtained on these machine were slower by 30% and 20% respectively. The results provided by Sensen [38] are not normalized with those of Karisch et al.[21]. Thus, to be consistent with these works (in the fact that results are not normalized) we decided to report our exact times in the tables below. However, one should multiple our time reports by factor of 1.42 to normalize our results with those of Karisch et al. and by 1.25 to normalize our results with those of Sensen.

Note that these small factors are insignificant as the results below show a large difference of up to three orders of magnitude between the running times of the different algorithms. We can assume that using other modern machines and/or other implementations would not change these differences significantly. Thus, we believe the comparison below has meaning.

Solutions to a number of classes of graphs were reported by [21, 38]. We have

Graph	N	Density	Cut	Our time	Sensen	Karisch et al.
DB5	32	12	10	**0**	0	6
DB6	64	6	18	**8**	8	07:49
DB7	128	3	30	14:32:12	4:17	23:18:29
DB8	256				85:15:10	
SE6	64	8	9	**0.8**	3	
SE7	128		16	15:12:10	28	
Star 50	50		25	**0**	1	
Grid 5x10	50		5	**0.02**	1	
Rand 0.1	60	10	35	**0.4**	10	
Rand 0.05	60	5	13	**0.02**	12	

Table 6: De-Bruijn, Shuffle exchange, Star, Grid and random graphs.

solved the same classes of graphs and describe them in the same notation. The classes of graphs are as follows:

- The first class includes De-Bruijn graphs. See [21] for a comprehensive description and discussion of these graphs. We denote De-Bruijn graphs with dimension d as DBd.

- Shuffle exchange graph with dimension d are denoted SEd.

- Simple star graph with 50 vertices. It is denoted $Star50$.

- Two dimension grid graph of 5x10 is referred to as $Grid5x10$.

- Random graphs with 60 vertices. Each possible edge exists in the graph with a probability of 0.1 and 0.05. We denote these graphs by $Rand0.1$ and $Rand0.05$ respectively.

- In [2] a library of graphs was created and was referred to as BCR-library in [21, 38]. We have also optimally solved these graphs. The first set of graphs from the BCR-library are randomly generated instances where the degree of nodes of the graph was first fixed. Then edges belonging to the graph received weight uniformly drawn from [1,10]. These graphs are denoted in the BCR-library by $v.d$, $t.d$, $q.d$, $c.d$ and $s.d$ where d is the fixed degree.

- Toroidal grid instances from the BCR-library. They are denoted by $h \times kt$ where they have $n = hk$ vertices and $m = 2hk$ edges.

- Equicut of mixed grid instances from the BCR-library. They are referred to as $h \times km$. These are dense graphs where edges of a planar grid got weights which were uniformly distributed from [1,100] and all other edges were uniformly distributed from [1,10].

Graph	N	Density	Cut	Our time	Sensen	Karisch et al.
v0.90	20	10	21	**0**		1
v0.00	20	100	401	**0.02**		1
t0.90	30	10	24	**0**		1
t0.50	30	50	397	**0.09**		17
t0.00	30	100	900	**1.13**		3
q0.90	40	10	63	**0**		4
q0.80	40	20	199	**0**		01:09
q0.30	40	70	1056	**19**		01:02
q0.20	40	80	1238	43		25
q0.10	40	90	1425	01:45		41
q0.00	40	100	1606	02:52		4
c0.90	50	10	122	**0**		10
c0.80	50	20	368	**1.5**		03:04
c0.70	50	30	603	**21**		04:02
c0.30	50	70	1658	31:36		02:44
c0.10	50	90	2226	03:16:00		02:39
c6.90	56	10	177	**3.16**		30
c4.90	54	10	160	**0.16**		01:39
c8.90	58	10	226	**2.45**		08:46
c0.00	50	100	2520	08:12:02		02:20
c2.90	50	10	123	**0**		12
s0.90	60	10	238	**1.28**		04:57

Table 7: Random graphs from the BCR library.

All our results below were obtained by DFBnB with the most powerful heuristic that we have, namely with f_3. The experiments were conducted on a Pentium 4, 1.6 MHz, with 512MB of main memory.

Results are presented in Tables 6, 7 and 8. The columns are ordered as follows:

- The name of the graph,
- The number of vertices N,
- The average degree of the graph,
- The optimal GPP solution.
- The time in **hh:mm:ss.xx** that it took our program to solve the graph. Note that digits after the decimal point *xx* correspond to fractions of a second.
- The time reported by Sensen [38].
- The time reported by Karisch et al. [21].

Graph	N	Density	Cut	Our time	Sensen	Karisch et al.
4x5t	20	21	28	**0**		1
6x5t	30	14	31	**0**		3
8x5t	40	10	33	**0**		6
21x2t	42	10	9	**0**		5
23x2t	46	9	9	**0**		02:05
4x12t	48	9	24	**0**		17
10x6t	60	7	42	**0**		05:50
5x10t	50	8	33	**0**		6
2x10m	20	100	118	**0**		1
6x5m	30	100	270	**0.14**		1
2x17m	34	100	316	**0.77**		29
10x4m	40	100	436	5.2		2
5x10m	50	100	670	10:00		2
13x4m	52	100	721	18:25		34
4x13m	52	100	721	16:03		34
9x6m	54	100	792	47:00		12

Table 8: Toroidal and mixed grid instances from the BCR-library.

These tables show that for each of the methods compared, there exists a graph where it was significantly faster than the other approaches. The values in **bold face** correspond to cases where our results were the best.

Our approach seems to perform best for graphs with small number of vertices or for large graphs with low average degree. For such graphs, while the other approaches usually take a number of seconds, we solve them in a fraction of a second, which is up to three orders of magnitude faster. In particular, we outperformed the results reported by Sensen [38] for $Star50$, $Grid5x10$, $Rand0.1$, $Rand0.05$ and $SE6$. See Table 6. While we can meet his results for $DB5$ and $DB6$ we were behind for $DB7$. Note that Sensen also solved $DB8$. We outperformed Karisch et al. for all De-Bruijn graphs.

The results from Table 7 show that we have outperformed Karisch et al. for all the graphs of size 30 and smaller or for graphs of size 40 with average degreee lower than 70%. For graphs of size 50 and higher our results outperformed those of Karisch et al. for graphs with relatively low average degree.

The same phenomenon exists for the grid graphs of Table 8. Results from our approach outperformed those of Karisch et al. for any small sized graph. Again, for larger graphs our results outperformed those of Karisch et al. when the graphs had low average degrees.

We can conclude from these experiments that different attributes of the graphs cause a different behavior of the different algorithm. In general, it would be beneficial to use our approach for any small graph or for larger graphs with low average degrees. In these classes of graphs it seems that our pruning technique is the most effective among the three algorithms.

7 Conclusions and future work.

We have shown a number of admissible heuristics that optimally solve the graph partitioning problem. We have managed to optimally partition random graphs of sizes up to 128 vertices with a search space of size $O(2^{128})$. We have shown experimentally that our approach is a strong competitor to the best existing optimal GPP solvers and identified the cases where our algorithm performs best.

In general, it seems that introducing new domain independent search algorithms by suggesting new methods for expanding nodes does not lead to a large reduction in the search effort. Thus, more accurate heuristic functions might be the method of choice for significantly reducing the search effort. While our heuristics are somewhat tailored to GPP they implement a general method of finding better heuristics by looking deeper into interactions and conflicts between unsolved subgoals. This seems to pay off at the end with a much better overall time for obtaining a solution for GPP. Our best heuristic seems to outperform the simple one by a couple of orders of magnitude.

We believe that this basic idea of looking into interactions and conflicts between subgoals is general and might be applicable for other domains as well. The pattern database systems that were discussed above are one way of implementing this idea. In fact, we developed our current heuristics for the GPP while trying to implement the pattern database idea for this domain. For more details about the evolutions of our heuristics from the pattern databases methods see [10].

Another contribution of our work is that we have shown how to apply search techniques from artificial intelligence to successfully solve a problem from another area of computer science. While the notion of developing more accurate heuristics proved powerful, it is usually used only in artificial intelligence problems such as games and puzzles. We have demonstrated that this technique is more general and that it is applicable in solving problems from other areas of computer science that can be formulated as search problems.

Our approach cannot be directly compared to other GPP algorithms that, unlike ours, are designed to return suboptimal solutions. Naturally, since GPP is NP-hard, we can only find optimal solutions to problems of relatively small size. However, our experimental results show that our approach is indeed competitive to the best existing optimal GPP solvers. We showed that on a large number of graphs the running time of our algorithm outperformed previous approaches by a number of orders of magnitude.

An interesting idea would be to combine our algorithm with the work from [34] which used Lagrangian relaxation. While their algorithm does not guarantee a feasible optimal solution, it does give the size of the optimal solution rather quickly. Once we know the size of the optimal solution we can run our algorithm with that size as a bound and stop as soon as the first optimal solution is found. In this search both IDA* and DFBnB converge and only develop nodes whose cost value is smaller or equal to the size of the expected optimal solution. No node will be visited more than once, and no node with a cost that is greater than the optimal solution will be visited.

This work can be taken further in the following directions:

- We have only solved the GPP problem when a partitioning to two groups is needed. Future work can generalize our approach to the m-way partitioning.

- We have presented a sequence of heuristics. Future work can expand this sequence and develop more complicated heuristics for this problem.

8 Acknowledgments

I wish to thank Olga Shoikhet and Oleg Taubenshlak for their help with the comparison to the other GPP solvers. Thanks to Richard E. Korf for suggesting the GPP as a candidate domain for advanced heuristics, for his preliminary ideas on such heuristics and for his helpful comments on this work. I also thank Sarit Kraus for her comments.

References

[1] Dimacs implementation challenge: The traveling salesman problem. *Avaialble at http://www.research.att.com/ dsj/chtsp/.*

[2] L. Brunetta, M. Conforti, and N. Rinaldi. A branch-and-cut algorithm for the equicut problem. *Math. Program.*, 78:243–263, 1997.

[3] T. N. Bui and B. R. Moon. Genetic algorithm and graph partitioning. *IEEE Transactions On Computers*, 45(7):87–101, 1996.

[4] P. P. Chakrabarti, S. Ghose, A. Acharya, and S. C. de Sarkar. Heuristic search in restricted memory. *Artificial Intelligence*, 41(2):197–221, 1989.

[5] J. Clausen and J. Larsson-Traff. Implementation of parallel branch and bound algorithms - experiences with the graph partitioning problem. *Ann. Oper. Res.*, 33:331–349, 1991.

[6] J. C. Cullberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[7] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A*. *Journal of the Association for Computing Machinery*, 32(3):505–536, 1985.

[8] J. F. Dillenburg and P. C. Nelson. Perimeter search. *Artificial Intelligence*, 65:165–178, 1994.

[9] S. Edelkamp. Planning with pattern databases. *Proceedings of the 6th European Conference on Planning (ECP-01)*, 2001.

[10] A. Felner. *Improving search techniques and using them on different environments, Ph.D thesis*. PhD thesis, Department of Computer Science, Bar-Ilan University, Ramat-Gan, Israel, available at http://www.ise.bgu.ac.il/facutly/felner, 2001.

[11] A. Felner, R. E. Korf, and Sarit Hanan. Addtive pattern database heuristics. *Journal of Artificial Intelligence Research (JAIR)*, 22:279–318, 2004.

[12] A. Ferreira, A. Martin, C. C. de Souza, R. Weismantel, and L. A. Wolsey. The node capacitated graph partitioning problem: A computational study. *Math Program.*, 81:229–256, 1998.

[13] M. Garey, D. Johnson, and L. Stockmeyer. Some simplefied np-complete graph problems. *Theort. Comput. Sci.*, 1:237–267, 1976.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SCC-4(2):100–107, 1968.

[15] B. Hendrickson and R. Leland. The chaco user's guide. *Technical Report*, SAND94-2692, 1994.

[16] I. T. Hernádvölgyi. Searching for macro operators with automatically generated heuristics. *Advances in Artificial Intelligence - Proceedings of the Fourteenth Biennial Conference of the Canadian Society for Computational Studies of Intelligence (LNAI 2056)*, pages 194–203, 2001.

[17] Feng-Hsiung Hsu. *Behind Deep Blue : Building the Computer that Defeated the World*. Princeton University Press, 2002.

[18] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated allealing: An experimental evaluation; part 1, graph partitioning. *Operations Research*, 37:121–133, 1989.

[19] E. L. Johnson, A. Mehrotra, and G. L. Nemhauser. Min-cut clustering. *Math Program.*, 62:133–151, 1993.

[20] A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129:219–251, 2001.

[21] S. E. Karisch, F. Rendl, and J. Clausen. Solving graph bisection problems with semidefinite programming. *Informs: Journal of Computing*, 12(3):177–191, 2000.

[22] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *Technical Report*, 95-035, 1995.

[23] B. Kernigham and S. Lin. An efficient heiristic procedure for partitioning graphs. *Bell Systems Technical J.*, 49:291–307, 1970.

[24] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[25] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 700–705, Providence, Rhode Island, July 1997.

[26] R. E. Korf and A. Felner. Disjoint pattern database heuristics. *Artificial Intelligence*, 134:9–22, 2002.

[27] R. E. Korf and L. Taylor. Finding optimal solutions to the twenty-four puzzle. In *Proc of AAAI-96*, pages 1202–1207, Portland Or., 1996.

[28] G. Manzini. BIDA*: an improved perimeter search algorithm. *Artificial Intelligence*, 75:347–360, 1995.

[29] J. B. Oommen and E. V. de St. Croix. Graph partitioning using learning automata. *IEEE Transactions On Computers*, 45(2):195–208, 1995.

[30] B. Patrick, M. Amulla, and M. Newborn. An upper bound on the time complexity of iteratively-deepening a*. *International Journal of Artificial Intelligence and Mathematics*, 5:265–277, 1992.

[31] F. Pellgrini and J. Roman. SCOTCH: a software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *HPCN-Europe, Springer LNCS 1067*, pages 493–498, 1996.

[32] H. Pirkul and E Rolland. New heuristic solution procedures for the uniform graph partitioning problem. *Computers and Operations research*, 1991.

[33] R. Preis and R. Diekmann. PARTY - a software library for graph partitioning. *Technical Report*, 1997.

[34] E. Rolland and H. Pirkul. A lagrangian based heuristic for uniform graph partitioning. *Computers and Operations research*, pages 87–99, 1999.

[35] S. J. Russell. Efficient memory-bounded search methods. *Proc of ECAI-92*, 1992.

[36] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. *CRPC Parallel Computing Handbook*, 2000.

[37] A. Sen and A. Bagchi. Fast recursive formulations for best-first search that allow controlled use of memory. In *Proc of IJCAI-89*, pages 297–302, Detroit, MI, 1989.

[38] N . Sensen. Lower bounds and exact algorithms for the graph partitioning problem with multicommodity flows. *Algorithms - ESA 2001*, pages 391–403, 2001.

[39] C. Walshaw. Parallel JOSTLE userguide. *Technical Report*, 1998.

[40] W. Zhang and R. E. Korf. Performance of linear-space search algorithms. *Artificial Intelligence*, 79:241–292, 1995.