AN OBJECT-ORIENTED FRAMEWORK FOR LOCAL SEARCH HEURISTICS

Alexandre A. Andreatta **Sérgio E.R. Carvalho** **Celso C. Ribeiro**

Department of Computer Science
Catholic University of Rio de Janeiro
R. Marquês de São Vicente 225
Rio de Janeiro 22453, Brazil
E-mail: {andreatt,sergio,celso}@inf.puc-rio.br

Abstract: In the study of heuristics for combinatorial problems, it is often important to develop and compare, in a systematic way, different algorithms, strategies, and parameters for the same problem. This comparison is often biased not only by different implementation languages, but also by different architectures. This paper proposes a framework, described using design patterns, modeling different aspects involved in local search heuristics, such as algorithms for the construction of initial solutions, methods for neighborhood generation, and movement selection criteria. Using this framework we fix a basic architecture and thus increase our ability to construct and compare heuristics.

1. Introduction

Hard combinatorial optimization problems most often have to be solved by approximate methods. Constructive methods build up feasible solutions from scratch. Among them we find the so-called greedy algorithms, based on a preliminary ordering of the solution elements according to their cost values. Basic local search methods are based on the evaluation of the neighborhood of successive improving solutions, until no further improvements are possible. As an attempt to escape from local optima, some methods allow controlled deterioration of the solutions in order to diversify the search.

In the study of heuristics for combinatorial problems, it is often important to develop and compare, in a systematic way, different algorithms, strategies, and parameters for the same problem. The central issue in this paper is that, by encapsulating, refining and relating the different concerns involved in local search, we increase our ability to construct and compare heuristics, independently of their implementations.

We use the object orientation paradigm, and provide an architectural basis for the implementation and comparison of different local search heuristics. The **NeighborSearcher** framework encapsulates, in abstract classes, different aspects involved in local search heuristics, such as algorithms for the construction of initial solutions, methods for neighborhood generation, and movement selection criteria. Encapsulation and abstraction promote unbiased comparisons between different heuristics, code reuse, and easy extensions.

This paper is organized as follows. In the next section, we introduce the main aspects and relevant issues concerning local search and metaheuristics based on local search, and shortly describe some of the most important and/or frequently used algorithms. In section 3 we present the **NeighborSearcher** framework, using a small variation of the pattern description format defined in [11]. In section 4 we describe implementation issues, including the design patterns used in framework construction and accompanying type hierarchies. In section 5 we present some concluding remarks concerning the main issues involved in this work, the most relevant consequences of the use of the proposed methodology for the construction of this framework, as well as the description of some extensions, applications and work-in-progress.

2. Local Search and Metaheuristics

Local search methods for the solution of combinatorial optimization problems are a class of approximate algorithms based on the systematic exploration of neighborhoods of successive solutions. Starting at some initial feasible solution, they are based on the idea that a solution may be improved through atomic changes, until no further improvements are possible. Accordingly, neighbor solutions are obtained from the current one by the application of an adjacency relation defined over the set $\boldsymbol{X}$ of feasible (and also unfeasible, in some cases) solutions. Then, the neighborhood $\boldsymbol{N}(\boldsymbol{x})$ of a given solution $\boldsymbol{x}$ is a mapping $\boldsymbol{x} \rightarrow \boldsymbol{N}(\boldsymbol{x}) \subseteq \boldsymbol{X}$. This concept of neighborhood is the central issue to all algorithms in this class of local search methods.

The simplest local search algorithm is *Iterative Improvement*. The neighborhood of the current solution is searched for a lower cost solution. If such a solution is found, then it will become the new, improved current solution; otherwise the algorithm stops. In that case, a local optimum has been found. The *Steepest Descent* algorithm is a variant of this scheme. Here, instead of just picking any lower cost solution in the neighborhood, we choose the one with the smallest cost within the neighborhood.

Variable Neighborhood Search is another recent extension of these approaches. Here, instead of using only one neighborhood definition, the current neighborhood is enlarged whenever a local optimum is found with respect to it. Starting with small neighborhood definitions may speed up the search, while the progressive neighborhood enlargement allows finding a solution within some controlled quality level satisfying some desired termination condition.

The use of several common ideas such as fast techniques for repeatedly constructing initial solutions, solution improvement by local search, controlled randomization, search diversification when no further improvements seem possible, and search intensification into promising regions of the solution space, among others, has recently led to the development of new classes of heuristics for the solution of combinatorial optimization problems, which considerably improve the efficiency of approximate algorithms and the size of solvable instances of difficult problems found in practice. Among them, we may cite Simulated Annealing, Tabu Search, Greedy Randomized Adaptive Search Procedures, Scatter Search, Genetic Algorithms, and Ant Colonies, among others. These methods are usually called *metaheuristics*, in the sense that the process of finding a good solution (eventually optimal) consists in applying at each step a subordinate heuristic, which has to be designed for each particular problem. As an attempt to escape from local optima, some of them allow controlled deterioration of the solutions, in order to diversify the search. We illustrate these ideas below, through the presentation of two techniques in this class.

Tabu Search is an adaptive procedure for solving combinatorial optimization problems, which guides an improvement heuristic to continue exploration without being confounded by an absence of improving movements, and without falling back into a local optimum from which it previously emerged (see e.g. Glover [12,13] and Glover and Laguna [14]). Tabu search goes beyond local search by employing a strategy for modifying the neighborhood of the current solution as the search progresses, effectively replacing it by another neighborhood. Depending on the search stage, this modified neighborhood may be a restriction or an extension of the regular neighborhood. As a consequence, and contrary to the basic local search scheme, moves towards solutions that deteriorate the cost function are permitted under certain conditions.

A *Greedy Randomized Adaptive Search Procedure* (GRASP) is an iterative process, where each GRASP iteration consists of two phases: construction and local search [7]. The construction phase builds a feasible solution, whose neighborhood is explored by local search. The best solution over all iterations is retained. In the construction phase, a feasible solution is built, one element at a time. At each construction iteration, the next element to be added is determined by ordering all elements in a candidate list with respect to a greedy function that measures the benefit of selecting each element. The adaptive component of the heuristic arises from the fact that the benefits associated with every element are updated at each iteration of the construction phase, to reflect the changes brought on by the selection of the previous element. The probabilistic component of a GRASP is characterized by randomly choosing one of the best candidates in the list, but usually not the top candidate. This way of making the choice allows for different solutions to be obtained at each iteration, but does not necessarily jeopardize the power of its adaptive greedy component [15].

As we have seen above, many concepts and ideas are common to most local search based methods and metaheuristics. Among them, we may cite algorithms for construction of initial solutions, neighborhood definitions, move selection criteria, local search procedures, and termination criteria. This is the main motivation for the construction of **NeighborSearcher**.

3. The NeighborSearcher Framework

To describe the framework, we use an adaptation of the pattern language employed in [11], providing Intent, Motivation, Applicability, Structure, Participants, and Collaborations. We discuss implementation issues in the next section.

Intent.
To provide an architectural basis for the implementation and comparison of different local search strategies.

Motivation.
In the study of heuristics for combinatorial problems, it is important to develop and compare, in a systematic way, different heuristics for the same problem. It is frequently the case that the best strategy for a specific problem is not a good strategy for another. It follows that, for a given problem, it is often necessary to experiment with different heuristics, using different strategies and parameters.

Modeling in separate classes the different concerns involved in local search, and relating these classes in a framework, we increase our ability to construct and compare heuristics, independently of their implementations. Implementations can easily affect the performance of a new heuristic, for example due to programming language, compiler, and other platform aspects.

As pointed by Barr et al. [4], although no set standard exists for publishable algorithmic research, it is generally accepted that a heuristic method is a relevant contribution if:

- it produces high quality solutions faster than others;
- it identifies better quality solutions;
- it is less sensible to tuning of parameters and to problem variations;
- it is easier to implement;
- it is faster and more precise in the solution of a new or important problem;
- it is applicable to a large family of problems; or
- it is new and creative in its own right.

Our framework addresses several issues listed above, as it will be seen in what follows. The classification of different aspects of heuristics simplifies their implementations and invites new extensions, in the form of subclasses. The reuse of code that we can accomplish, implementing common aspects of a family of heuristics, allows us a better platform for comparison, since a large part of the code is common to all members of the family.

Applicability.
The **NeighborSearcher** framework can be used in situations involving:

- Local search strategies that can use different methods for the construction of the initial solution, different neighborhood relations, or different movement selection criteria.
- Construction algorithms that utilize subordinate local search heuristics.
- Local search heuristics with dynamic neighborhood models.

Structure.
Figure 1 below shows the classes and relationships involved in the **NeighborSearcher** framework. We use OMT notation [16,17] for static views.

Participants.

- *Client:* contains the combinatorial problem instance to be solved, its initial data and the pre-processing methods to be applied. Contains also the data for creating the *SearchStrategy* that will be used to solve the problem. Generally, it can have methods for processing the solutions obtained by the *SearchStrategy*.
- *Solution:* encapsulates the representation of a solution for the combinatorial problem. Defines the interface the algorithms have to use to construct and to modify a solution. Delegates to *IncrementModel* or to *MovementModel* requests to modify the current solution.
- *SearchStrategy:* constructs and starts the *BuildStrategy* and the *LocalSearch* algorithms, also handling their intercommunication, in case it exists.
- *BuildStrategy:* encapsulates constructive algorithms in concrete subclasses. Investigates and eventually requests to *Solution* modifications in the current solution, based on an *IncrementModel*.

- *LocalSearch:* encapsulates local search algorithms in concrete subclasses. Investigates and eventually requests to *Solution* modifications in the current solution, based on a *MovementModel.*
- *Increment:* groups the data necessary to an atomic modification of the internal representation of a solution for constructive algorithms.
- *Movement:* groups the data necessary to an atomic modification of the internal representation of a solution for local search algorithms.
- *IncrementModel:* modifies a solution according to a *BuildStrategy* request.
- *MovementModel:* modifies a solution according to a *LocalSearch* request.

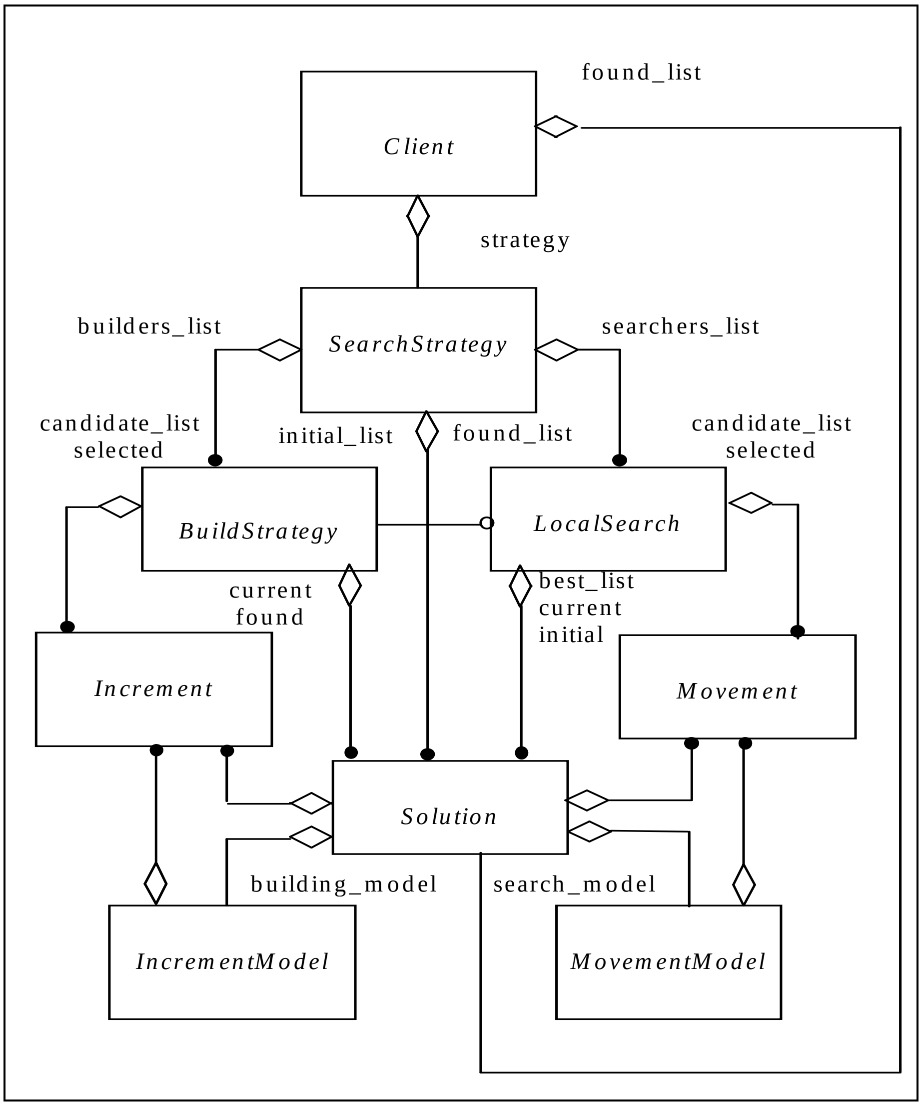

Figure 1: The **NeighborSearcher** framework

Collaborations.
The *Client* wants a *Solution* for a problem instance. It delegates this task to its *SearchStrategy*, which is composed by at least one *BuildStrategy* and one *LocalSearch.* The *BuildStrategy* produces an initial *Solution* and the *LocalSearch* improves the initial *Solution* through successive movements.

The *BuildStrategy* and the *LocalSearch* perform their tasks based on neighborhood relations provided by the *Client*.

The implementation of these neighborhoods is delegated by the *Solution* to its *IncrementModel* (related to the *BuildStrategy*) and to its *MovementModel* (related to the *LocalSearch*). The *IncrementModel* and the *MovementModel* are the objects that will obtain the *Increment*s or the *Movement*s necessary to modify the *Solution* (under construction or not).

The *IncrementModel* and the *MovementModel* may change at run-time, reflecting the use of a dynamic neighborhood in the *LocalSearch*, or of having a *BuildStrategy* that uses several kinds of *Increment*s to construct a *Solution*. The variation of the *IncrementModel* is controlled inside the *BuildStrategy* and the variation of the *MovementModel* is controlled by the *LocalSearch*. This control is performed using information made available by the Client and accessible to these objects. Figure 2 illustrates this scenario.

4. Implementation Issues

To construct the framework, we use design patterns presented in [11]. We identify these patterns below. In this section we also present some class hierarchies that could be used in the framework instantiations.

4.1 Strategy

In this section we examine the application of this pattern to the framework classes *Client*, *Search Strategy*, *BuildStrategy*, and *LocalSearch*. Figure 3 below shows the relationships among them.

The *Client* class contains an instance of a combinatorial problem, to be solved by a local search heuristic. Instance data is usually pre-processed, to verify whether simplifying hypotheses are satisfied and/or to perform problem reductions (variable fixations and constraint elimination). Also, after the search is performed, specific post-optimization techniques are usually applied. This suggests the encapsulation of search strategies in other classes, not in *Client*. Even the pre- and post-processing techniques should be isolated in distinct class hierarchies, once more using the Strategy pattern. However, to simplify our analysis, we will consider only the separation of local search heuristics (from the context of the combinatorial problem).

In our model, the *SearchStrategy* is responsible for finding solutions to the problem indicated by the *Client*. The operations and data structures used by the *SearchStrategy* are of no concern to the *Client*. The *Client* needs information to instantiate the *SearchStrategy*. The *SearchStrategy* class represents a meta-procedure that uses constructive and local search heuristics to generate solutions. For example, GRASP could be represented as a concrete subclass of *SearchStrategy*.

In summary, we isolate in distinct classes the problem and the search strategy, thus encapsulating search information and allowing greater facility in the construction and extension of a family of search strategies. Using the Strategy pattern, the *Client* defines the desired local search via a data structure visible to *SearchStrategy*, which then creates a corresponding *LocalSearch* object, which encapsulates the chosen local search algorithm. Code 1 illustrates a possible implementation for this class diagram.

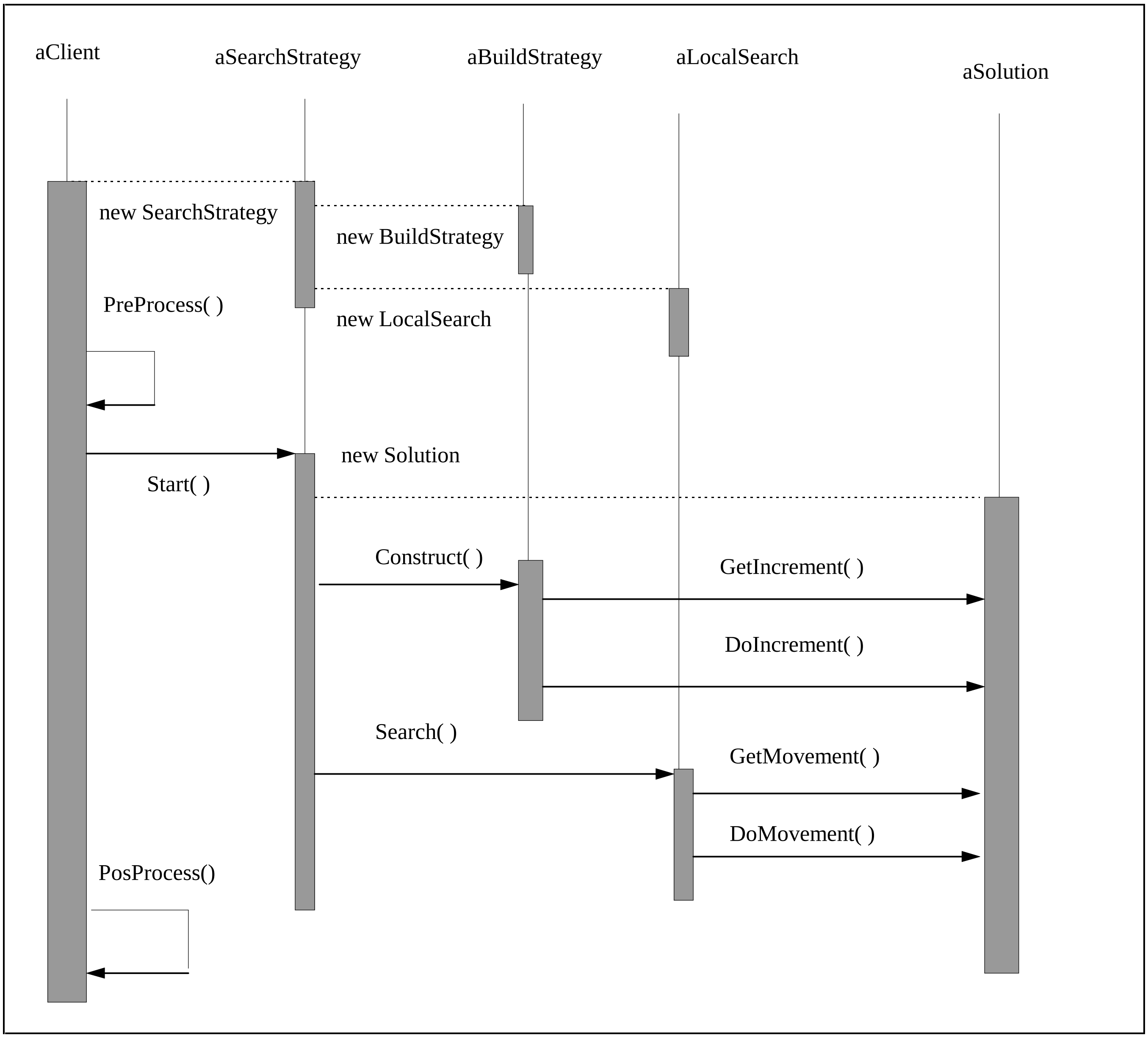

Figure 2: Collaborations in the framework

We notice that this piece of code should be taken just as an example, since one could implement the Start procedure without constructing the complete set of initial solutions before beginning the exploration by the searchers. This would be the case when several initial solutions should be constructed for being improved by a few searchers, or by a unique searcher, as is the case with GRASP. Also, there is usually only a single *BuildStrategy* and a single *LocalSearch,* and not several objects of each class.

The same motivation can be applied to the constructive and local search heuristics related to the *SearchStrategy* class: they can also be modeled in distinct classes, again using the Strategy pattern. In this way, families of algorithms for building initial solutions, and families of algorithms for local search, can be constructed with ease. Figure 4 shows an example of a class hierarchy modeling local search algorithms considered from the point of view of the movement selection heuristic.

The algorithms in the concrete classes of the above hierarchy, if implemented via conditionals within the same procedure would create confusing and hard to extend code. This would prevent the inclusion of new heuristics and the extension of those already included, better modeled as their subclasses.

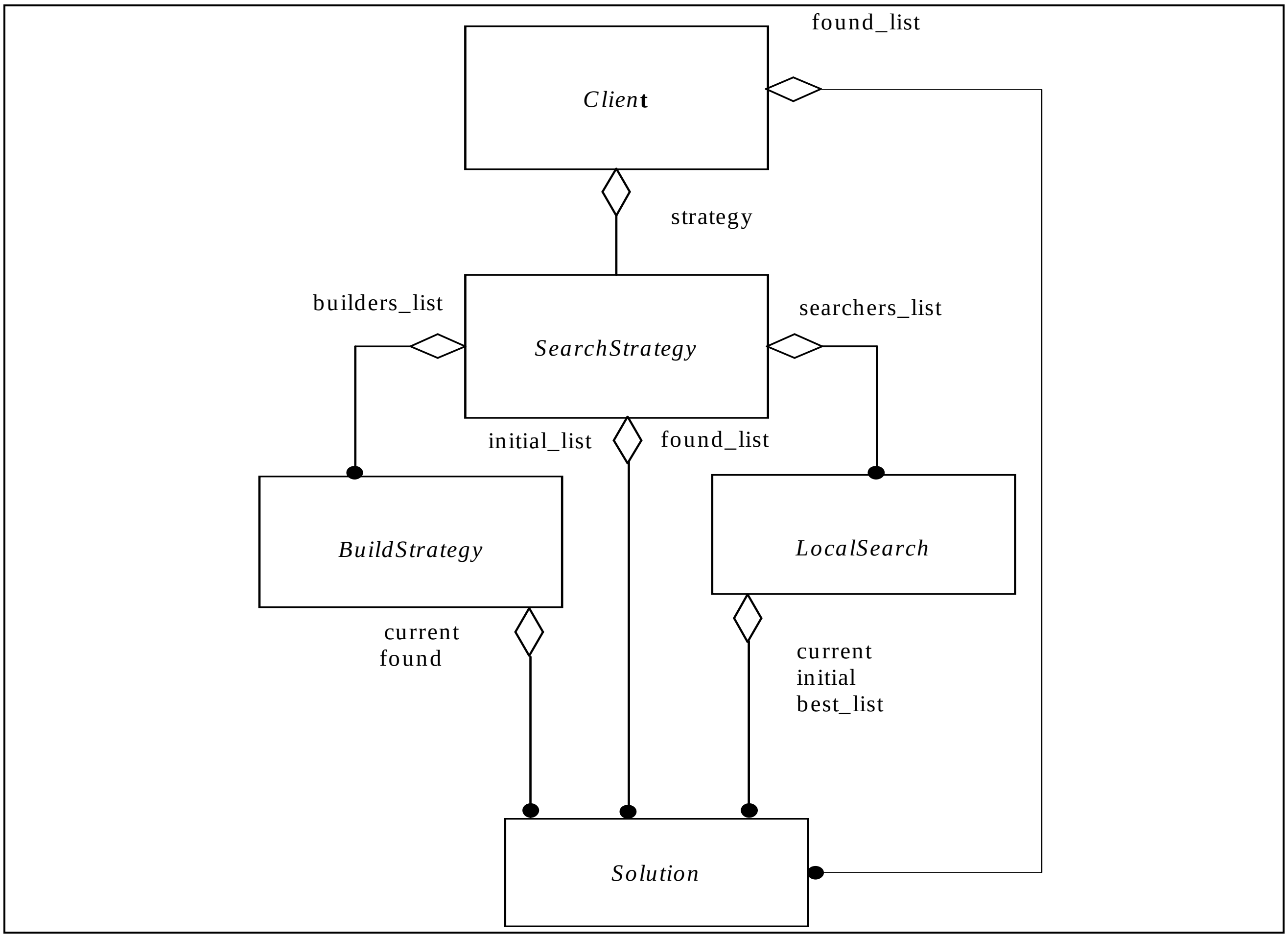

Figure 3: Application of the Strategy pattern

4.2 Template Method

In the last section we used the Strategy pattern to implement *LocalSearch*. The Template Method pattern could also be used in this derivation, provided we could factor out common steps from the local search algorithm. This can be done, for example, in the specialization of *IterativeImprovement*. To specialize *LocalSearch*, however, the different control mechanisms used in different algorithms do not allow such commonalities to be detected (unless we make the algorithm so abstract that we lose the code reuse facilities that the Template Method pattern offers).

We illustrate an attempt to use the Template Method in Code 2. In this example, the implementation of the Selection method declared in *LocalSearch* procedures is deferred to *IterativeImprovement*, *TabuSearch*, and any other concrete subclass of the hierarchy.

```
class Client{
   public:
      List <Solution> Solve( );
      ...
   protected:
      SearchStrategy strategy;
      List<Solution> found_list;
      virtual void PreProcess( );
      void LookFor( );
      virtual void PosProcess( );
      ...
};
List<Solution> Client::Solve( ){
   PreProcess( );
   LookFor( );
   PosProcess( );
   return found_list;
}
void Client::LookFor( ){ found_list = strategy.Start( this ); }

class SearchStrategy{
   public:
      virtual List<Solution> Start( Client * owner );
   protected:
      List<BuildStrategy> builders_list;
      List<LocalSearch> searchers_list;
   ...
};
List<Solution> SearchStrategy::Start(Client * owner ){
   BuildStrategy builder;
   LocalSearch searcher;
   List<Solution> initial_list;
   ...
   for( ; ; ){
     builder = builders_list.Next( );
     initial_list.Append( builder.Construct( owner ) );
     if( builders_list.IsEmpty( ) ) break;
   }
   for( ; ; ){
     searcher = searchers_list.Next( );
     for( ; ; ){
       found_list.Append(searcher.Search(initial_list.Next()));
     if( initial_list.IsEmpty(  )) break;
     }
    if( searchers_list.IsEmpty( )) break;
   }
   ...
}
```

Code 1: Applying the Strategy pattern

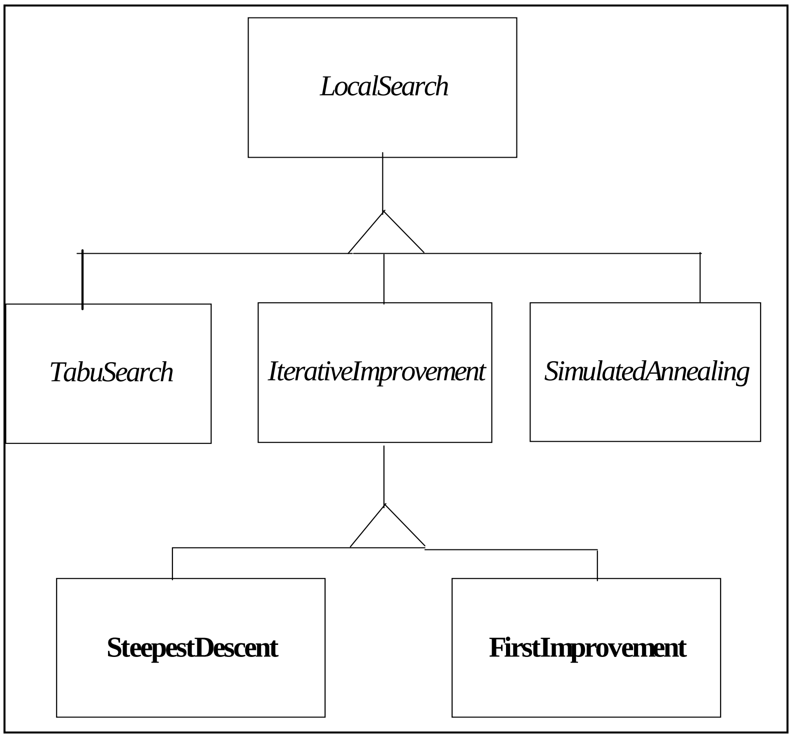

Figure 4: A LocalSearch hierarchy

4.3 State

The State pattern is used in the implementation of dynamic neighborhoods. A neighborhood is an attribute of a solution, hence the construction of different neighborhood types could be specified in the class *Solution*. However, due to the diversity of neighborhood relations that can be defined, it is convenient that *Solution* delegates this task, thus avoiding the inclusion, in its definition, of specialized data structures and generating algorithms.

This delegation of algorithms could again be considered as an application of the Strategy pattern. However, due to the fact that the object responsible for the construction of the neighborhood changes classes during the construction process (in other words, the neighborhood relation changes), this delegation is better identified as an application of the State pattern. We show the classes and relationships involved in this pattern in Figure 5. The corresponding Code 3 follows.

A *Movement* (or an *Increment*) is the information that is used by a *Solution*'s operation to modify its internal representation values or to provide another *Solution* with modified representation values. A *Movement* is just the data structure that contains the necessary information to implement the kind of movement associated to a particular neighborhood relation. Each concrete *MovementModel* is related to exactly one *Movement* concrete class. If the *MovementModel* changes during execution, we have a dynamic neighborhood. Delegating the construction of a neighborhood to a *MovementModel* results in the localization of the algorithmic peculiarities of the neighborhood relation, which is an application of the Strategy pattern. Since we do not have a unique method to localize (we must also localize the modification of the solution itself), and since we can use polymorphism on the *MovementModel*, the delegations from *Solution* to *MovementModel* are better implemented as applications of the State pattern. Actually, the implementation of dynamic neighborhoods consists in the use of polymorphism on a *MovementModel* inside its class hierarchy.

Families of construction algorithms can be defined according to different growth models. As in local search algorithms, the neighborhood of a solution in a construction algorithm is an attribute of a solution, which delegates the construction of the neighborhood to a subclass of *IncrementModel*, as shown in Figure 6.

If the building model object does not change classes during execution, the diagram above represents an application of Strategy; otherwise we have a State pattern application. It is not uncommon that a constructive algorithm uses more than one type of *IncrementModel*.

```
class LocalSearch{
   public:
      ...
      List<Solution> Search( Solution initial );
   protected:
      Solution current;
      ...
      virtual Boolean StopCriteria( );
      virtual Movement Selection( List<Movement> movement_list );
}
List<Solution> LocalSearch::Search( Solution initial){
      Movement selected, obtained ;
      List<Movement> candidate_list;
      ...
      for ( ; ; ){
         if ( StopCriteria( )) break;
         ...
         for ( ; ; ){
            if ( current.NbhAlreadyScanned( )) break;
            obtained = current.GetMovement( );
            candidate_list.TryInsertion(obtained);
         }
         selected = Selection(candidate_list);
         current.DoMovement(selected);
         ...
      }
}
class IterativeImprovement: public LocalSearch{
   protected:
   Boolean StopCriteria( );
   Movement Selection(...);
   ...
};

class TabuSearch: public LocalSearch{
   protected:
   ...
   List<Attribute> tabu_list; // Attribute is known by TabuSearch
   Boolean StopCriteria( );
   Movement Selection(...);
};
```

Code 2: Applying the Template Method pattern

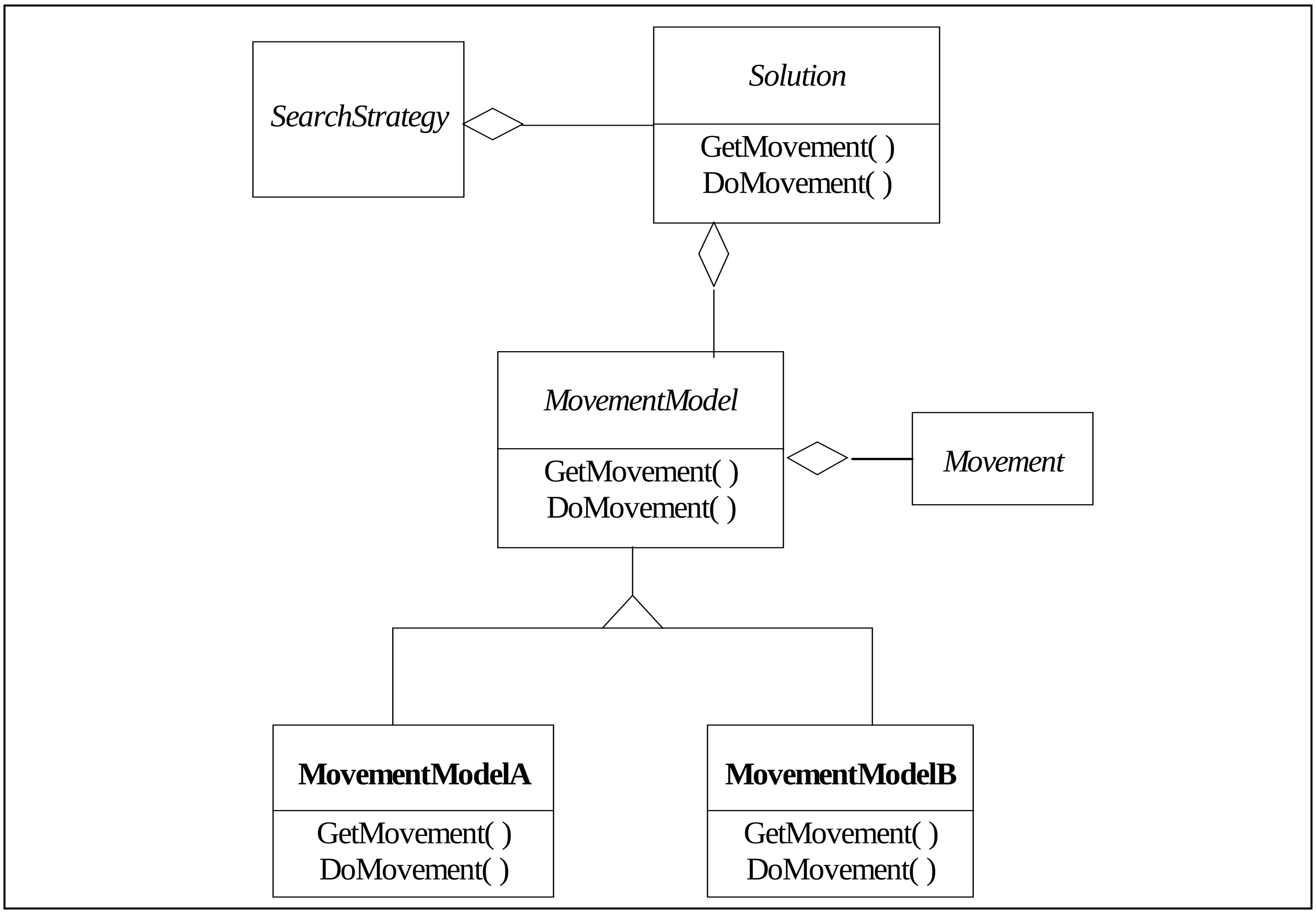

Figure 5: Application of the State pattern

```
class Solution{
   public:
      Movement GetMovement( );
      void DoMovement( );
      ...
   protected:
      MovementModel search_model;
};
Movement Solution::GetMovement( ){
   return search_model.GetMovement( );
}
void Solution::DoMovement(Movement movimento){
   search_model.DoMovement(movimento);
}
class MovementModel{
   public:
   Movement GetMovement( );
   void DoMovement(Movement mov);
};
```

Code 3: Applying the State pattern

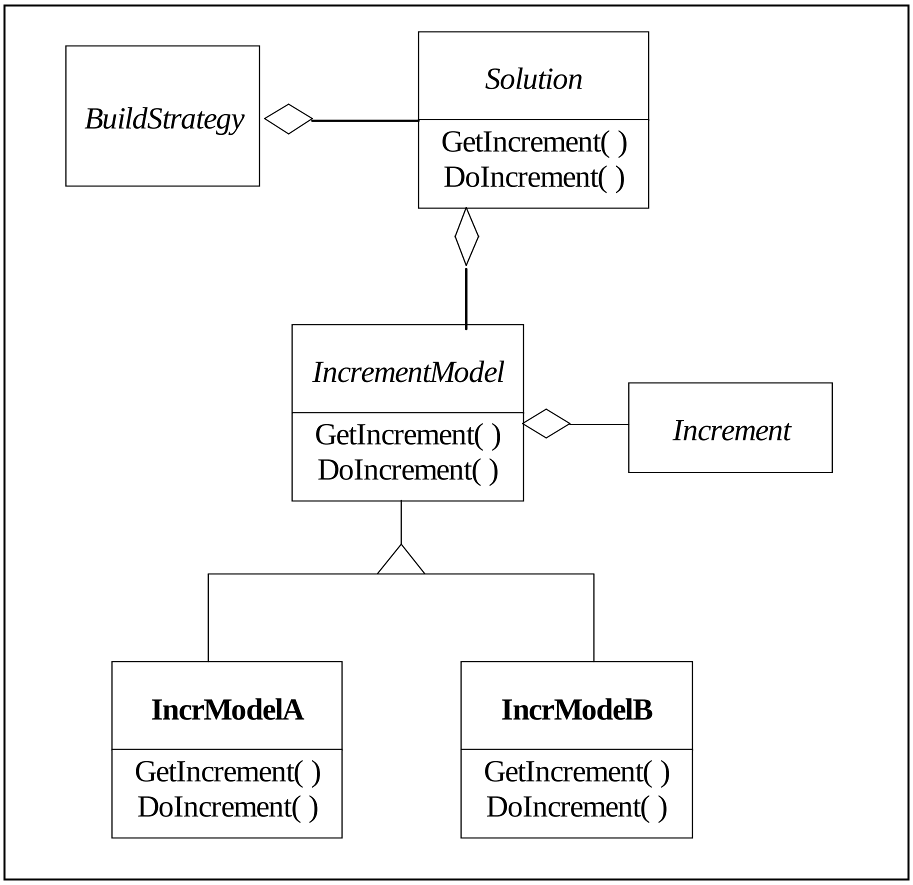

Figure 6: Other application of the State pattern

In the implementation of local search heuristics (via *LocalSearch* objects) that use dynamic neighborhoods, it is convenient that changes in neighborhood relationships be requested to *Solution* objects by *LocalSearch* objects, following a pre-defined transition model. *LocalSearch* controls transitions among neighborhood relationships. This control can use parameters made avaiable by the Client .

5. Concluding Remarks and Current Use

We propose in this work an architecture for the implementation and comparison of local search heuristics. The **NeighborSearcher** framework encapsulates, in abstract classes, different aspects involved in local search heuristics, such as algorithms for the construction of initial solutions, methods for the limitation of possible neighborhoods, and criteria for the selection of movements. Encapsulation and abstraction promote unbiased comparisons between different heuristics, new extensions, and code reuse. The classification of different aspects of heuristics simplifies their implementations and invites new extensions, in the form of subclasses. The reuse of code we can accomplish, implementing common aspects of a family of heuristics, allows us a better platform for comparison, since a large part of the code is common to all members of the family.

We created the framework from design patterns, carefully considering their use, often balancing a pattern against others. We have used the C++ language. This choice was motivated by the need of generating efficient code to manage problems instances of great size. Moreover, although C++ supports object-oriented programming, it is possible to relax the paradigm whenever one needs more speed (relaxing the data encapsulation restriction, for instance).

The development of object-oriented tools in the domain of neighborhood search metaheuristics is a trend that follows the effort of developing object-oriented tools that encapsulate exact algorithms for combinatorial problems, where the ABACUS [19] system is likely to be the most successful example. Ferland et al. [8] recently proposed a generic system in the domain of local search heuristics. They describe the use of object orientation in a system that implements four local search heuristics, essentially related to resource allocation problems. They propose two class hierarchies. In relation to our work, the first hierarchy can be said to define *Solution* descendants dynamically, which can be incorporated in the **NeighborSearcher** framework committing to a concrete *Solution* class dynamically (for example, based on the *Client* pre-processing operation). The second hierarchy is actually an application of the Strategy pattern, or the definition of a hierarchy of search heuristics, just as we propose for the class *LocalSearch*. Design patterns are not used to describe their architecture. Our framework presents some improvements with respect to their solution:

- It contains a hierarchy of local search strategies, and allows the inclusion of a hierarchy of solutions, thus containing the architecture proposed there.
- It allows the use of dynamic neighborhood models and uses a class hierarchy in the definition of the several neighborhood relations, while in the previous work only a static neighborhood model is used.
- It allows the implementation of constructive heuristics that utilize subordinate local search heuristics available in the *LocalSearch* hierarchy.

More recently, Woodruff [21] developed a class library concerning neighborhood search metaheuristics for problems whose solutions can be represented by characteristic, permutation, or integer vectors. As in our framework **NeighborSearcher**, the solution, problem instance and movement model concepts are partitioned. Solution representation and semantic are encapsulated in separated classes, since his framework is designed to take advantage of the common representation of different combinatorial problems. Fink and Voss [9] are also currently developing a framework for the same domain. Their project is based mainly in the use of static polymorphism, with intensive use of generic classes (using C++ templates).

We are currently using the **NeighborSearcher** framework to develop, implement, test, and compare heuristics for the phylogeny problem [18], which is one of the main problems in comparative biology. A phylogeny (or an evolutionary tree) is a tree structure that mainly relates groups of species or homologue genes in populations of distinct species, indistinctly denoted by taxons [3,20]. This relation is based on the similarity over a set of characteristics. This problem is NP-hard in general and in the common restricted cases [5,6,10]. We are currently developing a set of constructive methods and local search heuristics (iterative improvement, tabu search, GRASP, and variable neighborhood search, among others) for the phylogeny problem. The framework proposed here was developed as part of this effort [1]. The instantiation of the proposed framework to the phylogeny problem not only allowed the reproduction and a systematic comparison of algorithms dispersedly published in the literature, but also, and more important, the development of a whole family of local search methods and their effective comparison in terms of solution quality and computational times. Thorough computational results are presented in [2].

References

[1] A.A. Andreatta, *A framework for the development of local search heuristics with an application to the phylogeny problem*, PhD Dissertation (in preparation), Computer Science Department, Catholic University of Rio de Janeiro, Rio de Janeiro, 1997.

[2] A.A. Andreatta, S.E.R. Carvalho, and C.C. Ribeiro, "A comparative study of heuristics for the computation of phylogenetic trees", INFORMS Montréal Spring Meeting, Montréal, 1998.

[3] F.J. Ayala, "The myth of Eve: molecular biology and human origins", *Science* 270 (1995), 1930-1936.

[4] R.S. Barr, B.L. Golden, J.P. Kelly, M.G. Resende, and W.R. Stewart, "Designing and reporting on computational experiments with heuristic methods", *Journal oh Heuristics* 1 (1995), 1-32.

[5] H. Bodlaender, M. Fellows, and T. Warnow, "Two strikes against the perfect phylogeny problem", in *Proceedings of the International Conference on Automata, Languages and Programming* (Vienna, 1992), *Lecture Notes in Computer Science* 577, 273-283, Springer-Verlag, Berlin.

[6] W.H. Day, D.S. Johnson and D. Sankoff, "The computational complexity of inferring rooted phylogenies by parsimony", *Mathematical Biosciences* 81 (1986), 33-42.

[7] T. Feo and M.G. Resende, "Greedy randomized adaptive search procedures", Journal of Global Optimization 6 (1995), 109-133.

[8] J.A. Ferland, A. Hertz, and A. Lavoie, "An object-oriented methodology for solving assignment-type problems with neighborhood search techniques", *Operations Research* 44 (1996), 347-359.

[9] A. Fink and S. Voss, "Comparing different meta-strategies within a unifying framework for heuristic optimization", presented in EURO-INFORMS Conference, Barcelona, 1997.

[10] L.R. Foulds and R.L. Graham, "The Steiner problem in phylogeny is NP-Complete", *Advances in Applied Mathematics* 3 (1982), 43-49.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns - Elements of reusable object oriented software*, Addison Wesley, Reading, 1994.

[12] F. Glover, "Tabu search - Part I", *ORSA Journal on Computing* 1 (1989), 190-206.

[13] F. Glover, "Tabu search - Part II", *ORSA Journal on Computing* 2 (1990), 4-32.

[14] F. Glover and M. Laguna, "Tabu search", Chapter 3 in *Modern Heuristic Techniques for Combinatorial Problems* (C.R. Reeves, Ed.), 70-150, Blackwell Scientific Publications, 1993.

[15] M.G. Resende and C.C. Ribeiro, “A GRASP for graph planarization”, *Networks* 29 (1997), 173-189.

[16] J. Rumbaugh, “OMT: The object model”, *Journal of Object Oriented Programming* 7 (1995), no. 8, 21-27.

[17] J. Rumbaugh, “OMT: The dynamic model”, *Journal of Object Oriented Programming* 7 (1995), no. 9, 6-12.

[18] D.L. Swofford and G. Olsen, “Phylogeny reconstruction”, in *Molecular Systematics* (D. M. Hillis and C. Moritz, eds.), 411-501, Sinauer, Sunderland, 1990.

[19] S. Thienel, “ABACUS 2.0: User’s guide and reference manual”, Universität zu Köln, 1997.

[20] E.O. Wiley, D. Siegel-Causey, D.R. Brooks and V.A. Funk, *The compleat cladist - A primer of phylogenetic procedures*, Special publication no. 19, University of Kansas, Museum of Natural History, 1991.

[21] D.L. Woodruff, “A class library for heuristic search optimization”, Technical report, University of California at Davis, 1997.