Two Different Approaches of Modeling the Teacher Relocation Problem in a Constraint Logic Programming System and Their Comparison

Nagehan Ilhan, Zeki Bayram
Computer Engineering Department, Eastern Mediterranean University
Famagusta, Turkish Republic of Northern Cyprus
{nagehan.ilhan, zeki.bayram}@emu.edu.tr

Abstract—**We propose two ways of modeling the teacher relocation problem in the ECL^{i}PS^e constraint logic programming system, implement them and compare their performance through simulation. The first one relies on logic variables that take on boolean values. The second one involves the use of the built-in predicates "element" and "occurrences." Simulations confirm that the approach that uses the built-in predicates performs better in general.**

I. INTRODUCTION

In [1] we defined the "teacher relocation problem," and proposed a model for it in the constraint logic programming language ECL^{i}PS^e [4] that made use of boolean-valued logic variables. The teacher relocation problem concerns school teachers wishing to relocate to other schools and who state their preferences in terms of which schools they want to move to. The problem, in simple terms, is to come up with a relocation scheme in which a utility function (the "happiness" of teachers) is maximized while not violating quota constraints of schools. This is an instance of the class of discrete optimization problems, and requires exponential search for its solution.

In this paper we improve upon our previous solution by adapting a different approach to modeling the problem, which makes use of two high-level built-in constraint predicates that are available in ECL^{i}PS^e, namely the "element" predicate and the "occurrence" predicate. We compare the two solutions by running simulations using both models. We conclude that using high-level built-in constraint solving predicates is advantageous in this case compared to the previous solution of using Boolean-valued logic variables.

Constraint Logic Programming (CLP) is indeed very convenient for both modeling and solving discrete optimization problems, since the problem can be formulated as a set of variables and constraints between these variables. The goal is to find an assignment of values to variables such that none of the constraints are violated. There can be a cost function which has to be minimized/maximized in the final solution. Once the model is set up, the built-in search and constraint solving mechanisms of the CLP system take care of the rest [2][3][5].

The remainder of this paper is as follows. In section II, we formally define teacher relocation problem. In section III we model it using two different strategies in ECL^{i}PS^e, the first strategy being the one described in [1]. We then test both models under different scenarios in section IV. Results show that the modeling approach using built-in predicates out-performs the other one. In section V we have a brief survey of related work and finally in section VI we have the conclusion and future research directions.

II. DESCRIPTION, FORMAL SPECIFICATION AND TIME COMPLEXITY OF THE TEACHER RELOCATION PROBLEM

Each teacher who wishes to relocate can state two choices in order of preference. An implicit, but undesirable third choice on the part of the teacher is to stay in his/her current school. Teachers who do not wish to relocate do not make any choices, and are not forcibly relocated.

Teachers state their choices in order of preference. A specific teacher's happiness is given very naturally by *order(teacher, assignment(teacher))*, where *order(t,s)* is the position (first, second etc.) of school *s* in the preference declaration of teacher *t*. Here, smaller numbers mean more happiness.

The goal of the system is to maximize the overall happiness of teachers, which corresponds to minimizing the value of the

objective function

$$\sum_{t \in TeachersWhoWishToRelocate} order(t, assignment(t)) \quad \textbf{(1)}$$

subject to the constraint in each school that the number of teachers at the school never exceeds its capacity (understaffing is a possibility, and occurs frequently in North Cyprus). This is modeled by the use of quotas, which specify the teacher deficiency (vacant positions) in each school before relocation is initiated. So for each school, the following constraint must be satisfied in a final solution:

$$incomingTeachers \leq outgoingTeachers + quota \quad \textbf{(2)}$$

Another obvious constraint is that a teacher cannot be assigned to more than one school, hence the function *assignment*.

For N teachers, the best case time complexity is N, where each teacher is assigned to his/her first choice without violating any constraints. For the worst case analysis, all possible combinations of assignments must be considered, which gives us 3^N.

III. TWO DIFFERENT MODELS USING CLP

In this section, two different models for the problem specified in Section II are presented, using the ECL^{i}PS^e Constraint Programming Language. We start with the data structures used to represent teachers, their choices and school information.

A. Data Representation

To store the information about teachers, we use the structure *teacher(Teachername,* ALSchool*)* for each teacher. *Teachername* is the name of the teacher and *ALSchool* is the current school of the teacher. Each teacher name is assumed to be unique.

A teacher's choices are represented as *choice(Teachername, SchoolName, ChoiceNo, AppYear),* where *Teachername* is the name of the teacher, *SchoolName* specifies the name of the school which the teacher wants relocate to, *ChoiceNo* represents the order of preference of this school and *AppYear* is the application year of the teacher for relocation. So, *choice('Geoge Washington', 'Talented Kids High School', 1, 2005)* would mean that George Washington's first choice would be to move to Talented Kids High School, and he first made his application to relocate in 2005.

Information about schools is given by *school(Schoolname, Countyname, Quota, MaxCapacity),* where *SchoolName* is the name of the school, *Countyname* specifies the county which the school belongs to, *Quota* is the number of vacancies at the school, and *MaxCapacity* specifies the maximum number of teachers which the school can employ.

```
solve(Cost):-
   findall(X,teacher(X,_),Teacherlist),
   increment_loop(Teacherlist,1,Cost),!.
```

Fig. 1. The "solve" predicate

Note that not all the information represented in the database is used in the solutions that we present, but they are included to allow the possibility of different utility functions in the future.

B. Techniques Common to Both Approaches

An incremental strategy is adopted in both approaches. First, an assignment is tried where only the first choice of each teacher is taken into account. If this does not yield a solution, then the first two choices are considered. If still no solution is generated, then the first three choices are considered (the third choice being implicit – stay at your own school, i.e. do not relocate). Our simulations in [1] confirm the advantage of using this incremental approach.

Another technique common to both approaches is the usage of a "base value" for a solution. A base value of the utility function is found for just *any* relocation scheme which does not violate the constraints. If a partial solution results in a utility function value that already exceeds the base value, then that branch of the search space is pruned. The base value is computed by the `solve1` predicate (used in Figures 2 and 3).

C. The First Modeling Approach

The main predicate in the program is `solve`, given in Fig. 1. Its output parameter `Cost` is the minimum cost of the solution if it exists. The solution itself is printed on the screen. `solve` generates the teacher list using the database, and passes it on to the `increment_loop` predicate, which actually does the job. The second actual parameter of `increment_loop` denotes that only the first choices will be considered when the search begins. `increment_loop` implements the incremental search technique.

The `increment_loop` predicate definition, given in Fig. 2, gets `Teacherlist` and `Minhappiness` as input parameters and outputs the `Cost`. Predicate `willbelabeled` generates the list `Clist1`, which will be flattened by the `flatten` predicate to generate a list of the form [(*Teachername, Schoolname, R, Cost*), ….] in its `Clist` output parameter. The domain of R is [-1,0] if *Schoolname* is either a school preferred by *Teachername* or it is his/her current school. Otherwise the domain of R is [0].

This is the main point of difference between this approach and the next: here we use -1 to represent that a teacher is assigned to a school, and 0 that s/he is not (our choice of -1, rather than 1, to represent a teacher being assigned to a school is technical and is not of major importance: the built-in `indomain` predicate assigns values to variables in

```
increment_loop(Teacherlist,
               Minhappiness,Cost):-
   willbelabeled(Teacherlist,Minhappiness,
              [],Clist1),
   flatten(Clist1,Clist),
   each_teacher(Teacherlist,Clist),
   findall(Y,school(Y,_,_,_),Schoollist),
   createoutoflist(Teacherlist,Schoollist,
               [],Outlist,Clist),
   flatten(Outlist,Outoflist),
   findinginoutlist(Schoollist,Clist,
               [],Schoolinlist),
   findinginoutlist(Schoollist,Outoflist,
               [],Schooloutlist),
   quotaconstraint(Schoollist,Schoolinlist,
               Schooloutlist),
   solve1(RefCost),
   length(Teacherlist,LT),
   bb_min((our_labeling(Clist, 0, RefCost,
          Cost)), Cost,bb_options with
          [from:LT]),
   (nonvar(Cost),showreplacement(Clist),!);
   (Minhappiness1 is Minhappiness+1,
               increment_loop(Teacherlist,
               Minhappiness1,Cost)).
```

Fig. 2. The "increment_loop" predicate for the first approach

increasing numeric order, and in our labeling predicate this would result in a teacher *not* being assigned to a school). In the second approach, we shall use the built-in `element` constraint predicate.

Then `each_teacher` predicate is called. It gets `Teacherlist` and `Clist` as input. It is the first constraint of the program which constrains a teacher to locate only to one school. A teacher cannot exist in more than one school at the same time. In order to prevent the relocation of a teacher to more than one school, the sum of all the *R* values of each teacher in `Clist` is constrained to be -1.

$$SumofRs\ \#\ =\ -1 \tag{3}$$

We also need to make sure that the constraint

$$(Incoming\text{-}Outgoing)\#<=Quota \tag{4}$$

is not violated for any school. The `quotaconstraint` predicate achieves this. It takes `Schoollist`, `Schoolinlist`, and `Schooloutlist` as parameters. It constrains each school such that the difference between the number of incoming teachers and outgoing teachers to the school is smaller than or equal to the quota of the school. Its parameters are generated as follows: The list of schools `Schoollist` is generated and passed as an input parameter to the `createoutoflist` predicate, together with `Teacherlist` and `Clist`. The predicate `createoutoflist` generates the `Outlist`, flattened into `Outoflist`, which has the same structure as `Clist`, except that the R value of a teacher becomes -1 in the school which he/she wants to move out from. `Outoflist` contains information about which schools teachers are moving out of.

Then `findinginoutlist` predicate is called two times. In first call, it takes `Schoollist` and `Clist` as input parameters and generates `Schoolinlist` as output. It keeps the information of the number of coming teachers to each school and has the structure [(*Schoolname*, *NumberOfComingTeachers*),…]. In second call of the `findinginoutlist` predicate, `Schoollist` and `Outoflist` are passed as input parameters to generate `Schooloutlist` which has the same structure with `Schoolinlist` but keeps the number of outgoing teachers for each school: [(*Schoolname*, *NumberOfOutgoingTeachers*)].

The `solve1` predicate is then called to get a baseline cost in its `RefCost` parameter. `solve1` just finds any feasible solution, without any consideration of optimality. We use this value to prune the search space when we make incremental assignments: if the current "cost" has already exceeded the reference cost, that branch of the search space is not explored any further. `Refcost` is passed as a parameter to `our_labeling` predicate, which performs assignments to teachers, keeping track of the value of the utility function (happiness of the teachers).

Next we have a call to the minimization predicate *bb_min(Goal,Cost,Options). bb_min/3* is a built-in predicate which finds a solution of the *Goal* that minimizes the value of *Cost* [4]. The goal to be satisfied is `our_labeling( Clist, 0, RefCost, Cost)`, where `Cost` is to be minimized. In our case, `Cost` is the cumulative happiness of teachers, as already discussed.

Then we have an OR (;) structure. If the program has already found the minimum cost, the `showreplacement` predicate is called, which prints the teacher names, their assigned schools, as well as the rank of the school in the preference order of the teachers. If no solution was found, the `Cost` variable is free, `Minhappiness` is incremented by 1 and `increment_loop` is called recursively to try to find a solution by including one more preference of the teachers to the search space.

D. The Second Modeling Approach

Similarly to the first approach, the main predicate is `solve` and the parameter `Cost` is the minimum cost of the solution (Fig. 1). The teacher list generated by the `solve` predicate is passed on to the `increment_loop` predicate, given in Fig. 3. The `increment_loop` predicate calls the `for_each_teacher` predicate that constrains the teacher choices with the value of `MinHappiness` and generates a `List` of the form [(*Teachernamei, Happinessi, Schooli*),..]. The `for_each_teacher` predicate assigns a school from the list of teacher choices for each teacher using built-in predicate *element*. The *element* predicate has signature

```
increment_loop(Teacherlist,
               MinHappiness,Cost):-
   findall(Y,school(Y,_,_,_),Schoollist),
   for_each_teacher(MinHappiness,
               Teacherlist,[],List),
   coming_teacher_list(List,[],ComersList),
   outgoing_teacher_list(Teacherlist,
               [],OutgoingList),
   constraint_quotas(Schoollist,ComersList,
               OutgoingList),
      solve1(RefCost),
      length(Teacherlist,LT),
   bb_min((our_labeling(List, RefCost, 0,
   Cost)),Cost,bb_options with
       [from:LT]), ! ,
  ((nonvar(Cost),showreplacement(List),!);
   (MinHappiness1 is MinHappiness+1,
               increment_loop(Teacherlist,
               MinHappiness1,Cost)).
```

Fig. 3. The "increment_loop" predicate for second approach

element(?Index, ++List, ?Value), where *Value* is the *Index'th* element of the integer list *List* [4]. The `for_each_teacher` predicate makes a call to `element( Happiness1, [ FirstSchool,SecondSchool, CurrentSchool], School1)`. The `School1` variable shows which school `Teacher1` is tentatively assigned to and `Happiness1` is the order of `Shool1` among the choices of `Teacher1`. As mentioned above, this is the main difference of the two approaches.

Then we have the following constraint, which implements the incremental search technique.

$$Happiness_i \ \#=< MinHappiness \qquad (5)$$

It constrains teacher *i* to be assigned to the first, first two, or first three choices, in that order (again, the third choice is the implicit "stay at your current school" option).

The `coming_teacher_list` predicate gets the `List` as an input parameter and generates `ComersList` which has the structure of [(*Teachernamei, AssignedSchooli*)]. This list depicts which teacher is assigned to which school. Similarly the `outgoing_teacher_list` predicate gets `Teacherlist` as an input parameter and generates `OutgoingList` which has the structure [(*Teachernamei, VacatedSchool*)].

Then, as before, we need to make sure that the capacity of each school is not exceeded in an assignment. The `constraint_quotas` predicate does this (Fig. 4). It takes `Schoollist, ComersList, OutgoingList` as input parameters and constrains each school such that the difference between the number of incoming teachers and outgoing teachers to the school does not exceed the quota of the school. The *occurrences/3* built-in predicate is used to determine the number of incoming and outgoing teachers to each school (the number of times a school *occurs* in the list of schools to which teachers are relocated is computed using this built-in predicate; similarly with outgoing teachers).

```
constraint_quotas([],_,_).
constraint_quotas([H|T],
                  Comers_list,
                  Outgoing_list):-
    pred1(Comers_list,[],Comers_list1),
    pred1(Outgoing_list,[],Outgoing_list1),
    occurrences(H,Comers_list1,NewTeachers),
    occurrences(H,Outgoing_list1,
                  OutgoingTeachers),
    school(H,_,Quota,_),
    NewTeachers-OutgoingTeachers #<=Quota,
    constraint_quotas(T,Comers_list,
                  Outgoing_list).
pred1([],A,A).
pred1([(H,S)|T],Acc,Acc2):-
append(Acc,[S],Acc1),pred1(T,Acc1,Acc2).
```

Fig. 4. The "constraint_quotas" predicate

The predicate *occurrences* has signature *occurrences(+Value, +List, ?N)* and constrains its arguments such that *Value* occurs exactly *N* times in *List* [4].

The remainder of the `increment_loop` predicate is the same as the one in the first approach.

IV. Comparison of the Two Approaches

We simulated both approaches to see the difference in their performance. We generated instances of the problem for different number of teachers and quotas in schools. The number of schools was fixed at 10. For a specific number of teachers, the program corresponding to each approach was run multiple times, where in every run the number of empty slots in each school was randomly determined from 1 up to a maximum number. The average time taken to find the optimum solution for a specific number of teachers was then noted for either approach.

Fig. 5 depicts the performance of the two approaches where the number of vacant slots in each school varies randomly from 1 to 5.

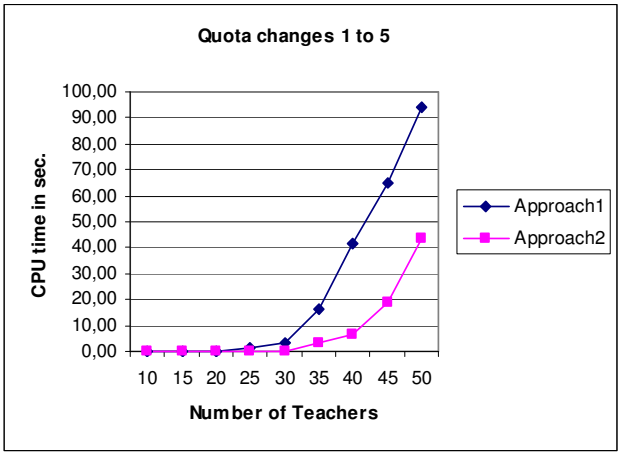

Fig. 5. Number of Teachers vs. CPU time - quota changes from 1 to 5

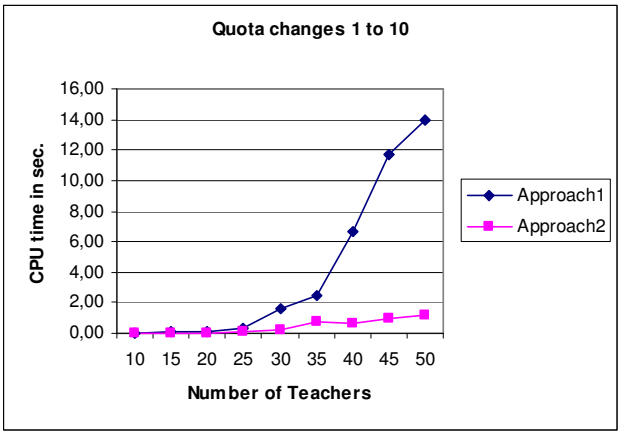

Fig. 6. Number of Teachers vs. CPU time - quota changes from 1 to 10

It is clearly seen that approach 2 takes less time, by a factor of 2 for 50 teachers, to solve the problem.

The solution of the problem becomes easier when we increase the number of empty slots in each school, because the teachers can be assigned more easily, possibly without requiring the relocation of other teachers from a school. In Fig. 6, the number of empty slots varies randomly from 1 to 10. We see that either approach requires far less time now to solve the problem, but the second approach still outperforms the first, and even to a greater degree (a factor of almost 14) than when the quotas changed from 1 to 5.

Fig. 7 depicts the worst case scenario, where empty slots of the schools are all zero. Approach 2 still performs better. However, it is not possible to obtain a solution in a reasonable amount of time when the number of teachers exceeds 30 using either approach. This is to be expected, given the exponential nature of the search space.

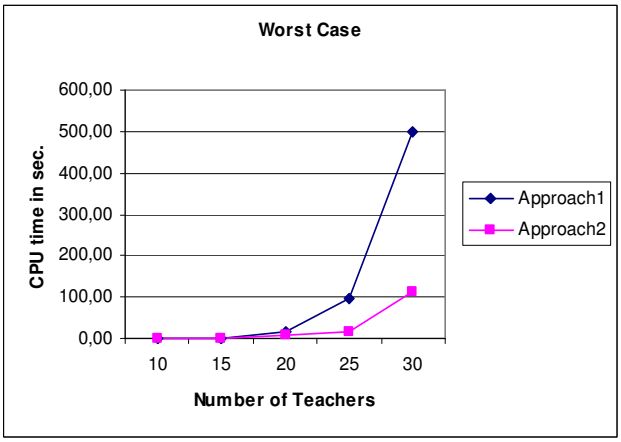

Fig. 7. Number of Teachers vs. CPU time - all quotas zero

V. RELATED WORK

Solution of many kinds of scheduling problems has been attempted using Constraint Logic Programming. We have not found, beside ours in [1], any that attempts to solve the teacher relocation problem in literature though. In [6], the Hospitals/Residents Problem is presented, which bears some resemblance to the teacher relocation problem. In the Hospitals/Residents Problem, each resident is paired with an acceptable hospital, in such a way that a hospital's capacity is never exceeded. Both the hospitals and residents have preferences, and a "stable" matching is attempted, where neither the resident, nor the hospital, would rather be paired with something/someone else. The authors investigate four different techniques, two of them being similar to our assignment of binary values to variables in first approach.

In [7], the author presents a solution to sport tournament scheduling using the finite domain library of ECL^{i}PS^e. He makes use of a constraint-based depth-first branch and bound procedure. The optimal solution is found in reasonable time except in some situations. He also proposes a local search procedure in order to provide an approximate solution in shorter time. The authors in [7] examine incremental search in AI applications by focusing on Lifelong Planning A*. In [9] the authors present a methodology to solve a job-shop scheduling problem using constraint logic programming. They investigate a new strategy to find the optimal solution which involves step by step decreasing of the upper and lower bounds of the search space of the branch and bound evaluation function. The authors in [10] present a solution to university timetabling problem using the ECL^{i}PS^e constraint logic programming language by stating the constraints in the most suitable order.

VI. CONCLUSION AND FUTURE RESEARCH DIRECTIONS

We modeled the teacher relocation problem, defined originally in [1], using two different approaches in the constraint logic programming system ECL^{i}PS^e and compared their performance under different scenarios. The first approach uses logic variables with boolean values. The second approach uses the built-in predicates "element" and "occurrence." In both approaches, incremental search and pruning techniques are used to speed up the solution process. Simulations confirm that the approach that uses the built-in predicates performs better in general. This result probably should not come as a surprise, given that built-in predicates are expected to be implemented to perform efficiently.

For future work, given that the framework is already established, we envisage making the utility function much more representative of the interests of all involved, such as schools, the ministry of education and even parents. A WEB interface can be used to accept the preferences of all parties involved.

REFERENCES

[1] N. Ilhan, Z. Bayram. "A constraint logic programming solution to the teacher relocation problem", accepted to the *2nd International Computer Engineering Conference (ICENCO 2006)*, 26-28 December 2006, Cairo, Egypt.

[2] M. G. Wallace. "Practical applications of constraint programming," *Constraints Journal*, 1(1), 1996.

[3] J. Jaffar, M. J. Maher. "Constraint logic programming: A survey," *Journal of Logic Programming*, 19 & 20, 1994, pp. 503-581.

[4] IC-PARC. ECLiPSe 5.7 User Manual, 2003.

[5] K. Marriot, P. Stuckey, *Programming with Constraints: An introduction*, The MIT Press, 1998.

[6] D. F. Manlove, G. O'Malley, P. Prosser, C. Unsworth, "A Constraint Programming Approach to the Hospitals / Residents Problem," in *Proceedings of the Fourth Workshop on Modeling and Reformulating Constraint Satisfaction Problems, held at the 11th International Conference on Principles and Practice of Constraint Programming*, (CP 2005), pp 28-43.

[7] A. Schaerf, "Scheduling Sport Tournaments using Constraint Logic Programming," in *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96)*, 1996, pp. 634-639.

[8] S. Koenig, M. Likhackev, Y. Liu, D. Furcy, "Incremental Heuristic Search in Artificial Intelligence", *AI Magazine*, 25(2), 2004, pp 99-112.

[9] J. Paralic, J. Csonto, M. Schmotzer, "Optimal Scheduling Using Constraint Logic Programming," in *Proceedings of 8th Symposium on Information Systems IS'97*, Varazdin, October 1997, pp. 65-72.

[10] M. Kambi, D. Gilbert, "Timetabling in Constraint Logic Programming", in *Proceedings of INAP-96: Symposium and Exhibition on Industrial Applications of Prolog*, Tokyo, Japan, 1996.