TASK QUARTERLY 11 No 1–2, 137–149

A HYBRID METHOD FOR SOLVING TIMETABLING PROBLEMS BASED ON THE EVOLUTIONARY APPROACH

MACIEJ NORBERCIAK

Institute of Applied Informatics, Wroclaw University of Technology, Wybrzeże Wyspiańskiego 27, 50-370 Wroclaw, Poland
maciej.norberciak@pwr.wroc.pl

(Received 30 December 2006; revised manuscript received 17 January 2007)

Abstract: Timetabling problems are often difficult and time-consuming to solve. Most of the methods of solving these problems are limited to one problem instance or class. This paper describes a universal method for solving large, highly constrained timetabling problems in various domains. The solution is based on an evolutionary algorithm framework and employs tabu search to quicken the solution finding process. Hyper-heuristics are used to establish the algorithm's operating parameters. The method has been used to solve three timetabling problems with promising results of extensive experiments.

Keywords: evolutionary algorithms, hybrid methods, timetabling

1. Introduction

Timetabling problems have been quite popular with researchers for more than thirty years now. Their practical importance should not be underestimated, as institutions involved in education, healthcare, transportation, sports, courts of law, production enterprises and many others devote considerable resources to establish effective plans of their actions. Over these years, many approaches to partial or complete automation of such tasks have been presented, roughly divisible into four types [1]:

- **Sequential methods**, using domain heuristics to order events and then assign them sequentially to valid periods of time (also called *timeslots*) so that no events in the period are in conflict with each other. Events are most often ordered so that those most difficult to schedule are assigned to timeslots first (this course of action being called a direct heuristic based on successive augmentation) [2].
- **Cluster methods**, in which events are collected in clusters so that no two events in a particular cluster are in conflict with each other. The clusters of events are formed and fixed at the beginning of the algorithm, which is the

main disadvantage of this approach, as it may result in a timetable of poor quality [3].

- **Constraint-based methods**, in which a timetabling problem is modeled as a set of variables (*i.e.* events) having a finite domain to which values (*i.e.* resources such as time periods) need to be assigned to satisfy a number of constraints; a number of rules is defined for assigning resources to events and, when no rule is applicable to the current partial solution, backtracking is performed until a solution is found that satisfies all constraints. Algorithms are usually allowed to break some constraints in order to produce a complete timetable (which must be done in a controlled manner), as it may be impossible to satisfy all constraints [4].
- **Meta-heuristic methods,** a variety of which have been investigated for timetabling, including simulated annealing, tabu search, evolutionary algorithms and hybrid approaches. Meta-heuristic methods begin with one or more initial solutions and employ search strategies to find the optimal solution, attempting to avoid local optima in the process [3, 5–8].

Most approaches use heuristics (or metaheuristics) because timetabling problems are considered to be NP-hard (although mathematical proof of this exists only for small-sized problem). Traditional combinatorial optimization methods most often come with a considerable computational cost and, although capable of producing high quality solutions, they are not suitable for solving large, highly constrained problems.

The above might suggest that AI-based automatic planning has already reached a level of relative maturity, all the problems have been solved in principle, and the research is focused on making the existing methods faster, more effective and yielding solutions of better quality for more complex and larger problems. (It is, of course, possible to find an instance large enough for a combinatorial optimization problem of a given class, where modern metaheuristics perform poorly or even fail, but these are rarely real-life problems.) Nevertheless, it must be pointed out that a vast majority of methods concern only a specific type of problems (*e.g.* [9, 10]) or a particular problem class (*cf.* [11, 12]) and require time and resources to be adapted for specific practical applications.

This paper presents an attempt to create a universal method capable of solving problems from different areas with minimum user interaction. Three problems from different classes have been chosen for testing, so that universality and flexibility of the method be assured. We begin with a typical university course timetabling problem, very popular and widely researched, thus facilitating access to test data, both real-life and artificially generated. A similar but more specific problem with different constraints is timetabling at the Faculty of Computer Science and Management of the Wroclaw University of Technology. Our last problem belongs to the personnel scheduling class and concerns monthly duty plan of a Polish hospital's ward. Its relatively small size makes facilitates analysis, while its constraints are unlike those of school timetabling.

2. Description of the problems

A typical timetabling problem consists in assigning a set of activities/actions/events (*e.g.* work shifts, duties, classes) to a set of resources (*e.g.* physicians, teachers, rooms) and time periods, fulfilling a set of constraints of various types. Constraints either stem from the very nature of timetabling problems or are specific to the institution involved. In other words, timetabling (or planning) is a process of putting in a sequence or partial order a set of events to satisfy temporal and resource constraints required to achieve a certain goal. It is sometimes confused with scheduling, which is a process of assigning events to resources over time to fulfill certain performance constraints, although some scientists consider scheduling as a special case of timetabling or vice versa [7].

Timetable problems are subject to numerous constraints, usually divided into two categories: *hard* and *soft*. *Hard* constraints are rigidly enforced and have to be satisfied in order for a timetable to be feasible. *Soft* constraints are desirable but not absolutely essential.

The first problem considered is a typical university course timetabling problem (UCTP). It consists of a set of events (classes) to be scheduled in a certain number of timeslots and a set of rooms of certain features and sizes in which the events can take place. There is a defined set of students attending each event and the number of timeslots is 45 (5 days, 9 timeslots each). Test sets for this problem come from International Timetabling Competition (ITTC) [13].

A feasible UCTP timetable is one in which all the events have been assigned a timeslot and a room, while the following hard constraints have been satisfied:

(1) only one event is scheduled in each room at any timeslot,
(2) each room is large enough for all the attending students and has all the features required for the event, and
(3) no student attends more than one class at the same time.

There are also three soft constraints defined, which are broken if:

(1) a student has a class in the last slot of the day,
(2) a student has more than two classes in a row,
(3) a student has a single class on a day.

The second problem – timetabling at the Faculty of Computer Science and Management of the Wroclaw University of Technology – is similar to the first, but has additional constraints related to teachers and the set of students attending each event is undefined (only the number of students and the faculty they attend is known) and has to be concluded from other data. In this problem, the number of timeslots is 35 (5 days, 7 timeslots each) and each event is attributed to a defined course (each class is a part of a particular university course). Some test sets for this problem come from real FCSM data while others have been generated artificially. In this problem, a feasible timetable is one in which all the events have been assigned a timeslot and a room so that the following hard constraints have been satisfied (apart from constraints (1) and (2) of UCTP):

– no teacher teaches more than one class at the same time,
– no teacher teaches any class in timeslots which are forbidden for him, and

– if a particular course has only one class assigned, no class with students from the same faculty is scheduled at the same timeslot with this course (this covers the obligatory courses which are usually taught for all the faculty's students).

The third problem concerns a typical hospital department where about a dozen physicians of various specialties are employed. Every day one or more doctors have duty, but the number of doctors on duty may vary from day to day. A planning horizon (*i.e.* a period of time for which the problem must be solved) amounts to one month. If specialties of physicians in a particular department are not homogenous (*e.g.* the casualty ward employs surgeons and anesthesiologists) doctors of particular specialty are often required to be on duty. The following hard constraints have been defined:

– all the timeslots (*i.e.* days) have a proper number of physicians of appropriate specialties assigned,
– no physician has duty on two (or more) consecutive days,
– no physician has duty more often than twice a week,
– at least one physician on each duty is able to perform his duties single-handedly (*viz.* has a sufficient degree of medical education and experience).

In order to consider and model issues of fairness and job satisfaction, the following soft constraints have been introduced:

– physicians have duties on their preferred days of the month and, symmetrically, have no duties assigned in timeslots they would rather avoid,
– if more than one physician have duty at the same time, their interpersonal preferences are taken into consideration (so that doctors have duty with persons they like).

3. Solution details

Evolutionary algorithms (EA) are considered to be a good general-purpose optimization tool due to their high flexibility and conceptual simplicity. Moreover, as they have been proven to solve timetabling problems effectively ([3, 14]), the EA framework has been chosen as the basis for a universal solver of timetable problems. The term "EA" is used in this paper in its most generic meaning to indicate any population-based metaheuristic optimization algorithm that uses mechanisms inspired by biological evolution, such as reproduction and mutation.

3.1. Representation of the solutions

In order to assure universality of the approach, each solution (genotype) of a particular problem's instance is represented directly – each timeslot has a list of events assigned and each event – a list of resources. Genotype length is constant for particular problem – in the case of hospital duties the genotype has the length of the number of physicians on duty multiplied by the number of timeslots, while the course timetable genotype's length equals the number of timeslots multiplied by the number of rooms. The data (*e.g.* constraints) required to describe a particular problem class is abstract and unified for all problem classes. Some weak constraints (those with more than two preference levels – desired and undesired) require a special description: a constraint is defined by its type and the number of preference levels. The latter must

always be odd; the first level is the most desirable, the last level is the least desirable and the middle level is neutral. Weak constraints may be of two types: timeslots which have preferred resources and resources which have preferred resource combinations. Examples of such constraints may be found in the duty assignment problem in the form of interpersonal preferences (doctors may "love", "like", "dislike" or "hate" one another, or merely be "indifferent").

3.2. The evaluation function

A penalty-based evaluation function has been used. Penalty for genotype g amounts to:

$$f_g = \sum_{i=0}^{i<t} \sum_{j=0}^{j<c} w_j n_{ij}, \tag{1}$$

where t is the number of timeslots, c is the number of constraint types (in the case of a weak constraint with more than two preference levels, all preference levels are considered to be separate constraints), w_j is the weight assigned to a particular constraint type, while n_{ij} is the factor determined by the penalization method. Four different methods of penalization have been considered wherein:

- a timeslot is penalized once for every type of constraint broken (*i.e.* n_{ij}, is either 0 or 1),
- a timeslot is penalized every time a particular type of constraint is broken,
- as in the first method, but the penalty is doubled for each subsequent constraint of a particular type being broken, and
- a binary penalty – if a timeslot with events planned breaks no constraints, the penalty for this timeslot amounts to 0 (or 1 otherwise); this is an exception from Equation (1), as no weights are used to determine the penalty's value.

A value of the evaluation (fitness) function for a solution, g, is calculated by dividing the lowest non-zero penalty value in the population by the penalty value for g:

$$F_g = \frac{f_{\min}}{f_g}. \tag{2}$$

After generating the initial population, the evolutionary algorithm begins to operate. A population is created in subsequent generations (iterations) by means of the classical genetic roulette, as described in [15], but 20% of the population is always preserved from the previous generation. 10% consists of the best solutions in terms of the evaluation function described above. The remaining 10% are the solutions most distant from the rest of the population, in order to preserve population diversity. The distance between two timetables can be measured in three ways:

- as the number of events planned with the same resources in the same timeslot in both timetables,
- as the number of pairs of events planned with the same resources in the same timeslot in both timetables; as described in [1], this method is favored as it allows to represent diversity as a single value average and did not have the drawback of the method where absolute positions of events in timetables are considered, or
- as search space coverage – how often the tuple ⟨event, resources, timeslot⟩ appears in the whole population.

The higher the score, the smaller the distance between the timetables.
Additionally, three methods of determining the weights have been proposed:

- unified weights (all weights amount to 1),
- weak constraints having the weight value of one, strong constraints having the weight equal to the number of weak constraints,
- automatic weight assignment – a procedure that allows establishing the weights basing on how frequently constraints of particular type are broken in randomly generated solutions; a set of solutions is generated at random and the least frequently broken constraint is assigned the weight of one, while other weights are established proportionally (the more frequent the constraint is broken, the higher the weight).

If a weak constraint has more than one preference level, in all cases:

- the neutral level has nil weight value,
- positive preferences have negative weight values (it is a prize rather than penalty; which may lead to negative penalty values, in such cases considered to be zero),
- the weight of preference level p of constraint c equals:

$$w_{cp} = \left| \frac{w_c}{2} \right| \cdot p, \tag{3}$$

where w_c is the weight value of constraint c established by means of one of the aforementioned methods.

3.3. Genotype initialization strategies

In most of the approaches either random or heuristic initialization is used to provide EA with an initial population of solutions. The random method is the least computationally complex and ignores the problem's domain knowledge. Heuristic approaches have proven to be more effective though, *i.e.* the final solution tends to be found faster than in the case of random initialization. Nevertheless, heuristics always employ event sequencing strategies – the events are placed in the timetable in the descending order of their "difficulty" to be planned, *i.e.* the events that are the most difficult to schedule are allocated first. Graph coloring or problem-specific heuristics is used. In the approach described in this paper, random initialization has been used as reference in evaluating the other method, viz. the peckish initialization method [16]. In this approach, sets of events (and resources) are chosen at random for each timeslot k. The one that breaks the least number of hard constraints is assigned to the timeslot. The number k is called the greediness level – when k equals 1 this method corresponds to random initialization; when k aspires to the number of combinations of events and resources, the algorithm becomes greedy. After assigning the weights (by means of any of the aforementioned methods), the greediness level is established. About a dozen sets of solutions are generated with ascending greediness levels (due to increasing computational complexity of the generation process, the highest greediness level considered has been arbitrarily set up at the number of timeslots in the particular solution). Then the average fitness is calculated for each set of solutions, along with their average generation time. The greediness offering the best score (the shortest time and the greatest average fitness) is chosen.

3.4. Genetic operators

In the classic genetic algorithm, some solutions are exposed to genetic operators, mutation and crossover, after selection during each iteration. The contents of operators' sets and their operation depend strongly on both specifics of the problem being solved and the chosen approach. In this particular case, a recombination operator would probably exhibit high computational and conceptual complexity. Even a simple, one-point crossover operator would be quite complicated – after swapping parts of different timetables the integrity of resulting solution would have to be assured, which requires making sure that no event appears twice in the timetable and removing the copies accordingly. Thus, only mutation operators are used, as in evolutionary programming, which rarely attempt to emulate specific genetic operators as observed in nature. Resources, events and timeslots can be mutated, thus yielding a set of three different types of mutation operators. In the "classic" EA mutation, the operator is "blind", *i.e.* it changes the solution at random, an approach proven to be ineffective (see [3]). The place in the genotype (tuple $\langle$event, resources,timeslot$\rangle$), which breaks the most constraints, *i.e.* the most difficult to schedule, is selected for mutation. If a number of places are equally difficult to schedule, one of them is chosen at random. Operators attempt to reschedule events so that they would eliminate a particular type of conflict (broken constraints of a particular type) caused by this event. k possible variants are examined, and that which breaks the least constraints of the particular type is chosen, like in the peckish initialization algorithm. Typically, either only one random change is considered [17] or a form of local search is employed [6]. The proposed method is a simple, yet effective alternative to that.

3.5. The tabu search phase

Preliminary tests have shown an interesting phenomenon occurring about half-way through the solution-finding process. If the average is being observed one can notice A steady, steep drop in the value of the penalty function can be observed for a particular solution until the population reaches a certain plateau, where the value oscillates slightly (within a margin of about 1%). This is a proof that, although directed genetic operators have been used, the algorithm continues to search the solution space somewhat blindly and tends to get stuck in local optima, which it manages to escape through random mutation only. Tabu search (TS) has been employed in order to speed up the solution-finding process and avoid the aforementioned oscillations. If the average penalty for a population deviates by less than 20% for fifty generations, the genetic roulette is stopped and tabu search begins to operate. The length of the tabu list has been arbitrary established and fixed at $10 \cdot k$, like all other parameters for the tabu search operation, in order to avoid introducing new variables into the method. The algorithm operates as follows:

(1) Find a place in solution (tuple $\langle$event, resources,timeslot$\rangle$) which breaks the most constraints (if there are a few such places, choose one at random).
(2) Generate k solutions with events rescheduled with different resources and/or timeslot.
(3) Choose the solution which has the lowest penalty score and is not on the tabu list, add it to the tabu list and go to (1). The chosen solution is now the current one.

The tabu search algorithm operates for 50 iterations, following which the evolutionary algorithm takes over again. Results of experiments with tabu search application are described in Section 4.

3.6. Establishing the operating parameters

Some parameters have to be established in order for the solution to work. Usually, the parameters are established arbitrarily, *e.g.* on the basis of domain knowledge, or experimentally. However, in "knowledge-poor" algorithms, designed to solve a range of problems, such approach is not applicable. It has recently been suggested [18], that hyper-heuristic methods can be used to deal with this problem. A hyperheuristic is a high-level procedure which searches over a space of low-level heuristics rather than directly over the space of problem solutions. It is different from the popular *metaheuristic*, or heuristics which control simpler heuristics for a narrow range of problems, in than the hyper-heuristic approach chooses from a range of heuristic approaches to robustly solve a wide range of problems. A hyper-heuristic can be thought of as a heuristic to choose or create heuristics and is a term more specific than that of *meta-algorithm*, which uses the results of individual algorithms for similar tasks or subtasks to perform the chosen task.

Current applications of hyper-heuristics in timetabling tend to use metaheuristics to search for permutations of graph heuristics which are then used for constructing timetables [18]. In the method described in this paper, a metaheuristic (or an evolutionary algorithm, in this case a second-level algorithm as described below) is used to find the best parameters for another metaheuristic (evolutionary programming, a first level algorithm described in Section 3). Automatic weight assignment and establishing greediness level procedures are both part of the hyper-heuristics.

An evolutionary algorithm has been used to find out which methods of penalization, measuring the distance between solutions and weight assignment, along with the order of conflict elimination and greediness level of genetic operators are the most effective in terms of solution quality and time required to reach a feasible solution. The genotypes represent the aforementioned parameters; the greediness level is a natural number not greater than the number of timeslots, the order of conflict elimination is an ordered sequence, the remaining attributes are nominal. A particular problem's solution is generated during each iteration of the algorithm using the parameters given in every genotype, so that the first-level algorithm operates with parameters from the second-level algorithm's genotype. In order to avoid infinite operations, the first-level algorithm ceases to operate after finding a feasible solution or after 1000 iterations, whichever occurs first. The genotypes of the second-level algorithm that have not given feasible solutions of the first-level algorithm are scrapped, while the others are evaluated. The value of the evaluation function is that of the binary penalty function for the best genotype in the first-level algorithm's population. The best set of parameters is memorized and the genetic operators of mutation and crossover are applied to the solutions. Mutation, acting with the probability of 0.2, changes one of the parameters at random; in the case of the conflict elimination order, it changes that order. Crossover swaps random parts of any two parameter sets with the probability of 0.05, treating the conflict elimination order as one parameter. In the case of the second-level algorithm, there is no threat that a crossover operator could compromise

data integrity, as may be the case in the first-level algorithm, because it merely mixes parameters from two solutions. The procedure is stopped after a fixed number of iterations or when no improvement has been achieved in two subsequent iterations.

4. Experimental results

Preliminary experiments have been conducted with the approach described in this paper. The first task was to prove that the method is capable of finding a feasible solution for all the test problems. Ten International Timetabling Competition sets have been used for the first problem, two real datasets for the second, and one real and nine artificially-generated ones for the third. A feasible solution was found for all the test sets. More extensive experiments were conducted on UCTP, as the problem is widely recognized and the results can be compared with those found in other publications. The results presented in the tables below have their fitness function values recalculated to match the method used for evaluating the solution in ITTC (all weights equal one, the second penalization method). As there are no analytical means to compare the complexity of particular problem instances, only empirical comparison is feasible. All the datasets depict problems similar in size: about 400 events, 200 students and a dozen rooms and features.

4.1. The first level algorithm: EA vs. tabu search

The first set of experiments has been conducted to ascertain whether EA can be sensibly used as a first-level algorithm. Many researches have reported excellent results using only a variation of tabu search, so the importance of this question cannot be underestimated. Every dataset has been solved twenty times for each method variation, *i.e.* EA only, TS only and EA with tabu search used to escape from local optima. Both algorithms operated until a feasible solution has been found or for 1000 generations otherwise. The greediness level for EA equaled 15, the tabu list's length was 150, while the population size was 500. The second penalization method with automatic weight assignment was used. The results are presented in Table 1.

The feasibility of the solutions found was not taken into account (most of the time the algorithms were unable to find one in 1000 iterations): it is possible that two solutions with the same score existed and only one of them was feasible.

Tabu search shows better average results, although not all of its "best" results are better than those of EA, which is probably due to tabu search operating in more systematically and predictably than EA. This comes with a price tag: EA is more likely to find a better solution by pure chance. The combination of TS and EA shows improvement in both best and average results; incorporating a tabu search phase accelerates the search process. It is possible that using only tabu search would produce a better overall solution but at a considerably greater computational cost of looking through the tabu list.

4.2. Experiments with the second-level algorithm

In all our experiments, the second-level EA had a population size of 100, the first-level – that of 500. The second-level EA was run for 1000 iterations, following which the best 10 specimens of the second-level algorithm were run through the first-level algorithm for 5000 generations.

Table 1. Experimental results: the first row is the best result obtained with the evolutionary algorithm only, the second is the average of the best solutions in 50 runs, the next two rows are the best and average results using tabu search, the last two – the evolutionary algorithm combined with a tabu search phase

Dataset	1	2	3	4	5	6	7	8	9	10
EA best	176	135	201	551	391	207	159	186	212	167
EA avg.	191	163	249	610	412	257	199	206	281	199
TS best	168	174	228	511	388	201	148	211	217	159
TS avg.	181	195	245	536	391	226	158	241	247	178
Both best	166	145	192	510	357	175	144	197	184	157
Both avg.	187	156	210	570	386	242	163	254	255	163

Dataset	11	12	13	14	15	16	17	18	19	20
EA best	186	254	295	281	184	166	317	101	302	165
EA avg.	199	263	301	297	190	167	327	109	317	179
TS best	179	243	271	269	155	128	281	91	275	159
TS avg.	187	251	289	285	179	134	287	98	278	165
Both best	163	231	262	261	164	126	262	84	269	157
Both avg.	176	244	278	278	176	139	276	88	271	159

The results archived by ITTC participants, which will be used as reference for the method presented in this paper, are shown in Table 2. The best results have been gathered from all the participants, but the winner has failed to achieve the best known solutions for all the problem instances. An experiment with the algorithm described in this paper was conducted 20 times and the averages of its best results are included in Table 2.

Table 2. Results of experiments with the second-level algorithm

Dataset	1	2	3	4	5	6	7	8	9	10
ITTC best	45	25	65	115	77	6	12	29	17	61
ITTC average	137	87	150	289	248	143	145	129	123	153
Without TS	158	103	156	399	336	146	125	110	154	153
With TS	141	101	145	340	271	138	107	98	146	139
Best 10 with TS	130	93	139	320	264	136	104	92	138	128

Dataset	11	12	13	14	15	16	17	18	19	20
ITTC best	53	110	109	93	62	22	79	31	44	0
ITTC average	148	206	234	229	149	124	207	89	257	101
Without TS	163	220	268	255	158	145	301	92	299	185
With TS	151	218	255	234	149	134	287	88	278	165
Best 10 with TS	146	211	248	227	136	126	252	74	267	151

In all the experiments, feasible solutions have been found for all the problem instances before the second-level algorithm ceased to operate. Most of the results are above the average of the ITC score, but none is near the best score achieved by competitors. Nevertheless, it has to be emphasized that the methods used in ITC were designed to perform only one task and the parameters of their operation were chosen intentionally to perform that task in the most effective way possible.

4.3. Diversity of feasible solutions: the duty planning problem

Another set of experiments has been conducted on the duty planning problem. The datasets solved were rather simple, so the algorithm was able to find feasible solutions (even with no penalty) quite fast. However, a less constrained problem often means a greater search space and further study is required to determine what part of the search space has been visited. The ultimate operational acceptability and subjective quality of the solutions with no penalty had to be determined by a human, so the more different solutions were offered by the algorithm, the wider was the range of selection. The task was to observe the diversity of feasible solutions in order to determine whether it was sufficient to for the user to choose from.

The experiments were conducted on nine datasets. One was a real dataset from the Department of Neurosurgery of the Wroclaw Medical University and contained data on ten neurosurgeons, three of who could not work alone. No time preferences and only two negative social preferences were defined. There were two doctors on duty on odd days of the month and one doctor on even ones. The remaining datasets were created artificially by adding 2 random social preferences for each doctor and 5 random temporal preferences to the real-data set (datasets 2–5) or 5 random social preferences and 10 random temporal preferences for each doctor in the real-data set (datasets 6–9).

The results are presented in Table 3. The figures given in the table are averages of fifty measurements. Algorithm run time was 250 iterations of the second-level algorithm, with the first-level population size of 100 solutions and the second-level of 30 sets of parameters. The second column from the left contains the number of iterations after the first solution with 0 or less penalty had been found. The average penalty of the entire population after completion of computations is given in the third column, while the fourth contains the average distances between solutions with no penalty after completion of computations measured by search space coverage (the third method) and divided by the number of timeslots, which gives the average distance between particular timeslots across the whole population. The number of 0 or less penalty solutions after the algorithm had stopped is given in the last column.

The most interesting value is the average distance between solutions, *e.g.* the value of 7.6 roughly means that nearly 90% of the population have no different

Table 3. Population diversity in the duty planning problem: experimental results

Dataset	Average iterations	Average penalty	Average distance	0 or less solutions
1	160	47.8	7.6	63
2	217	69.3	9.2	31
3	267	77.1	4.1	53
4	232	43.2	4.3	56
5	199	81.8	9.2	23
6	631	95.3	12.2	9
7	939	57.3	3.8	33
8	715	73.4	4.4	21
9	688	104.2	11.0	11

events and/or resources planned in any particular timeslot and there is about a dozen different types of timetables to choose from.

5. Conclusions and future work

Whether a universal, "knowledge-poor" method is able to perform better or at least comparably as well as domain-specific ones remains an open question. In terms of computational time, it appears impossible, as a general method searches the parameter space blindly. Our preliminary results are encouraging, but further research is required to improve the algorithm: employing a form of local search seems particularly promising in this regard. Nevertheless, universal methods will always have a distinctive advantage over specialized ones in they do away with the laborious and time-consuming process of redesigning and fine-tuning to fit the specific requirements of a particular problem.

Results of the second-level algorithm's operation should be investigated as it is possible that some methods of penalization, weight assignment and distance measurement may prove useless for all the problem's variations and may thus be removed from the search space. This would be especially important in the case of personnel scheduling problems such as the doctors' duty assignment problem described in this paper. In most cases, duty roster changes only slightly from one planning horizon to another (see [8, 19, 20]), so there is no need to search the parameter space every time a particular problem class is solved; the set of "best" parameters from previous runs can be used instead.

Acknowledgements

The research was supported with Grant No. 3 T11C 031 27 from Poland's State Committee for Scientific Research.

References

[1] Burke E K and Petrovic S 2002 *Eur. J. Operational Res.* **140** 266
[2] Burke E K, Newall J P and Weare R F 1998 *Proc. Int. ICSC Symposium on Engineering of Intelligent Systems*, ICSC Academic Press, Nottingham, pp. 574–579
[3] Myszkowski P and Norberciak M 2003 *Annales UMCS, Sectio Informatica*, Lublin, **I** 115
[4] Yakhno T and Tekin E 2002 *Proc. EurAsia-ICT 2002*, Teheran, Iran, Springer-Verlag, pp. 11–18
[5] Colorni A, Dorigo M and Maniezzo V 1990 *Proc. 1st Int. Workshop on Parallel Problem Solving from Nature, Lecture Notes in Computer Science* **496** 55
[6] Newall J P 1999 *Hybrid Methods for Automated Timetabling*, PhD Thesis, Department of Computer Science, University of Nottingham
[7] Ross P and Corne D 1995 *AISB Workshop, Lecture Notes in Computer Science*, Sheffield, UK (Fogarty T, Ed.), Springer-Verlag, **993**, pp. 94–102
[8] Valouxis C and Housos E 2000 *Artif. Intell. Med.* **20** 155
[9] Carter M W 2001 *Proc. PATAT 2000*, Constance, Germany (Burke E and Erben W, Eds), Springer-Verlag, pp. 64–84
[10] McCollum B 1998 *Proc. PATAT 1997*, Toronto, Canada (Burke E and Carter M, Eds), Springer-Verlag, pp. 237–253
[11] Ross P, Hart E and Corne D 1998 *Proc. PATAT 1997*, Toronto, Canada (Burke E and Carter M, Eds), Springer-Verlag, pp. 115–129
[12] Socha K, Knowles J and Sampels M 2002 *Proc. ANTS 2002*, Brussels, Belgium, Springer-Verlag, pp. 1–13

[13] 2002 *International Timetabling Competition*, http://www.idsia.ch/Files/ttcomp2002/
[14] Norberciak M 2003 *Proc. 9th Int. Conf. on Soft Computing MENDEL 2003*, Brno, Czech Republic, pp. 19–23
[15] Michalewicz Z 1996 *Genetic Algorithms + Data Structures = Evolution Programs*, Springer Verlag
[16] Corne D and Ross P 1995 *Proc. 1st Int. Conf. on the Theory and Practice of Automated Timetabling*, Napier University, Edinburgh, pp. 227–240
[17] Corne D, Ross P and Fang H-L 1994 *Improving Evolutionary Timetabling with Delta Evaluation and Directed Mutation, Parallel Problem Solving from Nature III*, Springer Verlag
[18] Burke E K, Petrovic S, Meisels A and Qu R 2004 *Computer Science Technical Report*, No. NOTTCS-TR-2004-9, University of Nottingham
[19] Norberciak M 2004 *J. Med. Inform. Technol.* **7** 83
[20] Ernst A T, Jiang H, Krishnamoorthy M and Sier D 2004 *Eur. J. Operational Res.* **153** 3