Annals of Operations Research 33(1991)379–402

BRANCH-AND-BOUND AS A HIGHER-ORDER FUNCTION

G.P. McKEOWN, V.J. RAYWARD-SMITH and H.J. TURPIN
School of Information Systems, University of East Anglia, Norwich NR4 7TJ, U.K.

Abstract

The branch-and-bound paradigm is presented as a higher-order function and illustrated by instantiations, providing two well-known branch-and-bound algorithms for the Steiner tree problem in graphs and one for the travelling salesman problem. We discuss the advantages of such a specification and various issues arising from sequential and parallel implementations of branch-and-bound kernels.

1. Introduction

Branch-and-bound (B&B) algorithms are commonly used to solve a wide variety of problems in operational research, combinatorial optimisation and artificial intelligence. However, there are many differences of opinion over precisely which algorithms fall into the category B&B. The term "branch-and-bound" was first used in the field of operational research to refer to a method for solving general, mixed-integer, linear programming problems [10,24]. Even within operational research itself, the term has been applied to a number of more specialised algorithms, such as Balas' 0–1 additive algorithm [4], and to algorithms exploiting the special structure of topological network problems [14,25,40]. More controversially, in recent years it has been claimed that a number of AND/OR graph search procedures used in the field of artificial intelligence can, in fact, be viewed as B&B algorithms [21,24].

In section 2, we describe the components of the B&B paradigm and use a B&B algorithm for the travelling salesman problem (TSP) as an illustration, although we do not suggest that the resulting algorithm is state-of-the-art for the TSP. A formal specification of each of the components is given in the appendix, where we present branch-and-bound as a *higher-order function.* Higher-order functions provide a mechanism for modularising algorithms in new and exciting ways. They enable us to encapsulate entire classes of algorithms within a single meta-algorithm, a process which we call *algorithm abstraction.* The latter has tremendous potential for easing the increasingly complex task of developing software for a variety of computer architectures. An eloquent case for the importance of higher-order functions is presented in [15].

We define a B&B algorithm to be any algorithm which can be instantiated as a particular instance of our B&B higher-order function. It may be shown [43] that all of the algorithms to which the term "branch-and-bound" is usually applied, from

any of the areas mentioned above, are indeed B&B algorithms according to our definition. Furthermore, a number of less obvious candidates are also seen to be of type B&B [43]. In section 3, we instantiate our higher-order function to yield two B&B algorithms for the Steiner tree problem in graphs – a problem, like TSP, of particular relevance to topological network design.

Previous attempts at general formulations of the B&B paradigm are given in [1,5,17,20,30,31]. The advantage of our approach is that we have produced a clear, higher-order functional specification of the algorithm type B&B. The high degree of modularity exhibited by our specification facilitates the production of *software skeletons* which can be used as a basis for implementing large subclasses of B&B algorithms. The use of these skeletons for B&B is described in section 4. In section 5, a serial implementation of the higher-order function is described. The language Modula-2 has been used for this implementation and the advantages arising from this choice of language are discussed. We conclude the paper with a brief discussion of parallel implementations of the B&B higher-order function. We believe that by providing the software for B&B skeletons, we will considerably simplify the implementation and enhance the portability of parallel B&B algorithms.

2. Abstraction of the branch-and-bound paradigm

2.1. THE SPACE OF PROBLEMS

We consider the B&B paradigm to be an approach for solving certain types of *constrained optimisation problems* (*COPs*). In general, a COP has the form

optimise $f : S \rightarrow W$

over $F = \{s \in S \mid s \text{ is feasible}\}$.

$s \in S$ is feasible iff it satisfies each constraint in a specified set of constraints. W is a totally-ordered set with ordering $\gtrsim$. For $w_1, w_2 \in W$, $w_1 \gg w_2$ iff $w_1 \gtrsim w_2$ and $w_1 \neq w_2$.

An *optimal solution* of the above COP is specified as follows:

if $F \neq \emptyset$, an optimal solution is some $s^* \in F$ such that $\forall s \in F : f(s^*) \gtrsim f(s)$;

if $F = \emptyset$, the optimal solution is ω, where $\omega \notin S$ may be interpreted as "undefined".

In a *discrete constrained optimisation problem* (*DCOP*), only a finite (although possibly very large) number of elements of S need to be considered in order to determine an optimal solution. The aim of a B&B algorithm is to limit the number of elements that need to be explicitly enumerated. This is achieved using a tree search, together with various strategies for pruning parts of the search tree. All of the nodes in such a tree correspond to DCOPs of the same type. The root node corresponds to an initial given problem and the path from the root to any other node corresponds to a sequence

of elementary operations which transforms the initial problem into the DCOP corresponding to that node. We denote by P the space of problems in which all of the DCOPs corresponding to nodes in the tree must lie.

The higher-order function for branch-and-bound is formally specified in fig. 2 (appendix). The problem- and algorithm-dependent functions for our higher-order B&B function are listed in a **uses functions** statement. The interpretation of each of these functions is described in the context of a B&B algorithm for the travelling salesman problem in this section, and formally in the appendix.

2.2. A BRANCH-AND-BOUND ALGORITHM FOR THE TRAVELLING SALESMAN PROBLEM

The travelling salesman problem (TSP) is a well-known NP-complete problem in topological network design. In this section, we describe a simple B&B algorithm to solve the TSP. We have deliberately avoided incorporating performance-enhancing techniques in favour of conveying the ease with which a B&B algorithm can be described in terms of the higher-order function. For a survey of state-of-the-art B&B codes for the TSP, see [6].

Given a graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ and costs $c_{ij} \in \mathbb{Z}^+ \cup \{\infty\}$ (for $i = 1, \ldots, n$ and $j = 1, \ldots, n$) are defined by:

$$c_{ij} = \begin{cases} \text{cost of edge } \langle i,j \rangle & \text{if } \langle i,j \rangle \in E, \\ \infty & \text{otherwise,} \end{cases}$$

the TSP is that of finding a minimum cost tour of all the vertices, visiting each vertex exactly once. Without loss of generality, we assume that the tour starts and ends at vertex 1. The TSP can now be expressed by:

$$\text{minimise} \sum_{i=1}^{n} c_{s_i s_{i+1}}$$

$$\text{subject to } \{s_2, \ldots, s_n\} = \{2, \ldots, n\},$$

$$s_1 = s_{n+1} = 1.$$

$v_1, \ldots, v_k$ $(2 < k \leq n + 1)$ is a simple cycle in G if each vertex $v_i \in V$ only appears once except that $v_1 = v_k$. For this example, we can view the solution space S as comprising all simple cycles including vertex 1. Thus, each element of S can be represented by a sequence of the form $s_1, \ldots, s_k$, where $s_1 = s_k = 1$ and $s_2, \ldots, s_k$ are disjoint integers in $\{2, \ldots, n\}$. $s \in S$ is feasible (i.e. represents a tour of all the vertices) iff $l(s) = n + 1$, where $l(s)$ denotes the length of the sequence. Since we wish to minimise the objective function, we define $W = \mathbb{Z}^+ \cup \{\infty\}$, ordered by $\leq$.

The first step in the B&B method is to establish whether a tour of all the vertices has been made. If so, then a feasible solution of the TSP has been found;

otherwise, we derive a set of new problems by branching from the most recently visited vertex to each unvisited vertex. These problems (called the *children* of the original problem) are added to the active list, and the B&B repeats the process on the problems that are selected to be solved next.

2.3. A STATE SPACE REPRESENTATION

Each node in the search tree represents a *problem-state* of type $X = P \times \Sigma^*$, where $(p, \sigma) \in X$ describes a problem $p \in P$ and the sequence $\sigma \in \Sigma^*$ of elementary operations transforming the initial problem p_0 into p. In the sequel, we assume that $x = (p, \sigma)$.

For our TSP example, a problem $p \in P$ is parameterised by a sequence of fixed vertices $(v_2 v_3 \ldots v_m)$ corresponding to a path, path(p), $1, v_2, \ldots, v_m$ of length m, and representing the problem:

$$\text{minimise} \sum_{i=1}^{n} c_{s_i s_{i+1}}$$

$$\text{subject to } \{s_2, \ldots, s_n\} = \{2, \ldots, n\},$$

$$s_1 = s_{n+1} = 1,$$

$$s_i = v_i, \qquad i = 2, \ldots, m,$$

We denote by free(p) the set of vertices not in path(p). Thus, $p_0 = ()$ is the parameterised initial problem, and free$(p_0) = \{2, \ldots, n\}$.

Σ denotes the set of elementary operations, and Σ^* the set of all sequences over Σ. $\varepsilon \in \Sigma^*$ is the empty sequence. Corresponding to each $\sigma \in \Sigma^*$ is a (partial) function $\boldsymbol{\sigma} : P \to P$. We define $\boldsymbol{\sigma}(p) = p$ if $\sigma = \varepsilon$, but otherwise $\sigma = a\tau$, $a \in \Sigma$ and $\tau \in \Sigma^*$, and $\sigma(p) = a(\tau(p))$. Then,

$$X = \{(p, \sigma) \in P \times \Sigma^* \mid \boldsymbol{\sigma}(p_0) = p\},$$

and $x_0 = (p_0, \varepsilon)$ represents the initial problem-state. For our TSP example, we set $\Sigma = \{2, \ldots, n\}$ and $\sigma \in \Sigma^*$ is a string of visited vertices and represents path(p). If $v \in$ free(p), then $v(p) = (v_2 \ldots v_m v)$.

By interpreting F as a function over P, we may denote P by a pair of functions (f, F), where $f : S \to W$ is the function to be optimised and $F(p) = \{s \in S \mid \text{is\_feasible\_for } (p)\ (s)\}$. Here, is_feasible_for is a higher-order truth-valued function of type $P \to (S \to \mathcal{B})$. Thus, $F(p)$ is the set of those $s \in S$ satisfying all of the constraints of problem p. Because $F(p) \subseteq F$ (Req1 – appendix), we do not really need a feasibility predicate parameterised by the problem. We therefore define a truth-valued function is_feasible : $S \to \mathcal{B}$ (where $\mathcal{B}$ denotes the type Boolean) such that

is_feasible Δ is_feasible_for (p_0)

As already observed, in the TSP example is_feasible(s) is true iff s contains all of the vertices in V.

In a B&B algorithm, each node $x \in X$ in the search tree is labelled with a value of $S \cup \{\omega\}$. We define label: $X \to S \cup \{\omega\}$ to be the function assigning labels to nodes in X, such that if label(x) $\in F(p)$, then label(x) is an optimal solution for p (Req2 – appendix). For the TSP algorithm that we are describing, label((p, σ)) = concat(path(p), 1).

2.4. BRANCHING AND BOUNDING

At any stage in a B&B algorithm, the best solution so far found for p_0 is called the *incumbent*. The incumbent is equal to ω until some element of F is known, and from then onwards it is an element of F giving rise to the best value of f so far found. On termination of the algorithm, the incumbent is an optimal solution for p_0. A problem-state $x \in X$ becomes *bounded* when it is known that branching from x would not lead to a better incumbent. A bounded problem-state will not be expanded by the B&B algorithm.

We define a dynamic function, bound: $X \to W$, satisfying the admissibility property of Hart et al. [13] and Nilsson [32], namely that unless x is bounded the value of an optimal solution for p is no better than bound(x). This is formally stated in Req3 (appendix). If λ = label(x) $\in F(p)$, then bound(x) = $f(\lambda)$; otherwise bound(x) is an estimate of the value of an optimal solution for p, unless x becomes bounded when bound(x) is set to $\perp$, the bottom element of W. For our TSP example, we define bound: $X \to W$ as follows:

$$\text{bound}((p, \sigma)) = c_{1v_2} + \sum_{i=2}^{l(p)-1} c_{v_i v_{i+1}} + e,$$

where $e = c_{v_{n1}}$ if $l(p) = n$, but otherwise e is a lower bound estimate on the cost of completing the tour (visiting all unvisited vertices and returning to vertex 1).

For each problem-state x, there is a set of children problem-states derived by applying to p all those elementary operations in Σ that are defined on p. We define a function, child: $X \to 2^X$, where child(x) is a finite (and possibly empty) set of the set of children problem-states of x comprising only those children of x that are to be considered in finding an optimal solution of the initial problem. Thus,

$$\text{child}(x) \subseteq \{(a(p), a\sigma \mid a \in \Sigma \text{ and } a(p) \text{ is defined}\}.$$

To guarantee that B&B returns an optimal solution for the initial problem, we impose the requirement that there is some $x' = (p', \sigma') \in$ child(x) such that the value of an optimal solution for p' is equal to that for p (Req4 – appendix). For the TSP,

$$\text{child}((p, \sigma)) = \{((v_2 \ldots v_{l(p)} v), v\sigma) \mid v \in \text{free}(p)\}.$$

Hence, child((p, σ)) = $\emptyset$ iff $l(p) = n$.

2.5. EXPANSION AND SELECTION

A fundamental aspect of a B&B algorithm is the *expansion* of problem-states. The process of expanding a problem-state x involves the following. If $\lambda = \text{label}((p, \sigma))$ is feasible for p and hence (since $F(p) \subseteq F$) is also feasible for p_0, then λ is considered as a possible optimal solution for p_0. Otherwise, x is *branched* from by generating child(x) and adding this set to the *active set* of problem-states that have been generated but not yet expanded. Initially, the active set is the singleton set containing (p_0, ε). A problem-state is removed from the active set when it is expanded and then becomes a member of the *dead set*, which consists of problem-states that have been both generated and expanded.

$x \in$ active becomes bounded (i.e. bound(x) is set to $\perp$) if any of the following tests hold:

(1) $F(p)$ is known to be $\varnothing$;

(2) $f(\text{incumbent}) \gtrdot \text{bound}(x)$; [*pruning by incumbent*]

(3) $\exists y \in \text{active} \cup \text{dead} : \text{is\_dom}(y, x)$; [*pruning by dominance*]

(4) $\exists y \in \text{active} \cup \text{dead} : y \neq x \wedge y = \text{isomorph}(x)$. [*pruning by isomorphism*]

Tests 1 and 2 are used in all B&B algorithms; tests 3 and 4 are optional, depending on the instantiation of B&B. B&B using only tests 1 and 2 we refer to as *basic B&B*.

The truth-valued function is_dom: $X \times X \to \mathcal{B}$ used in test 3 satisfies:

$$\text{is\_dom}(x, y) \triangleq (x, y) \in \text{dom} \wedge x \neq y,$$

where dom is a partial ordering defined on $X \times X$ such that if $x \neq y$, then $(x, y) \in$ dom only if it is known that y may be bounded if x is generated (Req5 – appendix). An equivalence relation E_{dom}, defined by

$$(x, y) \in E_{\text{dom}} \Leftrightarrow [(x, y) \in \text{dom} \vee (y, x) \in \text{dom}],$$

is used in the implementation to determine a subset of problem-states in active $\cup$ dead to consider for pruning x by dominance. We discuss this further in section 5.

Pruning by isomorphism is possible if an equivalence relation, iso, is apparent such that if $x \neq y$, then $(x, y) \in$ iso only if it is known that one of x and y may be bounded if both are generated (Req6 – appendix). Let $[x]$ denote the equivalence class to which x belongs. The dynamic function isomorph: $X \to X$ used in test 4 provides an algorithm for determining a representative problem-state of $[x]$ selected from $\{x\} \cup$ active $\cup$ dead.

The use of dominance and isomorphism is illustrated in our algorithm for the TSP because only one of the problem-states x and y need be expanded if each represents a path through the same set of vertices finishing at the same vertex. If $y \in$ active $\cup$ dead, then $x \in$ active is pruned by dominance if bound(y) $\gg$ bound(x), pruned by isomorphism if bound(y) = bound(x), and expanded otherwise.

Suppose $x = (p, \sigma)$ and $y = (p', \sigma')$, then is_dom: $X \times X \to \mathcal{B}$ and isomorph: $X \to X$ are defined by:

$$\text{is\_dom}(y, x) \equiv [(\text{free}(p) = \text{free}(p')) \wedge (v_{l(p)} = v'_{l(p')}) \wedge (\text{bound}(y) \gg \text{bound}(x))];$$

$$\text{isomorph}(x) = \begin{cases} y & \text{if } \exists y \in (\text{active} \cup \text{dead}) \text{ s.t. } (\text{free}(p) = \text{free}(p')) \\ & \quad \wedge (v_{l(p)} = v'_{l(p')}) \wedge (\text{bound}(x) = \text{bound}(x')), \\ x & \text{otherwise.} \end{cases}$$

Problem-states are *selected* in order of precedence from the *active* set to be expanded. Only problem-states that have not been bounded are selected. The precedence ordering is established by a priority function [38]. We define $\mathcal{V}$ to be some partially ordered set with ordering $\supseteq$. For $v_1, v_2 \in \mathcal{V}$, we define $v_1 \supset v_2$ iff $v_1 \supseteq v_2 \wedge v_1 \neq v_2$. priority: $X \to \mathcal{V}$ is a total function defined such that x has precedence over y iff priority(x) $\supset$ priority(y). For the algorithm that we are describing for the TSP, the precedence ordering establishes a best-first search. This is instantiated by specifying that $\mathcal{V} = W$, $\supseteq = \gg$, and priority(x) = bound(x) $\forall x \in X$.

The B&B function is defined on the set B_and_B_state = $2^X \times 2^X \times (S \times W)$. Our TSP algorithm is thus B&B instantiated in the way described above.

3. Branch-and-bound algorithms for the Steiner tree problem in graphs

We will illustrate the use of the branch-and-bound higher-order function by describing two basic B&B algorithms for the Steiner tree problem in graphs. This is an important problem in topological network design. The two algorithms we describe are given as instantiations of branch-and-bound by defining:

(a) the *types* of P, Σ (and hence of $X = P \times \Sigma^*$), S, $\mathcal{V}$ and W, and

(b) the *uses function* priority: $X \to \mathcal{V}$, is_feasible: $S \to \mathcal{B}$, label: $X \to S \cup \{\omega\}$, bound: $X \to W$, child: $X \to 2^X$, is_dom: $X \times X \to \mathcal{B}$ and isomorph: $X \to X$.

Given a graph $G = (V, E)$ with cost function $c: E \to \mathbb{Z}^+$, the Steiner tree problem in graphs (SPG) is that of connecting together some given subset of special vertices $K \subset V$ as cheaply as possible. The solution will always be a minimum spanning tree of some subgraph induced by $K \cup S$, for some subset S. This subset S of vertices used in addition to K is called the set of Steiner vertices.

The decision problem associated with SPG is known to be NP-complete [19]. However, some encouraging progress has been made concerning this problem. On the heuristic side, promising algorithms with guaranteed worst-case performances have been proposed by Takahashi and Matsuyama [42], Rayward-Smith [36,37,47], Plesnik [33], and Wu, Widmayer and Wong [48]. For both heuristic and certainly for

exact algorithms, it is wise to incorporate various problem reduction techniques to attempt to reduce the size of the problem. These are based on the inclusion/exclusion of certain vertices/edges/subgraphs. Details can be found in [3,11].

In this section, the two algorithms we present are both exact and based upon the branch-and-bound paradigm. The first is based on edge selection and the second on vertex selection. Other exact methods and further results on heuristics, reductions and special cases can be found in [16], which is a useful collection of all the major papers concerning SPG.

3.1. THE SHORE, FOULDS AND GIBBONS ALGORITHM

The first algorithm we describe was first published in [41]. We assume $V = \{1, 2, \ldots, n\}$, $K = \{1, 2, \ldots, k\}$ and the cost of edge $\langle i, j\rangle$ is $c_{ij} = c(\langle i, j\rangle)$. The edges of G are represented by the symmetric cost matrix $C = [c_{ij}]$, where $c_{ij} = \infty$ iff $i = j$ or $\langle i, j\rangle \notin E$.

The problem space P is a set of problems of the form "find a minimum cost subgraph of G which contains K and the edges E_1 but excludes the edges E_2". Thus, a problem $p \in P$ can be parameterised as (E_1, E_2). The initial problem p_0 is parameterised as $(\varnothing, \varnothing)$.

$\Sigma = \{e, e' \mid e \in E\}$. We use e to denote the inclusion of e in the Steiner tree and e' for the exclusion of e in the Steiner tree. If $p = (E_1, E_2)$ is some problem in P and $e \notin E_1 \cup E_2$, then

$$e(p) = (E_1 \cup \{e\}, E_2) \quad \text{and} \quad e'(p) = (E_1, E_2 \cup \{e\}).$$

The solution space S consists of sets of edges of the graph G. We define is_feasible(s) is true iff the set s of edges represents a connected subgraph of G containing all of the vertices of K.

label: $X \to S \cup \{\omega\}$ is defined by label$((p, \sigma))$ = label$(((E_1, E_2), \sigma)) = \omega$ if the graph $(V, E - E_2)$ does not contain a connected component containing all the vertices in K and all the edges in E_1. Otherwise, label$((p, \sigma))$ = label$(((E_1, E_2), \sigma)) = E_1$.

Since we wish to minimise the cost of the spanning tree, we set $W = \mathbb{Z}^+$ ordered by $\leq$ and then define bound: $X \to W$ as below.

If $C = [c_{ij}]$ is the original $n \times n$ cost matrix and $e = \langle u, v\rangle$ is some edge, then we define $e'(C)$ to be the same as C except $c_{uv} = c_{vu} = \infty$. $e(C)$ is defined to be an $(n-1) \times (n-1)$ matrix representing the graph where vertices u and v are merged. The cost of an edge between two merged vertices u and v, and a third vertex w, is defined by $\min\{c_{uw}, c_{vw}\}$. We can then recursively define $\boldsymbol{\sigma}(C)$ for any $\sigma \in \Sigma^*$ by defining $\boldsymbol{\varepsilon}(C) = C$ and $\boldsymbol{a\tau}(C) = \boldsymbol{a}(\boldsymbol{\tau}(C))$ for some $a \in S$, $\tau \in \Sigma^*$.

A lower bound for $x = (p, \sigma)$ is computed from the square $m \times m$ matrix $C^p = \boldsymbol{\sigma}(C)$. We define bound: $X \to W$ by bound(x) = bound$((p, \sigma)) = \sum_{\langle u, v\rangle \in E_1} c_{uv}$ + min$[b, c]$, where

$$b = \sum_{i=1}^{k} \min \left\{ c_{ij}^{p} \mid 1 \le j \le m \right\}$$

and

$$c = \left(\sum_{i=1}^{k} \min \left\{ c_{ij}^{p} \mid 1 \le j \le k \right\} \right) - \min \left\{ c_{ij}^{p} \mid 1 \le i \le k, 1 \le j \le k \right\}.$$

The search is a depth-first search with priority to inclusion edges, so we define $\mathcal{V} = \mathbb{Z}^+$ ordered by $\ge$ and set priority(x) = priority((p, σ)) = $|E_1|$.

The child function child$:X \to 2^X$ is defined whereby child((p, σ)) is determined by selecting an edge on which to branch which maximises an associated penalty. This penalty is computed from C^p as follows:

(1) Calculate a penalty vector $T = \{t_i \mid i = 1, \ldots, k\}$ by

(a) $c_i^* = \min\{c_{ij}^p \mid 1 \le j \le m\}$; k_i = the value of j producing c_i^*;

(b) $c_i^+ = \min\{c_{ij}^p \mid 1 \le j \le m, j \ne k_i\}$;

(c) $t_i = c_i^+ - c_i^*$.

(2) $t_r = \max\{t_i \mid i = 1, \ldots, k\}$.

The edge branched on from $x = (p, \sigma)$ is $e_p = \langle r, k_r \rangle$, and so

$$\text{child}(x) = \{(e_p(p), e_p\sigma), (e_p'(p), e_p'\sigma)\}.$$

The algorithm of Shore et al. is B&B with initial parameter list

$$(\{x_0\}, \varnothing, (\text{inc}_0, \text{valinc}_0)) = (\{((\varnothing, \varnothing), \varepsilon)\}, \varnothing, (E, \infty)).$$

Neither dominance nor isomorphism features, so we set is_dom to be the constant False function and isomorph to be the identity function.

3.2. BEASLEY'S ALGORITHM (MODIFIED)

The second B&B algorithm we describe is used by Beasley [7] and is based on formulating SPG as a shortest spanning tree problem (SST) with additional constraints. Computational results published by Beasley are encouraging, but the algorithm presented here is a simplification used to illustrate the B&B paradigm. Much of the success of Beasley's algorithm is perhaps attributable to the extensive use of reduction tests embedded within the B&B algorithm. We have omitted these in this paper, since they do not illustrate the B&B paradigm.

Let us assume $V = \{1, 2, \ldots, n\}$, $K = \{1, 2, \ldots, k\}$ and the edge between i and j can be represented either as $\langle i, j \rangle$ with cost c_{ij} or as $\langle j, i \rangle$ with cost c_{ji}. From G, construct G_0 by adding

(a) a new vertex 0 to G,

(b) for each $i > k$, a new edge $\langle 0, i\rangle$ of cost $c_{0i} = 0$, and

(c) an edge $\langle 1, 0\rangle$ of cost $c_{01} = 0$.

To solve SPG for G, we must solve SST for

$$G_0 = (V_0, E_0) = (V \cup \{0\}, E \cup \{\langle 0, i\rangle \mid i \in V - K \cup \{1\}\}),$$

subject to the additional constraint that the SST T computed must satisfy

$$\forall i > k, [\langle 0, i\rangle \in T \Rightarrow \text{degree}_T(i) = 1].$$

Now, let P_i denote all edges adjacent to vertex i in G. By defining $x_{ij} = 1$ if edge $\langle i, j\rangle \in E_0$ is in the optimal solution and 0 otherwise, we can construct an IP formulation of the constrained shortest spanning tree (CSST) problem as follows:

$$z_{ST} = \min \Sigma\{c_{ij} x_{ij} \mid \langle i, j\rangle \in E_0\}$$

subject to

$[x_{ij}]$ forms a spanning tree on (V_0, E_0),

$x_{0i} + x_{uv} \leq 1 \ \forall \langle u, v\rangle \in P_i, \ \forall i > k$ //degree constraint//, and

$x_{ij} \in \{0, 1\}$.

Note that the first of these constraints can easily be expressed equationally and that if z_{SP} is the value of the optimal solution to SPG on G, then $z_{SP} = z_{ST}$.

Following Beasley, we construct a Lagrangian lower bound program (LLBP) which uses Lagrangian multipliers to "price out" the degree constraint of a CSST problem. The resulting SST problem can then be solved using Prim's algorithm [34]. For any set of Lagrangian multipliers $\{s_{iuv} \geq 0 \mid i > k, \langle u, v\rangle \in P_i\}$, LLBP as defined below returns a lower bound for z_{SP}.

LLBP: minimise $\Sigma\{c'_{ij} x_{ij} \mid \langle i, j\rangle \in E_0\} - \Sigma\{s_{iuv} \mid i > k, \langle u, v\rangle \in P_i\}$

subject to

$[x_{ij}]$ forms a spanning tree on (V_0, E_0), and

$x_{ij} \in \{0, 1\}$,

where

$$\begin{aligned}
c'_{ij} &= \Sigma\{s_{juv} \mid \langle u, v\rangle \in P_j\} && \text{if } i = 0 \text{ and } k < j \leq n,\\
&= c_{ij} + s_{iij} && \text{if } \langle i, j\rangle \in E, k < i \leq n \text{ and } 1 \leq j \leq k,\\
&= c_{ij} + s_{jij} && \text{if } \langle i, j\rangle \in E, 1 \leq i \leq k \text{ and } k < j \leq n,\\
&= c_{ij} + s_{iij} + s_{jij} && \text{if } \langle i, j\rangle \in E, k < i \leq n \text{ and } k < j \leq n,\\
&= c_{ij} && \text{otherwise.}
\end{aligned}$$

Subgradient optimisation is used to choose the Lagrangian multipliers to maximise the lower bound obtained by LLBP; we denote this lower bound by z_{LB}.

The problem space P is a set of constrained SST problems presented as IP formulations. The initial problem p_0 is CSST as constructed above.

$\Sigma = \{i, i' \mid k < i \leq n\}$. We use i to denote the inclusion of i in the Steiner tree and i' for the exclusion of i in the Steiner tree. If p is some constrained SST problem, then $i(p)$ is p with the additional constraint that i must be in the Steiner tree, i.e. that

$$\Sigma\{x_{uv} \mid \langle u, v\rangle \in P_i\} \geq 2.$$

Similarly, $i'(p)$ has the additional constraint that

$$\Sigma\{x_{uv} \mid \langle u, v\rangle \in P_i\} = 0.$$

Each $\sigma \in \Sigma^*$ can also be used in an obvious way to construct a subgraph from G_0. If $p = \boldsymbol{\sigma}(p_0)$, then this subgraph will be denoted by $G_p = \boldsymbol{\sigma}(G_0)$.

The solution space S consists of sets of edges of the graph G_0. Defining $x_{ij} = 1$ if $\langle i, j\rangle \in S$ and 0 otherwise, we define

is_feasible(s) = true iff $[x_{ij}]$ satisfies the conditions of CSST.

If we wish the use a sequential (inclusion first) depth-first search, we define $\mathcal{V} = \Sigma^*$ ordered by

$$\sigma \supseteq t \text{ iff } \exists i, \exists j \geq 0\colon [\sigma_1\, \sigma_2 \ldots \sigma_j = \tau_1\, \tau_2 \ldots \tau_j \wedge \sigma_{j+1} = i \wedge \tau_{j+1} = i']$$

and priority$((p, \sigma)) = \sigma$.

The required solution is a set of edges forming a tree so label$:X \to S \cup \{\omega\}$ is defined by label$((p, \sigma)) = \omega$ if there is no feasible solution to p, i.e. there is no path in the graph G_p between 1 and vertex i for some $i \in K$, but otherwise label$((p, \sigma))$ is the set of edges selected by LLBP applied to p. Since we wish to minimise the cost of the tree, $W = \mathbb{Z}^+$ is ordered by $\leq$ and bound$:X \to W$ is defined by bound$((p, \sigma))$ $= z_{LB}$, i.e. the result of applying LLBP to p.

The child function child$:X \to 2^X$ can be defined as follows. Let V_p denote that subset of $V - K$ which has not been constrained to be in or out of the solution by the additional constraints of p. If $V_p = \varnothing$, then child$((p, \sigma)) = \varnothing$ but, otherwise, consider the tree constructed by applying LLBP to p. Let V_p' denote the vertices of V_p which have degree 1 in that tree and V_p'' those vertices with degree >1. If $V_p'' = \varnothing$, select $i \in V_p'$ to maximise c'_{0i}; otherwise select $i \in V_p''$ to maximize c'_{01}. Then define child$((p, s)) = \{(i(p), i\sigma), (i'(p), i'\sigma)\}$.

Finally, we set is_dom to be the constant false function and isomorph to be the identity function because, once again, we will use neither dominance nor isomorphism in this instantiation.

The function B&B can now be called with initial parameter list $(\{x_0\}, \varnothing,$ $(\text{inc}_0, \text{valinc}_0))$, where inc_0 and valinc_0 are a solution of SPG obtained using a reliable heuristic.

4. Kernels for B&B

The higher-order function can be implemented on a variety of architectures and an instantiation prescribed in a variety of languages. In section 5, we describe the sequential implementation of branch-and-bound kernels written in Modula-2 and requiring Modula-2 input which we have developed based on the higher-order function. Kernels to support parallel branch-and-bound written in an extended C and requiring input in C have also been developed, and are discussed in section 6. These parallel versions run on a Meiko transputer rack organised as an MIMD/MP (Multiple Instruction stream, Multiple Data stream Message Passing) computer.

Figure 1 shows how a B&B program can be constructed using one of our kernels. In order to construct a B&B program to solve a particular problem, the user must first select the appropriate B&B kernel. We provide guidelines to assist him or

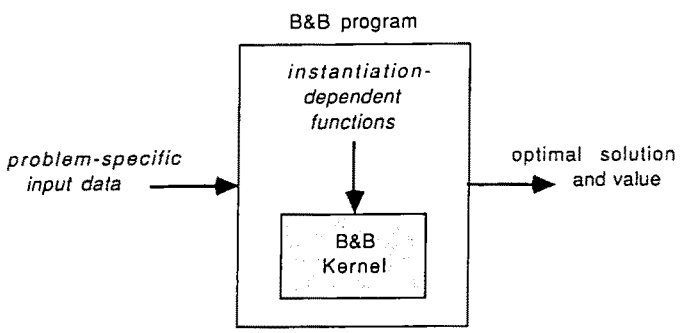

Fig. 1. Implementation of a branch-and-bound algorithm.

her in this choice. S(he) must then implement the instantiation-dependent functions to complete the B&B program. One B&B program may easily be transformed into another by selecting a different kernel and applying a simple transformation to the implementation of the instantiation-dependent functions. Thus, an implementation may be ported from a sequential architecture to a parallel architecture extremely quickly.

The resulting program finds an optimal solution to the specific problem entered by the user. From the problem-specific input data and the parameterisation of a problem in P, that problem can be constructed. By choosing to represent problems in this way, the amount of storage space required in an implementation to store the generated problem-states may be drastically reduced because the input data is not stored with each problem description, although a cost is incurred whenever it is necessary to construct a problem. For instance, in our TSP example the problem-specific data describes a graph and its associated edge costs. This data may be represented by a matrix requiring $O(n^2)$ storage space. However, each problem p is parameterised by a sequence of fixed vertices requiring only $O(n)$ storage space. In

a parallel implementation, the parameterisation of problems is even more important, although the input data must be made available to each processor because problem descriptions may be transferred between processors.

5. Kernels for sequential B&B

Our higher-order function approach toward developing kernels for B&B makes Modula-2 a natural choice of language. Algol-68 also possesses many features which make it an appropriate choice of programming language, but was not chosen due to lack of support at this University. The advantages of using Modula-2 for large programs such as this are that solutions to relatively self-contained subproblems can be developed and tested independently and that data structures can be hidden, thereby encouraging the use of top-down design. In addition, many software tools are already available for solving particular subproblems and can be easily incorporated into the code.

5.1. BASIC B&B

We deal first with the implementation of a kernel for basic B&B; that is, B&B without pruning by either dominance or isomorphism. Our implementation comprises two definition and implementation module pairs, Instance and HeapModule, and a program module, bbModule. The latter was constructed directly from the higher-order function making use of procedures to initialise, expand and select. It is a high-level description of B&B importing types, constants and uses-functions from Instance and an implementation of the Active set from HeapModule.

The definition module Instance defines the types of the higher-order function, whilst its corresponding implementation module defines the constants and uses-functions. These modules are completed by the user to instantiate a B&B algorithm. They are the only modules which need to be accessed and amended in order to describe an instantiation. The instantiation-dependent types P, Σ, S, $\mathcal{V}$, and W are specified by the user as type declarations. The representation and manipulation of the remaining types X, Σ^* and 2^X are defined in terms of these user-defined specifications. X is a record type containing two fields of type P and Σ, respectively. Σ^* and 2^X are both implemented as Queue types in the absence of the basic type String in Modula-2. However, generic types can be implemented in Modula-2. A generic ADT (abstract data type) module is a definition and implementation module pair supporting an ADT which manages elements of some element type, but is not concerned with the type of the elements [28]. A generic ADT module, Queues, has been used to implement types Σ^* and 2^X to avoid repetition of code and reduce development time. It will be seen below that Queues is used again in the implementation of B&B with dominance and/or isomorphism.

Procedures are used to specify the instantiation-dependent constants p_0, inc_0, and valinc_0, and uses-functions $\text{priority}: X \to \mathcal{V}$, $\text{is\_feasible}: S \to \mathcal{B}$,

label: $X \to S \cup \{\omega\}$, bound: $X \to W$, and child: $X \to 2^X$. The child function has been partially defined in this implementation to avoid requiring that components which are common to all instantiations of B&B are specified by the user. Recall, child$(x) \subseteq \{(a(p), a\sigma) \mid a \in \Sigma$ and $a(p)$ is defined$\}$. The specification of child: $X \to 2^X$ is completed by the user's definition of the following constituent parts:

define_map: $\Sigma \times P \to \mathcal{B}$
$(a, p) \mapsto$ true if $a(p)$ is a member of child(x) and false otherwise;

child_prob: $\Sigma \times P \to P$
$(a, p) \mapsto a(p)$;

first_op: $P \to \Sigma$
$p \mapsto a$, where a is the first elementary operation to be considered for problem p;

next_op: $\Sigma \to \Sigma$
$a \mapsto a'$, where a' is the next elementary operation to be considered after a;

more_ops: $\Sigma \to \mathcal{B}$
$a \mapsto$ false if a is the last elementary operation to be considered, true otherwise.

In terms of these components, child: $X \to 2^X$ is expressed as follows:

child(x) = {(child_prob(a, p), $a\sigma$) | $a \in$ Ops and define–map(a, p)},

where Ops is defined by the following algorithm:

```
a := first_op(p);
Ops := {a};
WHILE more_ops(a) DO
  a := next_op(a);
  Ops := Ops ∪ {a}
END (*WHILE*)
```

HeapModule provides an implementation of a heap of problem-state, priority-value, bound-value triplets. The heap is ordered by priority values [2]. The data structure is hidden from the program module and procedures which are needed to use the heap as a priority queue are made available through the definition module.

A library of functions commonly used in B&B algorithms is provided as a useful facility. In particular, we are interested in functions which "collapse" from one into the other and in adopting a functional approach to their implementation. For instance, best-first search is instantiated by defining priority(x) = bound(x) for all $x \in X$, and so we avoid computing both priority and bound values since their computation may typically be an expensive operation. We provide a library procedure best_first

which copies the value stored in the bound-value field of the heap element into the corresponding priority-value field.

Procedures for type-dependent input and output can also be entered by the user in the implementation module Instance. The procedure SetUp() has been designed to allow problem-dependent parameters to be defined at run-time. Problems are easily changed in this way without access to the implementation. For instance, the problem size can be defined by an input to the SetUp procedure. The procedures OutputSoln(solution:S) and OutputValue(value:W), respectively, print the optimal solution found and its value.

5.2. EXTENDED B&B

Now we discuss how the kernel for basic B&B has been developed into an implementation of a kernel for B&B which allows the incorporation of dominance and/or isomorphism. We call this *extended B&B.*

In addition to the definitions required for basic B&B, the instantiation-dependent uses-functions is_dom:$X \times X \to \mathcal{B}$ and isomorph:$X \to X$ must be defined. Extended B&B has basic B&B as its core, but in addition to a priority queue (heap), a set of generated problem-states must be maintained by the implementation of extended B&B. The generated problem-states are all those problem-states in active $\cup$ dead which have not been pruned. The generic type Queue is again used to implement this generated set Gen. The elements of the queue are implemented by variant records defined according to whether a problem-state is active or dead. Gen is a set of elements which are either pointers to the heap (representing active problem-states) or problem-state, bound-value and priority-value triplets (dead problem-states).

A problem-state $x \in X$ is pruned by dominance (isomorphism) if there is some y in Gen such that $(y, x) \in$ dom(iso). Implementing pruning by dominance and/or isomorphism is expensive and will only be worthwhile if the search space is pruned significantly. We have sought to encourage an efficient implementation by suggesting that Gen be divided into equivalence classes, only one of which need be searched to establish whether or not a problem-state should be pruned by dominance (isomorphism). Thus, rather than implement pruning by isomorphism by searching through the entire generated set, a table is used to store pointers to equivalence class representatives in Gen. The table IsoTable is implemented as an array of pointers to the generated set. When a problem-state is generated, it is installed as the representative in the table if one does not already exist, otherwise it is pruned. The user must provide an isomorphism-preserving function $I:X \to$ [1 . . . numIso], such that

$$(x, y) \in \text{Iso} \Rightarrow (I(x) = I(y)) \quad \forall x, y \in X,$$

where [1 . . . numIso] is the domain of I and numIso $\in \mathbb{N}$ is user-defined. The value of numIso can be as large as the system allows. Our implementation then provides a hashing function which maps values from [1 . . numIso] onto the domain of

IsoTable. Hashing techniques [2] are employed to increase the size of IsoTable should it become too full.

The case $I(x) = I(y)\ \forall x, y \in X$ guarantees that I can always be defined, although we advise the user to avoid this case. Ideally,

$$(I(x) = I(y)) \Rightarrow (x, y) \in \text{Iso} \qquad \forall x, y \in X,$$

and we encourage the user to aim for a function as close to this as possible in order to implement pruning by isomorphism efficiently.

Recall that some equivalence relation E_{dom} such that

$$(x, y) \in E_{\text{dom}} \Leftrightarrow [(x, y) \in \text{Dom} \vee (y, x) \in \text{Dom}]$$

must be identified in order to implement dominance. The user must provide an equivalence-class-preserving function $D: X \rightarrow [1 \ldots \text{numDom}]$ such that

$$(x, y) \in E_{\text{dom}} \Rightarrow (D(x) = D(y)) \qquad \forall x, y \in X,$$

where [1 . . numDom] is the domain of D and numDom $\in \mathbb{N}$ is user-defined. Again, the case $D(x) = D(y)\ \forall x, y \in X$ guarantees that such a function always exists. Ideally, $(x, y) \in E_{\text{dom}}$ iff $(D(x) = D(y))$, but in practice it is unlikely that this will be achieved. A table, DomTable, is exploited to implement dominance. This table is implemented by an array of queues, where each queue consists of pointers to Gen. Again, a hashing function is used to map problem-states onto DomTable. If there is some y in the set of problem-states referenced by $D(x)$ such that $(y, x) \in$ Dom either when x is generated or when it is considered for selection, then x can be pruned. By implementing dominance in this way, we effectively divide the search space of the generated set into a number of unions of equivalence classes.

For instance, for the instantiation of the travelling salesman problem described in section 2, a crude definition of the isomorphism- and equivalence-class-preserving function is:

$$I(x) = D(x) = l(p) + 1 \qquad \forall x \in X,$$

where $l(p)$ is the number of edges so far included in the tour, and numIso = numDom $= |E| + 1$.

Pruning by dominance and/or isomorphism is optional, but in order to instantiate an algorithm using one or both of these pruning methods, the user must be able to define the uses-functions is_Dom and/or isomorph, and also the dominance- and/or isomorphism-preserving functions D and/or I. The overheads of implementing dominance and/or isomorphism have to be weighed against the amount of work that is eliminated by pruning problem-states. Of course, this is instantiation-dependent and so the user must decide between the basic and extended B&B kernels. Extended B&B has been implemented by introducing Boolean parameters dom and iso, which are defined by the user. These constant values are passed from bbModule to procedures in HeapModule

for adding to or removing problem-states from the active set according to whether dominance and/or isomorphism is being used.

In deciding how to implement adding and removing problem-states, there are a number of factors to be considered. For adding problem-states, the two extremes are to:

$\mathbf{P_{11}}$ add all generated problem-states, and

$\mathbf{P_{12}}$ add only all those generated problem-states which cannot be pruned.

The work of determining whether a problem-state can be pruned has to be weighed against the overheads of storing a problem-state that could have been pruned. For selecting problem-states to remove, the choices lie between:

$\mathbf{P_{21}}$ selecting problem-states until a problem-state which cannot be pruned is selected, and

$\mathbf{P_{22}}$ pruning all those problem-states in the active set which can be pruned and then selecting a problem-state.

Again, the choice is between keeping the work to a minimum and keeping the size of the heap to a minimum. In our implementation, we employ strategies to keep work to a minimum unless memory overflow seems imminent. We discuss later how the likelihood of memory overflow is detected and propose a scheme of garbage collection. However, assuming for the moment that memory capacity is infinite, we first present our rules for the addition and selection of problem-states.

If iso(dom) is true, then upon generation it is determined whether a problem-state should be added to IsoTable (DomTable), thus establishing whether that problem-state should be pruned. Hence, a hybrid of rules $\mathbf{P_{11}}$ and $\mathbf{P_{12}}$ is employed for adding problem-states, namely:

$\mathbf{P_{13}}$ add only those problem-states which cannot be pruned by dominance and/or isomorphism.

Then a problem-state cannot be pruned by isomorphism at the selection stage, but may be pruned upon selection by incumbent and/or by dominance. Rule $\mathbf{P_{21}}$ is applied for selecting problem-states.

More realistically, the system memory size is a limitation of the implementation. If the heap should become so large that memory overflow seems imminent, then a "garbage collection" strategy is applied. This is to prune all those problem-states in the active set which can be pruned by incumbent and/or by dominance. If garbage collection fails to solve the problem, we are then obliged to implement rules $\mathbf{P_{12}}$ and $\mathbf{P_{22}}$ until such time when the heap size is significantly reduced. Modula-2 provides a Boolean function Available, which returns True if the amount of storage space requested is available and False otherwise. Another function, TSIZE, returns the amount of memory required to store a variable of a given type. This high-level

facility provides a way of detecting that memory overflow is imminent without requiring the user to input the system memory size.

Memory overflow may occur despite the use of garbage collection if there are too many problem-states in the heap that cannot be pruned. Our kernels provide implementations of exact B&B algorithms and so memory overflow is not avoided if garbage collection fails.

Backing store is a practical requirement of many B&B implementations with large memory demands, and so its use is incorporated into our implementation. In general, storing problem descriptions is the main cause of memory overflow. The problem-state field of each heap element has hitherto been described by a record in which one field represents a problem and another a string of elementary operations. We replace the problem field of the problem-state record by an index field, referring to the problem's storage location in backing store. This scheme, in which each heap element "points" to problems in backing store, enables the heap structure to be stored in main memory without overflow occurring. Our main objective in introducing the use of backing store is to prevent memory overflow, but reasonably fast access to problems is also desirable. Modula-2 provides facilities for sequential and direct access to indexed sequential files. All of the data items in a file must be of the same type, although flexibility may be introduced through the use of variant records. The use of backing store is optional since the overheads for storing and retrieving problems are significant and main memory is sufficient for some instantiations. However, file organisation is independent of the instantiation if backing store is used. Files will be volatile and transactions (to add or remove problems) frequent, and this has been taken into consideration in choosing the method of file organisation and access.

6. Kernels for parallel B&B

In this section, we consider the parallelisation of basic B&B. We are concerned with exploiting task-oriented parallelism because in general there is greater potential for this in B&B than for data-oriented parallelism. Furthermore, data-oriented parallelism is largely instantiation dependent, whereas we are interested in providing general guidelines for the B&B paradigm. Most of the parallel B&B algorithms that have been proposed [9,18,35] can be described either as synchronising the expansion phase or asynchronously parallelising both the expansion and selection phases of the B&B paradigm. We now show that both of these schemes are encompassed by a much broader class, which we call parallel (task-oriented) B&B. Each of the algorithms represented by this class is precisely sequential B&B in the case where only one processor is available. Thus, parallel B&B can be described as a *generalisation* of sequential B&B.

Sequential B&B can be viewed as a parallel algorithm employing a single processor in at least two ways. One way is to consider that the processor selects the problem-state of highest priority known to the system for expansion; an alternative

view is that the problem-state of highest priority known to the processor is selected. The first scheme we refer to as the Select Highest Overall (SHO) strategy, and the latter we call Select Highest Available (SHA).

The SHO strategy is a highly-synchronised process. At the start of each iteration, the m processors are each allocated exactly one of the m active problem-states of highest priority to expand if there are at least m active problem-states (otherwise, each of the active problem-states is expanded on a different processor). MANIP [45,46] is an MIMD shared memory computer specifically designed for enumerative-type algorithms, and in particular for basic B&B algorithms using the SHO strategy.

Under the SHA strategy, each processor asynchronously selects and expands problem-states from a pool of active problem-states to which it has access. There are three subschemes of the SHA strategy characterised by this pool. It may be common to all processors (**SHA(global)**), or to a group of processors (**SHA(shared)**), or it may be unique to each processor (**SHA(local)**). DIB [12] is an implementation, written in Modula, of a paralel backtracking kernel on the Crystal MIMD message passing computer at Wisconsin–Madison University. Basic B&B algorithms with depth-first priority ordering and using the **SHA(local)** strategy may be instantiated by DIB.

(Task-oriented) parallel B&B is expressed by the following algorithm in which /REPEAT/ and /DO/ denote "repeat in parallel" and "do in parallel", respectively.

```
Initialise;
/REPEAT/                    (*1 ≤ n ≤ m asynchronous processes*)
  WHILE (Sel := Select(Active) ≠ ∅) DO
    ∀x ∈ Sel /DO/
      expand(x)
    END (*For*)
  END (*While*)
UNTIL finished;             (*all m processors are idle*)
Return inc.
```

If $n = 1$, then this is SHO, and if $n = m$ it describes SHA. Both SHO and SHA(global) employ a global pool of active problem-states; the difference between them is that SHO is synchronised, whereas SHA(global) is not. Parallel B&B is represented by a transformation of the higher-order function that assumes that some partition of the active set exists and that problem-states are selected in order of priority from a collection of subsets of this partition [29,44].

We have implemented kernels for parallel branch-and-bound using the SHO and SHA(global) strategies on a Meiko transputer rack. Details are given in [44]. A transputer is a general purpose computer which can exchange data across communication links with each of four other transputers. Process interaction within a single transputer and between neighbouring transputers can be expressed in the programming language occam.

The higher-order function approach is particularly attractive because the provision of such kernels frees users from involvement in the use of occam and in the difficult problem of process allocation within an MIMD parallel computer. Although to date there is no Modula-2 compiler for the transputer, there are compilers for both FORTRAN and C, and the user will be able to instantiate his or her particular branch-and-bound algorithm using one of these high-level languages. Since both the SHO and SHA(global) strategies require that a global pool is maintained, we are also in the process of implementing basic B&B kernels using these strategies on an MIMD shared memory computer, a Sequent Balance, for comparison purposes.

The problem of selecting the "best" parallel B&B algorithm for a particular problem and architecture is not at all straightforward because of speed-up anomalies [8,22,23,26]. For instance, the worst-case scenario for any strategy using shared or local pools would seem to be that the problem-states with highest priority are all maintained in one pool; even so, this strategy may be better than one employing a global pool if the priority function is misleading. Other factors such as variable task processing times and communication overheads can also have a large impact on performance. Although our skeleton approach enables experimentation with different strategies, its main advantage is that rather than finding the "best" algorithm for a particular problem, it will provide a reasonably good algorithm quickly. This is illustrated in [29,44] by its use with instantiations of the higher-order function describing algorithms for SPG and for integer programming B&B. In addition to implementing further kernels for parallel implementations of basic B&B, we are also developing kernels for parallel extended B&B.

Appendix

The branch-and-bound higher-order function is formally specified in fig. 2. The problem- and algorithm-dependent functions for our higher-order B&B function are listed in a **uses functions** statement. The interpretation of each of these functions has been described in this paper, and we have imposed the following requirements which must be satisfied by any instantiation of our B&B higher-order function. First, we define some useful notation. Let $x = (p, \sigma)$, then:

$x^{(\tau)}$ denotes the problem-state $(\tau(p), \tau\sigma)$, where $\tau \in \Sigma^*$;

$X(x)$ denotes the set $\{y \in X \mid \exists \tau \in \Sigma^*, y = x^{(\tau)}\}$;

$\mathrm{opt}(x)$ denotes the value of an optimal solution for p if $F(p) \neq \emptyset$ and denotes $\perp$ otherwise.

Req1 $\forall p \in P, \forall s \in S$: is_feasible_for (p) $(s) \Rightarrow$ is_feasible_for (p_0) (s).

Req2 $(\mathrm{label}(x) \in F(p) \Rightarrow \mathrm{label}(x)$ is an optimal solution for $p) \wedge (\mathrm{label}(x) = \omega \Rightarrow F(p) = \emptyset)$.

Req3 $\forall x \in X: x$ is not bounded $\Rightarrow \mathrm{bound}(x) \gtrsim \mathrm{opt}(x)$.

Req4 $\{x' \in \text{child}(x) \mid \text{opt}(x') = \text{opt}(x)\} \neq \emptyset$.

Req5 $(x, y) \in \text{Dom} \wedge x \neq y \Rightarrow \text{opt}(x) \gg \text{opt}(y) \wedge (\forall y' \in X(y), \exists x' \in X(x) : (x', y') \in \text{Dom})$.

Req6 (i) $(x, y) \in \text{Iso} \Rightarrow \forall x' \in X(x), \exists y' \in X(y) : [\text{opt}(x') = \text{opt}(y')]$, and

(ii) $(x, y) \in \text{Iso} \Rightarrow \forall x' \in X(x), \exists y' \in X(y) : [(x', y') \in \text{Iso} \wedge \forall z \in X : [(x', z) \in \text{Dom} \Leftrightarrow (y', z) \in \text{Dom} \wedge (z, x') \in \text{Dom} \Leftrightarrow (z, y') \in \text{Dom}]]$.

To instantiate a particular instance of B&B, all of the uses functions must be specified, together with the types on which they are defined. The B&B function is defined on the set

$$\text{B\_and\_B\_state} = 2^X \times 2^X \times (S \times W).$$

The state manipulated by B&B comprises the active and dead sets and the incumbent and its corresponding cost value. In fig. 2, we use a function, second, defined as follows: let $R = A \times B$; then for any $(a, b) \in R$, second$((a, b)) = b$.

```
B&B({x0}, ∅, (inc0, valinc0)) // inc0, valinc0 are initial values for the incumbent and its cost//
                              // x0 = (p0, ε), where p0 is the initial problem//

B&B: B_and_B_state → B_and_B_state
                bs ↦ bs'

uses functions  priority: X → ℕ, is_feasible: S → 𝔹, label: X → S ∪ {ω},
                bound: X → W, child: X→ 2^X , is_Dom: X × X → 𝔹, isomorph: X → X

post (select2 = ∅ ⇒ bs' = bs) ∧ (select2 ≠ ∅ ⇒ bs' = B&B(expand(bs, select2))
where
    expand: B_and_B_state × 2^X  → B_and_B_state
      ((A, D, (inc, valinc)), Sel)  ↦ (A', D', (inc', valinc'))

    uses functions  is_feasible: S → 𝔹, label: X → S ∪ {ω}, bound: X → W, child: X→ 2^X

    pre   Sel ⊆ A ∧ Sel ≠ ∅

    post  (A' = A − Sel ∪ children(Sel)) ∧ (D' = D ∪ Sel)
          ∧ valinc ≥≥ best_new_val ⇒ (inc', valinc') = (inc , valinc)
          ∧ best_new_val >> valinc ⇒ (inc', valinc') = new_best
    where
          best_new_val = second(new_best)

          new_best = best (feasible_solns(newly_solved(Sel)))
```

Fig. 2 (continuation on next page).

children: $2^X \to 2^X$
$T \mapsto U$

uses functions child: $X \to 2^X$

post $U = \{y \in X \mid \exists x \in (T - \text{newly\_solved}(T)) \,.\, y \in \text{child}(x)\}$
where

newly_solved: $2^X \to 2^X$
$T \mapsto U$

uses functions is_feasible: $S \to \mathbb{B}$, label: $X \to S \cup \{\omega\}$

post $U = \{ x \in T \mid \text{is\_feasible}(\text{label}(x)) \}$

feasible_solns: $2^X \to 2^{S \times W}$
$Y \mapsto U$

uses functions is_feasible: $S \to \mathbb{B}$, label: $X \to S \cup \{\omega\}$, bound: $X \to W$

pre $\forall x \in Y \,.\, \text{is\_feasible}(\text{label}(x))$

post $U = \{ (\text{label}(x), \text{bound}(x)) \mid x \in Y \}$

best: $2^{S \times W} \to S \cup \{\omega\} \times W$
$U \mapsto (s, v)$

post $(U = \emptyset \Rightarrow (s, v) = (\omega, \bot))$
$\wedge\ (U \neq \emptyset \Rightarrow ((s, v) \in U \wedge \forall\ (s', v') \in U.\ v \geq v'))$

select2 = second(select(bs))
where

select: B_and_B_state $\to$ B_and_B_state $\times$ 2^X
(A, D, (inc, valinc)) $\mapsto$ ((A', D', (inc', valinc')), Sel)

uses functions priority: $X \to \mathbb{V}$, bound: $X \to W$, is_Dom: $X \times X \to \mathbb{B}$,
isomorph: $X \to X$

post $A'=A - \text{pruned}(A, D, (\text{inc},\text{valinc})) \wedge D'=D \wedge (\text{inc}',\text{valinc}') = (\text{inc}, \text{valinc})$
$\wedge\ \text{Sel} \subseteq A' \wedge [\ \text{Sel} = A' \vee \forall x \in \text{Sel} \,.(\sim\exists\ y \in A' - \text{Sel} \,.\text{priority}(y) \supset \text{priority}(x))]$

where

pruned: B_and_B_state $\to$ 2^X
(A, D, (inc, valinc)) $\mapsto$ Del

uses functions bound: $X \to W$, is_Dom: $X \times X \to \mathbb{B}$, isomorph: $X \to X$

post $\text{Del} = \{x \in A \mid \text{valinc} \geq \text{bound}(x)\}$
$\cup\ \{x \in A \mid \exists\, y \in A \cup D \,.\, \text{is\_Dom}(y,x)\}$
$\cup\ \{x \in A \mid \exists\, y \in A \cup D \,.\, y \neq x \wedge y = \text{isomorph}(x)\}$

Fig. 2. A higher-order function for branch-and-bound.

Acknowledgement

This work was supported by SERC grants GR/F 33063 and 87800580.

References

[1] N. Agin, Optimum seeking with branch-and-bound, Manag. Sci. 13(1966)B176–B185.
[2] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *Data Structures and Algorithms* (Addison–Wesley, 1982).
[3] A. Balakrishnan and N.R. Patel, Problem reduction methods and a tree generation algorithm for the Steiner network problem, Networks 17(1987)65–85.
[4] E. Balas, An additive algorithm for solving linear programs with zero-one variables, Oper. Res. 13(1965)517–526.
[5] E. Balas, A note on the branch-and-bound principle, Oper. Res. 16(1968)442–445.
[6] E. Balas and P. Toth, Branch and bound methods, in: *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, ed. Lawler, Lenstra, Rinnooy Kan and Shmoys (Wiley, London, 1985), pp. 361–401.
[7] J.E. Beasley, An SST-based algorithm for the Steiner problem in graphs, Networks 19(1989) 1–16.
[8] F.W. Burton, G.P. McKeown, V.J. Rayward-Smith and M.R. Sleep, Parallel processing and combinatorial optimisation, *Proc. CO81 Conf.*, Stirling University (1982).
[9] J. Clausen and J.L. Träff, Implementation of parallel branch-and-bound algorithms – experiences with the graph partitioning problem, NATO/ARW on Topological Network Design, Copenhagen (1989), Ann. Oper. Res., this volume.
[10] R.J. Dakin, A tree-search algorithm for mixed-integer programming problems, Comp. J. 8(1965) 250–255.
[11] C.W. Duin and A. Volgenaut, Reduction tests for the Steiner problem in graphs, Department of Operations Research, Faculty of Economic Sciences and Econometrics, University of Amsterdam (1988).
[12] R. Finkel and U. Manber, DIB – a distributed implementation of backtracking, ACM Trans. Prog. Languages and Systems 9(1987)235–256.
[13] P.E. Hart, N. Nilsson and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Systems and Cybernetics 4(1968)100–107.
[14] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, Rockville, MD, 1978).
[15] R.J.M. Hughes, Why functional programming matters, Comp. J. 32(1989)98–107.
[16] F.K. Hwang and D. Richards, a two-volume collection of important works in the area of Steiner trees, to appear in the series: *Advances in Discrete Mathematics and Computer Science* (Hadronic Press).
[17] T. Ibaraki, Branch-and-bound procedure and state-space representation of combinatorial optimization problems, Information and Control 36(1978)1–27.
[18] T. Ibaraki, Implementation and concurrent execution of branch-and-bound algorithms, Ann. Oper. Res. 10/11(1987), ch. 9.
[19] R.M. Karp, Reducibility among combinatorial problems, in: *Complexity of Computer Computations*, ed. R.E. Miller and J.W. Thatcher (Plenum Press, New York, 1972), pp. 85–103.
[20] W.H. Kohler and K. Steiglitz, Characterization and theoretical comparison of branch-and-bound algorithms for permutation problems, J. ACM 21(1974)140–156.
[21] V. Kumar and L.N. Kanal, A general branch-and-bound formulation for understanding and synthesizing AND/OR tree search procedures, Artificial Intelligence 21(1983)179–198.
[22] T.H. Lai and S. Sahni, Anomalies in parallel branch-and-bound algorithms, CACM 27(1984).

[23] T.H. Lai and A. Sprague, A note on anomalies in parallel branch-and-bound algorithms with one-to-one bounding functions, Inf. Proc. Lett. 23(1986)119–122.
[24] A.H. Land and A.G. Doig, An automatic method of solving discrete programming problems, Econometrica 28(1960)497–520.
[25] E.L. Lawler and D.E. Wood, Branch-and-bound methods: A survey, Oper. Res. 14(1966)699–719.
[26] G.-J. Li and B.W. Wah, Coping with anomalies in parallel branch-and-bound algorithms, IEEE Trans. Comp. C-35(1986)568–573.
[27] G.-J. Li and B.W. Wah, Computational efficiency of parallel approximate branch-and-bound algorithms, *Proc. 1984 Int. Conf. on Parallel Processing* (1984), pp. 473–480.
[28] I. Marshall and P. Messer, Conventions for generic abstract data type modules in Modula-2, School of Information Systems, University of East Anglia, Norwich, in preparation.
[29] G.P. McKeown, V.J. Rayward-Smith, S.A. Rush and H.J. Turpin, A framework for the implementation of parallel integer programming branch-and-bound on a transputer rack, submitted for publication.
[30] L.G. Mitten, Branch-and-bound methods: General formulation and properties, Oper. Res. 18(1970) 24–34.
[31] D.S. Nau, V. Kumar and L. Kanal, General branch-and-bound, and its relation to A* and AO*, Artificial Intelligence 23(1984)29–58.
[32] N.J. Nilsson, *Problem-solving Methods in Artificial Intelligence* (McGraw–Hill, New York, 1971).
[33] J. Plesnik, A bound for the Steiner tree problem in graphs, Math. Slovaca 31(1981)155–163.
[34] R.C. Prim, Shortest connection networks and some generalizations, Bell. Syst. Tech. J. 36(1957) 1389–1401.
[35] M.J. Quinn, *Designing Efficient Algorithms for Parallel Computers* (McGraw–Hill, New York, 1987).
[36] V.J. Rayward-Smith, The computation of nearly minimal Steiner trees in graphs, Int. J. Math. Educ. Sci. Tech. 14(1983)15–23.
[37] V.J. Rayward-Smith and A. Clare, On finding Steiner vertices, Networks 16(1986)283–294.
[38] V.J. Rayward-Smith, G.P. McKeown and F.W. Burton, The general problem solving algorithm and its implementation, New Generation Computing 6(1988)41–66.
[39] E.M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice* (Prentice–Hall, Englewood Cliffs, NJ, 1977).
[40] J.F. Shapiro, *Mathemetical Programming: Structures and Algorithms* (Wiley, New York, 1979).
[41] M.L. Shore, L.R. Foulds and P.B. Gibbons, An algorithm for the Steiner problem in graphs, Networks 12(1982)323–333.
[42] H. Takahashi and A. Matsuyama, An approximate solution for the Steiner problem in graphs, Math. Japonica 24(1980)573–577.
[43] H.J. Turpin, The branch-and-bound paradigm, Ph.D. Thesis, School of Information Systems, University of East Anglia, Norwich (1990).
[44] H.J. Turpin, G.P. McKeown, V.J. Rayward-Smith and S.A. Rush, Branch-and-bound on a transputer rack, submitted for publication.
[45] B.W. Wah, G.-J. Li and C.F. Yu, Multiprocessing of combinatorial search problems, IEEE Computer 18(1985).
[46] B.W. Wah and Y.W.E. Ma, MANIP – a multicomputer architecture for solving combinatorial extremum-search problems, IEEE Trans. Comp. C-33(1984)377–390.
[47] B.M. Waxman and M. Imase, Worst-case performance of Rayward-Smith's Steiner tree heuristic, Inf. Proc. Lett. 29(1988)283–287.
[48] Y.F. Wu, P. Widmayer and C.K. Wong, A faster approximation algorithm for the Steiner problem in graphs, Acta Informatica 23(1986)223–229.