An object oriented approach to generic branch and bound

A. de Bruin, G.A.P. Kindervater, H.W.J.M. Trienekens,
R.A. van der Goot, W. van Ginkel

Abstract

Branch and bound algorithms can be characterized by a small set of basic rules that are applied in a divide-and-conquer-like framework. The framework is about the same in all applications, whereas the specification of the rules is problem dependent. Building a framework is a rather simple task in sequential implementations, but must not be underestimated in the parallel case, especially if an efficient branch and bound algorithm is required. In generic branch and bound models, the basic rules can be clearly identified within the framework, and, hence, it can be developed independently from the application. Furthermore, it gives the user the opportunity to concentrate on the actual problem to be solved, without being distracted by user-irrelevant issues like the properties of the underlying architecture. In this paper, we will discuss an object oriented approach to generic branch and bound. We will show how object orientation can help us to build a flexible branch and bound framework, that is able to perform like any branch and bound algorithm that fits into some powerful taxonomies known from the literature. We will define an interface for the specification of the problem dependent parts, and we will give a first indication of how the user can tune the framework if a non-default behavior is desired.

1 Introduction

Branch and bound algorithms solve optimization problems by applying a small set of basic rules within a divide-and-conquer-like framework. These algorithms generate search trees, in which each node corresponds to a subset of the feasible solution set. A subproblem associated with a node is either solved directly, or its solution set is split, and for each subset thus generated a new node is added to the tree. The process is improved by computing a bound on the solution value a node can produce. If the bound is worse than the value of the best solution found so far, the node cannot produce a better solution, and, hence, it can be excluded from further examination. The order in which the nodes are selected for evaluation may be arbitrary, but a well-chosen order (e.g., depth first or best bound) will generally reduce the computational effort considerably.

Report EUR-FEW-CS-96-10
Erasmus University, Department of Computer Science
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

In the description of the algorithm, we can clearly identify four basic rules, as there are: the *branching* rule for the decomposition of nodes, the *bounding* rule for bound computations, the *selection* rule for choosing the next node to be evaluated, and the *elimination* rule for excluding nodes from possible evaluation.

On sequential computers, the specification of the rules makes up most of the work to obtain a useful algorithm. Moreover, a better structural understanding of the problem to be solved, i.e., a sharper specification of the basic rules, almost immediately leads to shorter computation times and larger tractable instances. To obtain an efficient parallel branch and bound algorithm, insight into the problem itself is no longer sufficient. The performance of implementations highly depends on the target architecture, and even on the search tree (to be) generated. This situation is highly undesirable from a user point of view. Instead of just coping with the problem itself, the user has to take nontrivial decisions concerning external matters.

Presently, several parallel branch and bound systems exist that alleviate the actual coding of an algorithm substantially. However, many of these systems are tailored to a special type of parallel implementation with a specific underlying architecture in mind. Furthermore, in some of the existing systems, there is no clear separation of the problem part from the general branch and bound part, i.e., the user has to have knowledge of the system itself. For instance, in the BOB system from Versailles [Le Cun & Roucairol, 1995], the user has to know the internal representation of the nodes, whereas in the PPBB library from Paderborn [Tschöke & Polzer, 1995], the user must be aware of the presence of a 'main' process.

This paper will deal with the construction of an object oriented branch and bound system that overcomes the above mentioned problems. We will start from a generic branch and bound approach, which constitutes an abstract branch and bound algorithm for an arbitrary (unspecified) problem. The abstract algorithm can be turned into a concrete one by filling in the problem specific parts. The approach can be modeled in object oriented programming languages, like C++, using polymorphism and inheritance. Our system is still under construction, but we will present a preliminary version that is operational on sequential computers. We will describe the system's interface (i.e., the parts to be filled in by the user), as well as the ways to alter the system's default behavior. We will argue that inheritance is a powerful concept to develop a flexible system, that makes it possible to solve different types of problems on various architectures efficiently.

The paper is organized as follows. In Section 2, generic branch and bound is introduced. Section 3 explains object orientation, and how it can be applied to implement generic branch and bound. This results in the definition of an object oriented generic branch and bound model. In Section 4, the model is made concrete by describing its syntax and semantics. The main advantage of the object oriented approach to generic branch and bound, its flexibility, is illuminated in Section 5. Finally, we present the conclusions and discuss future extensions to the object oriented generic branch and bound model.

2 Generic branch and bound

Based on the observation that branch and bound algorithms can be described in a problem independent way, several *models* have been developed [Lawler & Wood, 1966; Mitten, 1970; Ibaraki, 1976, 1977a, 1977b; Kumar & Kanal, 1983, and Nau, Kumar & Kanal, 1984]. A branch and bound model specifies a high-level abstract algorithm operating on abstract data types like 'subproblem', for which operators corresponding to the basic rules, such as 'branch', are defined. If in such a model the general branch and bound part can be clearly separated from the problem part, the model can be made practical. This approach is called *generic branch and bound*.

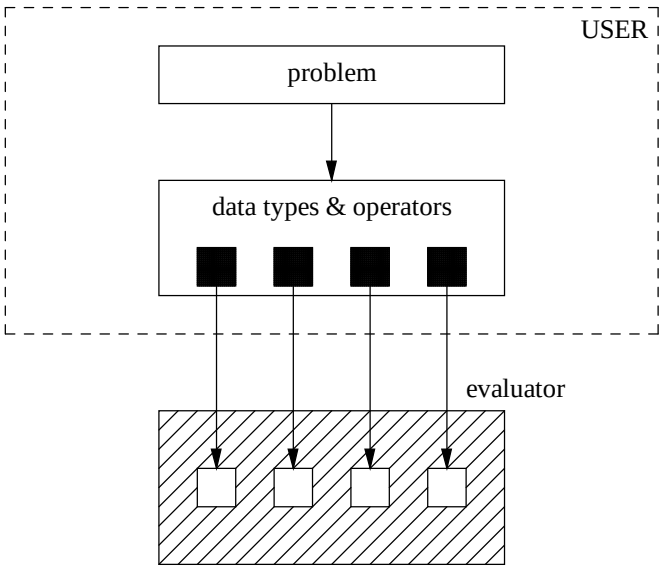

Figure 1: Generic branch and bound.

The implementation of the abstract algorithm results in a so called *branch and bound evaluator*, an incomplete branch and bound algorithm suited for general applications. In order to turn the abstract algorithm of the evaluator into a concrete one, the problem specific parts must be added, i.e., the implementation of the abstract data types and the associated high-level operators (cf. Figure 1).

Notice that the generic branch and bound approach defines a branch and bound algorithm with 'holes' in it, that should be filled with user defined problem specific parts. It contrasts the 'library' approach (see Figure 2), in which the user, in addition to the specification of the problem specific parts, has to program the solution process, thereby using library routines for the relief from low-level, often machine dependent, programming chores. Both approaches have been tested in practice. One of the most elaborate examples in the category of generic systems is the work done in East-Anglia [McKeown, Rayward-Smith & Turpin, 1991]. Belonging to the other class are, for instance,

the libraries developed in Paderborn (PPBB-Lib) [Tschöke & Polzer, 1995], and Versailles (BOB) [Le Cun & Roucairol, 1995].

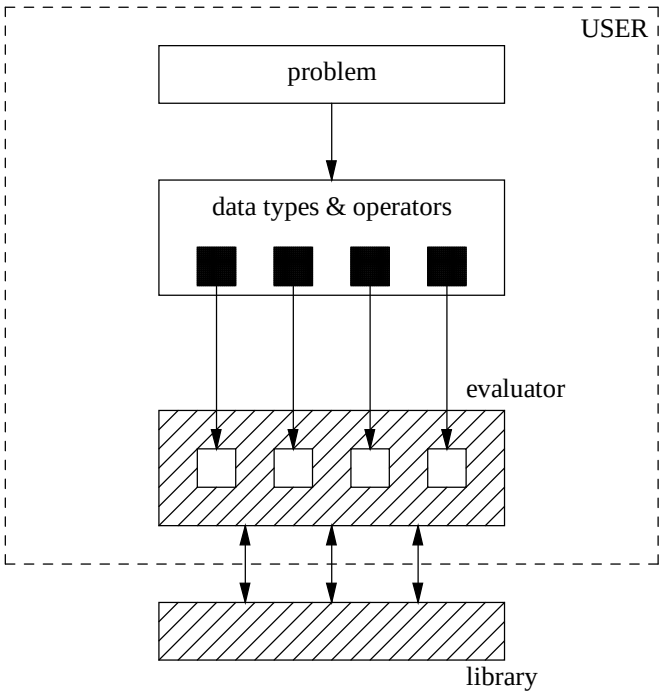

Figure 2: Library approach.

In the design of a branch and bound evaluator, an important parameter is the description of the holes to be filled in by the user, i.e., the interface of the system. On the one hand, the interface must enable the user to describe arbitrary problems without knowing too much of the solution process itself. On the other hand, the interface must allow for a broad class of different settings of the evaluator in order to obtain efficient algorithms in many situations. In the words of Jonathan Eckstein [1996]: "The holes may not be too big, but not too small either."

In the next sections, we discuss the implementation of generic branch and bound in an object oriented environment. We will present a preliminary version of a sequential branch and bound evaluator, and show how it can be extended to an evaluator that is robust, easy to use, and flexible enough to accommodate a broad range of different branch and bound implementations on parallel architectures of different types.

3 Object oriented generic branch and bound

The model described in the previous section can be implemented in an easy way using objects and polymorphism. Each hole in the branch and bound evaluator corresponds

to a particular operation to be performed by some object. In order for the evaluator and the objects to be able to cooperate, their interface has to be defined precisely. The polymorphism enables the evaluator to manipulate an object knowing only its interface, not its implementation.

One way of implementing the above is by using inheritance and abstract base classes in C++ [ISO Working Group WG21, 1995]. We opt for C++, because of its wide availability and efficiency.

An *abstract base class* is a class that defines an interface (a set of operations) through which objects of this class can be manipulated, without providing a complete implementation of the interface. Consequently, it is impossible to create objects in such a class. However, the user can derive a new class from the abstract base class through inheritance, and supply the missing implementation in the derived class. The derived class inherits the interface from the base class, and may be given its own (partially new) implementation of the operations. Since the interface is inherited, objects of the derived class can be manipulated as if they were objects of the base class.

In our implementation of the generic branch and bound model, three classes are used: PROBLEM, SOLUTION, and SOLVER.

Class PROBLEM is an abstract base class, objects of which represent nodes in the search tree (corresponding to subsets of the feasible solution set of a problem to be solved). The class has an interface related to the basic branch and bound rules. It contains, for example, operations to compute its bound, to compute its priority, to generate its children, and to compare itself with another node. The description of the class PROBLEM is independent of the actual problem to be solved. Some of the operations in the interface can be given a default behavior (e.g., the priorities are computed in such a way that the tree is searched in a depth first order), but others (e.g., the generation of the children) cannot be implemented without knowing the actual problem to be solved. Hence class PROBLEM is abstract. To keep the discussion simple, we assume in the following that none of the operations has a default behavior.

Class SOLUTION is also an abstract base class, representing feasible solutions to the problem to be solved. Its interface consists, amongst others, of operations to print itself, and to compare itself with other solutions. Again, as the actual problem to be solved is unknown, class SOLUTION is abstract.

Next, we will describe the class SOLVER. An object of the class SOLVER (also called *solver* for short) produces the solution to a problem by manipulating objects of type PROBLEM and SOLUTION. This is achieved by repeatedly activating operations of the objects currently present. The classes PROBLEM, SOLUTION, and SOLVER constitute an abstract branch and bound algorithm to solve arbitrary problems, i.e., they form a branch and bound evaluator, where the holes are defined by the (unimplemented) interface of PROBLEM and SOLUTION. Note that only the interfaces (and not the complete definition) of the classes PROBLEM and SOLUTION have to be known for a complete implementation of the class SOLVER. In other words, SOLVER is not abstract.

In the above description the classes PROBLEM and SOLUTION are abstract base classes. To solve an actual problem, the user has to define classes MY_PROBLEM and MY_SOLUTION, derived by inheritance from the corresponding base classes PROBLEM and SOLUTION. The new classes redefine the original ones in the sense that they contain data fields describing the nodes and an implementation of the interface (i.e., the

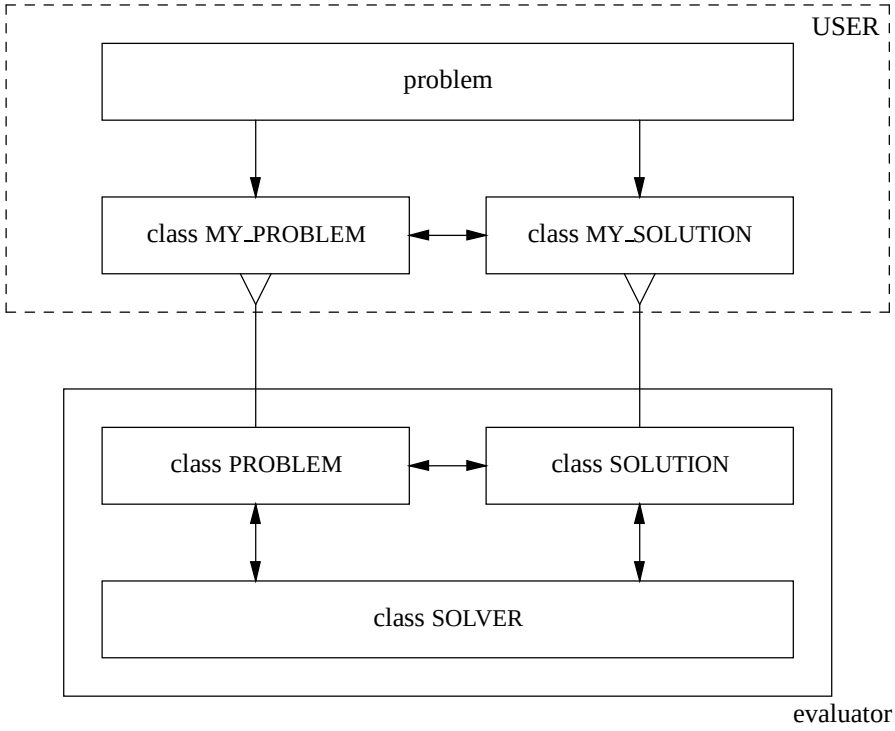

Figure 3: Object oriented generic branch and bound.

holes in the evaluator are filled), all with respect to the actual problem to be solved. Since MY_PROBLEM and MY_SOLUTION are special cases of PROBLEM and SOLUTION with the same interface, SOLVER can manipulate objects of the derived classes MY_PROBLEM and MY_SOLUTION as if they were objects of the original base classes. Having derived these two classes, the user solves an instance of the problem by instantiating the root node of the search tree (an object of class MY_PROBLEM) and by creating an initial solution. Both objects must be fed into the solver (an object of class SOLVER, which has to be created as well). The new situation is depicted in Figure 3.

Note that the inheritance mechanism, which is used for filling the holes, is also convenient if a user wants to experiment with different versions of the algorithm, in which (some of) the basic rules are implemented in a different way. By inheritance from the classes MY_PROBLEM and MY_SOLUTION, the user can redefine the behavior wherever wanted, without changing the original implementation.

4 The interface

In this section, we will present the abstract base classes PROBLEM and SOLUTION. We will describe both the syntax and the semantics of their interfaces in detail. Then, we will show the actions a user has to undertake to solve a real problem. For the understanding of the class definitions, we only assume some intuitive knowledge of C++. Details are of no importance here, and will not be dealt with.

Base class PROBLEM

As said before, class PROBLEM represents the nodes in the search tree. Its interface mainly consists of the basic branch and bound rules. We start by giving the syntax, and then explain each of the operations in more detail.

```
class PROBLEM
{
  public:
    virtual void compute_bound (const SOLUTION& current_best_solution) = 0;
    virtual void compute_priority (const SOLUTION& current_best_solution) = 0;
    virtual PROBLEM* next_child (const SOLUTION& current_best_solution) = 0;
    virtual int is_leaf () = 0;
    virtual SOLUTION* give_solution () = 0;
    virtual int compare_priority (const PROBLEM& problem) = 0;
    virtual ~PROBLEM () = 0;
};
```

The semantics of the operations is explained below.

compute_bound. Operation compute_bound is meant to compute a bound to the solution the node is able to produce. For each node, it is activated only once by the system.

compute_priority. This operation is meant to compute a priority of the node, that can be used in the selection process. For each node, it is activated only once by the system.

next_child. Each time next_child is called, a newly created child of the node should be returned, and zero if all children have been generated already.

is_leaf. Function is_leaf should return a nonzero value if the node does not have to be branched from anymore (is a leaf in the search tree), and zero otherwise. The function should be able to do so by inspecting the internal state of the node. For each node, it is activated only once by the system.

give_solution. During each of the above operations, a feasible solution to the original problem may be found as a sort of byproduct. To assure that the new solution is not getting lost, give_solution is activated after each activation of the above operations. If a solution has been detected, give_solution should return this solution, and zero otherwise.

compare_priority. Function compare_priority should compare the current node with the one given as an argument. It should return a negative value if the current node

is to be preferred over the other one, a positive value if it is the other way around, and zero if there is no preference.

~**PROBLEM**. The destructor of class PROBLEM. It removes objects of its class, when they are no longer used.

In the first three operations of PROBLEM (compute_bound, compute_priority, and next_child), the current best solution is passed as an argument. In many situations, the current best solution will not be used in these operations. However, Volgenant & Jonker [1982] use the current best solution value in the bound computation, and Lageweg, Lenstra & Rinnooy Kan [1977] use the solution itself in the selection process. Although such branch and bound algorithms do not seem to occur very often, we still want to be able to execute them using our branch and bound evaluator.

The choice of passing the current solution as an argument in some of the operations is a rather ad hoc one. Perhaps a different mechanism to provide operations with external information may turn out to be more advantageous, and ill perhaps be adopted in the future.

Base class SOLUTION

As in the previous subsection, we give the description of SOLUTION first, and explain it in more detail later on.

```
class SOLUTION
{
  public:
    virtual void print () = 0;
    virtual int compare_solution (const SOLUTION& solution) = 0;
    virtual int compare_bound (const PROBLEM& problem) = 0;
    virtual ~SOLUTION () = 0;
};
```

The semantics of the operations are explained below.

print. Operation print is meant to show the solution.

compare_solution. Function compare_solution should compare the current solution with the one given as an argument. It should return a negative value if the current solution is to be preferred over the other one, a positive value if it is the other way around, and zero if there is no preference.

compare_bound. This function should compare the current solution with the node given as an argument. It should return a negative value if the current solution is better than any solution possibly produced by the node (the bound of the node is worse than the value of the current solution), a positive value if it is the other way around, and zero if there is no decisive answer.

~**SOLUTION**. The destructor of class SOLUTION.

Solving a problem

If the user wants to solve an actual problem, new classes derived from the abstract classes PROBLEM and SOLUTION have to be defined. In this way, we obtain the classes MY_PROBLEM and MY_SOLUTION. The derived classes have to provide the data describing the nodes (MY_PROBLEM) and solutions (MY_SOLUTION), as well as a complete implementation of the operations in the interface. Furthermore, constructors have to be added to enable the initialization of the branch and bound process. We start with the class MY_PROBLEM.

```
class MY_PROBLEM : public PROBLEM
{
  public:
    virtual void compute_bound (const SOLUTION& current_best_solution);
    virtual void compute_priority (const SOLUTION& current_best_solution);
    virtual PROBLEM* next_child (const SOLUTION& current_best_solution);
    virtual int is_leaf ();
    virtual SOLUTION* give_solution ();
    virtual int compare_priority (const PROBLEM& problem);
    virtual ~MY_PROBLEM ();
    MY_PROBLEM ();                    // Default constructor.
  protected:
    // Data and operations with respect to the problem to be solved.
};
```

In the same way, we obtain the class MY_SOLUTION

```
class MY_SOLUTION : public SOLUTION
{
  public:
    virtual void print ();
    virtual int compare_solution (const SOLUTION& solution);
    virtual int compare_bound (const PROBLEM& problem);
    virtual ~MY_SOLUTION ();
    MY_SOLUTION ();                   // Default constructor.
  protected:
    // Data and operations with respect to the problem to be solved.
};
```

After the definition and implementation of the above classes, the user has to create the root node containing the data of the instance to be solved, and an initial solution. These are then given to a solver, which in turn produces the solution of the instance.

As a last remark in this section, we mention that the interfaces of the classes MY_PROBLEM and MY_SOLUTION only deal with nodes and solutions as an atomic entity. Their internal structure is irrelevant. As a consequence, the user is completely free whether or not to store the bound and the priority of the node explicitly. The only thing

to be guaranteed is that operations like compare_priority never fail, and that next_child, for example, really delivers a not yet generated child. A disadvantage may be that some bookkeeping is left to the user.

5 Towards flexibility

We now turn to the solver part of the evaluator. Unfortunately, the basic rules do not imply a unique branch and bound algorithm. For instance, both sequential and parallel branch and bound algorithms keep track of the set of nodes that still have to be considered for further examination (the *active set*). In most algorithms, the bound on the optimal solution a node can produce, has been computed already for each of the nodes in the active set. There exist branch and bound algorithms, however, in which the bound of a nodes in the active set is the (sometimes slightly improved) bound of its direct ancestor, and the node's own bound is computed just before it is branched. An example hereof is Carlier's algorithm for the single-machine scheduling problem [Carlier, 1982]. The above boils down to different orders in which the basic rules are applied. This phenomenon can be observed even more in the parallel case, where a broad variety of different implementations exists (see, for instance, De Bruin, Kindervater & Trienekens [1996]).

As each implementation is efficient in specific situations, a generic solver has to be flexible to accommodate as many implementations as possible. One approach is to extend the solver with *controls* that may be applied by a user in order to realize the desired behavior of the solver. Another option is to use inheritance by defining a default solver and a series of derived actual solvers. The latter approach has been successfully undertaken in a library for solving search problems [Bouthoorn, 1993]. As we feel that it offers more opportunities, our system will be built along these lines.

The idea is to use the class SOLVER as a base class, and to derive a set of new classes through inheritance (cf. Figure 4). In the derived classes, operations are redefined in order to obtain a different branch and bound algorithm. Now, the user can make a choice from different branch and bound solvers, each with their own specific behavior. In case this is insufficient, the user can even define his own class MY_SOLVER from any supplied SOLVER-class using inheritance.

Presently, class SOLVER contains one function, that completely models a (sequential) branch and bound algorithm in which the active set contains nodes with both their bound and priority computed. We intend to adopt a model from Miller & Pekny [1989], which relates to the discussion above. Miller & Pekny maintain two sets of nodes. In the first set, the nodes do not have their bound computed yet, in the other they do. Extending this idea, we obtain a model in which we have nodes with attributes (such as a bound), where each basic rule determines one of the attributes. Our solver will consist of a series of node sets, where the basic rules move nodes from one predetermined set to another predetermined set, while computing the attribute in question.

For the implementation of the model, the first thing to do is to identify the basic operations in the solver, such that they can be easily redefined in derived classes and at the same time offer the right amount of flexibility.

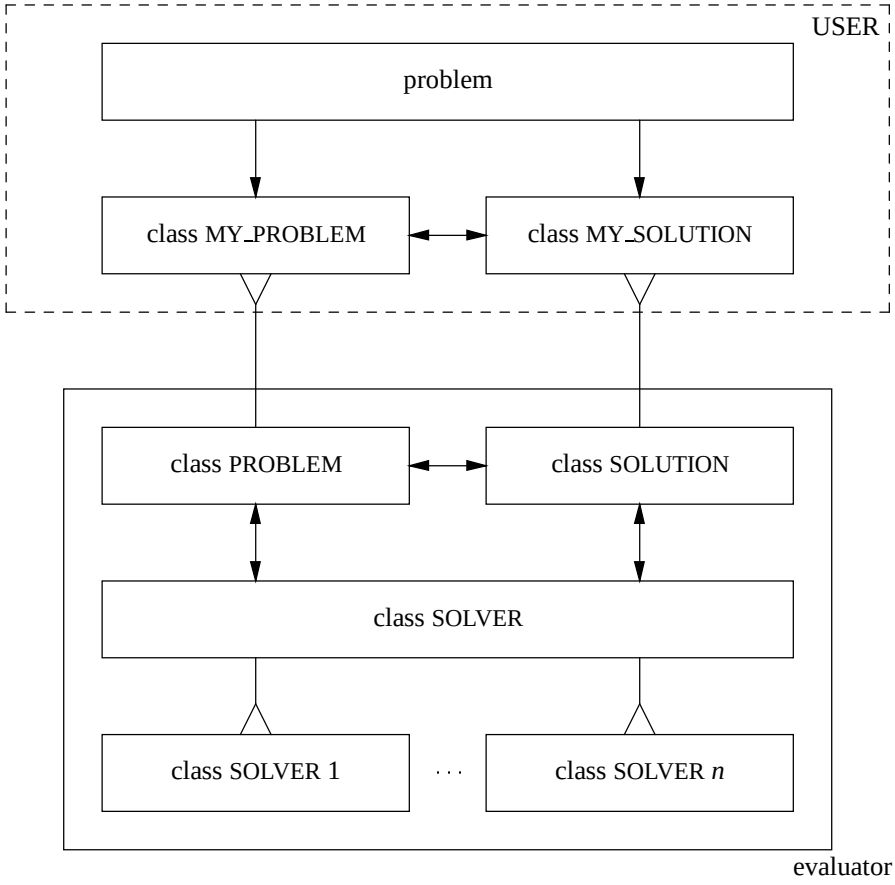

Figure 4: A flexible SOLVER.

6 Summary and future work

In the previous sections, we have described the development of a flexible object oriented branch and bound system. The object oriented approach has many advantages. Especially, it effectuates a clear separation of the problem specific parts and the branch and bound parts, where the linking between the actual problem and the abstract algorithm is done automatically through inheritance. Hence, a user may focus his attention completely on the problem itself.

We have limited experience with our system. Right now, it implements a single instance of branch and bound, and is mainly used to test the interface. Whether the interface is practical, can only be decided upon after the implementation of more branch and bound algorithms.

Future expansions to the system will deal with the solver part. We are currently working on the Miller & Pekny idea, described in the previous section. Our ultimate goal is to build a (parallel) generic branch and bound machine, that is robust, easy to use, and flexible enough to efficiently accommodate a broad range of different branch and bound implementations on parallel architectures of different types.

Acknowledgements

We thank the members of the 'SCOOP'-community for their contribution in the discussions on this topic. Special thanks go to the groups from Versailles and Paderborn for giving us the opportunity to work with their branch and bound libraries.

References

[1] P.M. Bouthoorn (1993). *C++ Search Class Library*, Technical Report, ICCE, University of Groningen.

[2] J. Carlier (1982). The one-machine sequencing problem. *European J. Oper. Res. 11*, 42–47.

[3] A. de Bruin, G.A.P. Kindervater, H.W.J.M. Trienekens (1996). Towards an abstract parallel branch and bound machine. A. Ferreira, P. Pardalos (eds.). *Solving Combinatorial Optimization Problems in Parallel: Methods and Techniques*, Lecture Notes in Computer Science 1054, Springer, Berlin, 145–170.

[4] J. Eckstein (1996). Quotation from a discussion meeting at POC 96, Versailles.

[5] T. Ibaraki (1976). Theoretical comparisons of search strategies in branch-and-bound algorithms. *Int. J. Comput. Inform. Sci. 5*, 315–344.

[6] T. Ibaraki (1977a). On the computational efficiency of branch-and-bound algorithms. *J. Oper. Res. Soc. Japan 20*, 16–35.

[7] T. Ibaraki (1977b). The power of dominance relations in branch-and-bound algorithms. *J. Assoc. Comput. Mach. 24*, 264–279.

[8] ISO Working Group WG21 (1995). *Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++*, American National Standards Institute (ANSI), Washington DC.

[9] V. Kumar, L.N. Kanal (1983). A general branch and bound formulation for understanding and synthesizing And/Or tree search procedures. *Art. Intelligence 21*, 179–198.

[10] B.J. Lageweg, J.K. Lenstra, A.H.G. Rinnooy Kan (1977). Job-shop scheduling by implicit enumeration. *Management Sci. 24*, 441-450.

[11] E.L. Lawler, D.E. Wood (1966). Branch-and-bound methods: a survey. *Oper. Res. 14*, 699–719.

[12] B. Le Cun, C. Roucairol, (1995). *BOB: a Unified Platform for Implementing Branch-and-Bound like Algorithms*, Research Report 95/16, PR*i*SM Laboratory, University of Versailles - St. Quentin en Yvelines.

[13] G.P. McKeown, V.J. Rayward-Smith, H.J. Turpin (1991). Branch-and-bound as a higher-order function. *Ann. Oper. Res. 33*, 379–402.

[14] D.L. Miller, J.F. Pekny (1989). Results from a parallel branch and bound algorithm for the asymmetric traveling salesman problem. *Oper. Res. Lett. 8*, 129–135.

[15] L.G. Mitten (1970). Branch-and-bound methods: general formulation and properties. *Oper. Res. 18*, 24–34.

[16] D.S. Nau, V. Kumar, L.N. Kanal (1984). General branch and bound and its relation to A* and AO*. *Art. Intelligence 23*, 29–58.

[17] S. Tschöke, T. Polzer (1995). *A Portable Parallel Branch-and-Bound Library (PPBB-Lib)*, Technical Report, Department of Computer Science, University of Paderborn.

[18] A. Volgenant, R. Jonker (1982). A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation. *European J. Oper. Res. 9*, 83–89.

This work was partially supported by the Human Capital and Mobility project SCOOP — Solving Combinatorial Optimization Problems in Parallel — of the European Union.