Branch and Bound Algorithms - Principles and Examples.

Jens Clausen*

March 12, 1999

Contents

Abstract

A large number of real-world planning problems called combinatorial optimization problems share the following properties: They are optimization problems, are easy to state, and have a finite but usually very large number of feasible solutions. While some of these as e.g. the Shortest Path problem and the Minimum Spanning Tree problem have polynomial algoritms, the majority of the problems in addition share the property that no polynomial method for their solution is known. Examples here are vehicle

*Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark.

routing, crew scheduling, and production planning. All of these problems are $\mathcal{NP}$-hard.

Branch and Bound (B&B) is by far the most widely used tool for solving large scale $\mathcal{NP}$-hard combinatorial optimization problems. B&B is, however, an algorithm paradigm, which has to be filled out for each specific problem type, and numerous choices for each of the components exist. Even then, principles for the design of efficient B&B algorithms have emerged over the years.

In this paper I review the main principles of B&B and illustrate the method and the different design issues through three examples: the Symmetric Travelling Salesman Problem, the Graph Partitioning problem, and the Quadratic Assignment problem.

1 Introduction.

Solving $\mathcal{NP}$-hard discrete optimization problems to optimality is often an immense job requiring very efficient algorithms, and the B&B paradigm is one of the main tools in construction of these. A B&B algorithm searches the complete space of solutions for a given problem for the best solution. However, explicit enumeration is normally impossible due to the exponentially increasing number of potential solutions. The use of bounds for the function to be optimized combined with the value of the current best solution enables the algorithm to search parts of the solution space only implicitly.

At any point during the solution process, the status of the solution with respect to the search of the solution space is described by a pool of yet unexplored subset of this and the best solution found so far. Initially only one subset exists, namely the complete solution space, and the best solution found so far is ∞. The unexplored subspaces are represented as nodes in a dynamically generated search tree, which initially only contains the root, and each iteration of a classical B&B algorithm processes one such node. The iteration has three main components: selection of the node to process, bound calculation, and branching. In Figure 1, the initial situation and the first step of the process is illustrated.

The sequence of these may vary according to the strategy chosen for selecting the next node to process. If the selection of next subproblem is based on the bound value of the subproblems, then the first operation of an iteration after choosing the node is branching, i.e. subdivision of the solution space of the node into two or more subspaces to be investigated in a subsequent iteration. For each of these, it is checked whether the subspace consists of a single solution, in which case it is compared to the current best solution keeping the best of these. Otherwise the bounding function for the subspace is calculated and compared to the current best solution. If it can be established that the subspace cannot contain the optimal solution, the whole subspace is discarded, else it is stored in the pool of live nodes together with it's bound. This is in [2] called the eager

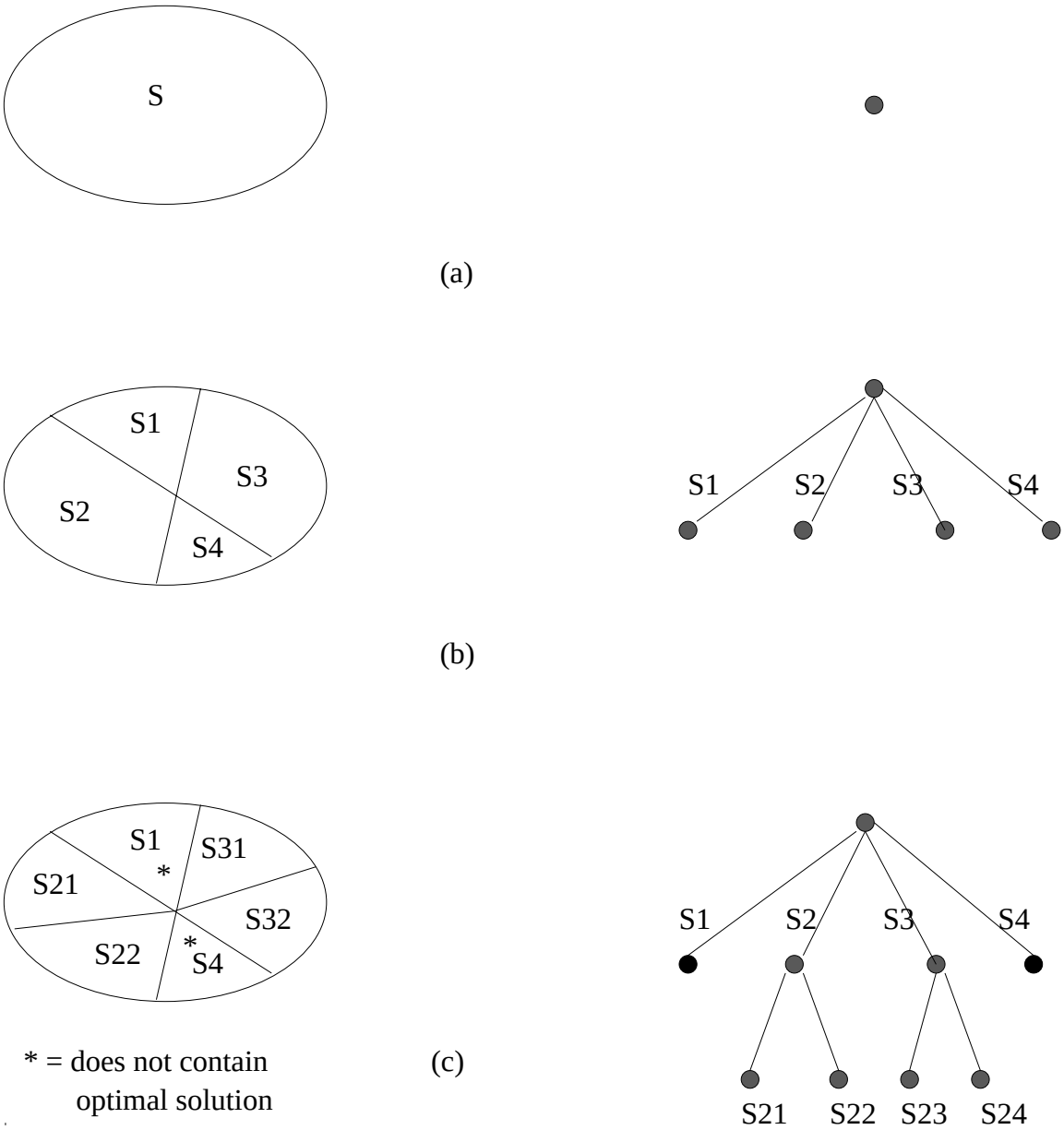


Figure 1: Illustration of the search space of B&B.

strategy for node evaluation, since bounds are calculated as soon as nodes are available. The alternative is to start by calculating the bound of the selected node and then branch on the node if necessary. The nodes created are then stored together with the bound of the processed node. This strategy is called lazy and is often used when the next node to be processed is chosen to be a live node of maximal depth in the search tree.

The search terminates when there is no unexplored parts of the solution space left, and the optimal solution is then the one recorded as "current best".

The paper is organized as follows: In Section 2, I go into detail with terminology and problem description and give the three examples to be used succeedingly. Section 2.1, 2.2, and 2.3 then treat in detail the algorithmic components selection, bounding and branching, and Section 2.4 briefly comments upon methods for generating a good feasible solution prior to the start of the search. I then describe personal experiences with solving two problems using parallel B&B in Section 3.1 and 3.2, and Section 4 discusses the impact of design decisions on the efficiency of the complete algorithm.

2 B&B - terminology and general description.

In the following I consider minimization problems - the case of maximization problems can be dealt with similarly. The problem is to minimize a function $f(x)$ of variables $(x_1 \cdots x_n)$ over a region of *feasible solutions*, S :

$$\min_{x \in S} \ f(x)$$

The function f is called the *objective function* and may be of any type. The set of feasible solutions is usually determined by general conditions on the variables, e.g. that these must be non-negative integers or binary, and special constraints determining the structure of the feasible set. In many cases, a set of *potential solutions*, P, containing S, for which f is still well defined, naturally comes to mind, and often, a function $g(x)$ defined on S (or P) with the property that $g(x) \leq f(x)$ for all x in S (resp. P) arises naturally. Both P and g are very useful in the B&B context. Figure 2 illustrates the situation where S and P are intervals of reals.

I will use the terms *subproblem* to denote a problem derived from the originally given problem through addition of new constraints. A subproblem hence corresponds to a subspace of the original solution space, and the two terms are used interchangeably and in the context of a search tree interchangeably with the term *node*. In order to make the discussions more explicit I use three problems as examples. The first one is one of the most famous combinatorial optimization problems: the Travelling Salesman problem. The problem arises naturally in connection with routing of vehicles for delivery and pick-up of goods or persons,

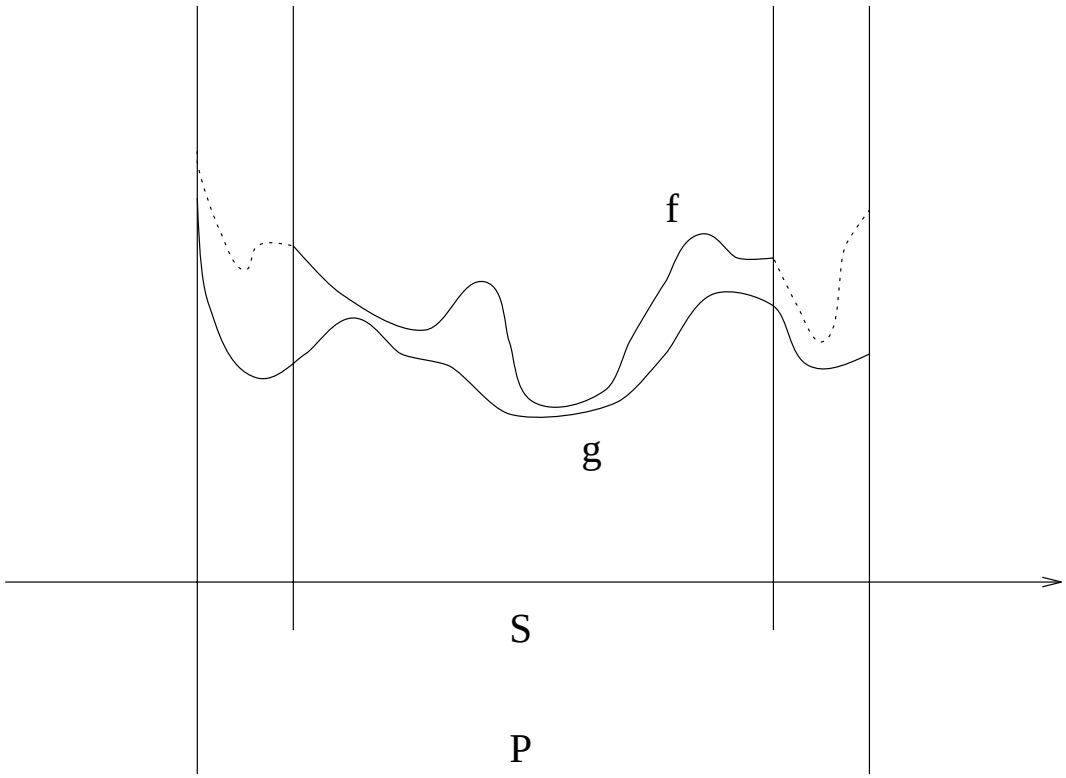
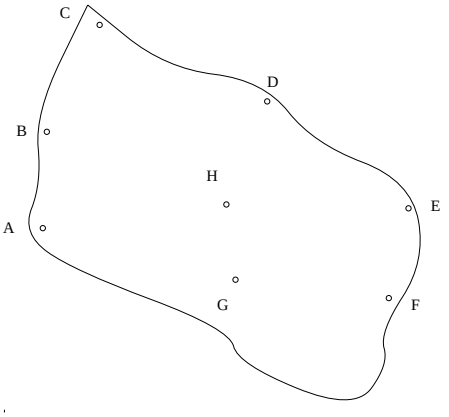


Figure 2: The relation between the bounding function g and the objective function f on the sets S and P of feasible and potential solutions of a problem.



	A	B	C	D	E	F	G	H
A	0	11	24	25	30	29	15	15
B	11	0	13	20	32	37	17	17
C	24	13	0	16	30	39	29	22
D	25	20	16	0	15	23	18	12
E	30	32	30	15	0	9	23	15
F	29	37	39	23	9	0	14	21
G	15	17	29	18	23	14	0	7
H	15	17	22	12	15	21	7	0

Figure 3: The island Bornholm and the distances between interesting sites

but has numerous other applications. A famous and thorough reference is [11].

Example 1: The Symmetric Travelling Salesman problem. In Figure 3, a map over the Danish island Bornholm is given together with a distance table showing the distances between major cities/tourist attractions. The problem of a biking tourist, who wants to visit all these major points, is to find a tour of minimum length starting and ending in the same city, and visiting each other city exactly once. Such a tour is called a *Hamilton cycle*. The problem is called the *symmetric* Travelling Salesman problem (TSP) since the table of distances is symmetric.

In general a symmetric TSP is given by a symmetric $n \times n$ matrix D of non-negative distances, and the goal is to find a Hamilton tour of minimum length. In terms of graphs, we consider a complete undirected graph with n vertices K_n and non-negative lengths assigned to the edges, and the goal is to determine a Hamilton tour of minimum length. The problem may also be stated mathematically by using decision variables to describe which edges are to be included in the tour. We introduce 0-1 variables $x_{ij}, 1 \leq i < j \leq n$, and interpret the value 0 (1 resp.) to mean "not in tour" ("in tour" resp.) The problem is then

$$min \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} d_{ij} x_{ij}$$

such that

$$\sum_{k=1}^{i-1} x_{ki} + \sum_{k=i+1}^{n} x_{ik} = 2, \qquad i \in \{1, ..., n\}$$

$$\sum_{i,j \in Z} x_{ij} < |Z| \qquad \emptyset \subset Z \subset V$$

$$x_{ij} \in \{0, 1\}, \qquad i, j \in \{1, ..., n\}$$

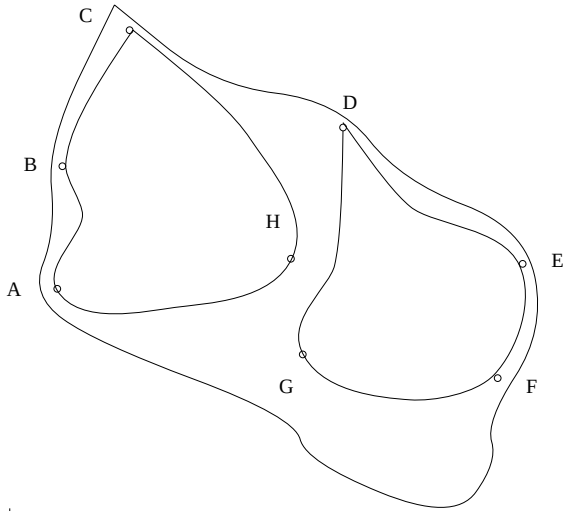


Figure 4: A potential, but not feasible solution to the biking tourist's problem

The first set of constraints ensures that for each i exactly two variables corresponding to edges incident with i are chosen. Since each edge has two endpoints, this implies that exactly n variables are allowed to take the value 1. The second set of constraints consists of the subtour elimination constraints. Each of these states for a specific subset S of V that the number of edges connecting vertices in S has to be less than $|S|$ thereby ruling out that these form a subtour. Unfortunately there are exponentially many of these constraints.

The given constraints determine the set of feasible solutions S. One obvious way of relaxing this to a set of potential solutions is to relax (i.e. discard) the subtour elimination constrains. The set of potential solutions P is then the family of all sets of subtours such that each i belongs to exactly one of the subtours in each set in the family, cf. Figure 4. In Section 2.1 another possibility is decribed, which in a B&B context turns out to be more appropriate.

A subproblem of a given symmetric TSP is constructed by deciding for a subset A of the edges of G that these must be *included* in the tour to be constructed, while for another subset B the edges are *excluded* from the tour. Exclusion of an edge (i, j) is usually modeled by setting c_{ij} to ∞, whereas the inclusion of an edge can be handled in various ways as e.g. graph contraction. The number of feasible solutions to the problem is $(n-1)!/2$, which for $n = 50$ is appr. 3×10^{62} □

The following descriptions follow [4, 5].

Example 2: The Graph Partitioning Problem. The Graph Partitioning problem arises in situations, where it is necessary to minimize the number (or weight of) connections between two parts of a network of prescribed size. We consider

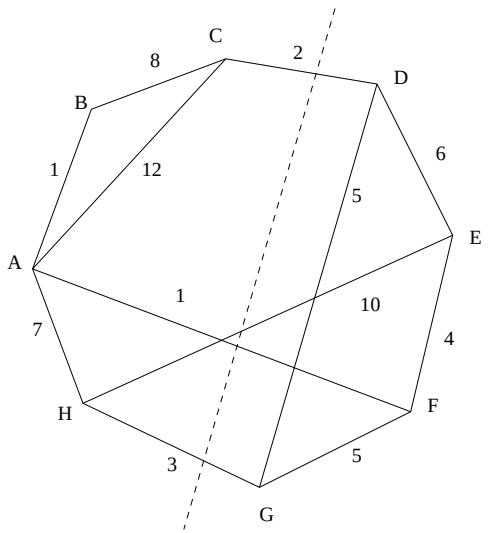


Figure 5: A graph partitioning problem and a feasible solution.

a given weighted, undirected graph G with vertex set V and edge set E , and a cost function $c : E \to \mathcal{N}$. The problem is to partition V into two disjoint subsets V_1 and V_2 of equal size such that the sum of costs of edges connecting vertices belonging to different subsets is as small as possible. Figure 5 shows an instance of the problem:

The graph partitioning problem can be formulated as a quadratic integer programming problem. Define for each vertex v of the given graph a variable x_v, which can attain only the values 0 and 1. A 1-1 correspondence between partitions and assignments of values to all variables now exists: $x_v = 1$ (respectively $= 0$) if and only if $v \in V_1$ (respectively $v \in V_2$). The cost of a partition is then

$$\sum_{v \in V_1, u \in V_2} c_{uv} x_v (1 - x_u)$$

A constraint on the number of variables attaining the value 1 is included to exclude infeasible partitions.

$$\sum_{v \in V} x_v = |V|/2$$

The set of feasible solutions S is here the partitions of V into two equal-sized subsets. The natural set P of potential solutions are all partitions of V into two non-empty subsets.

Initially V_1 and V_2 are empty corresponding to that no variables have yet been assigned a value. When some of the vertices have been assigned to the sets (the corresponding variables have been assigned values 1 or 0), a subproblem has been constructed.

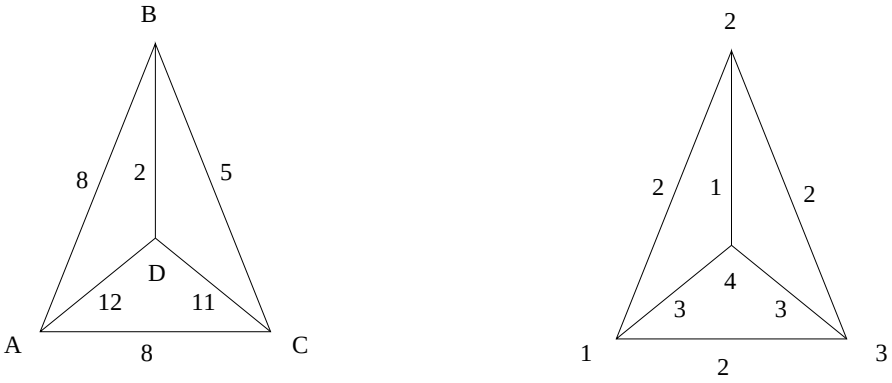


Figure 6: A Quadratic Assignment problem of size 4.

The number of feasible solutions to a GPP with $2n$ vertices equals the binomial coefficient $C(2n, n)$. For $2n = 120$ the number of feasible solutions is appr. 9.6×10^{34}. □

Example 3: The Quadratic Assignment Problem. Here, I consider the Koopmans-Beckman version of the problem, which can informally be stated with reference to the following practical situation: A company is to decide the assignment of n of facilities to an equal number of locations and wants to minimize the total transportation cost. For each pair of facilities (i, j) a flow of communication $f_{i,j}$ is known, and for each pair of locations (l, k) the corresponding distance $d_{l,k}$ is known. The transportation cost between facilities i and j, given that i is assigned to location l and j is assigned to location k, is $f_{i,j} \cdot dl, k$, and the objective of the company is to find an assignment minimizing the sum of all transportation costs. Figure 6 shows a small example with 4 facilities and 4 locations. The assignment of facilities A,B,C, and D on sites 1,2,3, and 4 respectively has a cost of 224.

Each feasible solution corresponds to a permutation of the facilities, and letting S denote the group of permutations of n elements, the problem can hence formally be stated as

$$\min_{\pi \in S} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{i,j} \cdot d_{\pi(i),\pi(j)}$$

A set of potential solutions is e.g. obtained by allowing more than one facility on each location.

Initially no facilities have been placed on a location, and subproblems of the original problem arise when some but not all facilities have been assigned to locations.

Again the number of feasible solutions grows exponentially: For a problem with n facilities to be located, the number of feasible solutions is $n!$, which for $n = 20$ is appr. 2.43×10^{18}. □

The solution of a problem with a B&B algorithm is traditionally described as a search through a search tree, in which the root node corresponds to the original problem to be solved, and each other node corresponds to a subproblem of the original problem. Given a node Q of the tree, the children of Q are subproblems derived from Q through imposing (usually) a single new constraint for each subproblem, and the descendants of Q are those subproblems, which satisfy the same constraints as Q and additionally a number of others. The leaves correspond to feasible solutions, and for all $\mathcal{NP}$-hard problems, instances exist with an exponential number of leaves in the search tree. To each node in the tree a *bounding function* g associates a real number called the *bound* for the node. For leaves the bound equals the value of the corresponding solution, whereas for internal nodes the value is a lower bound for the value of any solution in the subspace corresponding to the node. Usually g is required to satisfy the following three conditions:

1. $g(P_i) \leq f(P_i)$ for all nodes P_i in the tree
2. $g(P_i) = f(P_i)$ for all leaves in the tree
3. $g(P_i) \geq g(P_j)$ if P_j is the father of P_i

These state that g is a bounding function, which for any leaf agrees with the objective function, and which provides closer and closer (or rather not worse) bounds when more information in terms of extra constraints for a subproblem is added to the problem description.

The search tree is developed dynamically during the search and consists initially of only the root node. For many problems, a feasible solution to the problem is produced in advance using a heuristic, and the value hereof is used as the current best solution (called the *incumbent*). In each *iteration* of a B&B algorithm, a node is *selected* for exploration from the pool of *live* nodes corresponding to unexplored feasible subproblems using some selection strategy. If the eager strategy is used, a *branching* is performed: Two or more children of the node are constructed through the addition of constraints to the subproblem of the node. In this way the subspace is subdivided into smaller subspaces. For each of these the bound for the node is calculated, possibly with the result of finding the optimal solution to the subproblem, cf. below. In case the node corresponds to a feasible solution or the bound is the value of an optimal solution, the value hereof is compared to the incumbent, and the best solution and its value are kept. If the bound is no better than the incumbent, the subproblem is discarded (or *fathomed*), since no feasible solution of the subproblem can be better that the incumbent. In case no feasible solutions to the subproblem exist the subproblem is also fathomed. Otherwise the possibility of a better solution in the subproblem cannot be ruled out, and the node (with the bound as part of the information stored) is then joined to the pool

of live subproblems. If the lazy selection strategy is used, the order of bound calculation and branching is reversed, and the live nodes are stored with the bound of their father as part of the information. Below, the two algorithms are sketched:

Eager Branch and Bound

Initialize: Incumbent := ∞; LB(P_0) := $g(P_0)$; Live := $\{(P_0, LB(P_0))\}$

Repeat until Live = $\emptyset$
  Select the node P from Live to be processed; Live := Live $\setminus \{P\}$;
  Branch on P generating $P_1, ...P_k$;
  For $1 \leq i \leq k$ do
    Bound P_i : LB(P_i) := $g(P_i)$;
      If LB(P_i) = $f(X)$ for a feasible solution X
        and $f(X) < Incumbent$ then
          Incumbent := $f(X)$; Solution := X;
          go to EndBound;
      If LB(P_i) $\geq$ Incumbent then fathom P_i
        else Live := Live $\cup \{(P_i, LB(P_i))\}$
    EndBound;

OptimalSolution := Solution; OptimumValue := Incumbent

Lazy Branch and Bound

Initialize: Incumbent := -∞; Live := $\{(P_0, -\infty)\}$

Repeat until Live = $\emptyset$
  Select the node P from Live to be processed; Live := Live $\setminus \{P\}$;
  Bound P : LB(P) := $g(P)$
    If LB(P) = $f(X)$ for a feasible solution X
      and $f(X) < Incumbent$ then
        Incumbent := $f(X)$; Solution := X;
        go to EndBound;
    If LB(P) $\geq$ Incumbent then fathom P
      else Branch on P generating $P_1, ...P_k$;
          For $1 \leq i \leq k$ do
            Live := Live $\cup \{(P_i, LB(P))\}$;
  EndBound;

OptimalSolution := Solution; OptimumValue := Incumbent;

A B&B algorithm for a minimization problem hence consists of three main components:

1. a *bounding function* providing for a given subspace of the solution space a lower bound for the best solution value obtainable in the subspace,

2. a *strategy for selecting* the live solution subspace to be investigated in the current iteration, and

3. a *branching* rule to be applied if a subspace after investigation cannot be discarded, hereby subdividing the subspace considered into two or more subspaces to be investigated in subsequent iterations.

In the following, I discuss each of these key components briefly.

In addition to these, an *initial good feasible solution* is normally produced using a heuristic whenever this is possible in order to facilitate fathoming of nodes as early as possible. If no such heuristic exists, the initial value of the incumbent is set to infinity. It should be noted that other methods to fathom solution subspaces exist, e.g. dominance tests, but these are normally rather problem specific and will not be discussed further here. For further reference see [8].

2.1 Bounding function.

The bounding function is the key component of any B&B algorithm in the sense that a low quality bounding function cannot be compensated for through good choices of branching and selection strategies. Ideally the value of a bounding function for a given subproblem should equal the value of the best feasible solution to the problem, but since obtaining this value is usually in itself $\mathcal{NP}$-hard, the goal is to come as close as possible using only a limited amount of computational effort (i.e. in polynomial time), cf. the succeeding discussion. A bounding function is called *strong*, if it in general gives values close to the optimal value for the subproblem bounded, and *weak* if the values produced are far from the optimum. One often experiences a trade off between quality and time when dealing with bounding functions: The more time spent on calculating the bound, the better the bound value usually is. It is normally considered beneficial to use as strong a bounding function as possible in order to keep the size of the search tree as small as possible.

Bounding functions naturally arise in connection with the set of potential solutions P and the function g mentioned in Section 2. Due to the fact that $S \subseteq P$, and that $g(x) \leq f(x)$ on P, the following is easily seen to hold:

$$min_{x \in P}\ g(x) \leq \left\{ \begin{array}{l} min_{x \in P}\ f(x) \\ min_{x \in S}\ g(x) \end{array} \right\} \leq min_{x \in S}\ f(x)$$

If both of P and g exist there are now a choice between three optimization problems, for each of which the optimal solution will provide a lower bound for the given objective function. The "skill" here is of course to chose P and/or g so that one of these is easy to solve and provides tight bounds.

Hence there are two standard ways of converting the $\mathcal{NP}$-hard problem of solving a subproblem to optimality into a $\mathcal{P}$-problem of determining a lower bound for the objective function. The first is to use *relaxation* - leave out some of the constraints of the original problem thereby enlarging the set of feasible solutions. The objective function of the problem is maintained. This corresponds to minimizing f over P. If the optimal solution to the relaxed subproblem satisfies all constraints of the original subproblem, it is also optimal for this, and is hence a candidate for a new incumbent. Otherwise, the value is a lower bound because the minimization is performed over a larger set of values than the objective function values for feasible solutions to the original problem. For e.g. GPP, a relaxation is to drop the constraint that the sizes of V_1 and V_2 are to be equal.

The other way of obtaining an easy bound calculation problem is to minimize g over S, i.e. to maintain the feasible region of the problem, but modify the objective function at the same time ensuring that for all feasible solutions the modified function has values less than or equal to the original function. Again one can be sure that a lower bound results from solving the modified problem to optimality, however, it is generally not true that the optimal solution corresponding

to the modified objective function is optimal for the original objective function too. The most trivial and very weak bounding function for a given minimization problem obtained by modification is the sum of the cost incurred by the variable bindings leading to the subproblem to be bounded. Hence all feasible solutions for the subproblem are assigned the same value by the modified objective function. In GPP this corresponds to the cost on edges connecting vertices assigned to V_1 in the partial solution with vertices assigned to V_2 in the partial solution, and leaving out any evaluation of the possible costs between one assigned and one unassigned vertex, and costs between two assigned vertices. In QAP, an initial and very weak bound is the transportation cost between facilities already assigned to locations, leaving out the potential costs of transportation between one unassigned and one assigned, as well as between two unassigned facilities. Much better bounds can be obtained if these potential costs are included in the bound, cf. the Roucairol-Hansen bound for GPP and the Gilmore-Lawler bound for QAP as described e.g. in [4, 5].

Combining the two strategies for finding bounding functions means to minimize g over P, and at first glance this seems weaker than each of those. However, a parameterized family of lower bounds may result, and finding the parameter giving the optimal lower bound may after all create very tight bounds. Bounds calculated by so-called *Lagrangean relaxation* are based on this observation - these bounds are usually very tight but computationally demanding. The TSP provides a good example hereof.

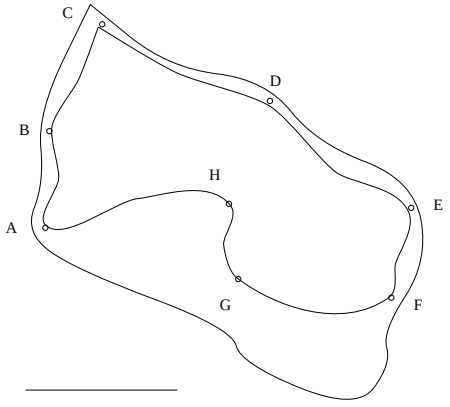
Example 4: The 1-tree bound for symmetric TSP problems. As mentioned, one way of relaxing the constraints of a symmetric TSP is to allow subtours. However, the bounds produced this way are rather weak. One alternative is the 1-tree relaxation.

Here one first identifies a special vertex, "#1", which may be any vertex of the graph. "#1" and all edges incident to this are removed from G, and a minimum spanning tree T_{rest} is found for the remaining graph. Then the two shortest edges e_1, e_2 incident to "#1" are added to T_{rest} producing the 1-tree T_{one} of G with respect to "#1", cf. Figure 7.

The total cost of T_{one} is a lower bound of the value of an optimum tour. The argument for this is as follows: First note that a Hamilton tour in G consists of two edges e'_1, e'_2 and a tree T'_{rest} in the rest of G. Hence the set of Hamilton tours of G is a subset of the set of 1-trees of G. Since e_1, e_2 are the two shortest edges incident to "#1" and T_{rest} is the minimum spanning tree in the rest of G, the cost of T_{one} is less than or equal the cost of any Hamilton tour.

In case T_{one} is a tour, we have found the optimal solution to our subproblem - otherwise a vertex of degree at least 3 exists and we have to perform a branching.

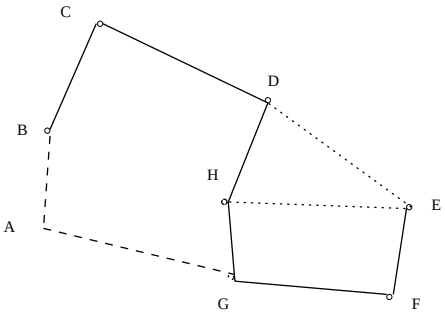
The 1-tree bound can be strengthened using the idea of problem transformation: Generate a new symmetric TSP problem having the same optimal tour as the original, for which the 1-tree bound is tighter. The idea is that vertices



Optimal tour, cost = 100

	A	B	C	D	E	F	G	H
A	0	11	24	25	30	29	15	15
B	11	0	13	20	32	37	17	17
C	24	13	0	16	30	39	29	22
D	25	20	16	0	15	23	18	12
E	30	32	30	15	0	9	23	15
F	29	37	39	23	9	0	14	21
G	15	17	29	18	23	14	0	7
H	15	17	22	12	15	21	7	0

(a)



Tree in rest of G

Edge left out by Kruskal's MST algorithm

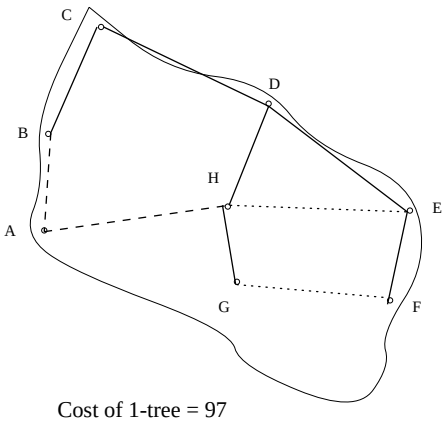
1-tree edge

Cost of 1-tree = 97

(b)

Modified distance matrix:

	A	B	C	D	E	F	G	H
A	0	11	24	25	29	29	16	15
B	11	0	13	20	31	37	18	17
C	24	13	0	16	29	39	30	22
D	25	20	16	0	14	23	19	12
E	29	31	29	14	0	8	23	14
F	29	37	39	23	8	0	15	21
G	16	18	30	19	23	15	0	8
H	15	17	22	12	14	21	8	0



Cost of 1-tree = 97

(c)

Figure 7: A bound calculation of the B&B algorithm for the symmetric TSP using the 1-tree bound with "#1" equal to A and Lagrangean relaxation for bounding.

of T_{one} with high degree are incident with too many attractive edges, whereas vertices of degree 1 have too many unattractive edges. Denote by π_i the degree of vertex i minus 2: $\pi_i := deg(v_i) - 2$. Note that the sum over V of the values π equals 0 since T_{one} has n edges, and hence the sum of $deg(v_i)$ equals $2n$. Now for each edge (i, j) we define the transformed cost c'_{ij} to be $c_{ij} + \pi_i + \pi_j$. Since each vertex in a Hamilton tour is incident to exactly two edges, the new cost of a Hamilton tour is equal to the current cost plus two times the sum over V of the values π. Since the latter is 0, the costs of all tours are unchanged, but the costs of 1-trees in general increase. Hence calculating the 1-tree bound for the transformed problem often gives a better bound, but not necessarily a 1-tree, which is a tour.

The trick may be repeated as many times as one wants, however, for large instances a tour seldomly results. Hence, there is a trade-off between time and strength of bound: should one branch or should one try to get an even stronger bound than the current one by a problem transformation ? Figure 7 (c) shows the first transformation for the problem of Figure 7 (b).

2.2 Strategy for selecting next subproblem.

The strategy for selecting the next live subproblem to investigate usually reflects a trade off between keeping the number of explored nodes in the search tree low, and staying within the memory capacity of the computer used. If one always selects among the live subproblems one of those with the lowest bound, called the *best first search strategy*, BeFS, no superfluous bound calculations take place after the optimal solution has been found. Figure 8 (a) shows a small search tree - the numbers in each node corresponds to the sequence, in which the nodes are processed when BeFS is used.

The explanation of the property regarding superfluos bound calculations lies in the concept of *critical* subproblems. A subproblem P is called *critical* if the given bounding function when applied to P results in a value strictly less than the optimal solution of the problem in question. Nodes in the search tree corresponding to critical subproblems have to be partitioned by the B&B algorithm no matter when the optimal solution is identified - they can never be discarded by means of the bounding function. Since the lower bound of any subspace containing an optimal solution must be less than or equal to the optimum value, only nodes of the search tree with lower bound less than or equal to this will be explored. After the optimal value has been discovered only critical nodes will be processed in order to prove optimality. The preceding argument for optimality of BeFS with respect to number of nodes processed is valid only if eager node evaluation is used since the selection of nodes is otherwise based on the bound value of the father of each node. BeFS may, however, also be used in combination with lazy node evaluation.

Even though the choice of the subproblem with the current lowest lower bound

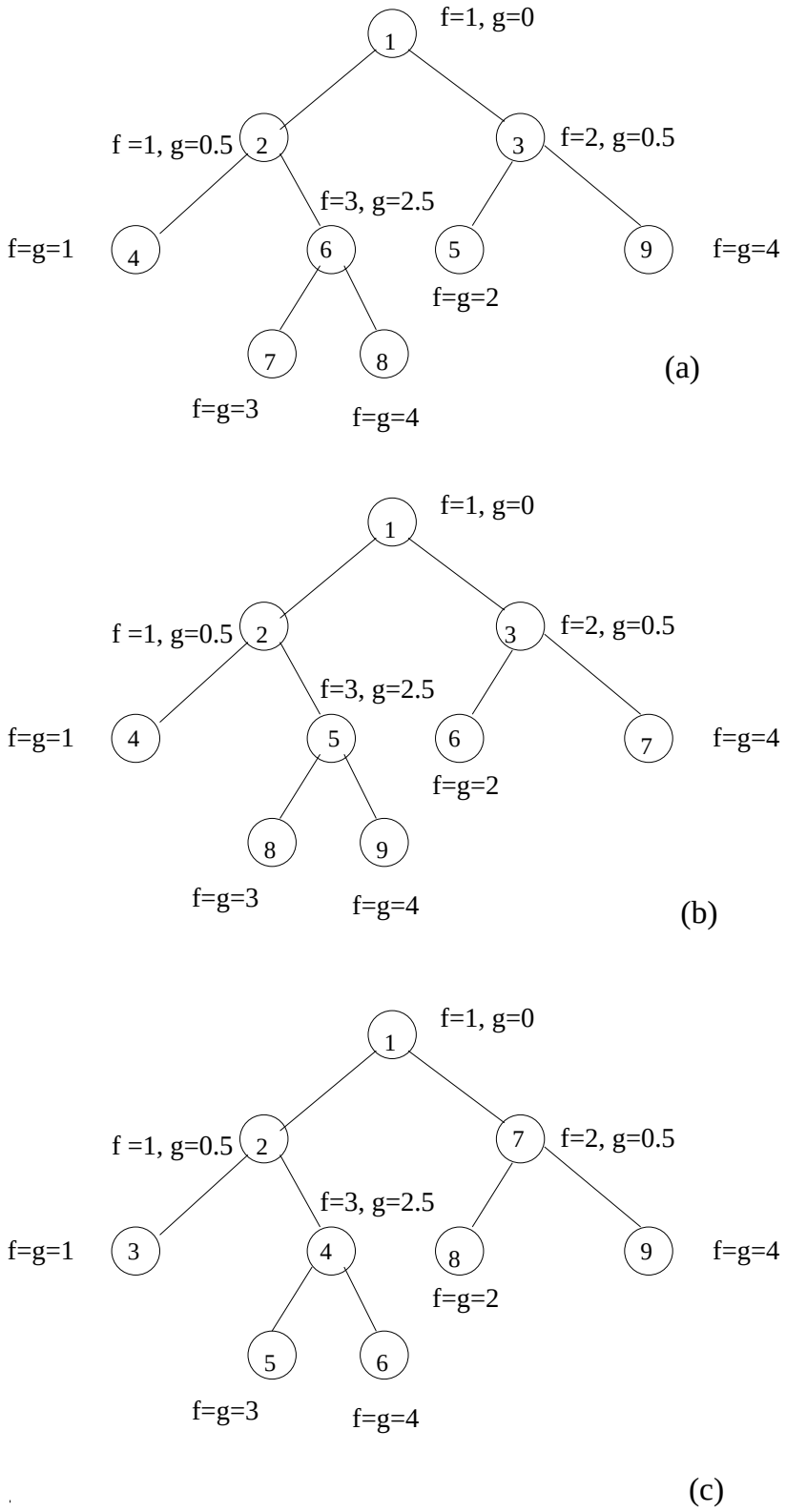


Figure 8: Search strategies in B&B: (a) Best First Search, (b) Breadth First Search, and (c) Depth First Search.

makes good sense also regarding the possibility of producing a good feasible solution, memory problems arise if the number of critical subproblems of a given problem becomes too large. The situation more or less corresponds to a *breath first search* strategy, in which all nodes at one level of the search tree are processed before any node at a higher level. Figure 8 (b) shows the search tree with the numbers in each node corresponding to the BFS processing sequence. The number of nodes at each level of the search tree grows exponentially with the level making it infeasible to do breadth first search for larger problems. For GPP sparse problems with 120 vertices often produce in the order of a few hundred of critical subproblems when the Roucairol-Hansen bounding function is used [4], and hence BeFS seems feasible. For QAP the famous Nugent20 problem [13] produces 3.6×10^8 critical nodes using Gilmore-Lawler bounding combined with detection of symmetric solutions [5], and hence memory problems may be expected if BeFS is used.

The alternative used is *depth first search*, DFS. Here a live node with largest level in the search tree is chosen for exploration. Figure 8 (c) shows the DFS processing sequence number of the nodes. The memory requirement in terms of number of subproblems to store at the same time is now bounded above by the number of levels in the search tree multiplied by the maximum number of children of any node, which is usually a quite manageable number. DFS can be used both with lazy and eager node evaluation. An advantage from the programming point of view is the use of recursion to search the tree - this enables one to store the information about the current subproblem in an incremental way, so only the constraints added in connection with the creation of each subproblem need to be stored. The drawback is that if the incumbent is far from the optimal solution, large amounts of unnecessary bounding computations may take place.

In order to avoid this, DFS is often combined with a selection strategy, in which one of the branches of the selected node has a very small lower bound and the other a very large one. The idea is that exploring the node with the small lower bound first hopefully leads to a good feasible solution, which when the procedure returns to the node with the large lower bound can be used to fathom the node. The node selected for branching is chosen as the one, for which the difference between the lower bounds of its children is as large as possible. Note however that this strategy requires the bound values for children to be known, which again may lead to superfluous calculations.

A combination of DFS as the overall principle and BeFS when choice is to be made between nodes at the same level of the tree is also quite common.

In [2] an experimental comparison of BeFS and DFS combined with both eager and lazy node evaluation is performed for QAP. Surprisingly, DFS is superior to BeFS in all cases, both in terms of time and in terms of number of bound calculations. The reason turns out to be that in practice, the bounding and branching of the basic algorithm is extended with additional tests and calculations at each node in order to enhance efficiency of the algorithm. Hence, the

theoretical superiority of BeFS should be taken with a grain of salt.

2.3 Branching rule.

All branching rules in the context of B&B can bee seen as subdivision of a part of the search space through the addition of constraints, often in the form of assigning values to variables. If the subspace in question is subdivided into two, the term *dichotomic* branching is used, otherwise one talks about *polytomic* branching. Convergence of B&B is ensured if the size of each generated subproblem is smaller than the original problem, and the number of feasible solutions to the original problem is finite. Normally, the subproblems generated are disjoint - in this way the problem of the same feasible solution appearing in different subspaces of the search tree is avoided.

For GPP branching is usually performed by choosing a vertex not yet assigned to any of V_1 and V_2 and assigning it to V_1 in one of the new subproblems (corresponding to that the variable of the node receives the value 1) and to V_2 in the other (variable value equal to 0). This branching scheme is dichotomic, and the subspaces generated are disjoint.

In case of QAP, an unassigned facility is chosen, and a new subproblem is created for each of the free location by assigning the chosen facility to the location. The scheme is called branching on facilities and is polytomic, and also here the subspaces are disjoint. Also branching on locations is possible.

For TSP branching may be performed based on the 1-tree generated during bounding. If all vertices have degree 2 the 1-tree is a tour, and hence an optimal solution to the subproblem. Then no further branching is required. If a node has degree 3 or more in the 1-tree, any such node may be chosen as the source of branching. For the chosen node, a number of subproblems equal to the degree is generated. In each of these one of the edges of the 1-tree is excluded from the graph of the subproblem ruling out the possibility that the bound calculation will result in the same 1-tree. Figure 9 shows the branching taking place after the bounding in Figure 7. The bound does, however, not necessarily change, and identical subproblems may arise after a number of branchings. The effect of the latter is not an incorrect algorithm, but a less efficient algorithm. The problem is further discussed as an exercise.

2.4 Producing an initial solution.

Although often not explicitly mentioned, another key issue in the solution of large combinatorial optimization problems by B&B is the construction of a good initial feasible solution. Any heuristic may be used, and presently a number of very good general heuristics as well as a wealth of very problem specific heuristics are available. Among the general ones (also called meta-heuristics or paradigms

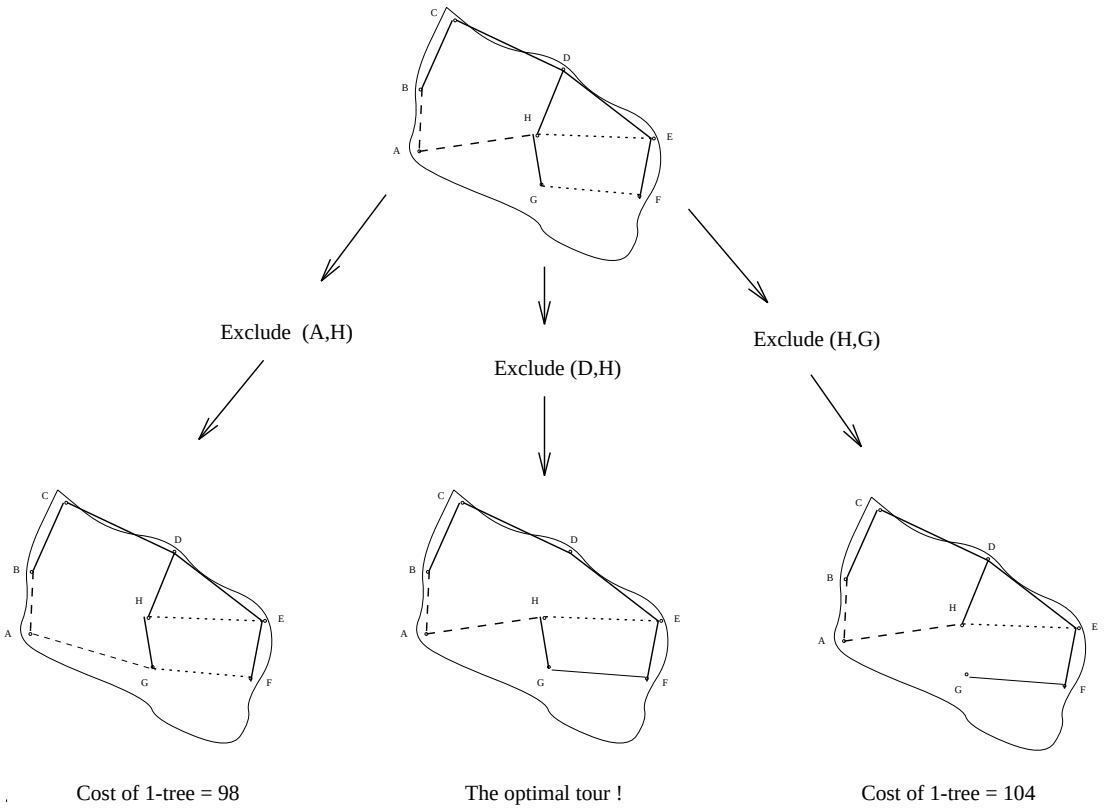


Figure 9: Branching from a 1-tree in a B&B algorithm for the symmetric TSP.

for heuristics), Simulated Annealing, Genetic Algorithms, and Tabu Search are the most popular.

As mentioned, the number of subproblems explored when the DFS strategy for selection is used depends on the quality of the initial solution - if the heuristic identifies the optimal solution so that the B&B algorithm essentially verifies the optimality, then even DFS will only explore critical subproblems. If BeFS is used, the value of a good initial solution is less obvious.

Regarding the three examples, a good and fast heuristic for GPP is the Kernighan-Lin variable depth local search heuristic. For QAP and TSP, very good results have been obtained with Simulated Annealing.

3 Personal Experiences with GPP and QAP.

The following subsections briefly describe my personal experiences using B&B combined with parallel processing to solve GPP and QAP. Most of the material stems from [3, 4] and [5]. Even though parallelism is not an integral part of B&B, I have chosen to present the material, since the key components of the B&B are

unchanged. A few concepts from parallel processing is, however, necessary.

Using parallel processing of the nodes in the search tree of a B&B algorithm is a natural idea, since the bound calculation and the branching in each node is independent. The aim of the parallel procesing is to speed up the execution time of the algorithm, To measure the success in this aspect, the *speed-up* of adding processors is measured. The relative speed-up using p processors is defined to be the processing time $T(1)$ using one processor divided by the processing time $T(p)$ using p processors:

$$S(p) = T(1)/T(p)$$

The ideal value of $S(p)$ is p - then the problem is solved p times faster with p processors than with 1 processor.

An important issue in parallel B&B is distribution of work: in order to obtain as short running time as possible, no processor should be idle at any time during the computation. If a distributed system or a network of workstations is used, this issue becomes particularly crucial since it is not possible to maintain a central pool of live subproblems. Various possibilities for *load balancing schemes* exist - two concrete examples are given in the following, but additional ones are described in [7].

3.1 Solving the Graph Partitioning Problem in Parallel.

GPP was my first experience with parallel B&B, and we implemented two parallel algorithms for the problem in order to investigate the trade off between bound quality and time to calculate the bound. One - called the CT-algorithm - uses an easily computable bounding function based on the principle of modified objective function and produces bounds of acceptable quality, whereas the other - the RH-algorithm - is based on Lagrangean relaxation and has a bounding function giving tight, but computationally expensive bounds.

The system used was a 32 processor IPSC1/d5 hypercube equipped with Intel 80286 processors and 80287 co-processors each with 512 KB memory. No dedicated communication processors were present, and the communication facilities were Ethernet connections implying a large start-up cost on any communication.

Both algorithms were of the distributed type, where the pool of live subproblems is distributed over all processors, and as strategy for distributing the workload we used a combined "on demand"/"on overload" strategy. The "on overload" strategy is based on the idea that if a processor has more than a given threshold of live subproblems, a number of these are sent to neighbouring processors. However, care must be take to ensure that the system is not floated with communication and that flow of subproblems between processors takes place during the entire solution process. The scheme is illustrated in Figure 10.

Regarding termination, the algorithm of Dijkstra et. al. [6] was used. The selection strategy for next subproblem were BeFs for the RH-algorithm and DFS

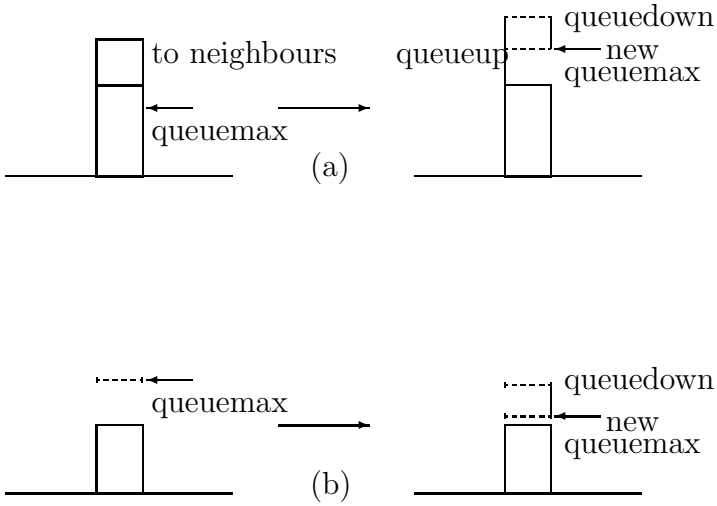


Figure 10: Illustration of the on overload protocol. (a) is the situation, when a processor when checking finds that it is overloaded, and (b) shows the behaviour of a non-overloaded processor

No. of proc.		4	8	16	32
CT	time (sec)	1964	805	421	294
	proc. util. (%)	97	96	93	93
	no. of bound calc.	449123	360791	368923	522817
RH	time (sec)	1533	1457	1252	1219
	proc. util. (%)	89	76	61	42
	no. of bound calc.	377	681	990	1498

Table 1: Comparison between the CT- and RH-algorithm on a 70 vertex problem with respect to running times, processor utilization, and number of subproblems solved.

for the CT-algorithm. The first feasible solution was generated by the Kernighan-Lin heuristic, and its value was usually close to the optimal solution value.

For the CT-algorithm, results regarding processor utilization and relative speed-up were promising. For large problems, a processor utilization near 100% was observed, and linear speed-up close to the ideal were observed for problems solvable also on a single processor. Finally we observed that the best speed-up was observed for problems with long running times. The RH-algorithm behaved differently - for small to medium size problems, the algorithm was clearly inferior to the CT-algorithm, both with respects to running time, relative speed-up and processor utilization. Hence the tight bounds did not pay off for small problems - they resulted idle processors and long running times.

We continued to larger problems expecting the problem to disappear, and

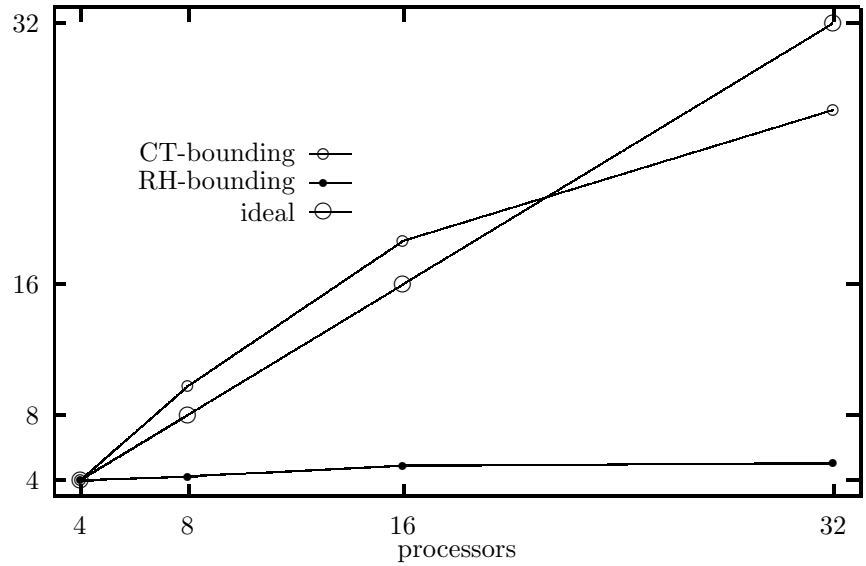


Figure 11: Relative speed-up for the CT-algorithm and the RH-algorithm for a 70 vertex problem.

	CT-algorithm			RH-algorithm		
No. Vert.	Cr. subpr.	B. calc.	Sec.	Cr. subpr.	B. calc.	Sec.
30	103	234	1	4	91	49
40	441	803	2	11	150	114
50	2215	3251	5	15	453	278
60	6594	11759	18	8	419	531
70	56714	171840	188	26	1757	1143
80	526267	901139	931	19	2340	1315
100	2313868	5100293	8825	75	3664	3462
110	8469580	15203426	34754	44	3895	4311
120	–	–	–	47	4244	5756

Table 2: Number of critical subproblems and bound calculations as a function of problem size.

Figure 11 and Table 1 shows the results for a 70-vertex problem for the CT- and RH-algorithms. We found that the situation did by no means improve. For the RH method it seemed impossible to use more than 4 processors. The explanation was found in the number of critical subproblems generated, cf. Table 2. Here it is obvious that using more processors for the RH-method just results in a lot of superfluous subproblems being solved, which does not decrease the total solution time.

3.2 Solving the QAP in Parallel.

QAP is one of my latest parallel B&B experiences. The aim of the research was in this case to solve the previously unsolved benchmark problem Nugent20 to optimality using a combination of the most advanced bounding functions and fast parallel hardware, as well as any other trick we could find and think of.

We used a MEIKO system consisting of 16 Intel i860 processors each with 16 MB of internal memory. Each processor has a peak performance of 30 MIPS when doing integer calculation giving an overall peak performance of approximately 500 MIPS for the complete system. The performance of each single i860 processor almost matches the performance of the much more powerful Cray 2 on integer calculations, indicating that the system is very powerful.

The processors each have two Inmos T800 transputers as communication processors. Each transputer has 4 communication channels each with bandwidth 1.4 Mb/second and start-up latency 340 μs. The connections are software programmable, and the software supports point-to-point communication between any pair of processors. Both synchronous and asynchronous communication are possible, and also both blocking and non-blocking communication exist.

The basic framework for testing bounding functions was a distributed B&B algorithm with the processors organized as a ring. Workload distribution was kept simple and based on local synchronization. Each processor in the ring communicates with each of its neighbours at certain intervals. At each communication the processors exchange information on the respective sizes of subproblem pools, and based here-on, subproblems are sent between the processors. The speed-up obtained with this scheme was 13.7 for a moderately sized problem with a sequential running time of 1482 seconds and a parallel running time with 16 processors of 108 seconds.

The selection strategy used was a kind of breadth first search. The feasibility hereof is intimately related to the use of a very good heuristic to generate the incumbent. We used simulated annealing, and as reported in [5], spending less than one percent of the total running time in the heuristic enabled us to start the parallel solution with the optimal solution as the incumbent. Hence only critical subproblems were solved. Regarding termination detection, a tailored algorithm were used for this purpose.

The main results of the research are indicated in Table 3. We managed to solve previously unsolved problems, and for problems solved by other researchers, the results clearly indicated the value of choosing an appropriate parallel system for the algorithm in question.

To get an indication of the efficiency of so-called static workload distribution in our application, an algorithm with static workload distribution was also tested. The results appear in Table 4. The subproblems distributed to each processor were generated using BeFS sequential B&B until the pool of live subproblems were sufficiently large that each processors could get the required number of

	Mautor & Roucairol		Fac. br. w. symmetry	
Problem	No. nodes.	Time (s)	No. nodes.	Time (s)
Nugent 15	97286	121	105773	10
Nugent 16.2	735353	969	320556	34
Nugent 17	–	–	24763611	2936
Nugent 18	–	–	114948381	14777
Nugent 20	–	–	360148026	57963
Elshafei 19	575	1.4	471	0.5
Armour & Buffa 20	531997	1189	504452	111

Table 3: Result obtained by the present authors in solving large standard benchmark QAPs. Results obtained by Mautor and Roucairol is included for comparison.

Problem	Dynamic dist.	Init. subpr. per proc.	Static dist.
Nugent 8	0.040	1	0.026
Nugent 10	0.079	1	0.060
Nugent 12	0.328	6	0.381
Nugent 14	12.792	24	13.112
Nugent 15	10.510	41	11.746
Nugent 16	35.293	66	38.925

Table 4: Result obtained when solving standard benchmark QAPs using static workload distribution. Results obtained with dynamic distribution are included for comparison.

subproblems. Hence all processors receive equally promising subproblems. The optimal number of subproblems pr. processors were determined experimentally and equals roughly $(p-8)^4/100$, where p is the number of processors.

4 Ideas and Pitfalls for B&B users.

Rather than giving a conclusion, I will in the following try to equip new users of B&B - both sequential and parallel - with a checklist corresponding to my own experiences. Some of the points of the list have already been mentioned in the preceding sections, while some are new.

4.1 Points for sequential B&B

- The importance of finding a good initial incumbent cannot be overestimated, and the time used for finding such one is often only few percentages of the total running time of the algorithm.
- In case an initial solution very close to the optimum is expected to be known, the choice of node selection strategy and processing strategy makes little difference.
- With a difference of more than few percent between the value of the initial solution and the optimum the theoretically superior BeFS B&B shows inferior performance compared to both lazy and eager DFS B&B. This is in particular true if the pure B&B scheme is supplemented with problem specific efficiency enhancing test for e.g. supplementary exclusion of subspaces, and if the branching performed depends on the value of the current best solution.

4.2 Points for parallel B&B.

First two points which may be said to be general for parallel computing:

- Do not use parallel processing if the problem is too easy - it is not worthwhile the effort. Usually, small speed-up results or even speed-up anomalies with speed-up less than 1.
- Choose the right hardware for your problem (or problem for your hardware, if you are a basic researchers). Trying to beat the best result of others for continuous problems requiring floating point vector calculations using a parallel system best at integers does not make sense.

Regarding parallel B&B, beware of the following major points:

- Centralized control is only feasible in systems with a rather limited number of processors. If a large number of processors are to be used, either total distribution or a combined design strategy should be used.
- If the problem in question has a bound calculation function providing strong bounds, then the number of live subproblems at any time might be small. Then only a few of the processors of a parallel system can be kept busy with useful work at the same time. Here it may be necessary to parallelize also the individual bound calculation to be able to exploit additional processors. This is usually much more difficult indicated by the fact the problem of solving the optimization problem providing the bound is often $\mathcal{P}$-complete, cf. e.g. [15], where parallel algorithms for the assignment problem are investigated.

- If on the other hand the bound calculation gives rise to large search trees in the sequential case, parallel B&B will most likely be a very good solution method. Here static workload distribution may lead to an easily programmable and efficient algorithm if the system used is homogeneous.
- When using dynamic workload distribution, the time spent on programming, testing, and tuning sophisticated methods may not pay off well. Often good results are possible with relatively simple schemes.
- When consulting the literature, be careful when checking test results. A speed-up of 3.75 on a system with 4 processors may at first glance seem convincing, but on the other hand, if almost ideal speed-up cannot be obtained with so few processors, the algorithm will most likely be in severe troubles when the number of processors is increased.

References

[1] A. de Bruin, A. H. G. Rinnooy Kan and H. Trienekens, "A Simulation Tool for the Performance of Parallel Branch and Bound Algorithms", *Math. Prog.* **42** (1988), p. 245 - 271.

[2] J. Clausen and M. Perregaard, "On the Best Search Strategy in Parallel Branch-and-Bound - Best-First-Search vs. Lazy Depth-First-Search", Proceedings of POC96 (1996), also DIKU Report 96/14, 11 p.

[3] J. Clausen, J. L. Träff, "Implementation of parallel Branch-and-Bound algorithms - experiences with the graph partitioning problem", *Annals of Oper. Res.* **33** (1991) 331 - 349.

[4] J. Clausen and J. L. Träff, "Do Inherently Sequential Branch-and-Bound Algorithms Exist ?", *Parallel Processing Letters* **4, 1-2** (1994), p. 3 - 13.

[5] J. Clausen and M. Perregaard, "Solving Large Quadratic Assignment Problems in Parallel", DIKU report 1994/22, 14 p., to appear in *Computational Optimization and Applications*.

[6] E. W. Dijkstra, W. H. J. Feijen and A. J. M. van Gasteren, "Derivation of a termination detection algorithm for distributed computations", *Inf. Proc. Lett.* **16** (1983), 217 - 219.

[7] B. Gendron and T. G. Cranic, "Parallel Branch-and-Bound Algorithms: Survey and Synthesis", *Operations Research* **42 (6)** (1994), p. 1042 - 1066.

[8] T. Ibaraki, "Enumerative Approaches to Combinatorial Optimization", *Annals of Operations Research* vol. **10, 11**, J.C.Baltzer 1987.

[9] P. S. Laursen, "Simple approaches to parallel Branch and Bound", *Parallel Computing* **19** (1993), p. 143 - 152.

[10] P. S. Laursen, "Parallel Optimization Algorithms - Efficiency vs. simplicity", Ph.D.-thesis, DIKU-Report 94/31 (1994), Dept. of Comp. Science, Univ. of Copenhagen.

[11] E. L. Lawler, J. K. Lenstra, A. H. G. .Rinnooy Kan, D. B. Shmoys (ed.), "The Travelling Salesman: A Guided Tour of Combinatorial Optimization", John Wiley 1985.

[12] T. Mautor, C. Roucairol, "A new exact algorithm for the solution of quadratic assignment problems", *Discrete Applied Mathematics* **55** (1994) 281-293.

[13] C. Nugent, T. Vollmann, J. Ruml, "An experimental comparison of techniques for the assignment of facilities to locations", *Oper. Res.* **16** (1968), p. 150 - 173.

[14] M. Perregaard and J. Clausen , "Solving Large Job Shop Scheduling Problems in Parallel", DIKU report 94/35, to appear in Annals of OR.

[15] C. Schütt and J. Clausen, "Parallel Algorithms for the Assignment Problem - Experimental Evaluation of Three Distributed Algorithms", *AMS DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **22** (1995), p. 337 - 351.

Exercises

(1) Finish the solution of the biking tourist's problem on Bornholm.

(2) Give an example showing that the branching rule illustrated in Figure 9 may produce nodes in the search tree with non-disjoint sets of feasible solutions. Devise a branching rule, which ensures that all subspaces generated are disjoint.

(3) The asymmetric Travelling Salesman problem is defined exactly as the symmetric problem except that the distance matrix is allowed to be asymmetric. Give a mathematical formulation of the problem.

(4) Devise a B&B algorithm for the asymmetric TSP. Since the symmetric TSP is a special case of the asymmetric TSP, this may also be used to solve

symmetric TSP problems. Solve the biking tourist's problem using your algorithm.

(5) Consider the GPP as described in Example 1. By including the term

$$\lambda \, (\sum_{v \in V} x_v - |V|/2)$$

in the objective function, a relaxed unconstrained problem with modified objective function results for any λ. Prove that the new objective is less than or equal to the original on the set of feasible solutions for any λ. Formulate the problem of finding the optimal value of λ as an optimization problem.

(6) A node in a B&B search tree is called *semi-critical* if the corresponding bound value is less than or equal to the optimal solution of the problem. Prove that if the number of semi-critical nodes in the search tree corresponding to a B&B algorithm for a given problem is polynomially bounded, then the problem belongs to $\mathcal{P}$.

Prove that this holds also with the weaker condition that the number of critical nodes is polynomially bounded.

(7) Consider again the QAP as described in Example 3. The simplest bound calculation scheme is described in Section 2.1. A more advanced, though still simple, scheme is the following:

Consider now partial solution in which m of the facilities has been assigned to m of the locations. The total cost of any feasible solution in the subspace determined by a partial solution consists of three terms: costs for pairs of assigned facilities, costs for pairs consisting of one assigned and one unassigned facility, and costs for pairs of two unassigned facilities. The first term can be calculated exactly. Bounds for each of the two other terms can be found based on the fact that a lower bound for a scalar product $(a_1, ..., a_p) \cdot (b_{\pi(1)}, ..., b_{\pi(p)})$, where a and b are given vectors of dimension p and π is a permutation of $\{1, ..., p\}$, is obtained by multiplying the largest element in a with the smallest elements in b, the next-largest in a with the next-smallest in b etc.

For each assigned facility, the flows to unassigned facilities are ordered decreasingly and the distances from the location of the facility to the remaining free locations are ordered increasingly. The scalar product is now a lower bound for the communication cost from the facility to the remaining unassigned facilities.

The total transportation cost between unassigned facilities can be bounded in a similar fashion.

(a)

Consider the instance given in Figure 6. Find the optimal solution to the instance using the bounding method described above.

(b)

Consider now the QAP, where the distances between locations are given as the rectangular distances in the following grid:

1	2	3
4	5	6

The flows between pairs of facilities are given by

$$F = \begin{pmatrix} 0 & 20 & 0 & 15 & 0 & 1 \\ 20 & 0 & 20 & 0 & 30 & 2 \\ 0 & 20 & 0 & 2 & 0 & 10 \\ 15 & 0 & 2 & 0 & 15 & 2 \\ 0 & 30 & 0 & 15 & 0 & 30 \\ 1 & 2 & 10 & 2 & 30 & 0 \end{pmatrix}$$

Solve the problem using B&B with the bounding function described above, the branching strategy described in text, and DFS as search strategy.

To generate a first incumbent, any feasible solution can be used. Try prior to the B&B execution to identify a good feasible solution. A solution with value 314 exists.