A Generic Object-Oriented Tabu Search Framework

Hoong Chuin Lau Wee Chong Wan Xiaomin Jia
School of Computing, National University of Singapore
{lauhc, jwan, jiaxiaom}@comp.nus.edu.sg

Extended Abstract

1 Introduction

Recent studies have reported many successful applications of tabu search in solving optimization problems. In most cases, tabu search applications are designed specifically to the problems, because different strategies are required to solve different problems. This leads to little reuse of developed applications. Conceivably, the development cost can 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 sophistication in allowing user-defined search strategies [Glover and Laguna, 1997] to be incorporated. To date, there is a lack of a widely accepted tabu search framework and only few research prototypes have gained limited popularity.

We propose a generic 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 and testing algorithms by minimizing efforts in programming.

2 Literature Survey

Harder [Harder, 2001] developed a Java-based tabu search framework *OpenTS* under the Common Optimization Interface for Operations Research (COIN-OR) initiative. His framework follows the object-oriented programming style and has a well-defined structure. The framework allows the definition of basic elements common to all tabu searches through interfaces and performs iterations based on these elements. The key elements are the solution structure, objective function, tabu list, move and move manager. OpenTS, however, restricts users to implement their strategies through the interfaces. This often leads to an inconvenient of implementing strategies that require interaction with one or more interfaces.

[Gaspero and Schaerf, 2000] proposed a local search framework EASYLOCAL++, which is not restricted to tabu search but comprises of a set of cooperating classes of local searches such as simulated annealing. The classes in EASYLOCAL++ are divided into six categories: Data classes, Helpers, Runners, Kicker, Solvers and Testers. *Data classes* store data, including the states of the search space, the moves and the input/output data. *Helpers* perform actions related to some specific aspects of the search such as the generation of neighborhood and prohibition of moves. *Runners* are the algorithmic core of EASYLOCAL++ responsible for performing a run of a local search. *Kickers* are used as a form of diversifying technique. *Solvers* control the search by generating the initial solutions, deciding how and in which sequence the *Runners* are to be activated. *Testers* represent a simple predefined interface of the user program. However, as with many other similar frameworks, EASYLOCAL++ does not cater an abstract class that could support user-defined strategies, and in our opinion, is a serious limitation as much of the development in meta-heuristics lies in the strategies involved.

[Andreas et al., 1998] proposed another generic tabu search - simulated annealing framework known as the Heuristic OpTimization FRAMEwork (HOTFRAME) that is based primarily on the use of template in C++ to obtain genericity. The authors suggest some potential advantages of their design such as run-time efficiency and enhanced decoupling of components that would lead to a black box reuse. Basically HOTFRAME can be divided into three templates, the *Solution Class*, the *Heuristic Class* and the *Neighborhood Iterator*. Unfortunately, although HOTFRAME can be used in most generic problems, it lacks the capabilities that allow developers to adaptively guide the search process.

In summary, the abovementioned frameworks either do not provide any control mechanism [Andreas et al., 1998, Gaspero and Schaerf, 2000] or implement decentralized controls that are attached to the abstract classes [Harder, 2001]. As most tabu search strategies affect one or more of the abstract classes, adding new strategies to a highly complex local search application may be difficult and tedious due to the scattered nature of such design. A centralized control mechanism, on the other hand, produces a better design, as it offers a more organized structure with all strategies implemented in a single abstract controller, which in turn, manages the rest of the abstract interfaces. Similarly none of the reviewed frameworks cater for a software library that supports the development of advanced search strategies.

3 Design and Architecture of TSF

TSF is composed of 4 categories of components: *interfaces*, *control mechanism*, *search engine,* and *strategies software library*. The *interfaces* define the basic components used by tabu search in an object-oriented architecture. The *control mechanism* is for advanced users to define rules that guide the search adaptively in response to events encountered. TSF eliminates the tedious routine task of programming the *search engine.* The TSF search engine interacts with the interfaces and collects information which is passed dynamically to the control mechanism to

re-adjust future search trajectory. Finally, TSF includes a *Strategy Software Library (SSL)* consisting of a set of software components to support various user-defined search strategies.

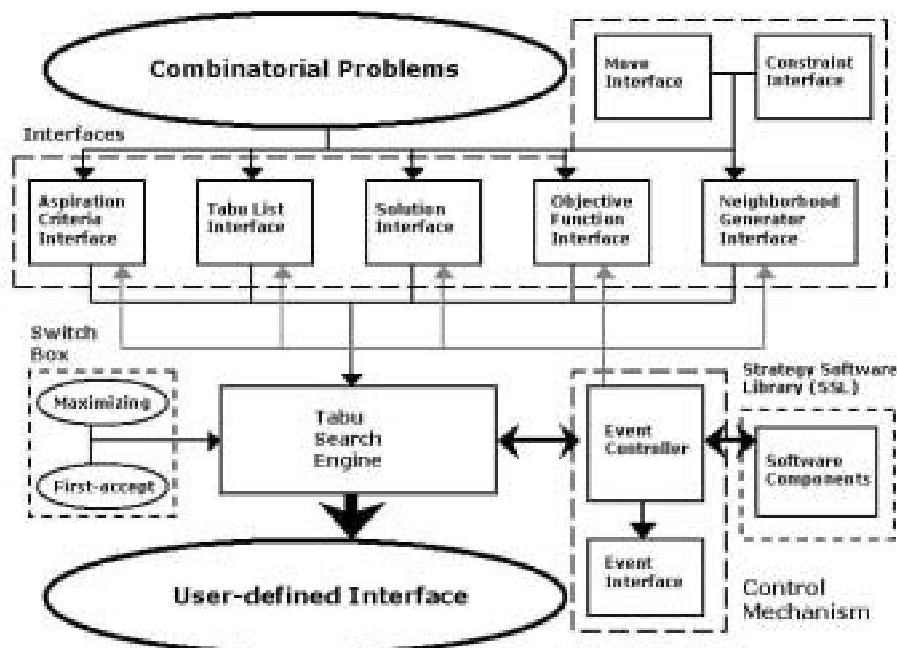

Figure 3.1: Tabu Search Framework Architecture

3.1 Interfaces

There are seven key interfaces that must be implemented. The *solution* interface is used as a representation for 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 the *Solution* objects directly. The *Objective Function* interface evaluates the objective value of the solution. The *Neighborhood* interface generates a list of neighbors with using *Move* and *Constraint* interfaces. The *Move* interface translates a *Solution* object into a new solution while the *Constraint* interface checks on the degree of violation for each constraint of a solution. The *Tabu List* interface records the tabu-ed solutions or moves. The *Aspiration Criteria* interface allows specification of aspiration criteria for tabu-ed moves.

3.2 Control Mechanism

3.2.1 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 more 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.

3.2.2 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. *No moves generated* is triggered when the *Neighborhood Generator* could not generate any moves.
6. *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 point. Figure 3.2 gives an example of code segment for such implementation.

The use of the *Event* interface is not limited to applying strategies. In fact, the *Event* can be used to collect useful information that may be important subsequently. For instance, in implementing the *Frequency and Recency* strategy, it is required for the tabu search to record on the number of occurrences a sub-structure/configuration in the solution appeared throughout the search. For

example, in the Traveling Salesman Problem, if we discover that the solution goes through customer 4 to customer 3 very frequently and especially in most local optima, this may imply that the global optimal may contain the sub-route of X–4–3–X, where X refers to an arbitrary customer in the solution. In this case, we may even implement a soft constraint based on this observation, which in turn will reduce the size of the neighborhood generated in each iteration. By collecting more information during the search and adaptively by modifying our search strategy based on this information, a more intelligent search is resulted.

```
// 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 3.2: A code sample implementation of a user-defined event

3.3 Strategy Software Library (SSL)

To improve the search further, various advanced strategies may need to be incorporated into the basic tabu search engine. These strategies include *intensification among elite solutions*, *probabilistic diversification*, *candidate lists*, *and very large-scale neighborhood (VLSN)*. In TSF, *SSL* provides a set of software components to support each of these strategies. Although not as powerful as specific strategies tailored to 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*.

3.3.1 Probabilistic Diversification

Probabilistic diversification refers to diversifying search when it is caught in a local optimum. One simple diversification strategy is to randomly choose a portion of the solution to be reconstructed and fuse with an unchanged portion. *SSL* supports this strategy by providing a *Random Generator* that generates a portion of the solution that is to be removed according to a prescribed probability distribution (such as uniformly random distribution).

3.3.2 Intensification search on Elite solutions

SSL supports this strategy by storing a list of elite solutions during tabu search. 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.

3.3.3 Very Large-Scale Neighborhood (*VLSN*)

VLSN [e.g. Ahuja et al., 2003] works on the principle that in a large neighborhood, there is a higher probability of obtaining a much better solution in single iteration. 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.

3.3.4 Candidates List

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.

4 Experimentation on VRPTW

We developed a VRPTW application and benchmark on Solomon’s test instances [Solomon, 1987]. Five 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. 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.

5 Experimentation on 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., 2003] proposed a VLSN strategy for QAP, which implements complex moves involving multiple swap. 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). To facilitate our testing, a set of test cases (CHR, BUR, NUG, SKO) were used 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 works very well for most of test cases. In CHR set of 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.

6 Conclusion

We presented TSF, an object-oriented framework for tabu search. TSF imposes no restriction on the domain representation and yet provides a well-defined structure for controlling the search. 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. *Strategies Software Library* further supports the development of enhancing solution quality. Finally, 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.

Reference:

[Andreas et al., 1998] A. Fink, S. Voß: HotFrame: A Heuristic Optimization Framework. In: S. Voß and D.L. Woodruff (Eds.), Optimization Software Class Libraries, Kluwer, Boston (2002).

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

[Gaspero and Schaerf, 2000] L. Di Gaspero and A. Schaerf, EasyLocal++: An object-oriented framework for flexible design of local search algorithms, Kluwer Academic Publishers, 2001.

[Harder, 2001] R.Hearder, IBM OpenTS Homepate, http://opents.iharder.net, 2001.

[Ahuja et al., 2003] R. K. Ahuja, K. C. Jha, J. B. Orlin, D. Sharma, Very Large-Scale Neighborhood Search for the Quadratic Assignment Problem, *INFORMS*, 2003.

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

Appendix A

Table A: Performance comparison on Solomon's VRPTW problem set.

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

Legend:

CR2 W. Chiang and R. A. Russell, A Reactive Tabu Search Metaheuristic for Vehicle Routing Problem with Time Windows, *INFORMS Journal on Computing*, *Vol 8, No 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*, VOL. *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*, 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

Table B: Performance comparisons on QAP problem set.

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

Legend:

CHR: N. CHRISTOFIDES and E. BENAVENT. An exact algorithm for the quadratic assignment problem. *Operations Research*, *37-5:760-768*, 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