Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2001), *15*, 67–80. Printed in the USA.

Improving efficiency of procedures for compositional synthesis by using bidirectional search

AMARESH CHAKRABARTI

Engineering Design Centre, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, United Kingdom

(RECEIVED September 1, 1998; ACCEPTED April 5, 2000)

Abstract

This article is an attempt to improve the efficiency of procedures for compositional synthesis of design solutions using building blocks. These procedures have found use in a wide range of applications, and are one of the most substantial outcomes of research into automated synthesis of design solutions. Due to their combinatorial nature, these procedures are highly inefficient in solving problems, especially when the database of building blocks for synthesis or the problem size is large. Previous literature often focuses on improving only the algorithm part of a procedure, although it is both its algorithm and database which together determine the overall efficiency of the procedure. This article reports the construction and analysis of an improved algorithm, based on bidirectional search, for efficient compositional synthesis of design solutions using a set of building blocks.

Keywords: Automated Compositional Synthesis; Bidirectional Search; Concept Generation; Efficient Exhaustive Search Algorithm; Engineering Design

1. INTRODUCTION

Computational systems for compositional synthesis, where a set of building blocks is composed into networks as solutions for design problems, have been found suitable for various applications (Pahl & Beitz, 1984; Prabhu & Taylor, 1988; Hoover & Rinderle, 1989; Ulrich & Seering, 1989; Finger & Rinderle, 1990; Hoeltzel & Chieng, 1990; Kota & Chiou, 1992; Malmqvist, 1993; Chakrabarti & Bligh, 1994, 1996*a*, 1996*b*, 1996*c*; Welch & Dixon, 1994; Sushkov et al., 1996). Each of these systems requires a database of building blocks. Building blocks are simpler, constituting elements of the solutions found in the domain of application. For instance, Ulrich and Seering use bond graph terminology (Paynter, 1961) to describe various conceptual building blocks (such as springs and gears) in the physical systems domain, Hoover and Rinderle (1989) use various gear pairs, Chakrabarti and Bligh (1994) use various motion elements, and Kota and Chiou (1992) use matrices representing the properties of kinemåtic pairs as building blocks. The algorithms are essentially combinatorial in nature, often with a "generate and test" flavor.

The two parameters that synthesis procedure researchers should be, and often have been, most concerned with are: (1) the effectiveness of a synthesis procedure, and (2) its efficiency. A synthesis procedure should be both effective and efficient. Effectiveness is the ability to synthesize new and interesting concepts; the issue of the effectiveness of a compositional synthesis procedure has been discussed elsewhere (Chakrabarti, 1998).

The importance of having a comprehensive database of building blocks has long been recognized as essential for generating new and interesting concepts, and work has been going on in a number of groups towards developing comprehensive databases of building blocks for generating ideas (Roth, 1970; Selutsky, 1987; Ishii et al., 1994; Tsourikov, 1995; Taura et al., 1996; Sushkov et al., 1996; Chakrabarti et al., 1996; Khang, 1998). However, efficiency of a synthesis procedure, which is defined here as the inverse of computational effort (i.e., computation and/or memory) required for the procedure to generate a given solution set, is particularly important for the procedure to be able to generate new and interesting concepts using a reasonable amount of effort, especially when, to achieve this, it may have to

Reprint requests to: Amaresh Chakrabarti, Engineering Design Centre, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge CB2 1PZ, United Kingdom. E-mail: ac123@eng.cam.ac.uk

Fig. 1. The general form of design problems under consideration.

use very large databases. A procedure consists of two things: an algorithm and a database of building blocks. Previous efforts to improving efficiency have concentrated primarily on the algorithm part. We wish to improve the efficiency of both the algorithm and the database to bring about an overall improvement. This article concentrates on the efficiency aspect of a compositional synthesis algorithm.

In some earlier articles, an algorithm was proposed for exhaustive compositional synthesis of solution principles (Chakrabarti, 1995; Kiriyama & Johnson, 1995; Chakrabarti et al., 1997). A comparison between the solutions generated by the program and those considered independently by the designers (Burgess et al., 1995) in a case study reported by Chakrabarti (1996) showed that the computer suggested a wider range of interesting principles than designers considered on their own. This demonstrated that solutions generated by such computational-synthesis procedures could be helpful in expanding the designers' minds into a wider range of new and interesting ideas than possible at present. However, when the database of building blocks is large, the algorithm tended to be inefficient: it took a long time to run, and the memory required was huge. This article presents a new algorithm, based on bidirectional search, which, given the same database of building blocks, generates the same set of solutions using much less computation time and memory.

2. DESIGN PROBLEMS CONSIDERED

The design problems considered here are those which can be expressed in terms of a function transforming a given input into a required output. For instance, a sensing problem can be expressed in terms of an input signal to be sensed (such as an acceleration in the case of an accelerometer problem), and an output medium (Chakrabarti et al., 1997) in which this signal is to be sensed (such as an electrical voltage). In the medical-device domain, a drug infusion problem could be expressed in terms of an input signal producing the effect of a drug flow of some amount. In a mechanical device domain, an example would be a door-locking problem whereby a given input motion of the door handle leads to a retraction motion of the latch. Although not all design problems can be expressed in terms of inputs and outputs (Umeda & Tomiyama, 1997), it is a representation which finds its use in a large variety of problems in many domains of application. The common form for these functions is shown in Figure 1.

3. DESIGN SOLUTIONS CONSIDERED

A design solution (also referred to in this article as a solution principle) under the scope of this work is one that can be expressed as a composition of building blocks such that they are connected via their inputs and outputs to transform the given input of the design problem into its required output. Each building block, therefore, has an input and an output, and the building blocks, constituting a solution principle, transform the given input into a number of intermediate I/O variables before producing the required output. An accelerometer solution principle, for instance, can be a composition of three building blocks, an *inertia* block to transform the input acceleration into an inertia force, a *spring* building block to transform this force into a change in position, and a *capacitance* block to alter the voltage across a capacitor as a result of a change in capacitance due to the position change. A building block, in the context of the door-locking problem, could be a *cam* block transforming the rotation of the handle into a translational motion and a *tie-rod* block transferring this motion to a different output position. These building blocks, as seen from the above two examples, could be at different levels of abstraction, varying between abstract physical effects such as inertia, to concrete physical devices such as cams and tie-rods. A large part of design solutions in existence, as well as a substantial number of computer programs for design synthesis, are compositional in nature (e.g., Hoover & Rinderle, 1989; Ulrich & Seering, 1989; Finger & Rinderle, 1990; Kota & Chiou, 1992). Therefore, this form of solutions (Fig. 2) is fairly generic in its use in research as well as in applications.

4. THE SYNTHESIS ALGORITHMS

The original algorithm uses unidirectional search, whereas the new algorithm uses bidirectional search.

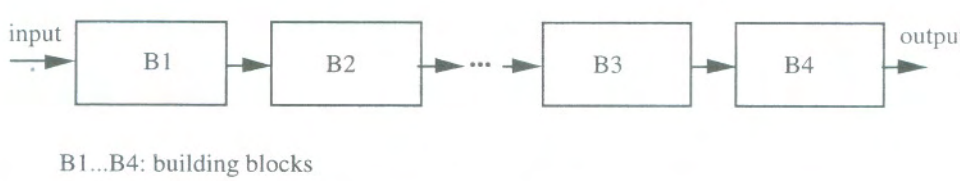

Fig. 2. The general form of the design solutions considered.

4.1. Unidirectional search algorithm

Figure 3 is used to illustrate how this algorithm works. Suppose the synthesis problem is defined as that of synthesizing, using a database of 19 building blocks, all possible solutions, each having three building blocks, for transforming a given "input" into a required "output." The algorithm proceeds by first finding all the building blocks that can take the given "input" as its input (in the case in Fig. 3, there are three such building blocks: B1, B2, and B3). For each of these building blocks, its output is now taken as the given "input," and search proceeds to find possible alternative building blocks that can take this new "input" as its input. In the case in Figure 3, B1 can be connected only to B4, B2 to B5 or B6, and B3 to B7 or B8. By now, two of the allowable number of three building blocks for forming a solution have already been consumed, as the search has proceeded through two steps. For each of the partial solutions that are formed (such as B1 connected to B4), its output is determined as that of its final building block (in the case of B1 connected to B4, it is the output of B4), and this is taken as the given input for searching for building blocks in the next (and in this case, last) step. This produces, in the case in Figure 3, B9 and B10 as possibilities for connecting to B4, B11 and B12 for connecting to B5, and so on, forming chains having three building blocks such as B1 connected to B4 connected to B9. The output of each such chain (which is given by the output of the last building block in the chain, such as that of B9 in the case of the chain containing B1, B4, and B9), is now checked to see if it is the same as the required output. The chains for which this is the case are stored as possible solution alternatives to the given problem (in the case in Fig. 3, these are B2-B5-B11, B2-B6-B13, and B3-B8-B17), while the others are discarded. The algorithm, therefore, consists of two steps: (1) formation of chains of compatible building blocks of specified size capable of accepting as input the given "input," and (2) checking these for their capability of providing the required "output."

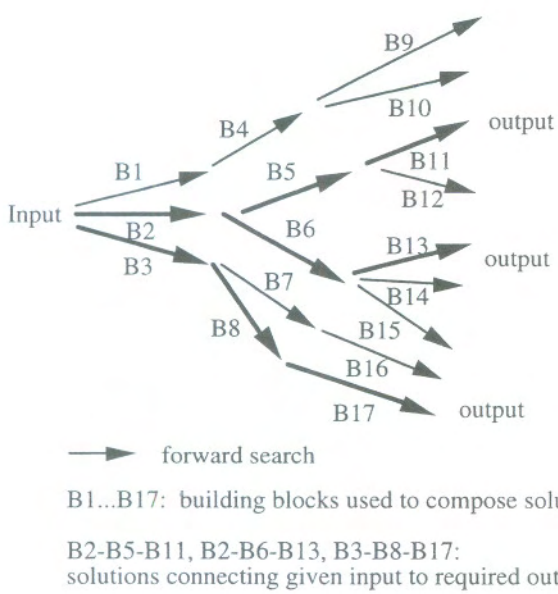

Fig. 3. Unidirectional search process.

4.2. Bidirectional search algorithm

The concept of bidirectional search is not new. It has been investigated in the past, for a number of search problems, as a potential solution to computational problems associated with unidirectional search. Pohl (1971), and De Champeaux and Sint (1977) used various heuristic versions of bidirectional search for solving shortest-path and network-flow problems, and Ishida (1996) used it for real-time search of mazes and n-puzzles, with various degrees of success. However, bidirectional search has never been used in exhaustive compositional design synthesis, where the problem is to find all possible paths of specified size between an input state and an output state. In bidirectional search, search proceeds from two directions. The central idea is that, instead of generating complete chains of building blocks having specified size before checking their capability of providing the required output, two search processes, one starting from the input and the other from the output, should synthesize two sets of partial chains, one of which can take as input the given input whereas the other can provide the required output, and together have the same number of building blocks as allowed in a single solution. Each chain from one set would then be checked against each from the other to ensure that they can be connected together, that is, the output from the chain that can accept the given "input" as its input matches the input of the chain that can provide the required "output."

Figure 4 illustrates one way in which a bidirectional algorithm may operate, for the problems discussed in Sec-

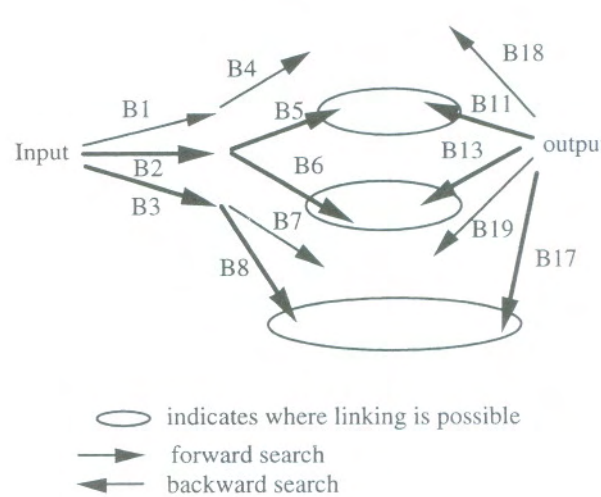

Fig. 4. Bidirectional search process.

tion 2, using a given database. First, the allowable number of steps for search (i.e., the allowable number of building blocks in a single solution) is divided into the number of steps for forward and backward search. In the case shown in Figure 4, it was decided that forward search should generate partial chains having two building blocks, and backward search should generate chains having a single building block. The next step is to generate these partial chains in the same way as in the case of unidirectional search above. This process produces a set of five forward chains (B1-B4, B2-B5, B2-B6, B3-B7, and B3-B8), all of which can accept the given input, and a set of five backward chains (B18, B11, B13, B19, and B17), all of which can provide the required output. The next step is to check their compatibility, that is, matching the output of the forward chains with the inputs of the backward chains. This should lead to concatenation of the compatible chains into complete chains capable of taking the given "input" and providing the required "output." In the case in Figure 4, chains B2-B5, B2-B6, and B3-B8 are compatible, respectively, with B11, B13, and B17, forming complete chains B2-B5-B11, B2-B6-B13, and B3-B8-B17. So, the algorithm has four main steps: (1) determination of the allowable size of partial solutions from forward and backward search, (2) synthesis of forward partial chains as a result of forward search, (3) synthesis of backward partial chains as a result of backward search, and (4) concatenation of compatible forward and backward chains into complete solution chains.

5. RESEARCH METHOD FOR COMPARISON

The research method would be to check to see if (1) the same solution set is generated for the same problem using the same database of building blocks by both the algorithms, at various required solution sizes (i.e., the number of building blocks allowed to compose a single solution); (2) bidirectional search takes less memory and computation time than unidirectional search for solving the same problem using the same database; and, (3) the computational performance of bidirectional search gets increasingly better than that of unidirectional search as the size of the database and the required size of solutions are increased. These measures take different forms from those used by previous researchers in bidirectional search (Pohl, 1971; De Champeaux & Sint, 1977; Russell & Norvig, 1995; Ishida, 1996) due to the difference in the nature of the problem investigated. Among the applications investigated in the past, shortest-path problems (Pohl, 1971; De Champeaux & Sint, 1977) are the closest to the present research. However, the task in these problems is to identify the shortest path of any size (i.e., number of arcs in the path), rather than all possible paths of a prespecified size as in the present problem. Therefore, *intersection* (i.e., that the two search trees, forward and backward, could grow into complete trees without any guarantee of meeting each other) is a serious issue in these problems. This has been addressed, and an improved algorithm developed to alleviate this, by De Champeaux and Sint (1977), where the heuristic function ensures that intersection occurs, although only with a substantial increase in computation. For these problems, $O(b^d)$ (Russell & Norvig, 1995), where b is the branching factor and d the depth of tree searched (both to be defined in Section 6), provides a reasonable measure for the bounds on space and time required for computation. However, the present algorithm is different in at least two ways: (1) it requires that all possible paths between the given input and output are identified, and (2) that each path has the same, prespecified size. These mean that (1) there is a substantial additional computational requirement for concatenating the partial paths, created in forward and backward search, into complete paths from the input to the output node, (2) the check for compatibility between forward and backward partial solutions could be left until they are spawned to their prespecified size, and (3) the problem of intersection is no longer relevant, as one could guarantee intersection by independently generating trees of prespecified sizes so that the sum of their lengths equals the allowable solution size. It is, therefore, possible here to provide closer estimates of computational performance of these algorithms than is possible to infer from estimates in the existing literature, for the class of problems investigated. The following section provides these estimates.

6. ESTIMATION OF COMPUTATIONAL PERFORMANCE OF THE ALGORITHMS

Two estimations are made. One is regarding the amount of memory required, and the other in terms of the number of computations required.

6.1. Comparison of memory required

A search could be seen as a process of spawning a tree, at the beginning of which only the input node is known. At each step of spawning the tree, a set of building blocks from the database is selected and stacked, the input of each of which is the same as that of the input node. At the next step, the same is done for each building block in the stack, where the output of the building block acts as the input to be used for selecting another set of building blocks from the database. The process continues until as many steps as the number of building blocks allowed to be used in a solution are performed. In the case of bidirectional search, a process similar to the above is used twice, each with roughly half the number of steps in the above case, and at the end, the leaf nodes of one tree are compared to those of the other so as to connect the matching branches to form solutions.

Let d be the number of building blocks to be used in one solution, which we will call the *depth of the tree* spawned in one-directional search. The *branching factor*, which is the average number of building blocks selected at each step of the spawning process, is denoted by b here. The maximum

memory required to spawn a tree of depth d and having x number of leaves is given by the product $(b \cdot x)$ of the number of leaves (i.e., branch ends) of the tree and the size of each branch.

The memory required at the end of the first step is $b.1$, since there are b branches, each of size 1 (as each is an alternative building block matching the required input), to be stored for connecting to other building blocks at the next step. At the end of the second step, an average of b building blocks could be connected to each of the branches in the tree stored from the first step, requiring now the storage of a list of size $b.b$ (i.e., a tree having $b.b$ branches), where each element in the list (i.e., each branch of the tree) is of size 2 (i.e., having two building blocks), requiring a total memory of $2.b.b$. At the end of the third step, similarly, the memory required will have been increased to $3.b.b.b$, and so on. So, the maximum memory required for one-directional search for solutions of size d is given by

$$N_1 = d.b^d. \tag{1}$$

Let us formulate a bidirectional search with forward and backward tree depths of d_1 and d_2 respectively such that d, d_1, and d_2 are integers and $d = d_1 + d_2$, and $|d_1 - d_2| = 0$ or 1, depending on whether d is even or odd respectively. The maximum memory required in this search is given by

$$N_2 = d_1.b^{d_1} + d_2.b^{d_2}. \tag{2}$$

The ratio of maximum memory required for the two search processes is

$$\begin{aligned} N_2/N_1 &= (d_1.b^{d_1} + d_2.b^{d_2})/(d.b^d) \\ &= (d_1.b^{d_1} + d_2.b^{d_2})/(d.b^{d_1+d_2}) \\ &= d_1/[(d_1 + d_2).b^{d_2}] + d_2/[(d_1 + d_2).b^{d_1}]. \end{aligned} \tag{3}$$

6.1.1. Case A. For $d_1 = d_2$ (i.e., d is even)

$$N_2/N_1 = 1/b^{d_1}. \tag{4}$$

For $b \geq 1$, $b^{d_1} \geq 1$. Therefore, $N_1/N_2 \geq 1$ for $b \geq 1$. This means that bidirectional search requires more memory than unidirectional search for problems where d is even and $b > 1$.

6.1.2. Case B. For $d_2 = d_1 + 1$ (i.e., d is odd)

$$d_1/(d_1 + d_2) = d_1/(2.d_1 + 1) < 1/2, \tag{5}$$

$$d_2/(d_1 + d_2) = d_2/(2.d_1 - 1) = 1/(2 - 1/d_2). \tag{6}$$

This is maximum when d_2 is minimum, which is 2. Therefore, the maximum value of $d_2/(d_1 + d_2)$ is given by

$$d_2/(d_1 + d_2)_{\max} = 1(2 - 1/2) = 2/3. \tag{7}$$

Using the maximum values of the two ratios from Eqs. (5) and (6) into Eq. (3),

$$N_2/N_1 < (1/2b^{d_2} + 2/3b^{d_1}) = (1/2b + 2/3)/b^{d_1}. \tag{8}$$

$(1/2b + 2/3)/b^{d_1}$ is maximum when d_1 is minimum (as $b > 1$). Therefore,

$$N_2/N_1 < (1/2b + 2/3). \tag{9}$$

A sufficient condition for $N_2/N_1 < 1$ is the following:

$$(1/2b + 2/3) \leq 1 \rightarrow b \geq 1.5. \tag{10}$$

Therefore, even for small problem sizes, bidirectional search consumes less memory than unidirectional search for branching factors larger than 1.5. For instance, for a branching factor b of 2 and a depth of 4, $(N_2/N_1) = 1/2^2 = 1/4$.

6.2. Comparison of the number of computations required

As before, b is the branching factor, that is, the average number of building blocks at each stage of the tree-spawning process. We define d as before, as the number of times the tree is branched out, that is, the number of building blocks allowed to be used in composing a single solution. Let l be the number of building blocks in the database, and v the number of variables these building blocks share among them. If each building block is taken as a link between its input and output variables (nodes), then a database could be seen as a collection of links between the nodes shared by them. We can then define a synthesis problem as that of finding paths between any two prespecified nodes in the database using a predefined number of links between them. Thus, the average number of links available from any node in the database is the average branching factor for this problem, and is given by

$$b = l/v. \tag{11}$$

In the case of one-directional search, at the beginning of the process, each building block in the database needs to be checked to see if its input matches the required input of the problem, thus requiring l checking. Once those building blocks that match are selected (its number is the same on an average as the branching factor b), we need, at the second step, to check, for each of these, each of the building blocks in the database to see which of them match the output of the branches of the tree (each of which contains one building block) at the end of the first step. Thus, this second step ends up on an average with b matches for each building block from the first step, requiring a total of $b.l$ checking on branches of size 1, thus requiring $b.l$ computations; this produces a list of $b.b$ branches, each of the length of two building blocks. In the third step, each building block from the database needs matching against each of the past $b.b$

branches of length two, requiring a further $b.b.l$ checking and consequent $2.b.b.l$ computations, thus producing $b.b.b$ matches as a result of this. This goes on for d steps. Thus the total number of computations required at the end of d steps is

$$M_1 = 1.l + 1.b.l + 2.b.b.l + 3.b.b.b.l + \ldots + (d-1).b^{d-1}.l. \tag{12}$$

In the case of the bidirectional search, two smaller searches go on exactly as in the above one-directional search, with depths d_1 and d_2 such that d_1 and d_2 are integers and $d = d_1 + d_2$, and $|d_1 - d_2| = 0$ or 1, depending on whether d is even or odd respectively. At the end of the two search processes, two sets of partial solutions (the leaf nodes of the trees spawned) are produced, which need to be matched against each other to see which partial solutions from one set could be linked to which from the other so as to form complete solutions as a result. As the size of these sets are b^{d_1} and b^{d_2} (the number of matches at the end of search for d_1 and d_2 steps, respectively), the number of checks required is given by the product of these two, where the size of each branch on which the checking is performed on average is $0.5.(d_1 + d_2)$. So, the maximum total number of computations (which is the product of the number of checks and the size of the branches on which these checks are performed) is given by M_2 as follows

$$\begin{aligned} M_2 = {} & [1.l + 1.b.l + 2.b.b.l + \ldots + (d_1 - 1).b^{d_1-1}.l] \\ & + [1.l + b.l + b.b.l + \ldots + (d_2 - 1).b^{d_2-1}.l] \\ & + 0.5.(d_1 + d_2).b^{d_1}.b^{d_2}. \end{aligned} \tag{13}$$

Therefore, the difference between M_1 and M_2 is given by

$$\begin{aligned} M_1 - M_2 > {} & [d_1.b^{d_1}.l + (d_1+1).b^{d_1+1}.l + \ldots + (d-1).b^{d-1}.l] \\ & - [1.l + b.l + b.b.l + \ldots + (d_2 - 1).b^{d_2-1}.l] \\ & - d_2.b^{d_1}.b^{d_2}, \end{aligned} \tag{14}$$

as $d_2 \geq 0.5.(d_1 + d_2)$.

6.2.1. Case A. $d_1 = d_2 = 1$

$$M_1 - M_2 = (l + b.l) - (l + l + b^2) = b.l - l - b^2. \tag{15}$$

For $M_1 > M_2$, the following condition must hold

$$b.l > l - b^2 \rightarrow b/l < (1 - 1/b), \tag{16}$$

for a problem where d is 2 (as d_1 and d_2 are each 1, and $d = d_1 + d_2$). From Eq. (11), l is b/v. In this case v is at least 3 (as d is 2). Thus the minimum value of l is $3b$, which makes the maximum value of b/l as 1/3. Therefore, a sufficient condition for Eq. (16) to hold is

$$1/3 < (1 - 1/b) \rightarrow b > 1.5. \tag{17}$$

Therefore, for all values of b larger than 1.5, bidirectional search is more efficient than unidirectional search when solutions having two building blocks are generated.

6.2.2. Case B. $d_1 = 1$; $d_2 = 2$

$$\begin{aligned} M_1 - M_2 &= (l + b.l + 2.l.b^2) - (l + b.l + l + b.l + 2b^3) \\ &\rightarrow 2.l.b^2 - (l + b.l + 2b^3). \end{aligned} \tag{18}$$

For $M_1 > M_2$, the following condition must hold:

$$2.l.b^2 > (l + b.l + 2b^3) \rightarrow b/l < (1 - 1/b)(1 + 2/b). \tag{19}$$

For this case, d is 3 (as $d = d_1 + d_2$). From Eq. (11), l is b/v. In this case v is at least 4 (as d is 3, and the number of nodes in the database is larger than d). The minimum value of l is $4b$, which makes the maximum value of b/l as 1/4. Thus a sufficient condition for Eq. (16) to hold is

$$1/4 < (1 - 1/b)(1 + 1/2b) \rightarrow b > 1.22. \tag{20}$$

Therefore, for all values of b larger than 1.22, bidirectional search is more efficient than unidirectional search when solutions having three building blocks are synthesized.

6.2.3. Case C. $d_1 \geq 2$; $d_2 \geq 2$

From Eq. (14), M_1 is greater than M_2 as long as the following inequality holds

$$A > B, \tag{21}$$

where

$$A = d_1.b^{d_1}.l + (d_1+1).b^{d_1+1}.l + \ldots + (d-1).b^{d-1}, \tag{22}$$

$$B = 1.l + b.l + b.b.l + \ldots + (d_2 - 1).b^{d_2-1}.l + d_2.b^{d_1}.b^{d_2}. \tag{23}$$

Now the first term in A is larger than that in B as $d_1.b^{d_1}$ is larger than 1. Similarly, the second term in A is larger than that in B as $(d_1 + 1).b^{d_1+1}$ is larger than b. In a similar way each term in A is larger than its corresponding term in B. Now, A has d_2 terms, whereas B has $(d_2 + 1)$ terms. Therefore, a sufficient condition for $A > B$ to hold is for the last two terms of B to be smaller than or equal to the last term in A. This can be written as

$$(d-1).b^{d-1} \geq d_2.b^{d_1}.b^{d_2} + (d_2 - 1).b^{d_2-1}.l. \tag{24}$$

6.2.3.1. Case C1. $d_1 = d_2$. Equation 24, after using $d_1 = d_2$ and $d = d_1 + d_2$, becomes

$$b/[(2 - 1/d_2).l] + 1/[\{1 + 1/(1 - 1/d_1)\}.b^{d_1}] \leq 1. \tag{25}$$

The left hand side of Eq. (25) cannot be greater than the sum of the maximum values of its two terms. Therefore, if the sum of the maximum values of its two terms is less than or equal to 1, so will be the inequality. The first term is maximum when its denominator is minimum, which is true when d_2 is minimum (i.e., when d_2 is 2). As the minimum

value of l is $5b$ (since l is $v.b$, and since v, the number of variables in the database, has to be at least 5 for synthesizing solutions having four building blocks that must share among them at least five variables). So the maximum value of the first term is given by

$$b/[(2 - 1/d_2).l]_{\max} = b/[(2 - 1/2).l] = 2b/(3.5.b) = 2/15. \tag{26}$$

The second term cannot be larger than $1/2b^{d_1}$. Substituting this value and the value of the first term from Eq. (26) into Inequality (25) provides the following sufficient condition for Inequality (25) to hold:

$$2/15 + 1/(2b^{d_1}) < 1 \rightarrow b^{d_1} > 15/26. \tag{27}$$

This is true for $b \geq 1$. This means that for synthesis of solutions of size 4 onwards, bidirectional search is always more efficient than unidirectional search, as long as the branching factor b is 1 or more.

6.2.3.2. Case C2. $d_1 = d_2 - 1$; $d_1 \geq 2$; $d_2 \geq 3$. Equation 24, after using $d_1 = d_2 - 1$ and $d = d_1 + d_2$, becomes

$$b/[(2 - 2/d_2).l] + 1/(2.b^{d_1}) \leq 1. \tag{28}$$

Inequality 28 cannot be greater than the sum of the maximum values of its two terms. Therefore, if the sum of the maximum values of its two terms is less than or equal to 1, so will be the inequality. The first term is maximum when its denominator is minimum, which is true when d_2 is minimum (i.e., when d_2 is 3). As the minimum value of l is $6b$ (since l is $v.b$, and since v, the number of variables in the database, has to be at least 6 for synthesizing solutions having five building blocks that must share among them at least six variables). So the maximum value of the first term is given by

$$b/[(2 - 2/d_2).l]_{\max} = b/[(2 - 2/3).l] = 3b/(4.6.b) = 1/8. \tag{29}$$

The second term cannot be larger than $1/2b^{d_1}$. Substituting this value and the value of the first term from Eq. (29) into Inequality (28) provides the following sufficient condition for Inequality (28) to hold:

$$1/8 + 1/(2b^{d_1}) < 1 \rightarrow b^{d_1} > 8/14. \tag{30}$$

This is true for $b \geq 1$. This means that for synthesis of solutions of size 5 onwards, bidirectional search is always more efficient computationally than unidirectional search, as long as the branching factor b is 1 or more. For instance, the ratio of number of computations required for the two search algorithms, for a branching factor of 2 and for synthesizing solutions having four building blocks (i.e., $d = 4$) using a small database (with $l = 20$) is given by

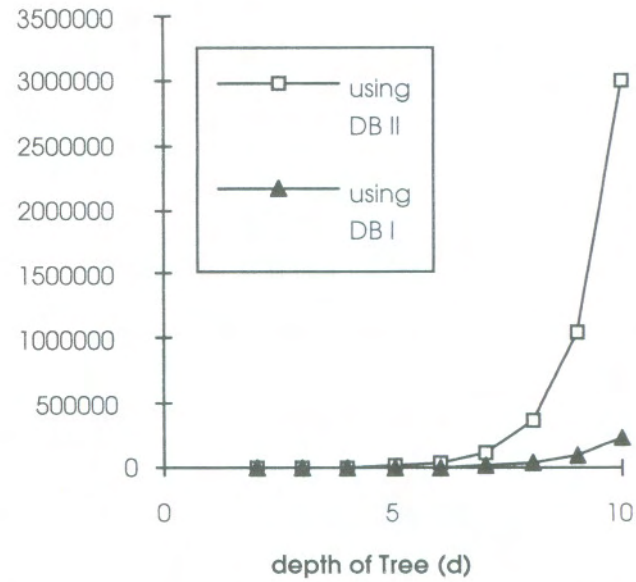

Fig. 5. Number of computations required against tree depth in unidirectional search.

$$M_1/M_2 = 20[1 + (1.2) + (2.2.2) + (3.2.2.2)]/$$
$$[20.\{1 + (1.2) + 1 + (1.2)\} + (2.2^2.2^2)]$$
$$= 700/152$$
$$= 4.6.$$

The plots in Figures 5–10 show, for the two search algorithms, how computational load and memory estimates vary with the searched tree depth (d), and how their ratios vary

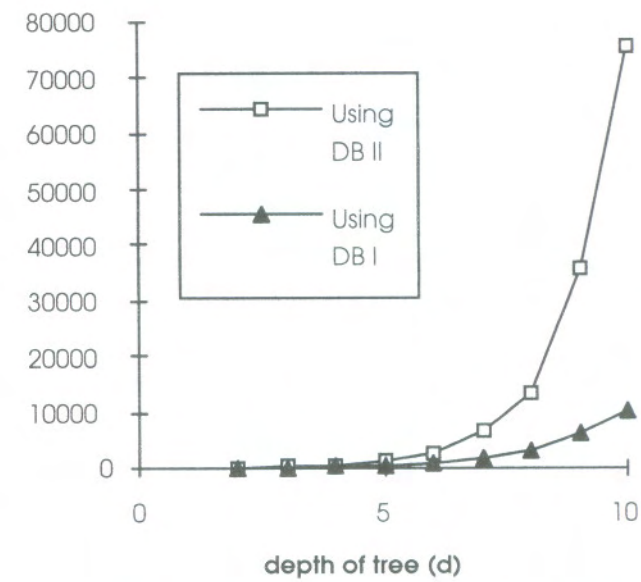

Fig. 6. Number of computations required against depth of tree in bidirectional search.

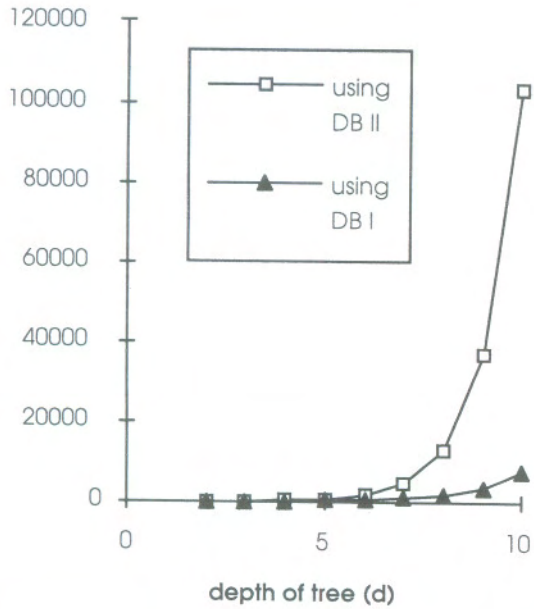

Fig. 7. Amount of memory required against depth of tree in unidirectional search.

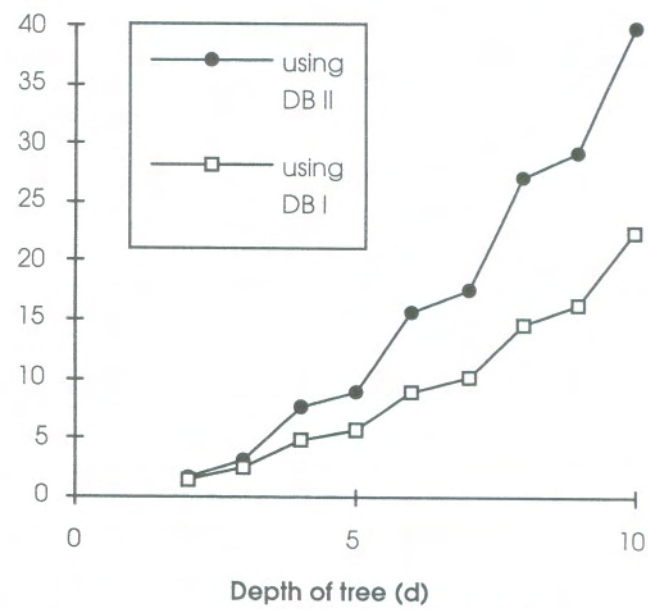

Fig. 9. Ratio of computations required *versus* depth of tree in the two-search algorithms.

with the tree depth. All are plotted for an accelerometer design problem. Figures 5 and 7 show these for the unidirectional search algorithm, and Figures 6 and 8 for bidirectional search. Figures 9 and 10 show the ratio of computation and memory estimates for these two search algorithms. Each figure has two plots, one using a small database (called database I; see the Appendix) which has 35 links (l) and 18 nodes (v), and the other using a larger database (called database II; see the Appendix) having 53 links and 21 nodes.

In general, it can be seen, by comparing the two plots in each of the figures, that the computation and memory required increases with the increase in the size of the database. Bidirectional search does better than unidirectional search, and, as can be seen clearly from Figures 9 and 10, it

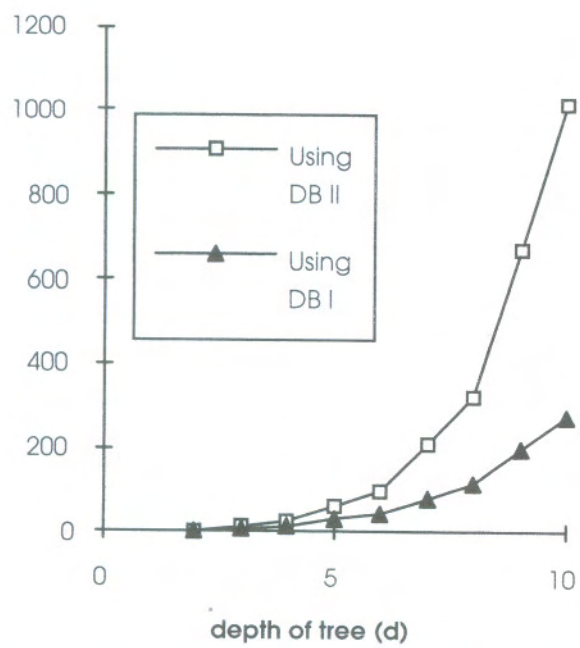

Fig. 8. Amount of memory required against depth of tree in bidirectional search.

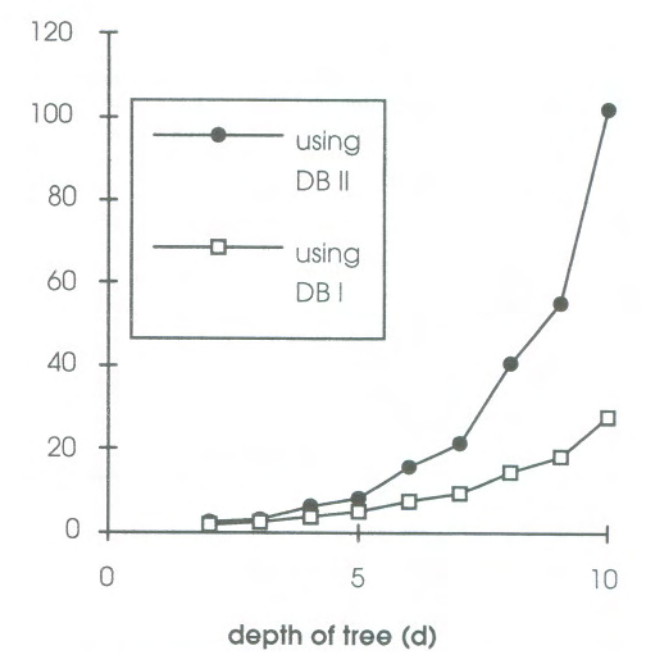

Fig. 10. Ratio of memory required *versus* depth of tree in the two-search algorithms.

does increasingly better than unidirectional search as the size of the solution or size of the database is increased. Even for a small database like database I (35 links) with a low branching factor (1.57), the memory ratio is always in favor of bidirectional search except for the smallest of tree depths (≤ 3), going up as high as 4 for a tree of depth 10 (see Fig. 10). The ratio of the number of computations follows a similar trend (Fig. 9). The effect of a larger database can be seen by comparing the two plots in each figure. The size of database II is less than twice that of database I, having 53 links, and has a branching factor of 2.52, which is a little more than 1.5 times that of database I. However, the ratio of memory required goes up from 20 for database I to 110 for database II at a tree depth of 10, and the ratio of computations goes up from 20 for database I to 40 for database II, showing the advantage of bidirectional search over unidirectional in synthesizing large solutions using large databases. As both the plot-sets show a similar trend, irrespective of their using different databases, this demonstrates the generic nature of the efficiency of bidirectional search over unidirectional search, for the class of problems investigated here.

7. COMPARISON OF THEORETICAL ESTIMATES WITH ACTUAL PERFORMANCE

Figure 11 is a plot of the ratio of the amount of computations required between unidirectional search and bidirectional search against solution size, for an acceleration-sensing problem using the same database of building blocks. Three plots are given: one is an estimate of the computation ratio using the average value of the branching factors in the database, another is an estimate using the actual values of branching factors, and the third one is a plot of the actual ratio of computations. Figure 12 is a similar plot for the ratio of memory required by the two algorithms for the same problem.

The estimate using average branching factors, in this particular case, provides an overestimate of the ratio. This is because in its calculation, the same branching factor is used to calculate computation and memory required for the forward and backward search in the bidirectional search. In reality, the backward branching factor in this case is much larger than the average branching factor. Consequently, backward search takes much more memory and computation in reality than is estimated using the average branching factor. In other words, the ratio of computation or memory in unidirectional search to bidirectional search is overestimated. If actual values of branching factors are used instead, the estimates follow the actual ratios more closely, as can be seen from Figures 11–12. Even then, as the tree depth increases, the estimate becomes more conservative than the actual ratio. This is because the estimate still provides an approximate model of how the algorithms operate, in which the exact number of calculations and the exact size of lists on which these calculations are performed every time a major step is to be taken (such as further proliferation of the tree, or concatenation of two partial branches) are approximated as a single calculation on an average-sized list. For instance, while seven different calculations, each on slightly

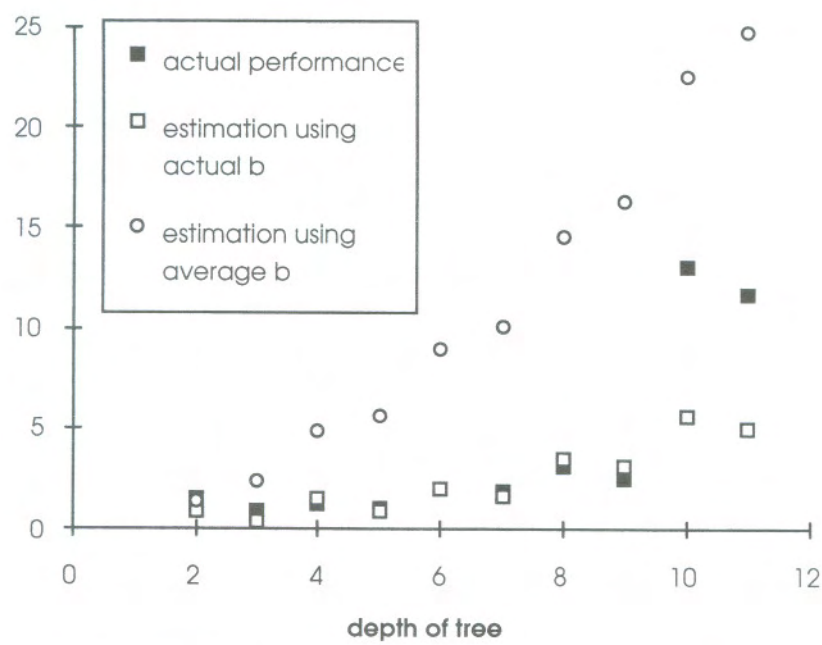

Fig. 11. Comparison of theoretical estimates with actual performance (computation).

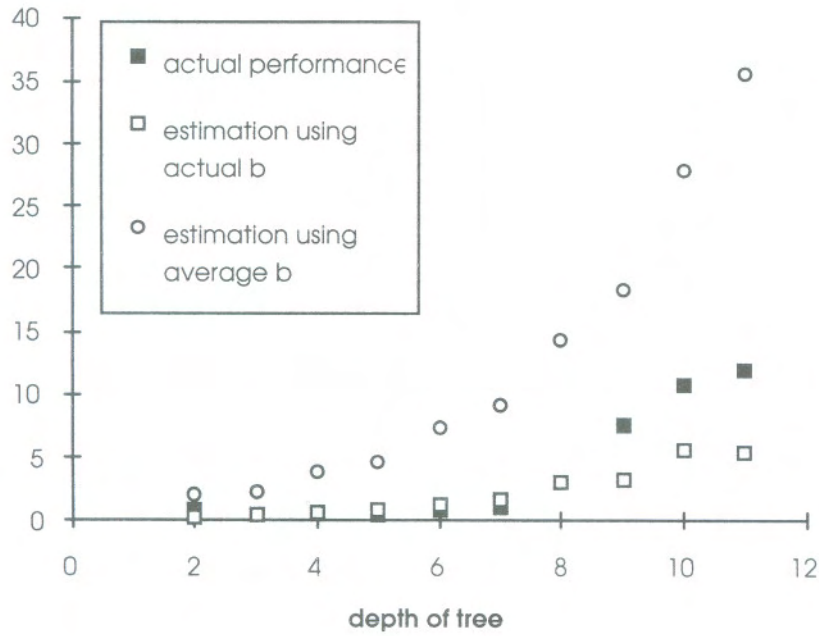

Fig. 12. Comparison of theoretical estimates with actual performance (memory).

different lists are performed on the list of partial solutions from forward and backward searches to concatenate them into complete solutions, this is represented in the estimate as a single calculation on a list of an average size. The estimate, when used with realistic values of the branching factors, however, provides a conservative estimate of how the algorithms compare in their performance.

8. OPTIMAL DIVISION OF PROBLEM SIZE IN THE BIDIRECTIONAL SEARCH ALGORITHM

The memory and computation required for the bidirectional search algorithm is dependent on the size of the tree that its forward and backward search processes need to explore; it is also dependent on the branching factor of the database. This branching-factor sensitivity was first noted by Pohl (1971) in the context of shortest-path problems, where he used this as a basis for a heuristic for dividing search efficiently between forward and backward search. However, the size of the paths to be compared in shortest-path problems can be variable, and therefore an optimum division of tree depths between forward and backward search is not possible. In the problems investigated in this research, a problem of depth of tree d is prespecified, and therefore can be subdivided into tree depths for forward and backward search in a finite number of alternative ways so long as their sum is equal to d, and as the forward and backward branching factors need not be the same, we need to ensure that the forward and backward search depths are chosen so as to give optimal or near-optimal computational performance. In the following, we discuss two cases, one where the forward branching factor is the same as the backward branching factor, and one where they are not.

8.1. Branching factors having the same value

Let the branching factor, as before, be b, and the forward and backward tree depths be d_1 and d_2 respectively. We derive first the optimal value for these for minimum memory and then those for minimum computation.

8.1.1. Minimum memory requirement

A theoretical estimate for the amount of memory required in bidirectional search is given by Eq. (2), which is repeated below:

$$N_2 = d_1.b^{d_1} + d_2.b^{d_2}. \tag{2}$$

This equation has a single minima, which means that any value of the above function will be greater than that given by values of d_1 and d_2 at the minimum value. This is written in terms of the following equations:

$$d_1.b^{d_1} + d_2.b^{d_2} < (d_1 - i).b^{d_1-i} + (d_2 + i).b^{d_2+i} \tag{31}$$

$$d_1.b^{d_1} + d_2.b^{d_2} < (d_1 + i).b^{d_1+i} + (d_2 - i).b^{d_2-i}, \tag{32}$$

where i is any integer value.

8.1.1.1. Case A: $d_1 = d_2$. Equation 31 can be simplified as

$$d_2.b^{d_2}.[b^i.(d_2 + i)/d_2 - 1] > d_1.b^{d_1}.[1 - (d_1 - i)/(b^i.d_2)] \rightarrow$$
$$b^i.(d_2 + i)/d_2 - 1 > 1 - (d_1 - i)/(b^i.d_2) \rightarrow$$
$$(b^i - i)[1 - 1/b^i + i.(b^i + 1)/(b^i.d_1)] > 0. \quad (33)$$

Equation 33 is always greater than zero for b greater than 1, for both $(b^i - 1)$ and $[1 - 1/b^i + i.(b^i + 1)/(b^i.d_1)]$ are greater than zero in this case.

As Eq. (32) is symmetrical to Eq. (31), that can also be proved in a similar way.

8.1.1.2. Case B: $d_2 = d_1 + 1$. Equation 31 can be simplified as

$$d_2.b^{d_2}.[b^i.(d_2 + i)/d_2 - 1] > d_1.b^{d_1}.[1 - (d_1 - i)/(b^i.d_2)] \rightarrow$$
$$(d_2./d_1).b.[b^i.(d_2 + i)/d_2 - 1] > [1 - (d_1 - i)/(b^i.d_2)] \rightarrow$$
$$(b^i - i).[(d_2./d_1).b - 1/b^i + i.(b^i + 1)/(b^i.d_1)] > 0. \quad (34)$$

Equation 34 is always greater than zero for b greater than 1, for both $(b^i - 1)$ and $[(d_2./d_1).b - 1/b^i + i.(b^i + 1)/(b^i.d_1)]$ are greater than zero in this case.

As Eq. (32) is symmetrical to Eq. (31), that can also be proved in a similar way.

8.1.2. Minimum computation requirement

A theoretical estimate for the amount of computation required in bidirectional search is given by Eq. (13) which is repeated below:

$$M_2 = [1.l + 1.b.l + 2.b.b.l + \ldots + (d_1 - 1).b^{d_1-1}.l]$$
$$+ [1.l + b.l + b.b.l + \ldots + (d_2 - 1).b^{d2-1}.l]$$
$$+ 0.5.(d_1 + d_2).b^{d_1}.b^{d_2}. \quad (13)$$

The value of M_2 for $(d_1 - i)$ and $(d_2 + i)$ can be obtained by removing the last i terms from the first list and adding further i terms to the second list in Eq. (13), and is given by

$$M_{2_{new}} = [1.l + 1.b.l + \ldots + (d_1 - 1).b^{d_1-1}.l]$$
$$- (d_1 - 1).b^{d_1-1}.l - (d_1 - 2).b^{d_1-2}.l \ldots - (d_1 - i).b^{d_1-i}.l$$
$$+ [1.l + b.l + \ldots + (d_2 - 1).b^{d_2-1}.l] + (d_2 + i).b^{d_1+i}.l$$
$$+ (d_2 + i - 1).b^{d_2+i-1}.l \ldots + d_2.b^{d_2}.l$$
$$+ 0.5.(d_1 + d_2).b^{d_1}.b^{d_2}, \quad (35)$$

$$(d_2 + i).b^{d_1+i}.l > (d_1 - 1).b^{d_1-1}.l, \quad (36)$$

because for $d_1 = d_2$: $(d_2 + i) \geq (d_2 + 1) > (d_1 - 1)$ and $b^{d_1+i} > b^{d_1-1}$, and for $d_2 = d_1 + 1$: $(d_2 + i) \geq (d_2 + 1) > (d_1 - 1)$ and $b^{d_1+i} > b^{d_1-1}$, even for $d_2 = d_1 - 1$: $(d_2 + i) \geq (d_2 + 1) .= d_1 > (d_1 - 1)$ and $b^{d_1+i} > b^{d_1-1}$.

All the other terms subtracted from the first list are smaller than $(d_1 - 1).b^{d_1-1}.l$, and all the other terms added to the second list are larger than $(d_2 + i).b^{d_1+i}.l$. Therefore each term added to the second list is larger than its corresponding term subtracted from the first list. The last term of the sum, $0.5.(d_1 + d_2).b^{d_1}.b^{d_2}$, is always the same irrespective of the values of d_1 and d_2, as long as their sum remains constant (which is the case here). This means that $M_{2_{new}}$ is always larger than M_2 for $d_1 = d_2$, $d_2 = d_1 + 1$, and $d_2 = d_1 - 1$. As these relationships are symmetrical, this means M_2 is minimum at these conditions. When d is even, this is true for $d_1 = d_2$, and when d is odd this is true for both $d_2 = d_1 + 1$ and $d_2 = d_1 - 1$. These are the same as for minimum memory requirement.

8.2. Branching factors having different values

Let the forward and backward branching factors be b_1 and b_2, and their corresponding tree depths be d_1 and d_2, respectively. We derive first the optimal value for these for minimum memory and then those for minimum computation.

8.2.1. Minimum memory requirement

The theoretical estimate for the amount of memory required in bidirectional search is given by

$$N_2 = d_1.b_1^{d_1} + d_2.b_2^{d_2}. \quad (37)$$

For the above to be the minimum value, the following equations must hold:

$$(d_1 + 1).b_1^{d_1} + (d_2 - 1).b_2^{d_2} > d_1.b_1^{d_1} + d_2.b_2^{d_2} \quad (38)$$

$$(d_2 + 1).b_2^{d_2} + (d_1 - 1).b_1^{d_1} > d_1.b_1^{d_1} + d_2.b_2^{d_2}. \quad (39)$$

8.2.1.1. $b_1 > b_2$. In Figure 13, the three curves $F1 = d_1.b_1^{d_1}$, $F2 = d_2.b_2^{d_2}$, and $F1 + F2 = d_1.b_1^{d_1} + d_2.b_2^{d_2}$ are plotted for $b_1 > b_2$. P is the point where $F1$ and $F2$ intersect. On the right side of this point, an increase in $F1$ for a given increase in d_1, given by $f1$, is necessarily greater than a decrease in $F2$, given by $f2$, for the same decrease in d_2 as the increase in d_1. This is because $F1$ has a higher slope than $F2$ at the same point, which gets even higher as one tries to climb $F1$ compared to the slope of $F1$ as one tries to descend down it.

This means that all values of $F1 + F2$ to the left of P have values larger than that at P. In other words the minimum value of $F1 + F2$ lies somewhere to the left of P including itself.

At P, values of $F1$ and $F2$ are the same, which means

$$d_{1P}.b_1^{d_{1P}} = d_{2P}.b_2^{d_{2P}}, \quad (40)$$

where d_{1P} and d_{2P} are values of d_1 and d_2 at P.

As $b_1 > b_2$, the above equation implies that $d_{1P} < d_{2P}$. This means that the minimum value for $F1 + F2$ will be found for d_{1m} and d_{2m} (i.e., values of d_1 and d_2 where $F1 + F2$ is minimum), such that $d_{1m} \leq d_{1P}$; $d_{2m} \geq d_{2P}$, such that

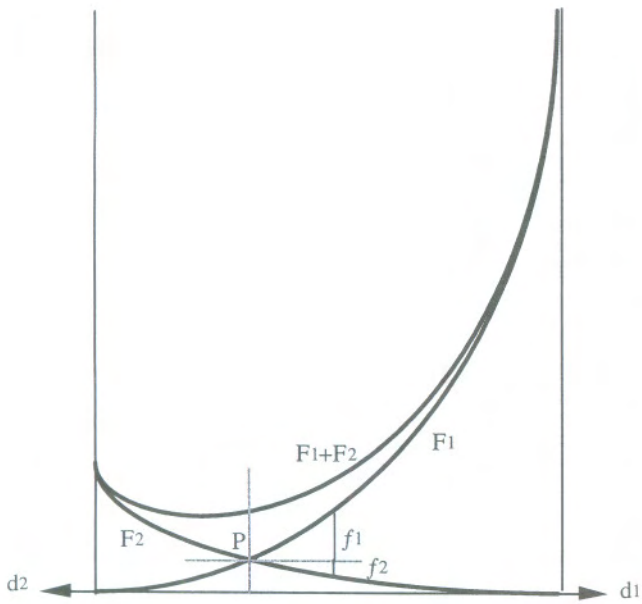

Fig. 13. Plot of curves $F1 = d_1.b_1^{d_1}$, $F2 = d_2.b_2^{d_2}$, and $F1 + F2 = d_1.b_1^{d_1} + d_2.b_2^{d_2}$.

$d_{1m} + d_{2m} = d_{1P} + d_{2P} = d_1 + d_2$. Values of d_{1m} and d_{2m} can be found by solving Eq. (37) numerically.

8.2.1.2. $b_2 > b_1$. As Eq. (37) is symmetrical in terms of b_2 and b_1, this too can be proved as above.

8.2.2. Minimum computation requirement

The theoretical estimate for the amount of computation required in bidirectional search is given by Eq. (13) which is repeated below:

$$M_2 = [1.l + 1.b.l + 2.b.b.l + \ldots + (d_1 - 1).b^{d_1-1}.l] + [1.l + b.l + b.b.l + \ldots + (d_2 - 1).b^{d_2-1}.l] + 0.5.(d_1+d_2).b^{d_1}.b^{d_2}. \quad (13)$$

The value of M_2 for $(d_1 - 1)$ and $(d_2 + 1)$ can be obtained by removing the last term from the first list and adding further one term to the second list in Eq. (13), and is given by

$$M_{2_{new}} = [1.l + 1.b.l + \ldots + (d_1 - 1).b^{d_1-1}.l] - (d_1 - 1).b^{d_1-1}.l + [1.l + b.l + \ldots + (d_2 - 1).b^{d_2-1}.l] + d_2.b^{d_2}.l + 0.5.(d_1 + d_2).b^{d_1}.b^{d_2}. \quad (41)$$

For its value to be less than M_2, the following equation must hold:

$$d_2.b_2^{d_2}.l > (d_1 - 1).b_1^{d_1-1}.l \rightarrow d_2.b_2^{d_2} > (d_1 - 1).b_1^{d_1-1}. \quad (42)$$

For this point to be minimum value, the following equation also must hold:

$$d_1.b_1^{d_1}.l > (d_2 - 1).b_2^{d_2-1}.l \rightarrow d_1.b_1^{d_1} > (d_2 - 1).b_2^{d_2-1}. \quad (43)$$

From Figure 13, $(d_{1P} - 1).b_1^{d_{1P}-1}$ is on the left of P on $F1$, and therefore less than $d_2.b_2^{d_2}$, which is the value of $F2$ at P. So Eq. (42) holds for values of $F1$ and $F2$ at P. Similarly, $(d_{2P} - 1).b_2^{d_{2P}-1}$ is on the right of P on $F2$, and therefore less than $d_1.b_1^{d_1}$, which is the value of $F1$ at P. So Eq. (43) holds for values of $F1$ and $F2$ at P. Therefore P is in fact the point whose values of d_{1P} and d_{2P} gives the minimum computation value. Again this will be true for $b1 > b2$, due to symmetry of Eq. (41).

Discussion in the above two subsections reveals that required memory and computation are both minimum for the same values of d_1 and d_2 only when memory required is minimum at point P of Figure 1 (as this is also where the amount of computation required is also minimum). If this is not the case, both cannot be minimized together. One heuristic that emerges is to use values close to P for a near-optimum performance.

9. SUMMARY AND CONCLUSIONS

The effectiveness and efficiency of an algorithm–database combination are defined here as the two essential features of the quality of a computational-synthesis procedure. This article focuses on the efficiency aspect of the algorithm of such procedures. The article proposes a bidirectional search algorithm for compositional synthesis of designs using a database of building blocks. Although bidirectional search has been investigated before, it has never before been investigated for compositional synthesis, which is distinct from the earlier problems in at least two ways: all possible paths between the input and output are sought, and each of these paths have the same size. The algorithm has been implemented in a computer program using LispWorks™ (Harlequin, 1991), and has been compared for its computational performance with a unidirectional synthesis algorithm. This comparison shows that the proposed algorithm is better in computational performance (memory and computation required) except for problems of very small size, and gets increasingly better for solutions of larger size using larger databases of building blocks. Theoretical estimates have also been made for optimum performance of the algorithm when it has unequal forward and backward branching factors, which provide useful heuristics for enabling the algorithm to provide (near-)optimal computational performance. The algorithm has been tested on several problems and databases. The relative computer performance has been consistent, which demonstrates the generic nature of its efficiency. This means that except for very sparse databases (i.e., with low branching factors), bidirectional search provides marked improvement in efficiency compared to unidirectional search. However, several things remain to be tested. One is the use of bidirectional search for multiple input-output problems, where the computational complexity of the bookkeeping task may overwhelm the advantages of bidirectional search. The other

issue is to investigate the use of bidirectional search in conjunction with further heuristics, such as using further intermediate states as goal states so as to turn this into a multidirectional search, when even with the added efficiency of bidirectional search, it is difficult for the search to be effective.

ACKNOWLEDGMENTS

This project has been funded by Engineering Production and Science Research Council (EPSRC) UK and Matsushita Electric, Japan. The author acknowledges Dave Brown for his support.

REFERENCES

Burgess, S., Moore, D., Edwards, K., Shibaike, N., Klaubert, H., & Chiang, H-S. (1995). Design application: The design of a novel micro-accelerometer. *Workshop on Knowledge Sharing Environment for Creative Design of Higher Quality and Knowledge Intensiveness*, January, University of Tokyo, Tokyo, Japan, 12–13.

Chakrabarti, A. (1995). Towards a support framework for the generation and exploration of conceptual design solutions, *Workshop on Knowledge Sharing Environment for Creative Design of Higher Quality and Knowledge Intensiveness*, University of Tokyo, Tokyo, Japan.

Chakrabarti, A. (1996). Evaluation of the program for synthesis of solution principles. *Workshop for Green Design and Micromechanisms*, University of Cambridge, Cambridge, United Kingdom.

Chakrabarti, A. (1998). *A measure of the newness of a solution set generated using a database of building blocks and the database parameters which control its newness*. Technical Report No. CUED-C-EDC/TR64. Cambridge University Engineering Department, Cambridge, United Kingdom.

Chakrabarti A. & Bligh T.P. (1994). Functional synthesis of solution-concepts in mechanical conceptual design. Part I: Knowledge representation. *Research in Engineering Design 6(3)*, 127–141.

Chakrabarti A. & Bligh T.P. (1996*a*). Functional synthesis of solution-concepts in mechanical conceptual design. Part II: Kind synthesis, *Research in Engineering Design 8(1)*, 52–62.

Chakrabarti A. & Bligh T.P. (1996*b*). Functional synthesis of solution-concepts in mechanical conceptual design. Part III: Spatial configuration. *Research in Engineering Design 8(2)*, 116–124.

Chakrabarti A. & Bligh, T.P. (1996*c*). An approach to functional synthesis of design concepts: Theory, application, and emerging research issues. *AI in Engineering Design, Analysis and Manufacturing 10(4)*, 313–331.

Chakrabarti, A., Johnson, A.L., & Kiriyama, T. (1997). An approach to automated synthesis of solution principles for micro-sensor designs. *Proc. International Conference on Engineering Design ICED'97*, 125–128.

De Champeaux, D. & Sint, L. (1977). An improved bidirectional heuristic search algorithm. *Journal of Association for Computing Machinery 24*, 177–191.

Finger, S. & Rinderle, J.R. (1990). *A transformational approach for mechanical design using a bond graph grammar*. EDRC Report no. 24-23-90, Carnegie-Mellon University, Pittsburgh, PA.

Harlequin (1991). LispWorks: The reference manual, Harlequin PLC. Cambridge, UK.

Hoeltzel, D.A. & Chieng, W-H. (1990). Knowledge-based approaches for the creative synthesis of mechanisms. *Computer-Aided Design 22(1)*, 57–67.

Hoover, S.P. & Rinderle, J.R. (1989). A synthesis strategy for mechanical devices, *Research in Engineering Design* 87–103.

Ishii, M., Tomiyama, T., & Yoshikawa, H. (1994). A synthetic reasoning method for conceptual design. In *Towards World Class Manufacturing*, (Wozny, M. & Olling, G., Eds.). Elsevier Science, North-Holland, Amsterdam.

Ishida, T. (1996). Real time bidirectional search: Coordinated problem solving in uncertain situations. *IEEE Transactions on Pattern Analysis and Machine Intelligence 18(6)*, 617–628.

Khang, J.H.L. (1998). Embodiment modelling with parameter trees, PhD Thesis. University of Cambridge, United Kingdom.

Kiriyama, T. & Johnson, AL. (1995). Functional modelling for the design of micromechanisms. *Workshop on Knowledge Sharing Environment for Creative Design of Higher Quality and Knowledge Intensiveness*, University of Tokyo, Tokyo, Japan.

Kota, S. & Chiou, S.-J. (1992). Conceptual design of mechanisms based on computational synthesis and simulation. *Research in Engineering Design 4*, 75–87.

Malmqvist, J. (1993). Computer-aided conceptual design of energy-transforming technical systems, *Proc. International Conf. on Engineering Design ICED'93*, 1541–1550.

Pahl, G. & Beitz, W. (1984). *Engineering Design: A Systematic Approach*, 2nd ed. (Design Council, London, Springer-Verlag).

Paynter, H.M. (1961). *Analysis and Design of Engineering Systems*, The MIT Press, Cambridge, MA.

Pohl, I. (1971). Bi-directional search. In *Machine Intelligence 6*, (Meltzer B, & Mitchie D., Eds.), 127–140, Edinburgh University Press, Edinburgh.

Prabhu, D.R. & Taylor, D.L. (1988). Some issues in the generation of the topology of systems with constant power-flow input-output requirements. *Proc. of The ASME Design Automation Conference*, 41–48.

Roth, K. (1970). Systematik der Machinen und ihrer Mechanischen Elementaren Funktionen, *Feinwerktechnik 74*, 453–460.

Russell, S. & Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*, Prentice Hall, London.

Selutsky, A.B. (Ed.). (1987). *Daring Formulae of Creativity*, Karelia, Petrozavodsk, Russia. (in Russian).

Sushkov, V., Alberts, L., & Mars, N.J.I. (1996). Innovative design based on sharable physical knowledge. In *Artificial Intelligence in Design '96*, (Gero, J.S. & Sudweeks F., Eds.), 723–742, Kluwer Academic, Dordrecht, The Netherlands.

Taura, T., Koyama, T., & Kawaguchi, T. (1996). Research on natural law database. *Joint Conference on Knowledge Based Software Engineering '96*, Sozopol, Bulgaria.

Tsourikov, V.M. (1995). Inventive machine: 2nd Generation. *AI and Society 7(1)*, 62–78.

Ulrich, K.T. & Seering, W.P. (1989). Synthesis of schematic descriptions in mechanical design. *Research in Engineering Design 1(1)*, 3–18.

Umeda, Y. & Tomiyama, T. (1997). Functional reasoning in design. *IEEE Expert: Intelligent Systems and Their Applications, 12(2)*, 42–48.

Welch, R.V. & Dixon, J.R. (1994). Guiding conceptual design through behavioral reasoning. *Research in Engineering Design 6(3)*, 169–188.

Dr. Amaresh Chakrabarti received a B.E. in Mechanical Engineering from the University of Calcutta in 1985, an M.E. in Mechanical Design from Indian Institute of Science in 1987, and a Ph.D. in Engineering Design from Cambridge University in 1991. He has since been associated with the Engineering Design Centre at Cambridge University as a Senior Research Associate. In 1994, his software FuncSION won a prize in the U.K. Morgans-Grampian Manufacturing Industry Achievements Awards competition. His main interest is in design methodology, particularly in the earlier phases of design. This includes requirements identification, functional representation, conceptual design, and engineering design research methodology.

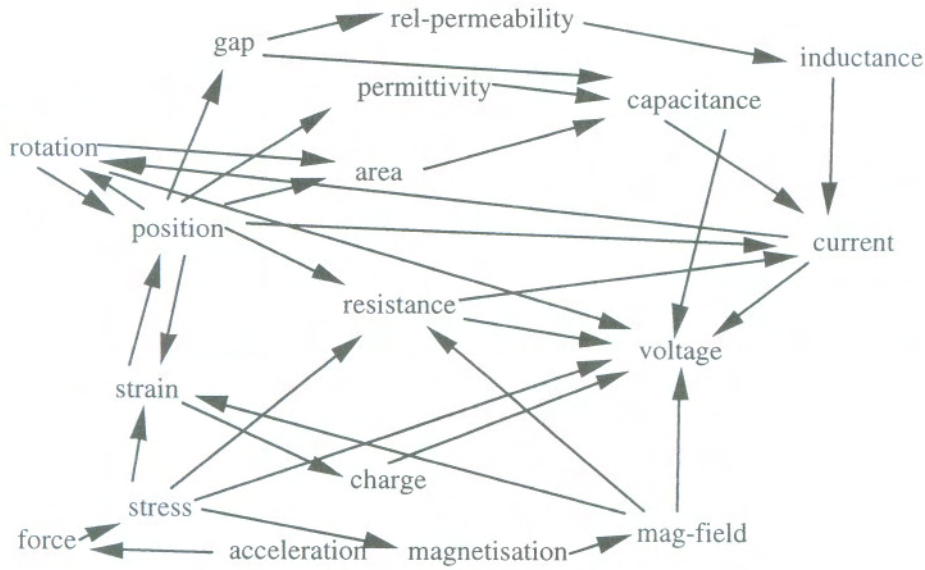

Fig. A1. Database I having 18 variables and 35 building blocks.

APPENDIX A: DATABASES USED IN ASSESSING PERFORMANCE OF THE ALGORITHMS

The databases are shown here in Figures A1 and A2 as a network of variables that are linked together by building blocks. So, each arrow is a building block, that is, a link between two variables.

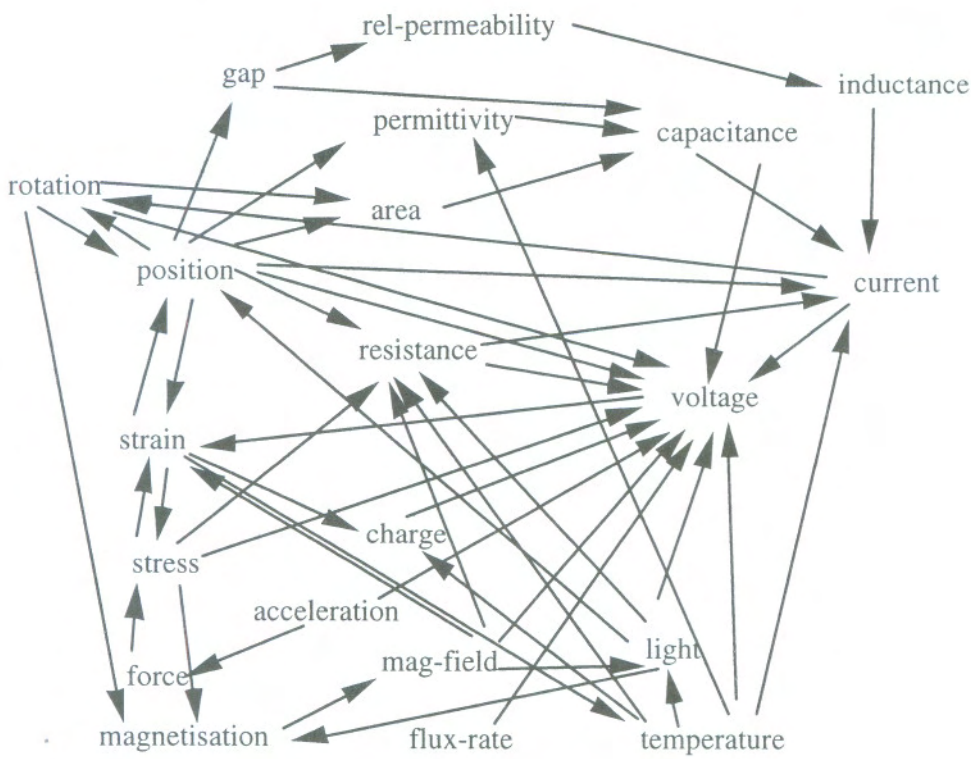

Fig. A2. Database II having 21 variables and 53 building blocks.