Chapter 9

A GENERIC OBJECT-ORIENTED TABU SEARCH FRAMEWORK

Hoong C. Lau, Xiaomin Jia and Wee C. Wan

School of Computing, National University of Singapore
3 Science Drive 2, Singapore 117543

{lauhc, jxiaomin, jwan}@comp.nus.edu.sg

Abstract: Presently, most tabu search designers devise their applications without considering the potential of design and code reuse, which consequently prolong the development of subsequent applications. In this paper, we propose a software solution known as *Tabu Search Framework (TSF),* which is a generic C++ software framework for tabu search implementation. The framework excels in code recycling through the use of a well-designed set of generic abstract classes that clearly define their collaborative roles in the algorithm. Additionally, the framework incorporates a centralized *process and control* mechanism that enhances the search with intelligence. This results in a generic framework that is capable of solving a wide range of combinatorial optimization problems using various tabu search techniques and adaptive strategies. The applications of TSF are demonstrated on the implementation of two NP-hard problems, the Vehicle Routing Problem with Time Windows (VRPTW) and Quadratic Assignment Problem (QAP). We show that TSF is able to obtain quality solutions within reasonable implementation as well as computation time.

Key words: Tabu Search, software framework, reusability, combinatorial optimization.

9.1 INTRODUCTION

Recent studies have reported many successful applications of tabu search [Glover and Laguna, 1997] in solving real-world optimization problems. In most cases, tabu search applications are designed specifically for their intended problems, with little or no consideration for design and codes reuse. Consequently, this leads to difficulties in recycling the developed work for subsequent applications. Conceivably, the software development cost could

be drastically reduced if a well-designed programming framework is available that allows the reuse of both designs and codes. The challenge of developing such a framework lies in the tension between its simplicity of use versus the sophistication involved in supporting user-preferred search strategies. To date, there is a lack of a widely accepted tabu search framework and only few research prototypes have gained limited popularity.

Another strong motivation for a tabu search programming framework is the demand for an unbiased platform that could ensure fairness in comparing different tabu search strategies. Typically, it is not unusual for a talented programmer to write better codes than his peers and thus could paint an unrealistic picture of a more efficient algorithm. However, when different algorithms for a given problem are implemented on a common framework, it implies that they have been implemented using common software components (such as using the same tabu search engine) and even underlying data structures, which indirectly enforces fairness when algorithms are compared, especially in terms of run-time performance. In addition, an optimized framework lowers the technical expertise of an algorithm designer by providing him with efficient components.

Perhaps an implicit third incentive for a framework can be seen from the software engineering point of view. By enforcing an object-oriented design (OOD), the framework imposes a development discipline on the algorithm designer. Consequently this paves a way for ease of code integration and maintenance, and for future extension. In addition, OOD also provides clarity in design, which allows algorithm designers to quickly grasp the conceptual flow of the framework. This results in more efficient use of the framework as well as less prone to programming errors.

In this paper, we propose a generic object-oriented framework that performs the standard routine of the tabu search and yet offers the robustness for algorithms designers and developers to incorporate their desired search strategies. *Tabu Search Framework (TSF)* is a C++ object-oriented software framework that uses a set of interfaces to epitomize the routine tabu search procedures. It has a centralized control mechanism to adaptively guide the search in accordance to events encountered dynamically. TSF also provides a set of supporting software tools, called the *Strategy Software Library,* which aids developers in their strategies development. In summary, TSF allows users to focus on designing, testing and comparing algorithms by minimizing efforts in programming.

This paper proceeds as follows. Section 9.2 gives a literature review of some existing local search frameworks. Section 9.3 presents the architecture of TSF. Section 9.4 gives 3 illustrations on using TSF to formulate user-defined

strategies. Section 9.5 presents experimental results. Section 9.6 presents the conclusion and future works.

9.2 LITERATURE REVIEW

In this section, we present a review of four existing frameworks, OpenTS, Localizer++, EasyLocal++ and HotFrame.

9.2.1 OPENTS

OpenTS [Harder, 2003] is one of the project initialized by *Computational Infrastructure for Operations Research (COIN-OR)* to spur the development of open-source software for the operations research community. It is a java-based tabu search framework that has a well-defined, object-oriented design. The generic aspect of the framework is achieved through inheritance, using well-structured interfaces, which includes *Solution*, *Move*, *Move Manager*, *Objective Function*, *Tabu List* and *Aspiration Criteria*. This unambiguous decomposition defined clearly the collaborative role of each interface in the algorithm. In addition, the author presumes that most TS applications adopt the "tabu-ing the move" strategy and hence provides "helper" classes such as *SimpleTabuList*, *ComplexMove* and *ComplexTabuList* classes to assist the implementation.

OpenTS also supports the implementation of adaptive strategies through the use of the *EventListener* objects. These listeners can be embedded into any of the interface-inherited objects and used later to adjust objects' parameters. However, the listeners only respond to a static set of search events and does not consider user-defined events such as recording the presence (or absence) of certain solution structures. This results in difficulty in implementing strategies that are based on the solution structures (such as recency and frequency based strategies). The absence of a centralized control mechanism also poses a limitation to the framework capability. For example, when two listeners are triggered in the same iteration, their order of execution follows a First-In-First-Out (FIFO) sequence, thus giving no control to the algorithm designer. It is also probable for two conflicting EventListener objects (such as intensification and diversification) to be performed together without designer's intention.

9.2.2 Localizer++

The literature presented another framework known as the Localizer++ [Michel and Van Hentenryck, 1999] that incorporates *Constraint Local Search (CLS)* in C++. The framework is structured into a two-level architecture, which composes of *Declarative* and *Search* components. The Declarative components are the core of the architecture and are used to maintain the complex data structure in local search. In addition, it also incorporates a *Constraint Library* that provides a set of frequently used constraints, such as the *AllDiff* constraint which verifies that every element in the data structure has a different value. The Search component on the other hand, operates around the Declarative component and is procedural in nature. Generally, this component implements the general procedure of local search and thus could be used to implement any meta-heuristics that follow to this general behavior (i.e. such as iterative local search and tabu search).

Localizer++ requires designers to formulate their problem into its mathematical equivalence form in order for the framework to recognize and subsequently manage the variables (thus achieving the genericty aspect). Algorithm designers are required to implement the routines of the local search such as the local moves and the selection criteria, and together with the Constraint Library, to construct the optimizer. Due to the numerous possible types of constraint, it is improbable for the Constraint Library to provide all forms of constraint and thus Localizer++ copes with this limitation by supporting the extension to the library through the addition of invariants. The framework also supports user-defined search strategies that are triggered at static points of the search (such as at the start or the end of the search) rather than dynamically in response to search events. New search procedures can be extended from Localizer++ through inheritance.

9.2.3 EASYLOCAL++

EasyLocal++ [Gaspero and Schaerf, 2001] is another object-oriented framework that can be used as a general tool for the development of local search algorithms in C++. EasyLocal++ relies on programming techniques such as the “Template Method” that specifies and implements the invariant parts of various search algorithms, and the “Strategy Method” for the communication between the main solver and its component classes, in order to achieve the generic aspect. The classes in EasyLocal++ can be classified into four categories, *Basic Data*, *Helpers*, *Runners* and *Solvers*. The Basic Data is a group of data structure with their managers and is used to maintain the states of the search space, the moves, and the input/output data. The

Basic Data classes are supplied to the other classes of the framework by means of template instantiation. The local search problem is embodied in the Helpers classes, which perform actions that are related to some specific aspects of the search, such as maintaining the states or exploring the neighborhood of a solution. The Runners represent the algorithmic core of the framework and are responsible for performing the routine of the meta-heuristic. Currently, EasyLocal++ supports several common meta-heuristics such as hill climbing heuristic, simulated annealing and tabu search.

EasyLocal++ can be easily deployed by first defining the data classes and the derived helper classes, which encode the specific problem description. These classes are then "linked" with the required Runners and Solvers and the application is ready to run. EasyLocal++ also supports diversification techniques through the *Kickers* classes. The Kickers objects are incorporated into the Solver and triggered at specific iteration of the search. Hence, this mechanism relies on the knowledge of the algorithm designer to determine the best moment to trigger the diversification. While this may be achievable for most experience designer, it may be demanding for unfamiliar implementer coping with a new problem. In short, the framework provides limited support for adaptive strategies and although it could be possible for such strategies to be incorporated, the authors do not present a clear mechanism to realize them.

9.2.4 HOTFRAME

HotFrame [Fink and Voß, 2002] is a more matured meta-heuristics framework implemented in C++ when compared with Easy Local++. The framework provides numerous adaptable components to incorporate a wide range of meta-heuristics and common problem-specific components. The supported meta-heuristics includes basic and iterated local search, SA and their variations, different variants of tabu search, evolutionary methods, variable depth neighborhood search, candidate list approaches and some hybrid methods. As a means of reducing the programming efforts of designers, HotFrame provides several reusable data structure classes to incorporate common solution spaces such as binary vectors, permutations, combined assignment and sequencing and also some standard neighborhood operations like bit-flip, shift, or swap moves. These classes can be deployed immediately or be used as base classes for subsequent customized derived classes. This design encourages software reuse especially for problems that can be formulated with the components that are already presence in the framework.

Meta-heuristics strategies can be implemented in HotFrame through the use of templates. The idea is to incorporate a set of type parameters that can be extended to support both problem-specific and generic strategies. A benefit of this design is that it gives HotFrame a concise and declarative system specification, which would decrease the conceptual gap between program codes and domain concepts. HotFrame also adopts a hierarchical configuration for the formulation of the search techniques in order to separate problem-specific with the generic meta-heuristic concepts. Generic meta-heuristic components are pre-defined in the configuration as a higher-level control while the problem-specific definitions are incorporated inside these meta-heuristic components to form a two level architecture (i.e. each problem-specific strategy will be embedded to a meta-heuristic scheme). Additionally, inheritance can be applied on these components to overwrite parts of the meta-heuristic routine with user preferred procedure.

9.2.5 Discussion

It can be seen that the focus of the above reviewed frameworks is primarily on relieving the mundane task of meta-heuristic routines from the algorithm designers. TSF has a slightly different objective: TSF is designed with the intention to support adaptive tabu search strategies. As such, in addition to performing the basic routines of tabu search, TSF has a centralized control mechanism to incorporate adaptive strategies. While it is true that most of the reviewed frameworks could also support adaptive strategies, the incorporation of such strategies may not be as straightforward (i.e. no dedicated mechanism is built for it). For example, while HotFrame could incorporate adaptive strategies by extending its procedures, such inheritance often requires the designer to "re-code" some of the meta-heuristic procedures. This may pose a problem to designers who are unfamiliar with the internal design of the framework, i.e. the framework can no longer be seen as a black box to the designer. TSF works differently from such conventional design by incorporating a communication link between the framework engine (which perform the tabu search routines) and the control mechanism. As the search proceeds, the engine will record the current search state and passes this information to the control mechanism. The control mechanism then processes information to determine if actions such as adjusting the search parameters or applying a diversification strategy are necessary. Analogously, the control mechanism can be seen as a feedback mechanism that adapts the framework engine to the search environment. Such feedback control provides a centralized mechanism for designer to collect information on the search spaces as well as to direct future search trajectory.

9.3 DESIGN AND ARCHITECTURE

9.3.1 Frameworks

Framework has a different concept from "*Software Library*". In a software library, implementers are required to design their program flow and then use the components in the library to develop their applications. The ILOG optimization suite [ILOG, 2003] (such as the CPLEX and Solver engines) is an example of a well-known software library. In ILOG optimization software library, the programmers formulate their problems as a specific constraint/mathematical-programming model, and a predefined engine operates on the formulated model to obtain solutions. A benefit of such software architecture is that it does not require the programmer to specify the algorithms to be performed on the problem. However, this can also be a drawback, as the predefined engine offers little if no capacity for the implementers to control of the search process.

On the other hand, in a framework, the reused code is the controller, which provides an overall "frame" for an application. By means of inheritance, the programmer can implement the child classes, which are subordinate to the framework. In an informal sense, frameworks tell the programmer "*Don't call us, we'll call you*." In summary, while library allows sharing of low-level codes, frameworks allow sharing of high-level concepts and control.

The essence of our proposed TSF framework revolves around four principal considerations:

(a) Genericity
This means that the framework should allow the user to implement any tabu search algorithm. Additionally the framework should not make any assumption on problem type or its solution representation, thus allowing flexibility to the formulation of applications.

(b) Reusability
The framework should adopt built-in tabu search routines that can be easily recycled across applications. Furthermore, the algorithm procedure should be disintegrated into distinctive interfaces so that the developed components could be recycled easily across different applications.

(c) Extensibility
The framework should able to extend easily to support not only some predefined tabu search strategies, but also user-defined procedures that

are specific to the problem domain. In addition, the possibility of extending the framework to form hybrids should not be ignored.

(d) Usage Friendliness
The framework should be easy to learn and understand from an algorithm designer and implementer perspective. It should have unambiguous interfaces that give clarity in execution flow.

To cater for these design goals, the components of TSF are categorized into 4 components: *interfaces*, *control mechanism*, *search engine,* and *strategies software library* (see Figure 9.1). The *interfaces* define the basic components used by tabu search in an object-oriented architecture. The *control mechanism* is for users to define strategies that guide the search adaptively in response to events encountered. TSF eliminates the tedious routine task of programming the *search engine* repeatedly. The search engine interacts with the interfaces and collects information that is passed dynamically to the control mechanism to influence future search trajectory. Finally, TSF includes an optional *Strategy Software Library* consisting of a set of software components to support various user-defined search strategies.

9.3.2 Interfaces

There are seven key interfaces that must be implemented, described as follows.

1. The *solution* interface allows the user to encapsulate the data structure(s) that represents the problem's solution. The framework does not impose any restriction on how the user defines the solution or the data structures used since it never manipulates *Solution* objects directly.
2. The *Objective Function* interface evaluates the objective value of the solution.
3. The *Neighborhood Generator* interface is an iterator that generates a collection of neighbors based on the *Move* and *Constraint* interfaces.
4. The *Move* interface defines the move neighborhood, where it constructs possible translations of *Solution* object. When the engine determines the best move in an iteration, the *Solution* object is then translated to its new state.
5. The *Constraint* interface computes the degree of violation for these translations.
6. The *Tabu List* interface records the tabu-ed solutions or moves.
7. The *Aspiration Criteria* interface allows specification of aspiration criteria for tabu-ed moves.

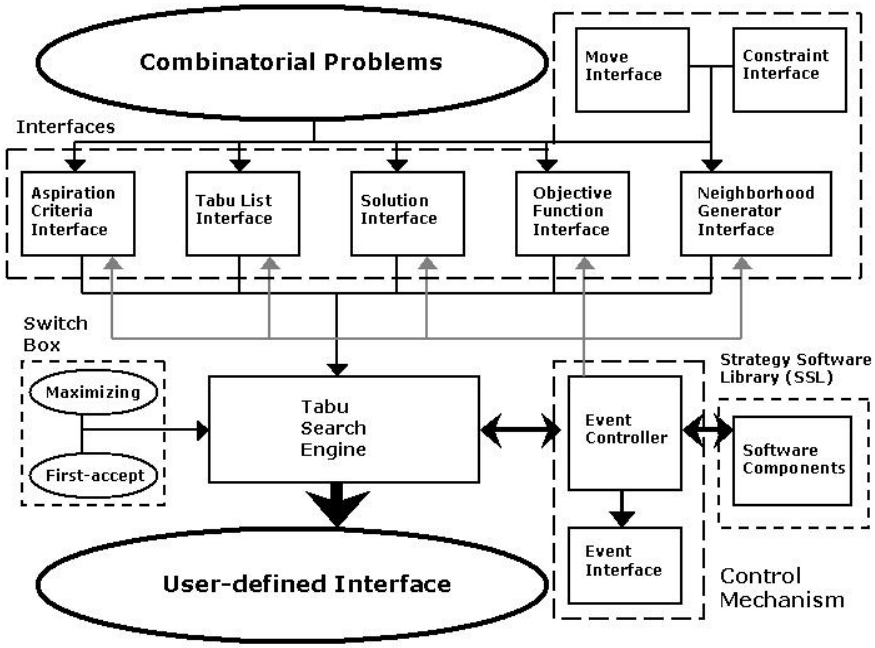

Figure 9.1. TSF Architecture

9.3.3 Control Mechanism

Switch Box

Like many machines, a set of switches is required to operate the Search Engine. A Switch Box is used in the framework to control the basic operations of the search engine. The two commonly used switches are the maximizing switch and the first-accept switch. The maximizing switch is used to control the engine to solve a maximizing or a minimizing problem. The First-Accept switch informs the tabu search engine to perform first accept or best accept strategy. The best accept strategy searches through all the feasible solutions in the neighborhood and select the best possible move. When run time is comparatively more crucial than the solution quality, we settle for the first neighbor with a better objective value, i.e. the first-accept strategy. All switches are capable of changing values dynamically set by an application.

Event Controller

It is often desirable for the search engine to respond to encountered events.

For example, a reactive tabu list would need to readjust its tenure in response to the success or failure in obtaining a better solution during the search. Hence we need a means of controlling the tabu search to make dynamic readjustments. TSF uses a centralized control mechanism. It has an *Event Controller*, which provides interaction between the search engine and its interfaces. When the tabu search engine detects the occurrence of pre-determined events, it conveys them to the *Event Controller*. The *Event Controller* then responds to these events in accordance to strategies defined by the user. Typically, these strategies will affect one or more of the elements in the interfaces, which in turn, re-adjust the search strategy adopted by the engine. For the reactive tabu list example, the number of non-improving moves encountered can be a "triggering-event", which causes the *Event Controller* to readjust the tabu tenure based on parameters. Some default events in TSF are as follows:

1. *Tabu Search Start* is triggered at the start of tabu search. Used to start a timer to record the total time spent in doing the search.
2. *Tabu Search Stop* is triggered at the end of tabu search. Used to stop the timer or to call an output program to display the results.
3. *New Best Solution Found* is triggered when a new best solution is found. Used for intensification strategies.
4. *Non-improving Move Made* is triggered when the tabu search engine executes a new non-improving move. Used for diversification strategies.
5. *End of Iteration* is triggered at the end of the iteration. This is extremely useful for collection of search information (such as similarity of solutions) or strategies that executed every iteration.
6. *No move generated is* triggered when the *Neighborhood Generator* could not generate any moves.
7. *No Solution Found* event is triggered when the search engine could not find a feasible solution. Used for implementing oscillating strategies.

TSF also allows users to define their own events by providing a generic *Event* interface. Users need to define the triggering event and implement the response to it. For example, suppose we want to apply *Random Restart* strategy at every *n* number of completed iterations. To do this, we simply need to implement an *Event* that is triggered at the every *n*th iteration. The response to this event is to restart the tabu search with a new starting solution. Figure 9.2 gives a fragment of code segment for such implementation.

```
// Base class for the Event interface
Class Event{
    ……
   virtual void IterationEvent (TabuSearch* TS){}; }

Class RandomRestart : Event // user-implemented class
{
  int max_limit = n; // User defined n iteration
  int count_NonImprovingMoves;

  virtual void IterationEvent (TabuSearch* TS){
    if (TS->isBadMove())
    int count_NonImprovingMoves++;
      if (count_NonImprovingMoves > max_limit)
        ApplyRandomRestart (TS);
  }
}
```

Figure 9.2. Code Implementation of a User-Defined Event

Some strategies may require history information to be collected during search. In TSF, the *Event* interface can be used to collect such useful information. For instance, implementing a *Frequency and Recency* strategy may require the tabu search to record the number of occurrences a sub-structure/configuration in the solution has appeared during the search. As an example, in the Traveling Salesman Problem, if we discover that the solution that visits customer 4 before customer 3 occurs very frequently in local optima, this may imply that the global optimal may contain the same sub-route with the structure X–4–3–Y, where X and Y refer to some arbitrary customers. In this case, the user may like to implement a soft constraint based on this observation, which in turn will reduce the size of the neighborhood generated. By collecting more information during the search and consequently adaptively modifying the search strategy, we can derive a more effective search.

9.3.4 Tabu Search Engine

The Tabu Search Engine component implements the control flow as follows:

1. Set initial solution as current and best found solution
2. ***Neighborhood Generator*** generates a list of feasible moves

3. ***Objective Function*** evaluates each move
4. Choose best non-tabu move using ***Tabu List*** and ***Aspiration Criteria***
5. Apply ***Move*** on the current solution
6. Update ***Tabu List*** and trigger related events
7. Go to step 1 until terminating condition(s) is reached

9.3.5 Strategy Software Library (SSL)

Strategy Software Library (SSL) provides optional components for designer to facilitate them in adding generalized strategies. By making assumptions on the solution representation (such as permutation of integers or array of bits), these components can be incorporated to build strategies such as *intensification among elite solutions*, *probabilistic diversification*, *candidate lists*, *and very large-scale neighborhood (VLSN)*. Although not as powerful as specific strategies tailored for a single problem type, these generic components provide a quick and easy means for developers to apply them in their search. In the following, we describe some of the components in *SSL*.

Probabilistic Diversification

Probabilistic diversification refers to diversifying the search when it is caught in a local optimum. If the designer adopts a permutation or bits array solution representation, *SSL* can readily supports this strategy by providing a *Random Generator* that determines the parts of the solution that will be removed in according to a preset probability distribution (such as uniformly random distribution). These randomly chosen portions are then reconstructed by random swapping (permutation representation) or flipping (bits array) and later recombined.

Intensification search on Elite solutions

SSL supports this strategy by storing a list of elite solutions during tabu search in the *Elite Recorder*. As solution objects inherited *Solution* interface, different solution representations can be easily stored as their base class. Each of these elite solutions is then used as a new initial solution for future tabu search. The rationale behind this strategy is to search the elite solutions more thoroughly and hence is classified as an intensification strategy. Developers simply need to declare the number of elite solutions to be collected in the preliminary search and TSF would search each of these points more thoroughly.

Very Large-Scale Neighborhood (VLSN)

VLSN [e.g. Ahuja et al., 2002] works on the principle that by generating a larger neighborhood, the hope is to increase the probability of obtaining better solution in a single iteration. Under the assumption that the solution representation is a permutation of integers, this strategy usually requires some permutation function to generate the large neighborhood. *SSL* provides two functions to support this strategy: *PermutationGenerator* and *NeighborhoodBooster*. *PermutationGenerator* provides the ease of permuting a solution to construct its neighborhood. *NeighborhoodBooster,* is used to increase the neighborhood size by combining multiple 2-opt moves into k-opt moves. User can also combine this strategy with the candidate list to keep the size of the neighborhood reasonable.

Candidate Lists

Candidates are often used to narrow the neighborhood size especially when VLSN is involved. A possible technique is to select neighbors that meet some certain criteria or constraint. *SSL* provides a *Filter* function that inherits from the *Constraint* Interface. It receives a solution, a move and a selection function, and "filters" unfavorable moves that do not meet the constraint.

9.4 ILLUSTRATION

In this section, we illustrate how TSF can deploy some common user-defined search strategies, namely *Diversification*, *Intensification* and *Reactive Tabu Search*[1]. These simplistic examples demonstrate how events can interact and incorporate the necessary actions to execute user-defined strategies.

9.4.1 Diversification

Diversification refers to strategies that transform one solution to a radically different solution. Usually, this transformation involves alternating part or the whole solution structure. When some forms of probability are involved in

[1] Note that although TSF can support adaptive strategies such as diversification, it is accomplished in a rather rudimentary fashion. Since the publication of the conference version of this work, the authors have developed an enhanced version of TSF known as Meta-heuristic Development Framework (MDF), in which TSF is a component of a bigger framework (please see [Lau et al., 2004]. In MDF, strategies such as adaptive methods and hybridization can be incorporated in a more elegant approach via the use of event handlers.

the transformation, it is known commonly as probabilistic diversification. The greatest difficulty in executing this strategy is to decide when to apply diversification, for if diversification is applied too often, this may result in accidentally missing out good local optima; otherwise if diversification is performed infrequently, the search may waste too much time exploring potentially useless solutions. The easiest approach to decide when to perform diversification is to execute a fixed number of iteration and then applied the strategy. TSF easily support this strategy by implementing an event that counts the number of iteration performed. This event performs like a hardware counter. When the counter-event reaches zero, it will execute the desired diversifying actions. A code fragment is shown on Figure 9.3.

Typically, this simplistic guiding rule is not very effective. An improved strategy is then to apply diversification when the search is caught in local optimal or experiences solution cycling. Unfortunately, both of these situations are extremely difficult to detect accurately. There exist many heuristics that help to predict their occurrences, and one such heuristic is to observe the number of non-improving moves made since the last local optimum found. This heuristic again can be easily implemented with TSF using the event controller. Here, we implement an adaptive counter that decrements when the search makes a non-improving move and resets itself when a new best solution is found. A code fragment for this adaptive counter is shown in Figure 9.4.

9.4.2 Intensification

Intensification refers to strategies that examine thoroughly on some specific solutions. As oppose to diversification, intensifying strategies improve on the solution quality by searching around the region close to the solution. As such, intensification is usually performed on elite solutions. Hence, intensification often requires two phases - the initial phase is used to identify “good” solutions while the next phase attempts to locate new local optimum around these identified solutions. TSF can implement this strategy using the event controller that track two events for each phase of search respectively. An elite solution recorder is essential to record on the elite solutions and user can use either the provide tool in SSL or to implement their own.

In the first phase, the `First-Phase event` first preset a number of iteration to collect the elite solutions. This is very similar to the counter-event discussed in section 9.4.1. In addition, the `First-Phase event` also records the solution whenever it encounters a new best-found solution. When the counter reaches zero, the `First-Phase event` will notify the event controller that it has completed its task and the event controller will proceed to replace the `First-Phase event` with the `Second-Phase event`.

The `Second-Phase event` is used to performed intensification on solution collected. One strategy in which intensification can be performed is to use a strategy that is analogous to backtracking. In this strategy, we conduct search on an elite solution for a fix number of iterations. If no better solution can be within these iterations, we would backtrack to our original solution and conduct a new search. To prevent conducting similar search, one approach is to preserve the tabu list. However, this may require a fair amount of memory depending on the number of iteration conducted. We proposed another approach, which is to interchange the tabu list each time we backtrack. The rationale behind this is that tabu list is used to guide the search and hence by using different tabu lists and interchanging them, we expect the search to move in different direction each time we revert to the original elite solution. A pseudo code for the events in both phases is presented in Figure 9.5.

9.4.3 Reactive Tabu Search

Our last illustration is on reactive tabu search, where we explain how two strategies can be incorporated into a single event. Reactive tabu search refers to strategies that adaptively adjusting tabu search parameters according to the search trajectory. Many complex heuristics have been proposed with this strategy, each with its own assumptions on the solution space. In fact a popular analogy is to visualize the solution space as a multi-dimensional terrain. The factors include objective value, similarity in the solution structure and time. Based on these factors, the reactive tabu search attempts to navigate along the terrain toward new local optima. In order to simplify our illustration, we only consider two factors, time and objective value and the parameter adjusted is limited to the tabu tenure. Time simply refers to number of iterations performed. Our simplified strategy works as follows. When we encounter a series of non-improving we lengthen our tabu tenure so as to prevent solution cycling. On the other hand, when we encounter a new best solution, we shorten our tenure in order to perform intensification. Hence we use TSF to implement an event called `Reactive-Event` to handle the two scenarios. First, when we encounter a series of non-improving moves, we will increase the tabu tenure by some `x` amount. On the other hand, when a new best solution is encountered, we will revert the tenure, discarding any move that have been kept for more than `n` iterations. The pseudo code for this implementation can be found in Figure 9.6.

```
Class CounterEvent : Event // user-implemented class
{
  int startValue = n; // User defined n iteration
  virtual void IterationEvent (TabuSearch* TS){
    startValue = startValue - 1;
    if (startValue <= 0){
      ApplyDiversification(TS->getCurrentSolution())
      StartValue = n; // reset counter
    }
  }
}
```

Figure 9.3: Code fragment implementing a counter event.

```
Class AdaptiveCounterEvent : Event
{
  int startValue = n; // User defined n moves
  virtual void NewBestSolutionFound (TabuSearch* TS){
    StartValue = n; // reset counter
  }
  virtual void NonImprovingMoveMade (TabuSearch* TS){
    startValue = startValue - 1;
    if (startValue <= 0){
      // Diversification is a user implemented method
      //   in class AdaptiveCounterEvent
      ApplyDiversification(TS->getCurrentSolution())
      StartValue = n; // reset counter
    }
  }
}
```

Figure 9.4: Code fragment implementing an adaptive counter event.

```
Class FirstPhaseEvent : Event
{
  int maxIter = n; // Maximum number of iteration
  virtual void NewBestSolutionFound (TabuSearch* TS){
    EliteRecorder.record(TS->getCurrentSolution());
  }
  virtual void IterationEvent (TabuSearch* TS){
    maxIter = startValue - 1;
    if (maxIter <= 0)
      TS->getEventController().NextEvent();
  }
}
Class SecondPhaseEvent : Event
{
  int allowedIter = x;
  virtual void NewBestSolutionFound (TabuSearch* TS){
    allowedIter = x;
  }
  virtual void IterationEvent (TabuSearch* TS){
    allowedIter = startValue - 1;
    if (allowedIter <= 0) BackTrack(TS);
  }
}
```

Figure 9.5: Code fragment implementing intensification strategy.

```
Class FirstPhaseEvent : Event
{
  int badMoveLimit = n; // Maximum allowed bad moves
  int increment = x; // Maximum allowed bad moves
  int defaultTenure = t; // Maximum allowed bad moves
  virtual void NewBestSolutionFound (TabuSearch* TS){
    TS->getTabuList().setTabuTenure(t);
    BadMoveLimit = n;   // reset limit
  }
  virtual void NonImprovingMovesMade (TabuSearch* TS){
    badMoveLimit = badMoveLimit - 1;
    if (badMoveLimit <= 0) {
      int tenure = TS->getTabuList().getTabuTenure();
      TS->getTabuList().setTabuTenure(tenure+x);
    }
  }
}
```

Figure 9.6: Code fragment implementing simplified reactive tabu search.

9.5 EXPERIMENTATION

In this section, we report on experimental results on VRPTW and QAP. It is interesting to first observe that a senior undergraduate computer science student proficient at C++ programming took 1 week to learn TSF by studying the documentation and interacting with the TSF developers. He took another 1 week to first implement an application (QAP), and 3 days to implement a second application (VRPTW).

9.5.1 Vehicle Routing with Time Windows (VRPTW)

We benchmark with Solomon's instances [Solomon, 1987]. 5 different moves were implemented: *Relocate*, *Exchange*, *Reverse*, *Cross* and *Distribute*. *Reverse* is used to reverse the sequence of customers within a same route and is useful when the time window is loose. *Cross* is an extended *Exchange* where a sub-section of a route is swapped with another. *Distribute* attempts to reduce a vehicle by distributing the vehicle's customers to other vehicles. The design of these moves exploits the advantage of minimizing the distance without minimizing vehicles. With the exception of *Relocate* and *Distribute*, the other moves are designed to minimize the total distance traveled. The tabu list records on the previously accepted moves and a different tenure is set for each type of moves. The values of the tenure are *Relocate: 1000*, *Exchange: 1000*, *Reverse: 300*, *Cross: 500* and *Distribute: 400* for a problem size of 100 customers. We also adaptively apply intensification or diversification strategies based on the quality of the solutions found. Based on the default events, each solution is classified into two categories: *improving solutions* and *non-improving solutions*. An improving solution has an objective value that is better than all previously found solutions. Non-improving solution refers to solutions whose objective value is the same or poorer than the best-found solution. TSF adaptively alternates between intensification and diversification by observing the frequency of non-improving solutions. Diversification will be applied using a greedy heuristic when the frequency of non-improving solution exceeds a certain threshold. **Appendix A** summarizes our experimental results. *TSF Results* are results obtained by TSF; *Best Results* are the best-published results.

9.5.2 Quadratic Assignment Problem (QAP)

In QAP, the *Solution* can be represented as a permutation on a set of *n* facilities. A typical move often involves swapping two elements in the solution. [Ahuja et al., 2002] proposed a VLSN strategy for QAP, which implements complex moves involving multiple swaps. Generally, VLSN

produces better solutions than typically swap move. Following the authors' proposal, we demonstrate that TSF is capable supporting this strategy through the use of two software components: *PermutatorGenerator* and *NeighborhoodBooster*. The *PermutatorGenerator* is used to construct the neighborhood from a solution by generating all the possible permutations. The *NeighborhoodBooster* then performs two further steps to generate a larger neighborhood. First, a selection criterion is used to accept only elite neighbors. These elite neighbors are then further permutated to result in more neighbors. The two steps are repeated for k times (and thus known as k-opt). The tabu list in this case, records the previously visited solution and has a tenure of $0.6n$, where n is the number of facilities.

We conducted our experiments on a set of test cases taken from the QAPLib [Burkard et al., 1991], and the results are summarized in **Appendix B**. *Gap* is calculated as *Gap = (Best Result – TSF Result)/Best Result *100%.* From the table, we can see that TSF performs well for most of test cases. In Chr test cases, except for Chr20a and Chr25a, the results obtained are optimal; the results for other test cases are within a small gap from optimality.

9.6 CONCLUSION

In this paper, we presented TSF, a C++ object-oriented framework for tabu search. TSF imposes no restriction on the domain representation and yet provides a well-defined programming scheme for customizing the search process. TSF differs from other frameworks in that it offers users the flexibility of incorporating various tabu search strategies through the *Event Controller* as the centralized control mechanism without compromising too much on the run-time efficiency and solution quality. The *Strategies Software Library* further supports the development of enhancing solution quality. Through the implementations of TSF on VRPTW and QAP, we illustrate that good results can be obtained with the framework within reasonable implementation time as well as good run-time.

TSF is one component within the Meta-Heuristics Development Framework (MDF) [Lau et al., 2004] that is work-in-progress. MDF encompasses other meta-heuristics such as the Simulated Annealing, Genetic Algorithms, and Ants Colony Framework. MDF aims to provide a generic, robust and user-friendly programming framework for promoting the development and integration of meta-heuristics algorithms. It also provides a platform for researchers to equitably compare and contrast meta-heuristics algorithms.

ACKNOWLEDGEMENT

The authors would like to thank the referees of the 5th Meta-Heuristic International Conference, and the referees of this journal version.

APPENDIX A:

Experimental results on VRPTW test cases

Test Cases	Best Results	Published by	TSF Results
R101	19/1650.80	RT	19/1686.24
R102	17/1486.12	RT	18/1493.31
R103	3/1292.85	HG	14/1301.64
R104	10/ 982.01	RT	10/1025.38
R105	14/1377.11	RT	14/1458.60
R106	12/1252.03	RT	12/1314.69
R107	10/1113.69	CLM	10/1140.27
R108	9/ 964.38	CLM	10/ 994.66
R109	11/1194.73	HG	12/1207.58
R110	10/1124.40	RGP	11/1166.65
R111	10/1096.72	RGP	11/1172.66
R112	9/1003.73	HG	10/1041.36
R201	4/1252.37	HG	4/1366.34
R202	3/1191.70	RGP	3/1239.22
R203	3/ 942.64	HG	3/1000.29
R204	2/ 849.62	CLM	3/ 781.86
R205	3/ 994.42	RGP	3/1063.29
R206	3/ 912.97	RT	3/ 955.34
R207	2/ 914.39	CR2	3 /866.35
R208	2/ 731.23	HG	2/1016.07
R209	3/ 909.86	RGP	3/ 979.30
R210	3/ 955.39	HG	3/ 968.32
R211	2/ 910.09	HG	3/ 865.51
RC101	14/1694.94	TBGGP	15/1698.50
RC102	12/1554.75	TBGGP	13/1551.32
RC103	11/1262.02	RT	11/1371.40
RC104	10/1135.48	CLM	10/1187.97
RC105	13/1633.72	RGP	14/1618.01
RC106	11/1427.13	CLM	12/1434.33
RC107	11/1230.54	TBGGP	11/1266.92
RC108	10/1139.82	TBGGP	10/1273.12
RC201	4/1406.94	CLM	4/1445.00
RC202	3/1389.57	HG	4/1204.45
RC203	3/1060.45	HG	3/1091.71
RC204	3/ 799.12	HG	3/ 826.27
RC205	4/1302.42	HG	4/1469.25
RC206	3/1153.93	RGP	3/1259.12
RC207	3/1062.05	CLM	3/1127.19
RC208	3/ 829.69	RGP	3/ 937.78

Legend:

CR2 W. Chiang and R. A. Russell, A Reactive Tabu Search Metaheuristic for Vehicle Routing Problem with Time Windows, *INFORMS Journal on Computing*, *8:4*, 1997

CLM J. F. Cordeau, G. Laporte, and A. Mercier, "A Unified Tabu Search Heuristic for Vehicle Routing Problems with Time Windows," *Journal of the Operational Research Society* 52, 928-936, 2001

HG J. Homberger and H. Gehring, "Two Evolutionary Metaheuristics for the Vehicle Routing Problem with Time Windows," *INFOR*, *37, 297-318*, 1999

RT Rochat, Y. and E. Taillard, Probabilistic Diversification and Intensification in Local Search for Vehicle Routing, *Journal of Heuristics, 1, 147-167*, 1995

RGP L.M. Rousseau, M. Gendreau and G. Pesant, "Using Constraint-Based Operators to Solve the Vehicle Routing Problem with Time Windows," *Journal of Heuristics*, *8, 43-58,* 1999

TBGGP E. Taillard, P. Badeau, M. Gendreau, F. Geurtin, and J.Y. Potvin, "A Tabu Search Heuristic for the Vehicle Routing Problem with Time Windows," *Transportation Science, 31, 170-186*, 1997

APPENDIX B:

Experimental results on QAP test cases

Test cases	Best results	TSF results	Gap
Chr12a	9552	9552	0.0
Chr12b	9742	9742	0.0
Chr12c	11156	11156	0.0
Chr15a	9896	9896	0.0
Chr15b	7990	7990	0.0
Chr15c	9504	9504	0.0
Chr18a	11098	11098	0.0
Chr18b	1534	1534	0.0
Chr20a	2192	2222	1.5
Chr20b	2298	2298	0.0
Chr20c	14142	14142	0.0
Chr22a	6165	6165	0.0
Chr22b	6194	6194	0.0
Chr25a	3796	3920	4.0
Bur26a	5426670	5432488	0.1
Bur26b	3817852	3824458	0.1
Bur26c	5426795	5427731	0.0
Bur26d	3821225	3821275	0.0
Bur26e	5386879	5387728	0.0
Bur26f	3782044	3782246	0.0
Bur26g	10117172	10118787	0.0
Bur26h	7098658	7098658	0.0
Nug12	578	578	0.0
Nug14	1014	1016	0.2
Nug15	1150	1152	0.2
Nug16a	1610	1610	0.0
Nug16b	1240	1240	0.0
Nug17	1731	1742	0.6
Nug18	1930	1930	0.0
Nug20	2570	2570	0.0
Nug21	2438	2456	0.7
Nug22	3596	3596	0.0
Nug24	3488	3500	0.3
Nug25	3744	3744	0.0
Nug27	5234	5348	2.2
Nug30	6124	6128	0.0
Sko42	15812	16020	1.3
Sko49	23386	23486	0.4
Sko56	34458	34664	0.6
Sko64	48498	48838	0.7
Sko72	66256	66702	0.7
Sko90	115534	116168	0.5
Sko100a	152002	153610	1.1
Sko100b	153890	155318	0.9
Sko100c	147862	149036	0.8
Sko100d	149576	151756	1.5
Sko100e	149150	150996	1.2
Sko100f	149036	150906	1.3

Legend:

Chr: N. Christofides and E. Benavent. An exact algorithm for the quadratic assignment problem. *Operations Research*, *3:5, 760-768*, 1989

Bur: R.E. Burkard and J. Offermann. Entwurf von Schreibmaschinentastaturen mittels quadratischer Zuordnungsprobleme. *Zeitschrift für Operations Research*, *21, B121-B132*, 1977

Nug: C.E. Nugent, T.E. Vollman, and J. Ruml. An experimental comparison of techniques for the assignment of facilities to locations. *Operations Research, 16, 150-173,* 1968

Sko: J. Skorin-Kapov. Tabu search applied to the quadratic assignment problem. *ORSA Journal on Computing, 2:1, 33-45,* 1990

REFERENCES

[Ahuja et al., 2002] R. K. Ahuja, J. B. Orlin, O. Ergun, and A. Punnen A Survey of Very Large-Scale Neighborhood Search for the Quadratic Assignment Problem, *Discrete Applied Mathematics 23, 75-102*, 2002.

[Burkard et al., 1991] R. E. Burkard, S.E. Karisch and F. Rendl. QAPLIB - A Quadratic Assignment Problem Library, *European Journal of Operational Research, 55:99, 115-119*, 1991.

[Fink and Voß, 2002] A. Fink, S. Voß: HotFrame: A Heuristic Optimization Framework. *In: S. Voß, D.L. Woodruff (Eds.), Optimization Software Class Libraries, Kluwer, Boston, 81-154*, 2002.

[Glover and Laguna, 1997] F. Glover and M. Laguna, Tabu Search, *Reading, Kluwer Academic Publishers, Boston/Dorderecht/London*, 1997.

[Gaspero and Schaerf, 2001] L. Di Gaspero and A. Schaerf, EasyLocal++: An object-oriented framework for flexible design of local search algorithms, *Reading, Kluwer Academic Publishers*, 2001.

[Harder, 2003] R. Harder, IBM OpenTS Homepage, see http://opents.iharder.net, 2003.

[ILOG, 2003] ILOG S.A. www.ilog.com, 2003.

[Lau et al., 2004] H. C. Lau, M. K. Lim, W. C. Wan and S. Halim. A Development Framework for Rapid Meta-heuristics Hybridization, *Proc.*

28th Annual International Computer Software and Applications Conference (COMPSAC), *362-367*, Hong Kong, 2004.

[Michel and Hentenryck, 1999] L. Michel and P. Van Hentenryck. Localizer++: A modeling language for local search. *INFORMS Journal of Computing, 11, 1-14*, 1999.

[Solomon, 1987] M. M Solomon. Algorithms for Vehicle Routing and Scheduling Problem with Time Window Constraints, *Operations Research 35, 254 – 265*, 1987.