AUTOMATIC TRANSPORT SYSTEM

Milan Schmotzer

Abstract

Modern technologies give us a possibility to build systems, which fully support automatic control of a transport. In the paper the *ABATS* (*Agent Based Automatic Transport System*) is presented. The system was created under the *ECL^{i}PS^e* platform, which can use the constraint logic programming technology. By mixing the re-active AI paradigm and the hypothesis space search focused one, by using fast prototyping Prolog facilities, by using specially created fast CLP based scheduling techniques and agent modules, the *ABATS* is able to serve as a basis for a fast building of automatic transport systems.

From new scheduling techniques, the *adaptive approximate min_max* and 7*:5 branch and bound* algorithms are explained.

1 MOTIVATION

There are a lot of people, which could benefit from an automatic transport system. For example, an automatic car can be very useful for a businessman who spends five hours daily on a road, for an artist who does not like to drive his car and rather wants to stay in a quasi-meditative state, for a doctor who losses a whole hour each day by driving of his car while going to work and it might be better for him if he could take a look in patients' folders. An automatically driven car can be very important for a physically impaired person who cannot see, hear or move his/her hands.

Today's researchers have enough hardware (fast computers, computer farms, good wireless computer networks) and software (CLP, agents) resources for creating of automatic as well as autonomous transport systems.

Why should one use an automatic transport system? Because a computer controlled vehicle

- "sees" better as the human in fog, in rain and at night,
- works independent from the user's (client's) health and psychical status,
- can effectively and (for the user) comfortable operate a vehicle,
- can automatically detect situations when the user's health is endangered (and bring the user for example to a hospital),
- can spare petrol, time and money (as well as the environment) by finding the shortest way to the user's chosen destination,
- can economize utilization of vehicle's resources,
- can detect a (possible) collision and for example slow down or turn the vehicle,
- can effectively react in the case of a collision.

Of course, the great benefit is, that the user can save its potency and relax, sleep, eat or work. Together with user's personal care system located in its handy (mobile phone) and in its home computer (please see [4] and [5]), the system can react on the user's needs and find for him/her the nearest cheep restaurant (or an exclusive one if preferred), motel and so on.

Moreover, an agent-based automatic transport system is more informed and therefore more robust and intelligent. It can

- drive the vehicle by using less stressed roads (it is important for avoiding of congestions),
- detect existing vehicle collisions and warn the public safety, the police and the nearest vehicles,
- send alarms in case of a jeopardy
  - the vehicle agent (vehicle central computer) can negotiate with some of the nearest hospitals central computers and (depending on the negotiation results) on-line choose the best one,

- the car agent can call for a help
  - a fast rescue service,
  - a drag away service,
  - a police-station,
  - firemen.

The vehicle user is not the only one who can benefit from an automatic transport system. De facto all vehicles can be grouped to multi-level organized transport systems. A local autonomous transport system can be used for a medicine drug and material cartage, automatic store systems, automatic material and people transport in a factory, automatic airlines systems and so on.

One should differ between privileged and non-privileged vehicles. The privileged ones are for example personal cars (automobiles) – a person owns a car and do not want to share it with any other user. On the other hand, the non-privileged vehicles are buses, cars, boats and airplanes and an autonomous transport system can utilize them more effectively (they are available for planning all the time when they are not in a use).

2 INTRODUCTION

For the purposes of this paper, let us define that an agent is an autonomous entity, which actively communicates with its environment, for, example with some other agents, and it processes obtained information. An agent can be a part of one or more multi-agent systems.

It seems very naturally if there is used an agent (as an autonomous entity) for a car driving. Because there are a number of cars on the road, they can collaborate together as a multi-agent system. An autonomous transport multi-agent system can share its knowledge in two ways:

1. There are no personal vehicles and no area agents (like an air dispatcher) – for example a commodity transport in a small firm. In such situations, the vehicles share a part of their internal knowledge and negotiate "who is going to do what".
2. There are area agents (or there exists a whole hierarchy of area agents depending on the size of controlled area). In such system it can be supposed that there are
   - vehicle agents (one agent for each vehicle),
   - hierarchy of area agents,
   - sensors and
   - senders on the road.

For building large systems, the second type of multi-agent systems is more effective than the first one because an area agent can

- provide an updated local road-map (an update is needed when some roads are not available from some reason),
- generate the optimised way through its area,
- activate a help agent who is able to help a vehicle agent to safely drive through a danger area,
- provide a vehicle agent with a basic safety information (limited speed, roadway humidity, other danger) by using of sensors (for detecting a roadway status) and senders (which can substitute road signs).

Area agents can be organized in a way that each local area agent is responsible for a transport in its local area and it will communicate with vehicles while they are located inside this area. A small number of such local agents can be grouped together by communicating with their area agent – country (area) agent. A continent agent acts as a “chief” of a number of country agents and it is responsible for a coordination of their work. Each transport type can have its own similar agent hierarchy. Each agent can negotiate with agents coordinating another type of transport on the same level. For example a country air agent can cooperate with a country train agent. A transport can be more effective if all such multi-agent systems will cooperate together.

The author of this paper has designed algorithms for creating of *ABATS – Agent Based Autonomous Transport System* – and they were implemented as the *ABATS* simulator under the *ECL^{i}PS^e* + Java environment.

ABATS Based Application Characterization

- fully automatic system,
- fully autonomous vehicle agents (no human interaction is needed),
- user comfortable,
- safe system (watches the user's health by co-working with user personal care system),
- real-time system,
- effective hierarchical agent architecture design.

The Vehicle Agent Architecture Characterization

- hybrid (reactive and reasoning) architecture,
- module based,
- rule oriented (rules are greatly supported by Prolog) and therefore
  - easily configurable,
  - easily extendable (other modules can easily be added),
  - easily re-programmable modules.

The agent has a transparent and an understandable design and therefore it is easy for implementing. Instead of intensive CPU-consuming algorithms, reactive (immediately reacting) techniques are used.

- reactive modules
  - anti-crash module (immediate menacing collisions detection system),
  - moving (driving) module.

But, especially when the CPU is sleeping most of the time (because it is very fast or the moving is relatively slow), it comes handy to be able to use planning and/or scheduling techniques in an application. In an *ABATS* based application the real time reasoning modules are using all free (otherwise unused) CPU time.

- real-time reasoning modules
  - (fast) planning module,
  - (fast) crash predicting module.

3 INTER-AGENT COMMUNICATION

To add an agent to a multi-agent system in a real world it is necessary to use a communication platform. Such platform should enable peer-to-peer communication connections among the agents. Each agent is an autonomous unit, which can communicate with the other agents in the system by message passing. At least one of the agent's modules contains a set of communication operations for sending and receiving messages as well as a set of operations determining how to react on received messages (e.g., changing the internal state of an agent, forwarding messages, calling for a help of new agents etc).

Each agent is attached to the whole system via the communication modules.

Agents are independent, autonomous entities communicating in the peer-to-peer way among them. An area agent helps to better organize traffic on its area and knows about busy or unavailable

roads, collisions etc. But the corresponding pieces of the control strategy are mostly “owned” by individual agents – to eliminate a risk of overcoming of the area agent power (overloading its CPU).

Because this paper is oriented on new planning techniques, it covers only personal vehicles (car) transport in which only a small set of inter-agent communication functions is needed. A vehicle agent may ask an area agent which roads are not available (it may ask for updated road-maps). A vehicle agent can ask another vehicle agent for its driving parameters, speed, plans and goals. However, details about communication are not covered in the paper.

4 AGENT MODULES

A vehicle agent can be built modularly. Each vehicle agent’s module can be built as a standalone agent (such hierarchically built "multi-agent" agents are called holonic agents). A vehicle agent consists of the next main modules:

- **Sensor module**. It watches the nearest environment, vehicles, roadway status and the status of user’s health.
- **User agent module**. It detects the user’s needs (specified destination, the user’s preferences) and so on.
- **Inter-agent communication module**. It is responsible for communicating with other *vehicle agents* (vehicle central computers).
- **Area-agent communication module**. It is used for communication with a *local area agent*.
- **Goal priorities module**. It generates goals and their priorities according to the user and vehicle needs. For example a goal for saving user’s health is more important than the user defined destination. If the system does not have enough power for going to the destination, a higher priority might be fuel tanking. The highest priority always has the user safety – the *driving anti-crash module*.
- **Planning module**. It hierarchically generates logical plans depending on priorities of goals.
- **Crash predicting module**. It generates high priority level constraints and goals in situations when a possible vehicle collision is detected. It is an important module – for example used while a street crossing. It computes the ways of the nearest vehicles (and asks them if they are able to communicate) and tries to avoid of close approaching to them by setting driving constraints or by directly setting driving parameters if its activity would not interfere with operations of the *driving parameter setting module*.
- **Driving anti-crash module**. It works in all situations when an immediate collision is detected. It has its own saving rules for speed and direction change. It is the most important module – it always overcomes all the other modules.
- **Driving plans module**. This module generates a small number of the most important mutually following detailed plans – for example in which direction the vehicle should go, where to turn and so on.
- **Driving parameter setting module**. It takes a number of user, vehicle and roadway parameters from the *sensor module*, computes optimal settings and sets the driving parameters (speed, brake and elasticity constraints). Because the roadway parameters always a bit vary, the last known parameter could not be the only one who dictates the driving parameters. It should rather be the most important one. Driving parameters should be changed slowly, for example through a PID controller, because user’s comfort and safety are important and because the roadway status may change soon (for example while going through a pool).
- **Driving module**. This is the driving plans executing module. Depending on the settings from the driving parameter-setting module it uses sensors and (in real time) follows plans from the *driving plans module*.

5 GENERATING OF A PLAN

We can look on some of the vehicle car agent's goals as on a finding of an optimised way to achieve goals by fulfilling all constraints. This is the reason why while by building such systems the constraints satisfaction techniques comes handy.

A vehicle agent's planning module has to find optimised way between important points like frontier passages, bridges, big cities and so on according to the constraints. The first thing it should do is to create a set of important points, which should be included in the created way (plan). But the status of some roads may vary. They suddenly may become unavailable in winter for example. Therefore rather than detailed planning of the whole way, only the optimal (most effective) way to the first (nearest) important point should be found in the second step.

The planning module is adaptive – it proves if there exists a way to the nearest important point of the planned way (otherwise the system should choose another set of important points) and creates a first feasible plan. While executing the first plan (for example while going through the first planned road) the system estimates optimistic executing (driving) time which remains to the next turn point and tries to generate better (more optimised) plan in a shorter time. If it will find more effective way from this turn point to the nearest important point, it will fulfil the new plan instead of the old one. If the system finds the optimal way, it will try to find the most effective way to the next important point and so on.

Depending on priorities the planning module should make still more and more detailed plans – it uses hierarchical planning. But there is a possibility to use similar (or the same) algorithms for almost all plans.

We need

- a very fast planning algorithm for proving of constraints satisfaction and for creating the first feasible plan,
- fast planning algorithms for generating more effective plans,
- a planning algorithm for finding the optimal way.

6 PLANNING BY USING CONSTRAINT LOGIC PROGRAMMING

Constraint logic programming [1] is a declarative programming paradigm derived from logic programming. CLP is particularly useful for solving constraint satisfaction problems, which are formulated as a set of variables and constraints between these variables. The goal is to find such an assignment of values to variables that none of the constraints is violated. In addition, there can be defined a cost function which has to be optimised in the final solution. Typical representative of this class of problems are scheduling applications (e.g. Job–Shop [2]).

CLP serves powerful tools to handle this kind of problems. Constraints are used actively to prune the search space in an a priori way. Moreover, there are tools for defining of new constraints and ways how the constraints are to be handled by the user. Optimisation is supported as well by predicates which implement the branch and bound like hypothesis space searching strategy. By this strategy often a large number of iterations are needed to find an optimal solution. In this paper a real time quasi-optimal plans generating strategy is presented.

Plan creating strategy consists of four steps.

1. a heuristic capable to quickly find an initial solution of a good quality (upper bound),
2. a heuristic capable to find a good lower bound for the optimisation algorithm,
3. a heuristic for fast finding a solution better than upper bound,
4. an effective strategy for finding the optimal solution within a minimal number of steps.

A planning problem is defined by a main goal and a set of constraints. The ideal goal is to find the most effective way – arrangement of roads through which the vehicle should go.

Technological constraints are physical constraints – for example an airplane may not fall down, a client cannot be in two vehicles in the same time, there have to be safe inter-vehicle distances, it should be kept in mind the maximal motor's working time and the maximal user comfortable time.

7 OPTIMISATION METHODS IN A CLP

Traditional Optimisation Methods in CLP

In the CLP an optimisation is usually achieved by using a higher order predicate incorporating a method known from mathematical programming, namely branch and bound. Predicates with this functionality can be found in CLP languages like *CHIP*, *ECL^{i}PS^e* and others.

The basic idea is that branch and bound searches for a solution to the problem and after finding a new solution it adds a further constraint that any new solution must be better than the current best one with respect to the evaluation function. This strategy fits also very well with the standard backtracking search of most sequential logic programming systems.

There are essentially two strategies (available in *ECL^{i}PS^e*).

- ***min_max*** Starting with known upper and lower bounds (C^{max} and C^{min} respectively) for the evaluation function C firstly finds some solution by standard backtracking search, using the initial upper and lower bound as a constraint $C^{min} \#\leq C \#\leq C^{max}$ to prune the search space. Whenever a new solution is found with cost C_n (an integer number), the search halts and restarts using the tighter constraint $C^{min} \#\leq C \#\leq C_n - 1$ to further prune the search space. If no further solutions were found or $C_n = C^{min}$, then the last found solution is optimal.
- ***minimize*** In this case the process is the same, but after finding a solution procedure does not restart the search continuing further in the search space. On one hand this approach brings an advantage in comparison with previous approach avoiding wasted effort in many cases because it does not re-traverse the initial empty part of the search tree. On the other hand another problem may occur, called "trashing". Trashing occurs when many solutions with the same cost are topologically close in the search space, what will result in a lot of wasted search.

The author has improved these strategies in order to minimize the number of iterations as much as possible. The algorithms are described in the following section.

Logarithmic min_max

The basic idea behind improved methods is not only to decrease the upper bound of the evaluation function, but also take the whole interval $<C^{min}, C^{max}>$ and split it in the middle trying this value as new upper bound of the evaluation function. If a solution is found, the upper bound decreases to the value of its evaluation function. If no solution exists, the lower bound increases. In both cases the searching interval will decrease in each step to less than the half of it.

Both traditional methods (*min_max* and *minimize*) can be improved in this way. We call the resulting methods *logarithmic min_max* and *logarithmic minimize* respectively.

In each iteration it divides the interval to half size owing to it needs about $log_2(Max–Min)$ proofs of a solution existence. The classic *min_max* algorithm needs *Max-Min+1* proves of a solution existence (in the worst case). The proving of the solution existence is done by setting a constraint that says that the cost of the solution cannot be greater than *Limit* and by finding a solution limited by that condition.

In the following the *logarithmic min_max* algorithm will be presented. Variable called *AllowedDeviation* denotes the required distance between the cost of the found schedule and the cost of an optimal one. The first step of the algorithm uses the "*fix*" operation. This mathematical operation cuts the decimal part of a real number.

1. Find (heuristically) an initial solution and take its cost as the upper bound (*Max*).
2. Determine the lower bound (*Min*) greater then zero.
3. Let *Limit = Max – fix(Max–Min)/2*.
4. If *Limit* ≥ *Max*, then let *Limit = Max – 1*.
5. Can it be proven that there exists a solution with cost less than *Limit*? If yes, then let *Max = its_cost*, otherwise if not, let *Min = Limit + 1*.
6. Let *Deviation = 100 * (Max – Min) / Min*.
7. If *Max = Min* or *Deviation* ≤ *AllowedDeviaton*, find a solution with cost equal or less than *max(Min, Max)* (we know that there exists a solution with such limitation) and **stop**, otherwise go to the step 3.

For a good performance of the algorithm, initial upper and lower bounds are not as important as for traditional *min_max*. Stating upper bound means to find an initial solution using a deterministic heuristic algorithm, i.e. quickly and as good as possible. Stating lower bound means to estimate the distance of a solution to the optimal one (under this bound does not exist any solution).

Proof of the solution existence is done by setting a constraint that the cost of the solution cannot be greater than *Limit* and then by finding a solution limited by this condition. It can be saved some time in the step 7 if the solution found in the step 5 can be stored.

Similar works the *logarithmic minimize* algorithm with the same difference as by the *minimize* when compared with *min_max*, i.e. after finding a new (better) solution the search continues and does not restart as it does by the *logarithmic min_max.* Both methods were implemented in the CLP language ECL^iPS^e [1]. The author achieved the results on a number of randomly generated Job–Shop scheduling problems. The results were presented in [3].

Logarithmic minimize

The author has also developed the *logarithmic minimize* algorithm. It is much faster than standard optimisation predicate *minimize*, because it minimizes a risk of trashing.

Approximate Logarithmic min_max

It may happen that while running the logarithmic *min_max* we repeatedly will get the some upper bound and only the lower bound (*Min*) will change. The reason is a simple fact that the *logarithmic min_max* (or the *logarithmic minimize*) algorithm quickly finds the optimal result (to be used as an upper bound) but the proof of its optimality takes too much time by decreasing the <*Min, Max*> intervals – whenever one tries to set the variable *Limit* to less number than *Max*, a searching for a result fails. Then we can try to set *Limit* directly to *Max–1* to prove optimality of a result with cost *Max* (which we have saved).

On the other hand, if we repeatedly will get the same minimum, it is possible that this is the cost of the optimal solution. In such a case we can save some time if we try to set *Limit* directly to *Min+1*.

Instead of setting the *Limit* directly to *Min+1* or to *Max–1*, we can try to use another splitting mechanism of the <*Min, Max*> interval. The author of this paper tried to split interval to non-equal parts (1:2 to 1:3, 1:4, 1:5...), but he got better results with the directly settings [6].

In this way he developed a modification of the *logarithmic min_max* and *logarithmic minimize* algorithms. If the *approximate* version of *logarithmic min_max* detects *N* proofs with the same upper bound, it directly sets *Limit* to *Max – 1* for the next iteration and if there cannot be found a result the cost of which is lower or equal to *Limit*, the *approximate logarithmic min_max* algorithm returns the last but one schedule as the optimal one.

The question is, which number should be set *N* to. There were tested *30* randomly generated Job-Shop problems (for 5, 8 and 10 machines) and the author tried to set the *N* to *2*, *3*, *4*, *5* and *6*. In the benchmark tests the best results gave the *approximate logarithmic min_max* with *N=3*.

Adaptive Approximate Logarithmic min_max

Another method how to increase the *min_max* speed is to take in mind the time which the algorithm spends by finding a result. If (compared to the previous success) the algorithm spends too much time while trying to find a solution under the *Limit* and then fails (this is a prove that there does not exist a result with a cost equal or less then *Limit*) then it is very probable that the *Cost* of the optimal result is nearer to *Limit* than to *Max*. On the other hand, when the proving of non-existence was much more faster than the successive finding of a result with the cost of *Max*, it is probable that the optimal cost is nearer to *Max*. The computing time (or number of backtracks) plays an important role here. In disjunctive scheduling or planning finding of a feasible result is often much more faster than a look for an optimal result. This is because in the optimal schedule there is only one way or a very small number of ways in search space and the branch and bound algorithm must do much more backtracks for finding them.

The final version of modified *min_max* algorithm is the *adaptive approximate logarithmic min_max*. Variables called *STime* and *FTime* denote the times used by successful finding of a result or by failed probe respectively.

1. Let *STime=0*, let *FTime=0*.
2. Find (heuristically) an initial solution and take its cost as the upper bound (*Max*).
3. Determine the lower bound (*Min*) greater then zero.
4. If *STime > 10*FTime* and *FTime* ≠ 0, then let *Limit = Max – fix(Max–Min)/3*
   else if *FTime > 10*STime* and *STime* ≠ *0*, then let *Limit = Max – 2*fix(Max–Min)/3*
   else let *Limit = Max – fix(Max–Min)/2*.
5. If the *Min* repeated 3 times, then let *Limit = Min+1*
   else if the *Max* repeated 3 times, then let *Limit = Max–1*.
6. If *Limit* ≥ *Max*, then let *Limit = Max – 1*.
7. If it can be proven that there exists a solution with cost less than *Limit*, then let *Max = its_cost* and set *STime* to time spanned by solution existence proving, if not, then let *Min = Limit + 1* and set *FTime* to time spanned by failed solution existence proving.
8. Let *Deviation = 100*(Max – Min)/Min*.
9. If *Max = Min* or *Deviation* ≤ *AllowedDeviaton*, find a solution with cost equal or less than *max(Min, Max)* (we know that there exists a solution with such limitation) and ***stop***, otherwise *go to* the step 4.

Results

The algorithms were tested on a number of randomly generated job-shop scheduling problems of size 10x10 (to prove the universality of the new created algorithms). In the next table, for each problem the resulted time (in seconds on PC AMD486 120MHz running under Linux operating system) is given. From the traditional methods only the *min_max* could be used for these problems (*minimize* suffered from trashing). The results are compared with new optimisation methods the *approximate logarithmic min_max* (in the table shown as *al min_max*) and the *adaptive approximate logarithmic min_max* (*aal min_max*). In all cases the same initial lower and upper bounds were used.

Table 1. Speed of the *min_max* algorithm and its logarithmic modifications

	min_max	al min_max	aal min_max
test file	time (s)	time (s)	time (s)
10_10_0	1276,87	79,07	79,04
10_10_1	250,44	30,53	30,60
10_10_2	364,81	223,09	207,82
10_10_3	4074,73	608,35	601,08
10_10_4	228,70	84,07	72,49
10_10_5	76,97	42,51	41,55
10_10_6	144,24	32,95	32,90
10_10_7	10647,00	8404,28	5855,17
10_10_8	241,05	87,20	84,72
10_10_9	951,59	613,41	613,79

8 OPTIMISATION CRITERIA

There are two important criteria, which the user may want to satisfy:

1. the vehicle should go from the starting point to the destination point as fast as possible,
2. the vehicle should go from the starting point to the chosen destination by maximum fuel saving.

Therefore the cost function for one length unit of a road R_i may look as follows:

$$C_u(R_i) = K*F_u(R_i)+1 \qquad (1)$$

where $C_u(R_i)$ is the cost of one length unit (for example one meter) of the road R_i,
$F_u(R_i)$ is fuel consumption for one length unit of the road R_i,
and K denotes the degree of compromise (it must be a number equal or greater then zero). If $K=0$ then the fuel consumption will be ignored and only the number of length units will be important – the algorithm will find the shortest way to the destination. If the K parameter is a high number, the fuel saving will be very important for the optimisation algorithm.

The (global) cost of N used roads will then be the sum of N "road costs":

$$C = \sum_{i=1}^{n} (L(R_i) * C_u(R_i)) \qquad (2)$$

The C is the cost of plan for the optimisation algorithm.
$L(R_i)$ is the used length of a road (R_i).

Upper Bound

The *logarithmic min_max* algorithm and all its modifications needs to know an upper bound of the cost of a solution (in our story the cost of a plan). If we know that the cost of the optimal plan will not be greater than some limit, this limit is called the upper bound. But it is not important to get very good upper bound because the interval <*Min, Max*> will be decreased to half size in the first iteration. The heuristic used for an upper bound finding works as follows:

1. Set the *RoadsSet* – set of roads on the planned way – to empty set.
2. Set the current point to start point.
3. Choose a new *StepSet* – set of *directly* "usable" roads – roads directly connected to the road on which the current point is now.
4. Sort the roads in the *StepSet* from the most effective to the less effective one.

5. If the destination lies on one of chosen road, add it to the *RoadSet* and exit.
6. Extract the first road from the *StepSet* and add it to the *RoadsSet*.
7. Move the current point to the start of the new road.
8. Go to step 3.

In a real application the algorithm backtracks. The whole algorithm depends on the sorting rule. For our purposes the most effective road is the most effective one for modified branch and bound algorithm. It is the road which end is closest to the destination from the ends of other roads located in the *StepSet*.

Lower Bound

The *logarithmic min_max* algorithm needs to know a lower bound. If we know that the cost of the result cannot be lower than some limit, this limit is called the lower bound.

We can choose the lower bound as the cost of a fictive road (abscissa on the road-map) starting at the start point and ending at the end point.

It is possible to increase the lower bound if we do some partial planning. For that purpose, we can use the *logarithmic min_max* algorithm. For speeding up the lower bound finding, the author tried to use partial plan of a small number of roads (the algorithm is similar to the one for upper bound searching). The *logarithmic min_max* algorithm stops when the lower bound of partial schedule is less than 10% closer to the cost of the optimal partial schedule. This algorithm allows us to find a good lower bound very quickly. But it is not necessary for the *logarithmic min_max*.

9 FAST OPTIMIZATION

For a looking for a sub-optimal plan the 7:5 heuristic modification of branch and bound algorithm was designed. It is very simple algorithm, which says:

1. Try to optimise only first 7 roads from the current point by using branch and bound algorithm.
2. If the destination was not reached, set the first 5 roads as optimal and set the current point to the fifth (not to the seventh) road.
3. Go to point 1.

Please note, than not all roads in the locally optimised way are used in the next step. Only the first five from seven roads (or tasks by Job-Shop solving) are used. The numbers 7 and 5 has been chosen experimentally.

With this simple algorithm (together with the *approximate logarithmic min_max* algorithm) it is possible to sub-optimally solve 10x10 Job-Shop problems in range of seconds. Large hypothesis space can be searched this way.

10 CONCLUSIONS

In the paper, the *ABATS* multi-agent system was presented. The vehicle agent tries to find a sub-optimal way (which can be found in seconds) and then it tries to find better (or even the optimal) one. Besides of intelligent agent technology, the *ABATS* uses some new artificial intelligence techniques for lower and upper bounds finding and some scheduling and planning techniques like *adaptive approximate logarithmic min_max*.

The traditional methods *min_max* and *minimize* have been improved in order to minimize necessary iterations during the search phase. As a result the performance of the new methods is much better with respect to the computational time.

Thanks to 7:5 heuristic optimisations, large hypothesis space can be searched in range of seconds.

11 FUTURE WORKS

All techniques have been designed and implemented into *ECL^{i}PS^e* as a part of the author's dissertation. Besides this, the author is designing the second generation of the *ABATS*. *ABATS II* is designed for an autonomous multi-level fault tolerant control of complex transport systems (mixed bus/train/airplane/ship transfer systems).

REFERENCES

1. Schmotzer, M.: Analysis and Design of New Methods to Solve Time Scheduling Using Constraint Logic Programming. Master's thesis, Technical University of Košice (1997)
2. French S.: Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop. Wiley & Sons, Chichester, England (1987)
3. Paralič J., Schmotzer M., Csontó J.: Optimal Scheduling Using Constraint Logic Programming. In: Proceedings of 8th Symposium on Information Systems IS'97, Varaždin (1997) 65-72
4. Camarinha-Matos L.M., Vieira W.: Using Multiagent Systems and the Internet in Care Services for the Ageing Society. In: Proceedings of the BASYS '98 - 3rd IEEE / IFIP International Conference on Information Technology for BALANCED AUTOMATION SYSTEMS in Manufacturing, Prague (1998) 33-47
5. Miksch S., Cheng K. and Hayes-Roth B.: An Intelligent Assistant for Patient Health Care. Report No. KSL 96-19, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, Stanford, California, USA (1996)
6. Schmotzer M.: Predictive approximate logarithmic min_max. Internal research report, Department of Cybernetics and AI, Technical University of Košice, Košice (1998)
7. Sudzina F.: Manažérske informačné systémy a marketingové informačné systémy, In.: Implementácia marketingových teórií do hospodárskej praxe Slovenskej Republiky, Českej Republiky a Poľskej Republiky, Ekonomická Univerzita v Bratislave, Podnikovohospodárska fakulta v Košiciach, vydavateľstvo Ekonóm, Košice (2002) 145-148

Biography

Milan Schmotzer, M.Sc. studied Technical Cybernetics and Artificial Intelligence at the Technical University of Košice. In 1997 he received his M.Sc. degree (Master thesis: Analysis and Design of New Methods for Time Scheduling Problems Solving in a Constraint Logic Programming Environment.). After his Ph.D. study at the Technical University of Košice he is working as a teacher at the Pavol Jozef Šafárik University in Košice. His research is focused on a CLP based planning and scheduling, multi-agent intelligent systems and knowledge systems, as well as on developing of new tumour-killing psychology and surgery techniques. Now he is setting up a larger project about a self-managed multi-agent transport system – a theme for his (AI) dissertation thesis.

Author

Ing. Milan Schmotzer
Pavol Jozef Šafárik University
Faculty of Science, Department of Informatics
Jesenná 5, 040 11 Košice, Slovak Republic
Tel.: +421 7 60291111 Fax: +421 7 60291111
e-mail:schmotze@kosice.upjs.sk