Searching Networks With Unrestricted Edge Costs

Parthasarathi Dasgupta, Anup K. Sen, Subhas C. Nandy, and Bhargab B. Bhattacharya, *Senior Member, IEEE*

Abstract—**Best-first and depth-first heuristic search algorithms often assume underlying search graphs with only nonnegative edge costs and attempt to optimize simple objective functions. Applicability of these algorithms to graphs with both positive and negative edge costs is not completely studied. In this paper, two new problems are identified: one in computational geometry and the other in the layout design of very large scale integrated (VLSI) circuits. The former problem relates to a weight-balanced bipartitioning of a given set of points in a plane. The goal of the second problem is to find an area-balanced staircase path in a VLSI floorplan. Formulations of these problems lead to an interesting directed acyclic search graph with positive, zero and negative edge costs and an objective function of general nature. These problems are NP-hard. To solve such general problems optimally, novel search schemes have been proposed in this paper. Experimental results reveal the efficacy and versatility of the proposed schemes, the depth-first scheme being the better choice. It is shown that the classical number-partitioning problem can also be formulated in this framework. The proposed depth-first search (*dfs*) scheme is capable of handling very large numbers, with a performance similar to the Complete Karmarkar–Karp (*CKK*) algorithm.**

Index Terms—**Artificial intelligence (AI), heuristic search, number partitioning, VLSI floor planning.**

I. Introduction

HEURISTIC search algorithms provide a powerful paradigm for solving computationally hard problems [14]. In this paper, we focus on solving two problems of recent interest: 1) *geometric bipartitioning of a point set* and 2) *area-balanced staircase bipartitioning of a VLSI floor plan*. The former problem arises in computational geometry and the latter is related to layout optimization of VLSI floorplans. The geometric bipartitioning problem attempts to find a monotone path through a given set of points in a plane with a minimum value of an objective function, where each point has an associated weight. The path is between two designated points in the set and the objective function is the difference of the sum of weights of the points on its two sides. The goal of the second problem, on the other hand, is to find a staircase cut in a VLSI floorplan such that the difference of the sum of areas of the rectangular blocks on its two sides is minimum. These problems can be shown to be NP-hard using a reduction from the number partitioning problem [8].

Our attempts to solve the above two problems using heuristic search have resulted in an interesting directed acyclic graph (*DAG*) with unrestricted, i.e., positive, zero and negative edge costs and a nonstandard objective function. Traditional heuristic graph search techniques [14] usually assume nonnegative edge costs and a very simple objective function. However, a mix of positive and negative edge costs have been considered in the past while studying the *shortest path problem* [1], [2], [19]. The well-known problem of finding a critical path in an acyclic digraph can be viewed as a shortest path problem with negative edge weights. Martelli [11] studied admissible search algorithms in networks with positive and negative edge costs. In all the earlier works, however, the goal was to minimize the sum of edge costs. In this paper, we develop new heuristic search algorithms for minimizing a general objective function in digraphs with unrestricted edge costs. We study best-first search (*bfs*) and depth-first search (*dfs*) strategies for solving the above class of problems. These techniques also enhance the applicability of other search algorithms of recent interest [18]. The proposed algorithms are implemented and run on several random examples and benchmarks. Empirical results are found to be very encouraging in terms of memory requirements and central processing unit (CPU) time.

We further show that the classical number partitioning problem [8] can be formulated as a search in a series-parallel acyclic digraph with unrestricted edge costs, where the objective function is the absolute value of the sum of edge costs along a path. Our experiments reveal that the proposed depth-first approach performs as good as the Complete Karmarkar-Karp (*CKK*) algorithm [10].

The rest of the paper is organized as follows. Sections II and III describe the two problems of geometric bipartitioning and area-balanced bipartitioning, respectively, and discuss their formulations as graph search problems. In Section IV, new heuristic search algorithms have been proposed. Experimental results are given in Section V. The number partitioning is discussed in Section VI. Finally, Section VII concludes the paper.

II. Geometric Bipartitioning Problem

This geometric optimization problem has applications to image processing [3], facility location and plant layout problems [16]. The formal description is as follows: Let A be a set of M points distributed arbitrarily in a rectangular plane whose bottom-left and top-right corners are $s(0,0)$ and $r(\alpha,\beta)$, respectively. Without loss of generality, we assume

Manuscript received April 10, 1998; revised July 23, 2001. This paper was recommended by Associate Editor B. Scherer.

P. Dasgupta is with the Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093 USA, on leave from the MIS Group, Indian Institute of Management (IIM), Calcutta, India (e-mail: partha@iimcal.ac.in).

A. K. Sen is with the MIS Group, Indian Institute of Management (IIM), Calcutta, India (e-mail: sen@iimcal.ac.in).

S. C. Nandy is with the Indian Statistical Institute (ISI), Calcutta, India (e-mail: nandysc@isical.ac.in).

B. B. Bhattacharya is with the Department of Computer Science and Engineering, University of Nebraska, Lincoln, NE 68588-0115 USA, on leave from the Indian Statistical Institute (ISI), Calcutta, India (e-mail: bhargab@cse.unl.edu).

Publisher Item Identifier S 1083-4427(01)09751-X.

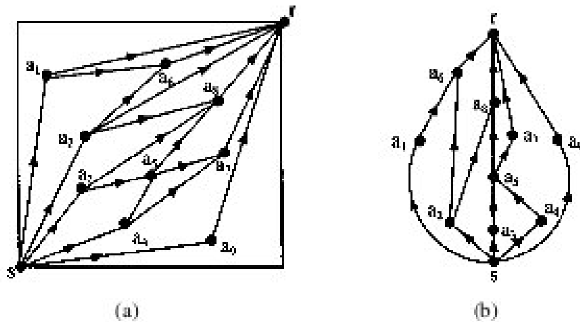


Fig. 1. (a) Example graph and (b) its transitive reduction.

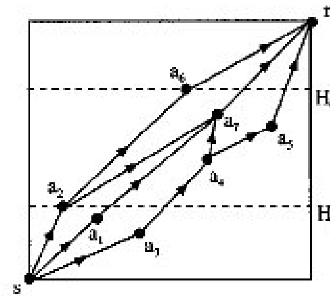


Fig. 2. Edge weight computation in geometric bipartitioning.

that the points s and r are members of A and for every pair of points $a_i(x_i, y_i)$ and $a_j(x_j, y_j)$ in A, $x_i \neq x_j$ and $y_i \neq y_j$. The problem is to partition the plane by a set of interconnected consecutive straight line segments $\{l_1, l_2, \ldots, l_k\}$, from s to r, such that the end points of each l_i, $i = 1, \ldots, k$, coincide with some points in A and each l_i has a positive slope. Any two consecutive segments l_i and l_{i+1} always share a common point in A. The starting point of l_1 and the end point of l_k are s and r, respectively. The path L formed by these line segments is called a monotone increasing path, or shortly, a *monotone path* (*MP*).

Definition 1: An *MP* is said to be maximal, if no other point of A can be included in it keeping the path monotone increasing.

Let A^L denote the set of points on a maximal monotone path (*MMP*) L. Each point a_i in A has an associated weight $w(a_i)$, which is a real number. The goal of geometric bipartitioning is to find a *MMP* L that partitions the set of points $A \setminus A^L$ into two parts such that the difference of the sum of the weights of all the points in these two parts is minimum. It can be easily verified that the number of *MMP*s in a plane may be exponential in M in the worst-case. Thus, checking every such path is computationally intractable. However, the special case when $w(a_i) = 1$ for all $i = 1, \ldots, M$, can be solved in polynomial time [6]. A restricted version of the geometric bipartitioning problem appears in [3], which considers only vertical and horizontal edges of a grid graph.

A. Formulation of the Problem

The above optimization problem can be captured using a directed acyclic search graph. Let us consider a directed graph $G(V, E)$ with $V = \{a_i \mid a_i \in A\}$ and $E = \{(a_i, a_j) \mid (x_i \leq x_j)$ and $(y_i \leq y_j)\}$. Costs associated with the points may be assigned to the edges of G through a suitable transformation shown below. The digraph G, defined above, is acyclic and the vertex s (r) has indegree (outdegree) 0.

Definition 2: An edge (a_i, a_k) in a graph G is said to be *transitive* if G also has a sequence of edges $(a_i, a_{i+1}), \ldots, (a_{k-1}, a_k)$.

Fig. 1 illustrates such a digraph and its transitive reduction. Any directed path from s to r is clearly an *MP* in G. Since we are interested in *MMP*s only, the transitive edges in G are removed to yield the transitive reduction G_T of G. The path $L = s \rightarrow a_3 \rightarrow a_5 \rightarrow a_8 \rightarrow r$ is an *MMP*, whereas the path $L' = s \rightarrow a_3 \rightarrow a_8 \rightarrow r$ is an *MP*. Given the set of points A, the graph G_T can be constructed in $O(M \log M + E^*)$ time using a plane-sweep technique, where E^* is the number of edges in G_T.

1) Determination of Edge Weights: To formulate the problem, we first transform point weights to edge weights as follows. Consider a *DAG* as in Fig. 2 and an edge, say (a_2, a_6). We draw two horizontal lines H_2 and H_6 through the points a_2 and a_6, respectively, up to the boundary of the floor. Let $W_l(a_2, a_6)$ [resp. $W_r(a_2, a_6)$] be the sum of the weights of the points to the left (resp. right) of the edge (a_2, a_6) lying within the band defined by H_2, H_6 and the left (resp. right) vertical boundaries of the rectangular floor. Then, the weight of the edge (a_2, a_6) is given by $[W_l(a_2, a_6) - W_r(a_2, a_6)]$. For example, if we assume that the weight of each point is unity in Fig. 2, then the weight of the edge $(a_2, a_6) = -3$.

Thus the weight of an edge may be a positive or a negative real number, or zero and the geometric bipartitioning problem reduces to that of finding a directed path L from s to r in the weighted *DAG* G_T, such that the *absolute value* of the total weight of all the edges along L is minimum.

III. Area-Balanced Bipartitioning in VLSI Design

An interesting problem of VLSI layout design is the staircase area-balanced partitioning problem [12], [13]. A VLSI floor plan [17] consists of a bounding rectangle that is divided into nonoverlapping smaller rectangles by a sequence of isothetic (axis-parallel) line segments, called *cutlines*. Each cutline splits a floor plan into two subfloorplans. A rectangle which has not been divided is called a *block*. A floorplan is *slicing* if it is either a block or there is a single cutline (*slice*) that partitions the enclosing rectangle into two slicing floorplans. Each block is a circuit module, whereas cutlines represent routing spaces or *channels*. Floorplans that are not slicing are called *nonslicing* (Fig. 3). For the convenience of routing, the channels are routed following certain order called the *safe (cycle-free) routing order*. For slicing floorplans, a safe routing order exists, whereas for nonslicing floorplans, no such order is possible. However, if the channel definition is generalized to *staircase* (monotone increasing) channels, then both the problems of hierarchical decomposition and that of finding a safe routing order can be solved for nonslicing floorplans. The utility of staircase channels as generalized routing regions has been reported in [7] and [12].

The *area-balanced bipartitioning problem* can be described as follows: given a floorplan with rectangular modules, find a staircase path through the cutline segments from one corner of

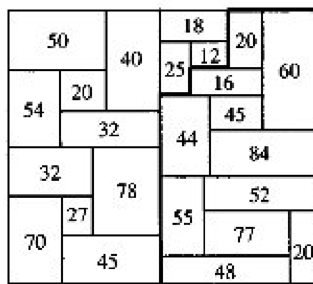


Fig. 3. Optimal area-balanced bipartition of a benchmark floorplan.

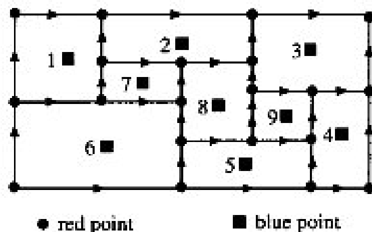


Fig. 4. Floorplan graph with two classes of points.

the floor to its diagonally opposite corner, that partitions the set of modules into two halves with *minimum difference of areas*. This problem has been shown to be NP-hard [13]. However, if the objective is to minimize the difference of number of blocks on two sides of the partition, then the problem can be solved in linear time [6].

An example of area-balanced bipartition for a benchmark problem from [20], is shown in Fig. 3. The area of each block is denoted by the number on the corresponding block.

A. Formulation of the Problem

A floorplan F can be represented by a *DAG* G_F with T-junctions as the vertices and cutline segments as directed edges which are oriented in either from left to right, or from bottom to top. Therefore, all directed paths in G_F from the bottom-left corner to the top-right corner are maximal monotone. Further, let us consider Fig. 4 where the T-junctions (circular nodes) are the red points and the centers of the rectangular blocks (rectangular nodes) are the blue points. With each blue point, we associate a weight equal to the area of that block. In the graph G_F, red points are the vertices and the computation of edge costs is similar to that of Section II. The number of possible *MMP*s in a floorplan may also be exponentially large.

The area-balanced bipartitioning problem can now be restated as follows: given a set of points of two colors (say, red and blue), find an *MMP* that passes through the red points and partitions the set of blue points into two equal weighted subsets. Thus, the problem here is to find a directed path in G_F between the two vertices corresponding to the opposite corners of the floorplan, such that the *absolute value* of the sum of edge weights along the path is minimum.

IV. Proposed Graph Search Methods

In this section, we develop search methods in an arbitrary directed acyclic graph G with unrestricted edge costs. We assume that the search graph G has N vertices and E edges. It has two designated nodes—*source* (s) and *goal* (r). A *path* in G is a sequence of consecutive vertices in G, whose start and end vertices are different. We assume that the goal node r is reachable from every other node of G. We define a *solution path* in G as a path in G from s to r. The cost of a path is the algebraic sum of the edge costs along the path. If the edge costs are unrestricted, the cost of a path may be either positive, or negative, or zero. For both the problems stated above, the objective is to find a solution path from s to r with minimum *absolute cost*. The existing heuristic search algorithms [14], [15] cannot be directly applied to solve this problem. Hence, every solution path may have to be examined completely to find the optimal one because a nonpromising path at any stage of the search may turn into a promising one later. Even in a tree search space, the first solution obtained by A^* algorithm may be suboptimal. A graph-search space further complicates the situation. No path can be discarded since the evaluation function is not order preserving [15]. In [11], it is shown that the A^* algorithm without heuristic estimates may expand an exponential number of nodes for search graphs with negative edge costs and sum-cost objective function.

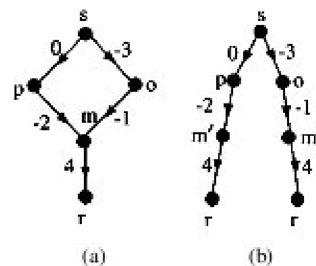


Fig. 5. (a) Search graph and (b) its equivalent tree.

A. Motivation of the Proposed Methods

In the graph search problems presented above, the objective function is nonadditive and nonmonotonic [15]. The cumulative cost along a path may vary in an arbitrary manner. Hence, no path can be discarded during the progress of the search.

For instance, consider the graph G shown in Fig. 5(a). There are two paths from node s to the node m. The absolute cost of the path $s \to p \to m$ is 2 and that of the path $s \to o \to m$ is 4. However, the path $s \to o \to m$ cannot be discarded because the optimal solution path is $s \to o \to m \to r$.

Now, let us consider the tree search space [Fig. 5(b)], corresponding to the graph of Fig. 5(a). The absolute cost at node m' is 4. However, the optimal path from s to r passes through m' and its cost is 0. Here, *bfs* never selects the node m' for expansion since the currently known best solution is observed in the path $s \to p \to m'' \to r$ with absolute cost 2. *dfs* prunes the path $s \to o \to m'$ since the current upper bound set by the path $s \to p \to m'' \to r$ is 2 and, in turn, misses to report the optimal solution path. Exhaustive search certainly yields the optimal solution, but it is extremely time consuming and almost impractical as the number of paths may be exponential in the number of nodes in the search graph. In the next two sections, two different approaches based on *bfs* and *dfs* paradigms are presented. Both have their respective advantages and disadvantages [14], [15].

B. Preliminaries

Given a node n in G, several solution paths may pass through it and the one with minimum absolute cost will be termed as the *optimal solution path* through n. The corresponding absolute cost is denoted by $f_{opt}(n)$.

The *most promising solution path* through a node n at an instant is the currently known best solution path through n, i.e., one with minimum absolute cost. Let $f(n)$ denote this absolute cost.

For a node n in G, $g(n, P_i)$ is the sum of costs of the edges from s to n along a path P_i. Consider a node n in G. There may exist several paths from n to the goal node r, each one having an associated cost. Let $l(n)$ and $u(n)$ denote the minimum and the maximum values of all these costs.

Let $f_{\text{low}}(n, P_i) = g(n, P_i) + l(n)$ and $f_{\text{high}}(n, P_i) = g(n, P_i) + u(n)$. Thus, $f_{\text{low}}(n, P_i)$ and $f_{\text{high}}(n, P_i)$ provide the lower and upper bounds, respectively, on the actual costs of the solution paths through n having P_i as the initial path segment.

Let $f(n, P_i)$ denote the *absolute cost* of the currently known most promising solution path through n having P_i as the initial path segment. Henceforth, this will also be referred to as the f-value. Now, two different situations may arise.

1) $f_{\text{low}}(n, P_i)$ and $f_{\text{high}}(n, P_i)$ are of the *same* sign.
2) $f_{\text{low}}(n, P_i)$ and $f_{\text{high}}(n, P_i)$ *different* signs.

In the former case, the node n is said to be *informed* along path P_i, while in the latter case, it is said to be *blind* along that path.

Lemma 1:

a) If a node n is found to be informed along a path P_i, the minimum absolute cost among all paths having initial path segment P_i, will be

$$f(n, P_i) = f_{\text{low}}(n, P_i), \text{ if } f_{\text{low}}(n, P_i) > 0$$
$$= |\, f_{\text{high}}(n, P_i)\, |, \text{ if } f_{\text{low}}(n, P_i) < 0.$$

b) If n is blind, the cost of the path from s to r with initial path segment P_i and having minimum absolute cost, will lie in the range $[f_{\text{low}}(n, P_i), f_{\text{high}}(n, P_i)]$.

Proof: Clear from definitions. ■

If a node n is found to be blind along a path P_i, Lemma 1 cannot uniquely determine $f(n, P_i)$. So, we set $f(n, P_i) = 0$ in this case.

At an instant, $f(n)$ is the minimum of $f(n, P_i)$s for all P_is explored up to that instant. Let $P_i^{\min}$ be the initial path segment having absolute cost $f(n)$. Since the edge costs may be negative, $P_i^{\min}$ is not set with the least cost path from s to n. We define $g(n) = g(n, P_i^{\min})$ and call it as *g-value* of node n. Note that, the path of minimum absolute cost from a node to a goal has no *a priori* significance for finding the optimal solution path.

C. Best-First Search (bfs) Method

We first discuss the *bfs* approach based on algorithm A^* [14], [15] and name it NA^*.

The proposed algorithm NA^* uses two lists, *OPEN* and *CLOSED*. The algorithm starts with node s in *OPEN* and keeping *CLOSED* empty. At every instant, the currently known best node (having minimum *g-value*) is selected from *OPEN* and put in *CLOSED*. The selected node is expanded and new successors, if any, are added to *OPEN*. At each node n in *OPEN* and *CLOSED*, NA^* maintains a currently accumulated cost $g(n, P)$, two bounds $\ell(n)$ and $u(n)$ and an absolute cost $f(n, P)$ where P is a path in the search graph.

A preprocessing procedure calculates $\ell(n)$ and $u(n)$ in a bottom-up manner starting from the node r and it takes $O(E)$ time, where E is the number of edges in the graph.

The search starts from the source node s and proceeds in a best-first manner. For our definitions of blind and informed nodes, we have the following result.

Lemma 2: An informed node along a solution path will never have a blind successor along that path.

Proof: Let n be an *informed* node on a solution path P and n' be a successor of node n. As n is informed, $f_{\text{low}}(n, P)$ and $f_{\text{high}}(n, P)$ are of the same sign. Now, consider the following inequalities:

1) $\ell(n) \le \ell(n') + c(n, n')$ and $u(n) \ge u(n') + c(n, n')$.
2) $f_{\text{low}}(n', P) = g(n, P) + c(n, n') + \ell(n') \ge f_{\text{low}}(n, P)$.
3) $f_{\text{high}}(n', P) = g(n, P) + c(n, n') + u(n') \le f_{\text{high}}(n, P)$.

If $f_{\text{low}}(n, P)$ and $f_{\text{high}}(n, P)$ are both positive, then $f_{\text{low}}(n', P) \ge 0$ [using inequality 2)] and $f_{\text{high}}(n', P) \ge f_{\text{low}}(n', P) \ge 0$.

Similarly, if $f_{\text{low}}(n, P)$ and $f_{\text{high}}(n, P)$ are both negative, then $f_{\text{high}}(n', P) \le 0$ [using inequality 3)] and $f_{\text{low}}(n', P) \le f_{\text{high}}(n', P) \le 0$. Hence the proof. ■

Lemma 3: The successor of a blind node along a solution path must be either informed or blind.

Proof: Follows from the fact that only r (goal node) has no successor and r is informed along all paths from s. ■

The computation of two bounds, $\ell(n)$ and $u(n)$ is described in the procedure *preprocess*. At every instant, NA^* selects a node n from *OPEN* with minimum $f(n)$. In the case of ties, priority is given to an informed node. Otherwise, the one at a higher depth is selected. During execution, a path P_1 to a node is discarded in NA^*, when an alternative path P_2 is obtained and $f(n, P_2) < f(n, P_1)$. Like A^*, in this case also, a node may be expanded more than once.

In NA^* algorithm, a global variable F is used to store the absolute cost of the currently known best solution path. The value of F is set to ∞ at the beginning; it is reset whenever an informed node n with $f(n) < F$ is found. F thus decreases in a step-wise fashion. An ordered list of nodes, called *outpath*, is globally maintained that represents the path segment from the start node s to n of the currently known best solution path. The search terminates when a node n with $f(n) \ge F$ is selected for expansion. If n is not the goal node r, *outpath* contains the initial path segment (i.e., from s to the node n) of the optimum path. The path from n to the goal nore r is determined as follows.

We use Lemma 2 to compute the $f_{\text{low}}(n', P)$ or $f_{\text{high}}(n', P)$ for each successor n' of n, depending on whether $f_{\text{low}}(n, P)$ $[f_{\text{high}}(n', P)]$ is positive or negative. By Lemma 2, all these successors will be *informed* and their successors will again be *informed* and so on. In order to proceed along the optimum path from the node n we choose the successor n^* such that

$$f_{\text{low}}(n^*, P) = f_{\text{low}}(n, P), \text{ if } f_{\text{low}}(n, P) > 0.$$
$$f_{\text{high}}(n^*, P) = f_{\text{high}}(n, P), \text{ if } f_{\text{low}}(n, P) < 0.$$

The process continues until the goal node r is reached.

A formal description of the algorithm is presented as follows.

```
Algorithm NA*
    input: A directed graph G having positive and
    negative edge costs.
    output: A path of minimum absolute cost in the
    given graph between nodes s and r.
begin
    OPEN := CLOSED := φ;
    for every node n ∈ G do begin ℓ(n) := ∞; u(n) :=
    −∞; end;
    preprocess(s);
    F := ∞; outpath := φ; g(s) := 0; state(s) := blind;
    if ℓ(s) and u(s) are of same sign then begin
        F := f(s) := min(| ℓ(s) |,| u(s) |);
        state(s) := informed; outpath := {s};
    end;
    else begin
        f(s) := f_lo; n := s; put n in CLOSED;
        while f(n) < F do
            for each successor n' of n  do begin
                g := g(n) + c(n, n');
                f_low := g + ℓ(n'); f_high := g + u(n');
                if f_low and f_high are of same sign then
                begin (*n' is informed *)
                f := min(| f_low |,| f_high |); state := informed;
                    if f < F then begin
                        F := f;
        outpath     :=     path(s, n') by tracing backward
    pointers;
                    end;
                end;
                else begin (*n' is blind *)
                    f := 0; state := blind;
                end;
                if (n' ∉ OPEN and n' ∉ CLOSED )
                    then begin
                    g(n') := g; f(n') := f; state(n') := state;
                        put n' in OPEN;
                direct backward pointer from n' to n;
                    end;
                else if (n' ∈ OPEN or n' ∈ CLOSED )
    and (f(n') ≥ f) then
                begin
            g(n') := g; f(n') := f; state(n') := state;
            redirect backward pointer from n' to n;
                    if n' ∈ CLOSED then
             remove n' from CLOSED and put n' in OPEN;
                end;
            end;
         select a node n from OPEN with minimum f(n);
         resolve ties in the order of a goal node, in-
    formed node, or a node at a higher depth
        endwhile;
    end
        output F as the solution cost;
        output solution path from outpath by tracing
    the remaining path, if any, to goal;
end.
```

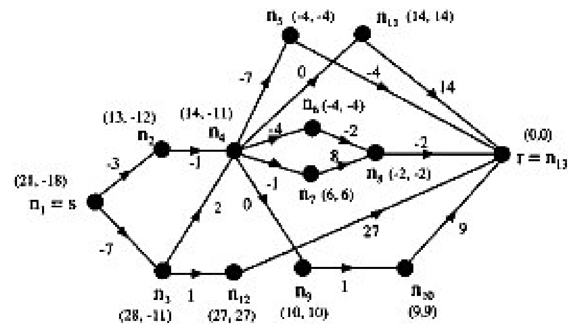


Fig. 6. Searchs graph G.

TABLE I
EXAMPLE ILLUSTRATING EXECUTION OF NA^*

Steps / Nodes	1	2	3	4	5	6
n_1	0,0,b	**0,0,b**				
n_2		-3,0,b	**-3,0,b**			
n_3		-7,0,b		-7,0,b	**-7,0,b**	
n_4			-4,0,b	**-4,0,b**	-5,0,b	**-5,0,b**
n_5				-11,15,i		-12,16,i
n_6				-8,12,i		-9,13,i
n_7				-5,1,i		-6,0,i
n_9				-4,6,i		-5,5,i
n_{11}				-4,10,i		-5,9,i
n_{12}					-6,21,i	
F	∞	∞	∞	1	1	0
outpath	ϕ	ϕ	ϕ	$sn_2n_4n_7$	ϕ	$sn_3n_4n_7$

```
Procedure preprocess (node : n)
begin
    if n has no successor then begin ℓ(n)    :=    0;
    u(n) := 0; end;
        for every successor n_i of n do begin
            preprocess (n_i);
        ℓ(n) := min(ℓ(n), c(n, n_i) + ℓ(n_i));
        u(n) := max(u(n), c(n, n_i) + u(n_i));
        end;
    end;
```

1) Example 1: We now illustrate the algorithm NA^* through an example. The term explicit graph at an instant means the part of the graph explored so far by NA^*.

Consider the digraph G in Fig. 6. There are 13 nodes $n_1, n_2, \ldots, n_{13}$ in the graph. The start node is $s(= n_1)$ and the goal node is $r(= n_{13})$. Edge costs are shown beside the edges and the bounds are shown beside every node as a pair $(\ell(.), u(.))$. Table I summarizes the stepwise execution of NA^* for this graph. Rows of the table correspond to the nodes of G and columns correspond to execution steps. Each entry in the table gives the g-*value*, f-*value* and the state (b means blind, i means informed) of the corresponding node at the concerned instant. An underscored entry represents a node (in the *OPEN* list) which is selected for expansion and a bold entry indicates a node (in the *CLOSED* list) which has just been expanded. An ordinary entry represents a node, which is generated and placed in *OPEN*; a blank cell indicates no change in the status of the corresponding node is made at that instant.

Nodes not in the explicit graph are not shown in Table I. Two additional rows are used to show the values of F and *outpath* at different instances. The total number of nodes expanded

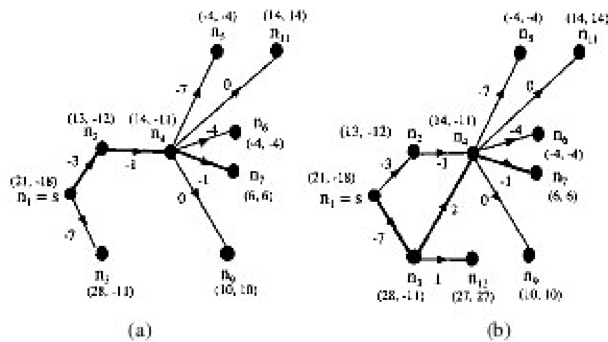


Fig. 7. Explicit search graphs.

and generated in this graph are six and ten, respectively. Node n_4 is expanded twice. At instant 1, the value of F is ∞; s is selected for expansion as it is blind and *outpath* is ϕ. When s gets expanded, nodes n_2 and n_3 are added to *OPEN* with $f(n_2) = f(n_3) = 0$, $g(n_2) = -3$, $g(n_3) = -7$ and both n_2 and n_3 are in the blind state. Assume n_2 is selected next for expansion; the choice is arbitrary as both n_2 and n_3 are at the same depth. NA^* proceeds in this manner and F remains ∞ until a node of the explicit graph is found to be informed. At step 4, n_4 is expanded and the nodes n_5, n_6, n_7, n_9 and n_{11} are inserted in *OPEN*. All these successors of n_4 are in informed state; so F is reset to 1 ($f(n_7) = 1$) and *outpath* now contains $\{s, n_2, n_4, n_7\}$ in this order [see Fig. 7(a); *outpath* is shown in bold]. Since there is still some node in *OPEN* with f*-value* $< F$, NA^* must examine them before deciding the optimal path segment. So, n_3 gets expanded at the next instant. NA^* terminates when the informed node n_7 is selected for expansion with $f(n_7) \le F$. Now, *outpath* contains $\{s, n_3, n_4, n_7\}$ as the initial segment of the optimal path and the optimal cost is 0. On termination, the explicit graph is shown in Fig. 7(b).

2) Properties of NA^*: We now prove some additional properties of NA^*.

Lemma 4: At any instant, $f(n)$ for a node n is the least among the absolute costs of all the currently known paths through n.

Proof: We refer to the formal description of NA^*. If n is blind, the result is obvious. Assume n is informed and n is reached along a different path. Irrespective of whether n is in *OPEN* or *CLOSED*, $f(n)$ is reset to the f*-value* computed along the path, if it is lower. ■

Lemma 5: Each solution path in G has at least one informed node.

Proof: Since each solution path in G must terminate at goal node r and r is always informed, the claim follows. ■

Remark 1: The costs of all informed nodes along a solution path are same.

Lemma 6: At any stage before NA^* terminates, there is a node n in *OPEN* such that $f(n) \le f_{\text{opt}}(s)$.

Proof: At any stage before NA^* terminates, *OPEN* contains at least one node from every solution path. Consider the solution path P that determines the value of $f_{\text{opt}}(s)$ and a node n on that path which is in *OPEN* . The node n may either be blind or informed. In the former case, $f(n) = 0$ and in the latter case, $f(n) = f(n) \le f_{\text{opt}}(s)$ (by Lemmas 4 and 5). ■

Lemma 7: Algorithm NA^* terminates at goal node r.

Proof: Since by Lemma 6, *OPEN* always contains a node n such that $f(n) \le f_{\text{opt}}(s)$, a failure exit is impossible. Each solution path in G has a finite cost since G is a directed acyclic graph and is finite. ■

Role of F: A simple *bfs* strategy would be to select the node n with minimum value of $f(n)$ from *OPEN* in turn until the goal is reached. It does not consider the current value of F. We now show that it does not guarantee an optimal solution. The following example indicates that it is not so. Consider a node n reached along a path P_i and observed to be informed. At a latter instant, the same node is reached along a different path P_j and is observed to be blind. Now, the cost of a blind node is less than or equal to that of an informed node. Thus n will now be explored along P_j and its earlier cost is not remembered. At a later stage, during the search along P_j, an informed node is reached. The cost of this informed node may be higher than the cost obtained at node n along path P_i. Thus, the optimal cost is not guaranteed. Thus to ensure the optimality, an upper bound F on the optimal cost is revised each time an informed node of lower cost is obtained.

Theorem 1: The algorithm NA^* always outputs the path having optimum absolute cost, on termination.

Proof: [By contradiction] Let us assume that the solution path obtained by NA^* is not optimal. At any instant before termination, *OPEN* contains at least one node from every solution path. The nodes in *OPEN* may be either blind or informed. If this node is blind, it will be expanded. Thus, as the search graph is acyclic and finite, one informed node from the optimal solution path must appear in *OPEN*. The cost of this node is smaller and by our assumption, it will be selected later, which is impossible. ■

Remark 2:

1) If edge weights are all nonnegative, the proposed NA^* algorithm terminates in $O(N^2)$ time in the worst case and can be viewed as a general form of Dijkstra's uniform cost method [5].
2) For the standard additive cost function with unrestricted edge weights, NA^* again finds the optimal path in $O(N^2)$ time in the worst-case if the graph does not have any cycle. As in 1), NA^* only needs the lower bound for every explored node and it explores a node at most once.

In algorithm NA^*, the number of changes in the parameter F is at most equal to the number of informed nodes in the search graph G. By Lemma 4, each solution path in G has at least one informed node. As the number of solution paths in G is exponential in the worst case, we have the following observation.

Remark 3: The number of nodes expanded by NA^* may be exponential in the worst-case.

D. Depth-First Search (dfs) Method

In this section, an alternative method called depth-first branch & bound for graphs with negative arc costs (NDFBB) is described. It is based on depth-first branch-and-bound technique (DFBB). The core of this algorithm is a recursive procedure SOLVE (n, g) which is used to explore the graph in a depth-first manner. The parameter n is an intermediate node and g is the cost of the current path from the start node s to the node n. As in

the case of DFBB method, an upper bound UB of the absolute cost is used in our algorithm. The algorithm starts by calling *SOLVE* $(s, 0)$. The procedure SOLVE (. , .) explores the successors of s in a depth-first manner by recursively calling itself. The downward movement along a path is controlled by an upper bound UB, a lower bound LB and the absolute cost of the currently known most promising path through the explored nodes. Usually, LB is set to 0. However, it can be set to some positive high value based on some *a priori* knowledge of the problem domain. In that case search terminates faster.

As in the previous case, each node n of the graph has two bounds $\ell(n)$ and $u(n)$. Initially, they are set to ∞ and $-\infty$, respectively, for all nodes except for r for which these are set to 0.

In this case, since the search graph is a tree, there exists only one path from s to any node n. Thus, $f(n) = f(n, P)$, $g(n) = g(n, P)$. Moreover, for simplicity, we will use the notations $f_{\text{low}}(n)$ and $f_{\text{high}}(n)$ for $f_{\text{low}}(n, P)$ and $f_{\text{high}}(n, P)$, respectively. The search starts from the source node s and proceeds in a depth-first manner. Let n be the current node under processing and is reached from s along a path P with a cost g. It computes $f_{\text{low}}(n) = g + \ell(n)$ and $f_{\text{high}}(n) = g + u(n)$ and then $f(n) = \min(|f_{\text{low}}(n)|, |f_{\text{high}}(n)|)$ which is used to reset UB to $\min(f(n), UB)$ during the search. In case UB is updated, the path P is stored in an array *outpath*.

Lemmas 2 and 3 suggest that if node n is informed along a path P, each of its successors will also be informed along the same path and have the same f*-value*.

If n is found to be blind along P and the updated value of UB is equal to LB, there is no need to explore further. Thus, if n is found to be blind along P, the successors of n are explored recursively until $UB = LB$ or some informed node is reached. On termination, UB will contain the minimum value of $f(n)$, considering all nodes n of the graph, i.e., the optimum value of the absolute cost. The optimum solution path is obtained from *outpath*.

The algorithm *NDFBB* is described as follows.

```
Algorithm NDFBB
begin
    UB := ∞; LB := 0;
    SOLVE (s, 0);
    output UB as the solution cost;
    let outpath contain a path from node s to an in-
    termediate node n;
    begin
        trace the remaining path from n to r and de-
    note it by rempath;
        output the concatenation of outpath and rem-
    path as the optimal path;
    end;
end.
Procedure SOLVE (node: n, cost: g)
begin
    push n in STACK;
    if UB = LB then prepare outpath by popping the
    STACK; return;
    if (UB > LB and n is informed) then begin
        prepare outpath by popping the STACK; return;
    end;
```

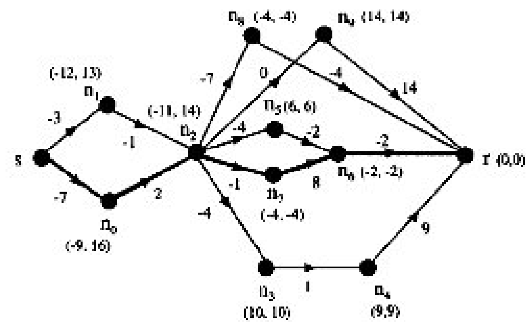


Fig. 8. Snapshot of our depth-first search *(dfs)* algorithm.

```
    if (UB > LB and n is blind) then
        (*f_low(n) and f_high(n) cannot give the absolute
        cost of the optimal path passing through n*)
    begin
        for every successor n_i of n do
        begin
            f_low(n_i) := g + c(n, n_i) + ℓ(n_i);
            f_high(n_i) := g + c(n, n_i) + u(n_i);
            f(n_i) : min(f_low(n_i), f_high(n_i));
            if f(n_i) < UB then
            begin
                UB := f(n_i);
                outpath    := the contents of STACK along
    with n_i;
            end;
            SOLVE (n_i, g + c(n, n_i));
        end;
        if (bounds of n are not yet defined) then
        begin
            ℓ(n) := min(ℓ(n), c(n, n_i) + ℓ(n_i));
            u(n) := max(ℓ(n), c(n, n_i) + u(n_i));
        end;
    end;
    pop the STACK; return;
end;
```

Theorem 2: The algorithm *NDFBB* terminates and always outputs the path having optimal cost.

Proof: Since we are dealing with finite acyclic graphs, the algorithm must terminate. The second part of the proof follows from the formal description of *NDFBB*. ■

E. Example 2

We now explain the algorithm with an example. Consider the graph in Fig. 8. It has 12 nodes $s, n_0, n_1, n_2, \ldots, n_9, r$, 16 edges and ten paths from s to r. The edge weights are shown beside the edges and values of $\ell(n)$ and $u(n)$ for each node n are shown along with the nodes. The optimal path is shown with thick lines and has absolute cost zero. We now explain the execution steps of the algorithm.

Initially s is explored and UB and LB are set to the $\min(|\ell(s)|, |u(s)|)$ and 0, respectively.

One of the successors of s, say n_0, is then expanded. The procedure SOLVE(.,.) recursively explores along the path $s \rightarrow n_0 \rightarrow n_2 \rightarrow n_3$ and finds that node n_3 is informed. It sets UB

TABLE II
SUMMARY OF RESULTS FOR GEOMETRIC BIPARTITIONING

Examples	# vertices	Av. absolute cost	*NDFBB*		NA*	
			Av. nodes generated	Av. CPU time (secs.)	Av. nodes generated	Av. CPU time (secs.)
EX1	50	0.00	52.55	0.00015	148.35	0.00046
EX2	100	0.00	104.17	0.0024	403.53	0.00146
EX3	150	0.00	155.22	0.0011	739.23	0.00316
EX4	200	0.00	204.99	0.00165	1155.92	0.00533
EX5	250	0.00	255.8	0.0019	1654.16	0.00813
EX6	300	0.00	305.35	0.0022	2222.63	0.0115
EX7	350	0.00	357.52	0.0024	2883.81	0.0156
EX8	400	0.00	408.04	0.0029	3608.82	0.0201
EX9	450	0.00	455.91	0.0031	4436.46	0.0254
EX10	500	0.00	510.48	0.0035	5327.53	0.0312
EX11	600	0.00	610.16	0.0046	7359.18	0.0447
EX12	800	0.00	816.82	0.008	12411.37	0.0716
EX13	1000	0.00	1014.42	0.012	15306.81	0.1181
EX14	5000	0.00	4562.25	0.042	19123.1	1.2522
EX15	10000	0.00	9218.12	0.095	24256.72	13.56
EX16	15000	0.00	14123.66	0.133	30123.74	147.12

to the value of the absolute cost of the path with initial segment $s \rightarrow n_0 \rightarrow n_2 \rightarrow n_3$, which is equal to 1.

Since $UB > LB$, the algorithm backtracks to n_2 and explores its successor n_5. Node n_5 is observed informed with absolute cost $= 10$.

As $UB < 10$, it is not updated. The algorithm backtracks to n_2 and explores n_7. Node n_7 is again observed informed with absolute cost $= 0$.

Now UB is updated to 0 and the algorithm terminates. Currently, *outpath* contains $s \rightarrow n_0 \rightarrow n_2 \rightarrow n_7$. From n_7 it reaches r through the path $n_7 \rightarrow n_6 \rightarrow r$. Thus, the reported optimal path is $s \rightarrow n_0 \rightarrow n_2 \rightarrow n_7 \rightarrow n_6 \rightarrow r$, with absolute cost $= 0$.

V. EXPERIMENTAL RESULTS

We have implemented both the algorithms NA^* and *NDFBB* in C on a DEC Alphastation 250 running at 266 MHz clock rate. Experiments are performed on various randomly generated problem instances. The average CPU time and average number of nodes generated are observed for 100 different instances of each of the examples.

In the implementation of NA^*, the *OPEN* list is carefully implemented as a priority queue. For each node, a bit is used which is set to 0 or 1 depending on the node being in *OPEN* or in *CLOSED*. The bounds are observed to be very sharp so as to prune off most of the nonpromising nodes.

For geometric bipartitioning, NA^* is found to be relatively slow and the number of nodes expanded is quite high. For area-balanced bipartition, both the algorithms perform almost equally well.

A. Geometric Bipartitioning

For each problem instance, M points are first generated in a plane rectangular region R. The DAG is created using a plane sweep technique. All pairs of points satisfying monotonicity and nontransitivity relation are connected by directed edges. The left-bottom and right-top corners are assumed to be the source and the goal nodes respectively of the graph. The weight of each node (a real number) is generated using a uniform random number generator. Edge weights are then computed as in Section II by sweeping a horizontal line from the bottom boundary of R to its top.

Performances for both *NDFBB* and NA^* are reported in Table II for M varying from 50 to 15 000. For NA^*, the number of node generations and CPU time, are much higher compared to those of *NDFBB*.

B. Area-Balanced Bipartitioning in VLSI Floorplans

We have considered a benchmark floorplan [20] with 24 blocks and some randomly generated floorplans with number of blocks ranging from 60 to 400. Area of each block is generated using a uniform random number generator. For testing the performance of the proposed algorithm we have scaled up the area values to order of 10^6. Results are summarized in Table III. Interestingly, for all the examples, the nodes generated by *dfs* is larger than that by *bfs*, though the CPU times are comparable.

VI. APPLICATION OF PROPOSED SCHEMES TO NUMBER PARTITIONING

Number partitioning is a well-known NP-complete problem [8]. Its corresponding NP-hard optimization problem may be stated as follows.

Instance: A finite set $\mathcal{A}$ of positive numbers, $a_1, a_2, \ldots, a_m$, $m = |S| \geq 2$.

Output: A partition $(\mathcal{A}_1, \mathcal{A}_2)$ of $\mathcal{A}(\mathcal{A}_1 \cup \mathcal{A}_2 = \mathcal{A})$ is such that $\mathcal{A}_1 \neq \phi$, $\mathcal{A}_2 \neq \phi$, $\mathcal{A}_1 \cap \mathcal{A}_2 = \phi$ and $|\sum_{a_j \in \mathcal{A}_1} a_j - \sum_{a_j \in \mathcal{A}_2} a_j|$ is minimum.

The number-partitioning problem is hard, in theory as well as in practice. Special purpose heuristics are known for this problem. e.g., the method by Karmarkar and Karp [9]. Subsequently, Korf reported an optimal solution technique called the *Complete Karmarkar–Karp (CKK)* algorithm using branch and bound [10].

A. Formulation of the Problem

Number partitioning problem can be mapped to a search problem in graphs with unrestricted edge costs. Without loss of generality, we consider partitioning of a set of integers. Let m

TABLE III
SUMMARY OF RESULTS FOR AREA-BALANCED BIPARTITIONING

Examples	# blocks	Av. absolute cost	*NDFBB*		NA*	
			Av. nodes generated	Av. CPU time (secs.)	Av. nodes generated	Av. CPU time (secs.)
EX1[20]	24	18	78	0.0000002	24	0.0000005
EX1	60	366853	97	0.0000011	25	0.0000005
EX2	70	58350	140	0.0000011	31	0.0000005
EX3	80	74292	164	0.0000015	40	0.0000013
EX5	90	36837	342	0.000008	159	0.000002
EX6	130	28620	256	0.000003	56	0.000001
EX7	160	47240	647	0.000009	281	0.00000201
EX8	175	25954	312	0.0000085	44	0.000002
EX9	200	39532	363	0.0000085	49	0.000000447
EX10	250	20186	1319	0.00008	698	0.00000816
EX11	350	1976	1317	0.000062	421	0.0000085
EX12	400	105835	366	0.0000082	173	0.0000011

TABLE IV
SUMMARY OF RESULTS FOR NUMBER PARTITIONING (10^{10})

Example	# numbers partitioned	Av. difference	*NDFBB*		NA*	
			Av. # nodes generated	Av. CPU time (secs.)	Av. # nodes generated	Av. CPU time (secs.)
NP1	10	51841168.46	46.76	0.00001	96.86	0.01
NP2	20	70287.65	13547.6	0.00001	41120.64	0.527
NP3	30	75.82	5928092.32	0.000054	28061220.8	420.14
NP4	40	0.47	94187662.87	0.000045	56113214.34	942.11
NP5	50	0.63	102354.97	0.000035	*	*
NP6	60	0.43	9092318.15	0.00003	*	*
NP7	70	0.49	9594.87	0.000025	*	*
NP8	100	0.47	596.95	0.000017	*	*
NP9	125	0.40	197.44	0.000017	*	*
NP10	150	0.48	158.34	0.000015	*	*
NP11	175	0.54	171.76	0.00001	*	*
NP12	200	0.56	195.38	0.00001	*	*

* extremely large number of nodes and CPU time

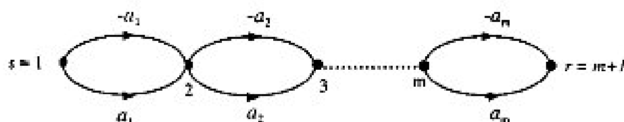


Fig. 9. Graph model for the number partitioning problem.

be the number of integers having elements $a_i, i = 1, \ldots, m$. The set of numbers is to be partitioned into two disjoint subsets such that the difference of the individual sums of the two subsets is minimum.

We construct a graph with nodes linearly ordered as shown in Fig. 9; the source node is s and the goal node is r. Each pair of nodes $(i, i+1)$, $i = 1, \ldots, m$ is connected with two directed edges having costs a_i and $-a_i$, respectively. In a solution path, all the edges having positive costs will form one subset; the edges having negative costs will form the other subset. The absolute value of the sum of the costs of all the edges along a solution path gives the difference of the sums of the numbers in the two subsets. Hence, the number partitioning problem can be solved by finding a path from s to r in this graph with minimum absolute cost.

B. Empirical Observations

A parameter R_n giving the range of the random integers is used. The value of m is varied from 10 to 200 and R_n is varied from 0 to 10 billion. A node in the search tree corresponds to a tentative decision on the relative positions of a subset of numbers of the input set. The numbers to be considered are always preserved in descending order.

The proposed *dfs* algorithm for this problem terminates when the value of UB is 0 or 1. Such cases are called *perfect partitions* [10]. In *NDFBB*, the descending order of the numbers to be partitioned is always maintained, considering $g(n, P)$ along path P from s to n for an intermediate node n as one of the numbers participating in the partitioning. If, for some node n, $g(n, P)$ is less than the remaining numbers and the descending order is disturbed, the problem is reconstructed, placing $g(n, P)$ in the right place in the set of numbers. Results obtained are as good as those observed using Korf's implementation in terms of the optimal partition value and the number of nodes generated. We could also optimally partition 40 48-bit double precision integers in a time of about 20 min.

The results obtained for random integers in the range of 10^{10} are shown in Table IV. The number of integers to be partitioned is assumed to be within a range from 10 to 200. The consolidated results for *NDFBB* appears in Fig. 10. Each data point is computed based on the average of 100 random problem instances. The horizontal axis shows the number of integers to be partitioned. For the continuous curve, the vertical axis represents the number of nodes generated, while for the dotted curve, it represents the average absolute cost of the solution.

It is interesting to note that the trends observed in our graphical representation are similar to those of Korf's [10]. NA^*,

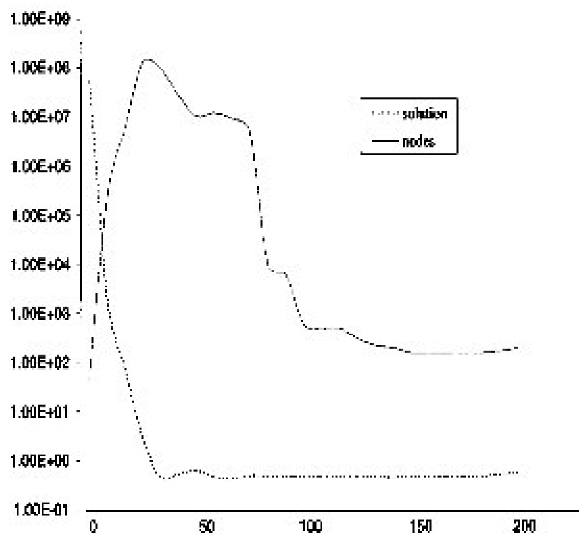


Fig. 10. Nodes generated for optimal partitioning of 10-digit integers.

however, quickly exhausts memory. For more than 40 integers, NA^* is unable to reach a solution within a reasonable amount of CPU time.

VII. CONCLUSION

The main contribution of this paper is to demonstrate the need of new search algorithms to solve practical problems, where edge costs in the search graph are unrestricted and the objective function is of more general nature. These problems arise in computational geometry and VLSI layout design. The classical number partitioning problem can also be formulated as a special case in this framework. Novel heuristic search algorithms are developed for solving such general problems. The proposed methods are guaranteed to terminate with the optimum solution. Further theoretical and experimental studies are needed for evaluating efficacy of these algorithms to other similar problems of interest.

ACKNOWLEDGMENT

The authors would like to thank Professor R. E. Korf of the University of California, Los Angeles, for providing the code for the *CKK* algorithm.

REFERENCES

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network Flows*. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[2] D. P. Bertsekas and J. N. Tsitsiklis, "An analysis of stochastic shortest path problems," *Math. Oper. Res.*, vol. 16, no. 3, pp. 580–595, 1991.

[3] F. Conti *et al.*, "On a 2-dimensional equipartition problem," *Euro. J. Oper. Res.*, vol. 113, pp. 215–231, 1999.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, pp. 269–271, 1959.

[6] P. S. Dasgupta *et al.*, "Monotone bipartitioning problem in a planar point set with applications to VLSI," *ACM Trans. Design Automat. Electron. Syst.*, to be published.

[7] S. Das, S. Sur-Kolay, and B. B. Bhattacharya, "Routing of L-shaped channels, switchboxes and staircases in Manhattan-diagonal model," in *Proc. Int. Conf. VLSI Design*, 1998, pp. 65–70.

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

[9] N. Karmarkar and R. M. Karp, "The differencing method of set partitioning," Comput. Sci. Div. (EECS), Univ. California, Berkeley, Tech. Rep. UCB/CSD 82/113, 1982.

[10] R. E. Korf, "A complete anytime algorithm number partitioning," *Artif. Intell.*, vol. 106, pp. 181–203, 1998.

[11] A. Martelli, "On the complexity of admissible search algorithms," *Artif. Intell.*, vol. 8, pp. 1–14, 1977.

[12] S. Majumder, S. C. Nandy, and B. B. Bhattacharya, "Partitioning VLSI floorplans by staircase channels for global routing," in *Proc. Int. Conf. VLSI Design*, 1998, pp. 59–64.

[13] S. Majumder *et al.*, "Area (number)-balanced hierarchy of staircase channels with minimum crossing nets," in *Proc. Int. Symp. Circuits Syst. (ISCAS)*, vol. 5, 2001, pp. 395–398.

[14] N. J. Nilsson, *Artificial Intelligence: A New Synthesis*. San Mateo, CA: Morgan Kaufmann, 1998.

[15] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA: Addison-Wesley, 1984.

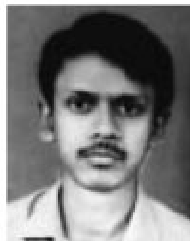
[16] A. Schobel, "Locating least-distant lines in the plane," *Eur. J. Oper. Res.*, vol. 106, pp. 152–159, 1998.

[17] N. Sherwani, *Algorithms for VLSI Physical Design Automation*, 3rd ed. Boston, MA: Kluwer, 1999.

[18] A. K. Sen and A. Bagchi, "Graph search methods for nonorder preserving evaluation functions: Applications to job sequencing problems," *Artif. Intell.*, vol. 86, pp. 43–73, 1996.

[19] R. E. Tarjan, *Data Structures and Network Algorithms*. Philadelphia, PA: SIAM, 1983.

[20] S. Wimer, I. Koren, and I. Cederbaum, "Optimal aspect ratios of building blocks in VLSI," *IEEE Trans. Computer Aided Design*, vol. 8, no. 2, pp. 139–145, 1989.

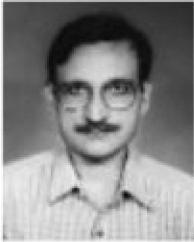


Parthasarathi Dasgupta received the B.Tech. degree in radiophysics and electronics and the M.Tech. and Ph.D. degrees in computer science from the University of Calcutta, India, in 1983, 1985, and 1997, respectively.

Currently, he is a Project Scientist with the Department of Computer Science and Engineering, University of California, San Diego. He is on leave from the MIS Group, Indian Institute of Management (IIM), Calcutta. His current research interests include VLSI CAD, AI and analysis of algorithms.

Anup K. Sen received the Ph.D. degree in computer science from the University of Calcutta, India, in 1990.

Currently, he is a Professor of Computer Science and Information Systems at the MIS Group, Indian Institute of Management (IIM), Calcutta. His previous assignments include long and short term visits to New Jersey Institute of Technology, Newark, Washington University, St. Louis, MO, and University of Iowa, Iowa City. His major research interests include the theory and applications of heuristic search algorithms. He is also interested in single machine job sequencing, metaheuristic techniques and constraint satisfaction problems. He has a number of journal publications in *Artifical Intelligence*, *European Journal of Operations Research*, and *Computers and Operations Research*. He has also published in well known refereed conference proceedings such as *International Joint Conference on Artificial Intelligence*, the *National Conference on Artificial Intelligence*, and the IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS.



Subhas C. Nandy received the M.Sc. degree in statistics from the University of Calcutta, India, the M.Tech. degree in computer science from the Indian Statistical Institute (ISI), Calcutta, and the Ph.D. degree in computer science from the University of Calcutta, in 1982, 1985, and 1996, respectively.

He is currently a Faculty Member at ISI. From 1998 to 1999, he was a Research Associate with the Japan Advanced Institute of Science and Technology (JAIST), Ishikawa, Japan. In the summer of 2000, he was a Guest Researcher at the City University of Hong Kong, Kowloon. His current research interests include the algorithmic aspects of computational geometry and VLSI design.

Bhargab B. Bhattacharya (SM'94) received the B.Sc. degree in physics, the B.Tech. and M.Tech. degrees in radiophysics and electronics, and the Ph.D. degree in computer science, all from the University of Calcutta, in 1971, 1974, 1976, and 1986, respectively.

Since 1982, he has been on the Faculty of the Indian Statistical Institute (ISI), Calcutta, where he became Full Professor in 1991. From 1985 to 1987, he was a Visiting Professor with the University of Nebraska, Lincoln (UNL). He has been on sabbatical from UNL since January 2001. In the summers of 1998, 1999, and 2000, he visited as a Guest Professor in the Fault-Tolerant Computing Group, Institute of Informatics, University of Potsdam, Germany. His research interests include logic synthesis, VLSI design and test and image processing. He has published more than 100 papers in archival journals and refereed conference proceedings.

Dr. Bhattacharya is a Fellow of the Indian National Academy of Engineering. He served on the Conference Committees of the International Test Conference (ITC), the Asian Test Symposium (ATS), the VLSI Design and Test Workshop (VDAT), International Conference on Advanced Computing (ADCOMP) and the International Conference on High Performance Computing (HiPC). For the International Conference on VLSI Design, he worked as Tutorial Co-Chair in 1994, Program Co-Chair in 1997 and as General Co-Chair in 2000.