30 SOLVING THE PROGRESSIVE PARTY PROBLEM BY LOCAL SEARCH

Philippe Galinier and Jin-Kao Hao

LGI2P, EMA-EERIE, Parc Scientifique Georges Besse, F-30000 Nimes, France.
galinier@eerie.fr, hao@eerie.fr

Abstract:

The Progressive Party Problem (PPP) is a complex, constrained combinatorial optimization problem. The goal of the problem consists of finding assignments of resources to variables for a given number of time periods while satisfying a set of multiple constraints. So far, two quite different approaches have been tried to solve the problem: Integer Linear Programming and Constraint Programming. All attempts with Integer Linear Programming failed, while Constraint Programming obtained better results. In this work, we present a third approach based on Local Search. We show that this approach gives better results than the previous ones and constitutes a very effective alternative to solve the PPP. We investigate different techniques for solving the problem, in particular issues related to search space, cost function and neighborhood functions.

30.1 INTRODUCTION

The Progressive Party Problem is a complex, constrained combinatorial problem. It consists in assigning resources to several variables for a given number of time periods while satisfying a set of multiple constraints. The PPP is interesting because it is known to be extremely hard for some mathematical method and remains unsolved. Moreover, the problem possesses many non trivial, heterogeneous constraints.

The problem has been proposed by a yacht club in order to organize a *party* taking place during several successive time periods [2, 8]. There are some host boats that receive the crews of other boats. For each time period, each guest crew must visit one of the host boats while respecting the following constraints:

- Each guest crew moves to a different host at each time period.

- Two crews meet each other at most once.
- The capacities of the host boats must be respected: for each time period and each host boat, the sum of the sizes of the crews who visit this boat during this time period cannot be greater than the capacity of the host.

Given these constraints, the problem consists in finding an assignment of the boats to the guest crews that respects all the constraints for the maximal possible number of time periods. There exists in the literature a well-known instance coming directly from the initial organization problem. Moreover, the basic version of the problem was to find, for this instance, a consistent assignment for a number of time periods fixed to six. So far, two different approaches have been tried to solve the above mentioned benchmark instance of the problem: Integer Linear Programming (ILP) [2] and Constraint Programming (CP) [8]. All attempts with Integer Linear Programming failed since no solution was found for the basic problem of 6 time periods. On the contrary, CP proved to be a more successful approach because it found solutions not only for the basic problem, but also for 7 and 8 time periods[1]. However, it was unknown if a solution would exist for more time periods.

In this work, we investigate a third approach based on Local Search to tackle the PPP. To model the PPP, we follow the CP approach, *i.e.* we consider the PPP as a Constraint Satisfaction Problem (CSP) [9, 14]. Based on this modeling, we introduce a search space, a cost function and two different neighborhoods, and experiment them with some well-known local search meta-heuristics. We show that this approach allows us to find better results than previous methods for the benchmark instance. Indeed, we are able to find solutions for 9 time periods in several seconds of CPU time.

The paper is organized as follows. In the next section, we present an overview of solving constraint satisfaction problems using Local Search and we represent the PPP using the formalism CSP (Section 30.2). Then, we present the different components of our local search algorithms (Section 30.3) and computational results (Section 30.4). In the last section, we give some conclusions of the work.

30.2 LOCAL SEARCH FOR CSP

Constraint Satisfaction Problem (CSP)

A CSP [9] [14] is defined by a triplet $(X, \mathcal{D}, \mathcal{C})$ with:

- *a finite set X of n variables :* $X = \{x_1, \cdots, x_n\}$.
- *a set $\mathcal{D}$ of associated domains :* $\mathcal{D} = \{D_{x_1}, \cdots, D_{x_n}\}$. Each domain D_{x_i} specifies the finite set of possible values of the variable x_i.

[1] Better results were obtained very recently, see Section 30.4 for more discussions.

- *a finite set $\mathcal{C}$ of p constraints :* $\mathcal{C} = \{C_1, \cdots, C_p\}$. Each constraint is defined on a set of variables and specifies which combinations of values are compatible for these variables.

Given such a triplet, the problem consists in finding a complete assignment of the values to the variables that satisfies all the constraints: such an assignment is then said consistent. Since the set of all assignments (not necessarily consistent) is defined by the Cartesian product $D_{x_1} \times \cdots \times D_{x_n}$ of the domains, solving a CSP means to determine a particular assignment among a potentially huge number of possible assignments.

The CSP formalism is a powerful and general model. In fact, it can be used to model conveniently some well-known problems such as k-coloring and satisfiability as well as many practical applications related to resource assignments, planning or timetabling. As we will see below, the PPP is also easily represented as a CSP.

Local Search (LS) LS, also called neighborhood search, constitutes a powerful approach for tackling hard optimization problems [12]: given a couple (S, f) with S being a finite set of configurations and f a cost function $f : S \to R$, to determine an element $s_* \in S$ such that $f(s_*) = \min_{s \in S} f(s)$. To solve an optimization problem, Local Search needs a so-called neighborhood function $N : S \to 2^S$ ($N(s) \subseteq S$ is called the neighborhood of $s \in S$). A Local Search algorithm begins with an initial configuration $s_0 \in S$ and then generates a series of configurations $(s_i)_{i \in \{0,1,\cdots\}}$ such that $\forall i \in \{0, 1, \cdots\}$, $s_{i+1} \in N(s_i)$. Well-known examples of LS methods include various descent methods, simulated annealing (SA) [7] and Tabu Search (TS) [4, 5]. The main difference among LS methods concerns the way of visiting the given neighborhood.

Several LS methods, called repair methods, have been developed to solve CSPs in Artificial Intelligence. One well-known example is the Min-conflict (MC) heuristic [10]. With this heuristic, one seeks for a best value for a given conflicting variable at each iteration[2]. This method corresponds in fact to a special descent method and cannot go beyond a local optimum. This method has been enforced by a noise strategy called *random-walk* [13]. Other extensions or similar methods are reported in [6, 11, 15]. Recently, more advanced LS methods such as Tabu Search and Simulated Annealing are also used to solving constraint satisfaction problems [3].

In all these studies, a *configuration* is a complete assignment of values to variables and the *search space* is defined by the Cartesian product $D_{x_1} \times \cdots \times D_{x_n}$. Two configurations are *neighbors* if and only if they are different at the value of a single variable. The *cost* of a configuration is simply the number of violated constraints by the configuration (a cost of zero corresponds to a consistent assignment, thus a solution).

[2]A variable is said conflicting if it is involved in some unsatisfied constraints.

Formulation of the PPP as a Constraint Satisfaction Problem Recall the PPP consists in finding an assignment of the boats to the guest crews that respects all the constraints for the maximal possible number of time periods. Let T be the number of time periods, G the number of guest crews and H the number of host boats. We use $c(g)$ to denote the size of a guest crew $g \in \{1, \cdots, G\}$, $C(h)$ the capacity of a host boat $h \in \{1, \cdots, H\}$ and $x_{g,t} \in \{1, \cdots, H\}$ the host boat visited by the crew g during the time period t. In the benchmark instance, there are $H = 13$ host boats and $G = 29$ guest crews. Table 30.1 indicates the capacities of the boats and the sizes of the crews.

capacity C	$\#boats$	size c	$\#crews$
10	2	7	1
9	1	6	1
8	6	5	3
7	1	4	8
6	1	3	2
4	2	2	14
	13		29

Table 30.1 Capacities of the boats and sizes of the crews

Note however that an obvious upper bound for T is 10. Indeed, there is a guest crew of size 7 and only 10 hosts have a capacity greater or equal to 7. Hence only this 10 hosts can be assigned to this guest crew in order to respect the capacity constraint. Moreover, this guest crew must visit a different host at each time period. Hence, the number of time periods cannot exceed 10.

We use P_n to denote the problem of finding a consistent assignment for a fixed number n of time periods in the benchmark instance of the problem. The instance is naturally solved by solving P_n for the different possible values of n (from 1 to 10). For a fixed value of n, P_n can be modeled as a CSP $(X, \mathcal{D}, \mathcal{C})$ as follows. The set of variables is naturally $X = \{x_{g,t}, 1 \leq g \leq G, 1 \leq t \leq T\}$ and all domains are equal to $D = \{1, 2, \cdots, H\}$: $\forall x \in X, D_x = D$. The constraint set $\mathcal{C}$ contains the following three different types of constraints:

- The first type of constraints imposes that each crew moves to a different host at each time period. For each g, $1 \leq g \leq G$, we impose the constraint: DIFF(g) $\Leftrightarrow x_{g,1}, x_{g,2}, \cdots, x_{g,T}$ are all different.

- The second type of constraints imposes that two crews meet at most once: for each pair of guest crews, the number of time periods when these two crews meet is 0 or 1. For each pair $\{g_1, g_2\}$, we impose the constraint: ONCE($\{g_1, g_2\}$) $\Leftrightarrow |\{t, 1 \leq t \leq T \mid x_{g_1,t} = x_{g_2,t}\}| \leq 1$.

- The third type of constraints imposes that the capacity of the host boats must be respected. For each time period t, $1 \leq t \leq T$ and each host boat h, $1 \leq h \leq H$, we impose the constraint: CAPA(h,t) $\Leftrightarrow \sum_{1 \leq g \leq G, x_{g,t} = h} c(g) \leq C(h)$

Note that although each DIFF constraint may be replaced by a set of $|T| * (|T|-1)/2$ binary constraints of difference, the constraints of types ONCE and CAPA cannot be defined simply using binary constraints.

30.3 SOLVING THE PPP WITH LOCAL SEARCH

We define below the components of our local search procedures that are the search space, the cost function, the neighborhood function and the meta-heuristic.

30.3.1 Search space

As presented above, a configuration is a complete assignment of the numbers in $D = \{1 \cdots H = 13\}$ to the variables of $X = \{x_{g,t}, 1 \leq g \leq G, 1 \leq t \leq T\}$: a configuration is a $G \times T$ table (G=29 and T = 6,7,8,9 or 10) whose elements are integers from $\{1 \cdots 13\}$. The search space is the set of all such assignments.

30.3.2 Cost function

As stated in Section 30.2, the cost of a configuration of a CSP is often defined as the number of violated constraints and one associates to each constraint C a penalty $f_C(s)$ which takes 1 or 0 according to whether the constraint is violated or not for the configuration s. This 0/1 penalty function works generally well for binary constraints, but is no more appropriate for n-ary constraints. In fact, this function cannot distinguish a "strongly" violated constraint from a "weakly" violated one. In the PPP, constraints are more complex, this is why we introduce a more informative multi-valued penalty function for each constraint.

Penalty for a constraint of type DIFF Recall that $DIFF(g)$ is satisfied if and only if $x_{g,1}, x_{g,2}, \cdots, x_{g,T}$ are all different. For a constraint $DIFF(g)$, we define the penalty as the number of pairs $\{x_{g,t_1}, x_{g,t_2}\}$ of variables having the same value. That is, for any DIFF constraint $C = DIFF(g)$ and any configuration s, $f_C(s) = |\{\{x_{g,t_1}, x_{g,t_2}\} \mid s(x_{g,t_1}) = s(x_{g,t_2})\}|$, where $s(x)$ denotes the value assigned to x in s.

Penalty for a constraint of type ONCE Given a constraint of type ONCE $C = ONCE(\{g_1, g_2\})$, we use $Meet_{g_1,g_2}$ to denote the number of time periods when these two crews meet: $Meet_{g_1,g_2}(s) = |\{t, 1 \leq t \leq T \mid s(x_{g_1,t}) = s(x_{g_2,t})\}|$. Recall that the constraint is satisfied for s if and only if $Meet_{g_1,g_2}(s) \leq 1$. If the constraint is violated ($Meet_{g_1,g_2} > 1$), we define the penalty f_C to be equal to $Meet_{g_1,g_2} - 1$ in order to incite the value of $Meet_{g_1,g_2}$ to decrease progressively until the constraint becomes satisfied (*i.e.* $Meet_{g_1,g_2} = 1$).

$$f_C(s) = \begin{cases} 0 & \text{if } Meet_{g_1,g_2}(s) \leq 1 \\ Meet_{g_1,g_2}(s) - 1 & \text{otherwise} \end{cases}$$

Penalty for a constraint of type CAPA Given a constraint of type CAPA $C = CAPA(h,t)$, let $\sigma_{h,t}$ be the overloading of the host h for the time period t: $\sigma_{h,t}(s) = C(h) - \sum_{1 \leq g \leq G, s(x_{g,t})=h} c(g)$. When $\sigma_{h,t} \leq 0$, the penalty is naturally 0 because the constraint is satisfied. The penalty is fixed to 1 if $\sigma_{h,t} = 1$ and increases progressively for higher values of $\sigma_{h,t}$, in order to incite $\sigma_{h,t}$ to decrease: $f_C = 1 + B * (\sigma_{h,t} - 1)$. We fixed $B = 1/4$ empirically.

$$f_C(s) = \begin{cases} 0 & \text{if } \sigma_{h,t}(s) \leq 0 \\ 1 + (\sigma_{h,t}(s) - 1)/4 & \text{otherwise} \end{cases}$$

Remark Note that the penalty $f_C(s)$ for a constraint C of type ONCE is exactly the minimum number of variables that must be changed in s in order to satisfy the constraint C. It is also roughly the case for the constraints of type CAPA, considering that the average size of a crew is about 4 (3.24 more precisely). This principle is applicable to other n-ary constraints in order to define an effective penalty function.

Cost function Finally, the cost function is the weighted sum of the penalty functions of all the constraints $C \in \mathcal{C}$:

$$\forall s \in S, f(s) = \sum_{C \in \mathcal{C}} p(C) * f_C(s)$$

where p is a weight function $p : \mathcal{C} \rightarrow R$.

The weight for each of three types of constraints is determined empirically. The best weights found are 2, 1 and 2 for DIFF, ONCE and CAPA respectively - more precisely, the weights are 4, 2 and 4, in order to have integer values for f.

30.3.3 Conflicting variables

Many search methods give a different role to a variable depending on whether it is in conflict or not. If a constraint C is satisfied, no variable is conflicting for this constraint. But if C is violated, some variables are conflicting. The notion of conflicting variable in the PPP is defined as follows.

For a DIFF constraint $C = DIFF(g)$, variable x_{g,t_1} is in conflict in a configuration s if and only if there exists another variable x_{g,t_2} $(t_1 \neq t_2)$ such that $s(x_{g,t_1}) = s(x_{g,t_2})$.

For a constraint $C = ONCE(\{g_1, g_2\})$: variables $x_{g_1,t}$ and $x_{g_2,t}$ are conflicting for the constraint C if the constraint is violated $(Meet_{g_1,g_2}(s) > 1)$ and $s(x_{g_1,t}) = s(x_{g_2,t})$.

For a constraint $C = CAPA(h,t)$: variable $x_{g,t}$ is conflicting for the constraint C if, first, the constraint is violated $(\sigma_{h,t}(s) > 0)$ and, second, $s(x_{g,t}) = h$.

30.3.4 Definition of the neighborhood function

Recall that a neighborhood function is any function $N : S \to 2^S$. In order to simplify the presentation, we will use the notion of *move* to define a neighborhood. Applying a move m to a configuration $s \in S$ leads to a new configuration denoted by $s \oplus m$. Let $M(s)$ be the set of all possible moves which can be applied to s, then the neighborhood of s is defined by: $N(s) = \{s \oplus m \mid m \in M(s)\}$.

For the PPP, we introduce two quite different types of moves denoted by $OneMove$ and $Swap$. From these two types of moves, we define two neighborhoods denoted by N_1 and N_2.

Neighborhood N_1 A move of type *OneMove* consists simply in changing the host boat affected to a given crew for a given period, *i.e.* the current value of a single variable x is replaced by a new one v. Such a move is denoted by the couple $(x, v) \in X \times D$ where $s(x) \neq v$. The neighborhood N_1 is defined from $M_1 = OneMove$.

- $M_1(s) = OneMove(s) = \{(x, v) \in X \times D \mid s(x) \neq v\}$,
- $s' = s \oplus (x, v) \Leftrightarrow \begin{cases} s'(x) = v \text{ and} \\ \forall y \in X - \{x\}, s'(y) = s(y) \end{cases}$
- $N_1(s) = \{s \oplus (x, v) \mid (x, v) \in M_1(s)\}$.

Neighborhood N_2 A *swap* move consists in reversing the host boats assigned to two crews g_1 and g_2 during the same time period t. It is denoted by a couple $(x_{g_1,t}, x_{g_2,t})$. Applying the move $(x_{g_1,t}, x_{g_2,t})$ to s consists in assigning the value $s(x_{g_2,t})$ to $x_{g_1,t}$ and inversely the value $s(x_{g_1,t})$ to $x_{g_2,t}$. The neighborhood N_2 combines the moves of *OneMove* and *Swap*.

- $Swap(s) = \{(x_{g_1,t}, x_{g_2,t}) \mid s(x_{g_1,t}) \neq s(x_{g_2,t})\}$,
- $s' = s \oplus (x_{g_1,t}, x_{g_2,t}) \Leftrightarrow \begin{cases} s'(x_{g_1,t}) = s(x_{g_2,t}) \text{ and} \\ s'(x_{g_2,t}) = s(x_{g_1,t}) \text{ and} \\ \forall y \in X - \{x_{g_1,t}, x_{g_2,t}\}, s'(y) = s(y) \end{cases}$
- $M_2(s) = OneMove(s) \cup Swap(s)$,
- $N_2(s) = \{s' = s \oplus m \mid m \in M_2(s)\}$.

Candidate list based on conflicting variables A common heuristic used to make the search more efficient is to restrict the choice of a move to the subset of $M(s)$ involving variables which are conflicting in s. Such an heuristic can be seen as a particular *candidate list strategy*. For the PPP, this conflict-based subset, denoted by $M_{1,CFL}(s)$ and $M_{2,CFL}(s)$ for the 2 neighborhoods, is defined as follows.

- $M_{1,CFL}(s) = \{(x, v) \in M_1(s) \mid x \text{ is conflicting in s }\}$
- $M_{2,CFL}(s) = \{(x_1, x_2) \in M_2(s) \mid x_1 \text{ or } x_2 \text{ is conflicting in s }\}$

Data : θ *:* parameter, N *:* neighborhood
Result : the best configuration found
begin
  generate a random configuration s
  while *not Stop-Condition* **do**
    choose randomly a move $m \in M$
    $\delta := f(s \oplus m) - f(s)$
    if $\delta \leq 0$ **then** $s := s \oplus m$
    else with probability $e^{-\delta/\theta}$ **do** $s := s \oplus m$
end

Figure 30.1 The Metropolis algorithm

30.3.5 Meta-heuristics tested

We experimented mainly two meta-heuristics: a *Metropolis algorithm* and a *Tabu Search* algorithm. Both algorithms use the above mentioned candidate list strategy.

Algorithm of Metropolis Metropolis algorithm is a simplified version of Simulated Annealing [7] using a constant value for its temperature parameter. The algorithm begins with an initial configuration in the search space S and then performs a series of iterations. At each iteration, a single neighbor (or equivalently a single move) of the current configuration is randomly chosen and then a probabilistic criterion is performed in order to decide if this neighbor is accepted or not. The principle of the Metropolis algorithm is to accept any move that does not increase the cost function, and to accept, in a controlled manner, deteriorating moves. So the algorithm is not trapped in local optima.

The method has one parameter θ called *temperature*: the higher the value of the temperature, the easier degradations are accepted (a zero temperature corresponds to a simple descent algorithm as all deteriorating moves are refused while an infinite temperature is a random walk in the search space as all moves are accepted).

More precisely, suppose that the move m has been chosen at the current iteration. Then, one considers the difference $\delta = f(s \oplus m) - f(s)$ that represents the effect of the chosen move on the cost function. If the move m does not increase the cost function ($\delta \leq 0$), then it is always accepted. If it does ($\delta > 0$), the move is accepted with a probability $e^{-\delta/\theta}$ so that a bigger deterioration is more rarely accepted.

Tabu algorithm A typical TS procedure [4] begins with an initial configuration in the search space S and then proceeds iteratively to visit a series of locally best configurations following the neighborhood. At each iteration, a *best* move m is applied to the current configuration s even if $s' = s \oplus m$ does not improve the current configuration in terms of the value of the cost function.

Data : k *:* tabu tenure; N *:* neighborhood
Result : the best configuration found
begin
  generate a random configuration s
  initialize the Tabu list to empty
  while *not Stop-Condition* **do**
    choose a best move m such that m is not tabu or satisfies the aspiration criterion
    introduce during k iterations the couple $< x, s(x) >$ in the Tabu list for all modified variables $x \in X$
    $s := s \oplus m$
end

Figure 30.2 The tabu search algorithm

This iterative process may suffer from cycling and get trapped in local optima. To avoid the problem, TS introduces the notion of *Tabu lists*. A tabu list is a special short term memory that maintains a selective history $\mathcal{H}$, composed of previously encountered solutions or more generally pertinent attributes of such solutions. A simple TS strategy based on this short term memory $\mathcal{H}$ consists in preventing solutions of $\mathcal{H}$ from being reconsidered for next k iterations (k, called tabu tenure, is problem dependent). Now, at each iteration, TS searches for a best neighbor from this dynamically modified neighborhood *N(*$\mathcal{H}$*,s)*, instead of from *N(s)* itself. Such a strategy prevents Tabu from being trapped in short term cycling and allows the search process to go beyond local optima. Moreover, note that Tabu restrictions may be overridden under certain conditions, called *aspiration criterion*.

A move for the PPP corresponds to changing the value of one variable (*OneMove*) or two variables (*Swap*). When a variable x is involved in a move, its current value v is replaced by a new one v'. At this time, the couple $< x, v >$ is classified tabu for the next k iterations and that means that the value v is not allowed to be re-assigned to x during this period. Nevertheless, a tabu move leading to a configuration better than the best configuration found so far is always accepted (*aspiration criterion*).

More precisely, a move $(x, v) \in OneMove$ is declared tabu iff the couple $< x, v >$ is tabu. A move $(x_{g_1,t}, x_{g_2,t}) \in Swap$ is declared tabu iff $< x_{g_1,t}, s(x_{g_2,t}) >$ or $< x_{g_2,t}, s(x_{g_1,t}) >$ is tabu.

Remarks: *Stop-Condition:* the algorithm stops if $f(s) = 0$ or if a given limit is reached concerning the time, the number of iterations or the number of moves.

We also tested a simple *Descent Method*. This algorithm chooses at each iteration a best possible neighbor that does not degrade the performance and hence it is easily trapped in a local optimum.

The efficiency of Tabu is greatly influenced by the ability of finding quickly a best move at each iteration. For moves of type $OneMove$, we use a data structure that permits to find in constant time the performance of a given move. This data structure is updated each time a move is performed.

30.4 COMPUTATIONAL RESULTS

In this section, we first compare the best results of Local Search with the best results obtained by Integer Linear Programming (ILP) and Constraint Programming (CP) for the PPP. Then we present and analyze in detail the results obtained by our Local Search procedures (LS).

30.4.1 Comparison of LS, ILP and CP

Table 30.2 presents the best known results of the three methods ILP, CP and LS. The published results of CP are denoted by CP1. From the table, we see that ILP fails to solve any of P_6 to P_{10}. CP1 solves P_7 and P_8 in 27 and 28 minutes respectively (using a SPARCstation IPX), but fails to solve P_9 and P_{10}. Recently, new strategies using CP were reported leading to much better results (CP2) [1]. Indeed P_7 and P_8 are now solved in a few seconds (after several hundreds of backtracks) by CP2. The problem P_9 is also solved, but using several hours (and millions of backtracks). Using LS we solve the problem up to 9 time periods [3] and fail for 10 periods. More detailed results will be presented below.

problem	ILP	CP1	CP2	LS
P_6	fail	27 min.	a few sec.	$<$ 1 s.
P_7	fail	28 min.	a few sec.	$<$ 1 s.
P_8	fail	fail	a few sec.	1 s.
P_9	fail	fail	hours	4 s.
P_{10}	fail	fail	fail	fail

Table 30.2 Results of the ILP, CP and Local Search for the PPP

30.4.2 Results of Local Search

We tested the two LS algorithms presented in Section 30.3 (algorithm of Metropolis denoted by Mt and Tabu algorithm denoted by TS) and used the two neighborhoods N_1 and N_2. So we tested four different procedures (denoted by Mt-N_1, TS-N_1, Mt-N_2 and TS-N_2). With these four procedures, we tried to solve the increasingly difficult problems from P_6 up to P_{10}. All these four procedures solved P_9 (and easier problems) but none of them could solve P_{10}. Note however that we don't know if P_{10} has a solution. Our local search

[3]The running time is obtained on a Sun ULTRA 1 (128 RAM, 134MHz). The LS algorithms are implemented in C++.

algorithms give repeatedly solutions violating a single constraint (of any of the three types) for P_{10}. Table 30.3 presents the results obtained by these four procedures for P_7, P_8 and P_9.

Each procedure was run 20 times. For each problem, the left column indicates the average number of moves ($\#moves$) and of iterations ($\#iter$) and the right column the average computing time (in seconds). Recall that for Tabu, each iteration leads to a move (hence, $\#moves = \#iter$), while for Metropolis several iterations are needed to lead to a move. Each method has been run with different values of its parameter on the problem P_9. The best value found has been used for the two other problems. These values are $\theta = 130$ for Mt-N_1 and Mt-N_2, $k = 9$ for TS-N_1 and $k = 6$ for TS-N_2.

	P_7		P_8		P_9	
	$\#moves/\#iter$	time	$\#moves/\#iter$	time	$\#moves/\#iter$	time
Mt-N_1	1,228/38,438	1.6	2,635/111,628	4.3	24,458/2,064,295	66.4
TS-N_1	330	0.5	1,366	1.7	51,507	67.5
Mt-N_2	494/5,724	0.6	682/11,582	1.0	1,488/63,685	3.6
TS-N_2	110	2.0	171	3.3	347	6.5

Table 30.3 Results of Local Search procedures for solving the PPP

From Table 30.3, we first note that all methods can solve the problems P_7 and P_8 very easily (in less than 5 seconds and 3,000 moves). P_9 can be solved very easily using the enlarged neighborhood N_2. Using N_1 to solve P_9 needs more effort (more than one minute for Mt-N_1 and TS-N_1). Moreover, we observe that the results obtained do not strongly depend on the method used since the number of moves and computing time are often similar when the two methods are used to solve the same problem using the same neighborhood.

method	problem	success rate	method	problem	success rate
Desc-N_1	P_3	26	Desc-N_2	P_5	93
	P_4	9		P_6	52
	P_5	3		P_7	11
				P_8	1

Table 30.4 Results of the Descent Method

Table 30.4 shows results obtained by the two Descent procedures with the two different neighborhoods (denoted by Desc-N_1 and Desc-N_2). Each procedure is run 100 times for each problem. The table indicates the number of success for the 100 runs. Each run corresponds to a single execution of the algorithm (there is no retry) and the algorithm is stopped after 1,000 iterations without improvement of the cost function. We observe that the Descent Algorithm can solve the PPP up to 5 time periods with the neighborhood N_1 and up to 8 time periods with N_2. These results illustrate the power of Local Search for the problem in that a problem that a simple descent method can solve can be estimated very easy for Local Search.

30.4.3 Comparisons between the two neighborhoods

From the above results, we observe a big difference of performance with the two neighborhoods. We compare now precisely the performance of the two neighborhoods. Concerning the number of iterations of TS, we observe that the ratio between the two neighborhoods is of 1 to 3 for P_7 (330 moves using TS-N_1 and 110 moves using TS-N_2), but becomes 8 and 148 for P_8 and P_9. This means that this ratio increases when the problem becomes more difficult and, in particular, the ratio increases dramatically for P_9.

Concerning computing time, we can compute from Table 30.4 that an iteration using the enlarged neighborhood N_2 costs (for Tabu) 12 times more than using N_1. Although P_7 and P_8 are solved more quickly using the limited neighborhood, the advantage turns strongly to the enlarged neighborhood for the most difficult problem P_9.

Moreover N_2 allows a simple descent method to solve up to P_8 while N_1 cannot go beyond P_5. Therefore the enlarged neighborhood N_2 is more powerful than N_1 especially for solving difficult problems. Experiments on P_{10} confirm this remark, as the two algorithms TS and Mt can easily find solutions that violate a single constraint using N_2, but not using N_1. So we can hope that an improved procedure using N_2 will find a solution for P_{10}, if there exists one.

	[0..5[	[5..10[	[10..15[	[15..20[	[20..25[	[25..30[	[30..35[
$\psi_1(\%)$	1.7	1.4	15.6	37.3	42.3	42.8	52.0
$\psi_2(\%)$	10.3	41.1	51.2	60.6	67.1	67.9	69.5

Table 30.5 Percentage of configurations which are not local optima for N_1 and N_2

Now we present another measure in order to explain the power difference of the two neighborhoods. We observed for the PPP a well-known phenomenon: starting with a high value of the cost function, the cost value decreases very quickly at the beginning of the search and then oscillates when the search becomes more and more difficult. The reason of the initial fast decrease of the cost function is that most of configurations have at least one neighbor which has a smaller cost: these configurations are not local optima for the neighborhood considered. On the contrary, many configurations encountered at the end of the search are local optima. So, the percentage of local optima tends to be related to the difficulty of the search. To study more precisely this point, the following experiment was carried out.

We used a local search procedure to solve P_9. At each iteration we tested if the current configuration is a local optimum for the two neighborhoods N_1 and N_2. Using this information, we computed the percentages $\psi_1(f)$ and $\psi_2(f)$ of configurations of cost f that are not local optima for N_1 and N_2 respectively. Table 30.5 gives a summary of this experiment. The different columns represent the values of the cost function grouped in classes. For example, the first column means that 1.7% and 10.3% of the encountered configurations having a cost of 0, 1, 2, 3 or 4 are not local optima for N_1 and N_2 respectively.

For high values of f, i.e., at the beginning of the search, we observe that the percentage is high for both neighborhoods: for example, for $f \in [30, 35[$, this percentage is about 50% and 70% for N_1 and N_2 respectively. Hence it is generally easy to find a move that improves the current configuration, using indifferently one of the two neighborhoods. For smaller values of f, this percentage decreases dramatically for N_1, while it remains much higher for N_2. For example, for $f \in [0, 4[$, $\psi_1(f) = 1.7\%$ and $\psi_2(f) = 10.3\%$. Hence there are much more useful improving moves in N_2 than in N_1. These results are averages computed on 100 runs performed with the algorithm TS-N_2 on the problem P_9, but experiments carried out with the other LS procedures show similar properties.

Besides, we note that N_2 is about only two times larger than N_1: in P_9, the size of N_1 is $29*9*(13-1) = 3132$ while the size of N_2 is $3132+9*(29*28/2) = 3132 + 3654 = 6876$.

In summary, we see that the neighborhood N_2 contains many solutions of high quality that are not present in N_1 while its size is about only two times larger than N_1. This is probably an important factor that explains the observed difference of efficiency between the two neighborhoods.

30.5 CONCLUSIONS

In this paper we have presented a local search approach for the Progressive Party Problem. Based on a formulation of the PPP as a Constraint Satisfaction Problem, the proposed approach introduces two different neighborhoods and penalty-based cost function for handling complex and heterogeneous constraints.

The approach was tested on the available benchmark instance. Results were compared with previous ones obtained with Integer Linear Programming and Constraint Programming. Numerical experiments showed that both a simple tabu algorithm and a metropolis algorithm give better results since they solve the problem up to 9 time periods in only several seconds of cpu time. Even a descent is successful for up to 8 time periods if the swap neighborhood is used. Therefore, LS should be considered as one of the most competitive approaches for the PPP.

Until now, all reported studies on the PPP concern a particular instance arising form the initial party organization problem. It is now natural and interesting to know how different approaches will behave for other instances of the problem.

Acknowledgements

We would like to thank the referees of this paper for their useful comments.

References

[1] N. Beldiceanu, E. Bourreau, P. Chan and D. Rivreau, Partial Search

Strategy in CHIP, *presented at MIC'97*, Sophia-Antipolis, 1997.

[2] S. C. Brailsford, P. M. Hubbard and B. M. Smith, The Progressive Party Problem: A Difficult Problem of Combinatorial Optimization, *Computers and Operations Research*, 23:845-856, 1996.

[3] P. Galinier and J. K. Hao, Tabu Search for Maximal Constraint Satisfaction Problems, *Lecture Notes in Computer Science 1330*, pp196-208, 1997.

[4] F. Glover and M. Laguna, Tabu Search, *USA, Boston: Kluwer Academic Publishers*, 1997.

[5] P. Hansen and B. Jaumard, Algorithms for the Maximum Satisfiability Problem, *Computing*, 44:279-303, 1990.

[6] J. K. Hao and R. Dorne, Empirical Studies of Heuristic Local Search for Constraint Solving, *Lecture Notes in Computer Science 1118*, pp194-208, 1996.

[7] S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi, Optimization by Simulated Annealing, *Science*, 220:671-680, 1983.

[8] B. M. Smith, S. C. Brailsford, P. M. Hubbard and H. P. Williams, The Progressive Party Problem: Integer Linear Programming and Constraint Programming Compared, *Constraints*, 1(1/2):119-138, 1996.

[9] A.K. Mackworth, Constraint Satisfaction, *in S.C. Shapiro (Ed.) Encyclopedia on Artificial Intelligence, John Wiley & Sons, NY*, 1987.

[10] S. Minton, M.D. Johnston and P. Laird, Minimizing Conflicts: a Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems, *Artificial Intelligence*, 58(1-3):161-206, 1992.

[11] P. Morris, The Breakout Method for Escaping from Local Minima, *Proc. of AAAI-93*, pp40-45, 1993.

[12] C.H. Papadimitriou and K. Steiglitz, Combinatorial Optimization - Algorithms and Complexity, *Prentice Hall*, 1982.

[13] S. Selman, H.A. Kautz and B. Cohen, Noise strategies for improving local search, *Proc. of AAAI-94*, pp337-343, Seattle, WA, 1994.

[14] E. Tsang, Foundations of Constraint Satisfaction, *Academic Press*, 1993.

[15] N. Yugami, Y. Ohta and H. Hara, Improving Repair-based Constraint Satisfaction Methods by Value Propagation, *Proc. of AAAI-94*, pp344-349, Seattle, WA, 1994.