AI Planning with Time and Resource Constraints

Filip Dvořák and Roman Barták

Department of Theoretical Computer Science and Mathematical Logic
Faculty of Mathematics and Physics, Charles University in Prague
Malostranské nám. 2/25, 118 00 Praha 1, Czech Republic
filip.dvorak@runbox.com, roman.bartak@mff.cuni.cz

Abstract

Introduction of explicit time and resources into planning that typically focuses on causal relations between actions is an important step towards modelling real-life problems. In this paper we propose a suboptimal domain-independent planning system Filuta that focuses on planning, where time plays a major role and resources are constrained. We benchmark Filuta on the planning problems from the International Planning Competition (IPC) 2008 and compare our results with the competition participants.

Introduction

In this paper we focus on fully observable, deterministic temporal planning with resources (Ghallab, Nau, & Traverso, 2004). In particular, the world state is specified using a set of multi-valued state-variables where different states are distinguished by different values assigned to the state-variables. The values of all state-variables are specified for the initial state, while the goal state is specified by required values of certain state-variables. Actions have known duration, require particular values of certain state-variables for execution (precondition) and change values of some state-variables at some time point of execution (effect). Resource constraints can then be naturally described using the state-variables, where the value is changed relatively (increased or decreased) rather than being set absolutely. The planning task is to find a set of actions allocated to time such that the time evolution of state-variables is feasible (each state-variable has a unique value at each time point and this value is consistent with actions being executed at this point) and the final values of state-variables satisfy the goal condition. The quality of plan is measured by time needed to reach the goal state – makespan. Plans with a smaller makespan are preferred. Filuta is a sub-optimal domain-independent planning system that solves the above sketched planning problems.

We will first describe the formal representation of the planning problem consisting of temporal databases modelling evolution of state-variables and resource models. Then we will show how to solve the planning problem by integrating search decisions with maintaining consistency of temporal databases and resource models. Finally, we will demonstrate the quality of Filuta by comparing it with the state-of-the-art planners from IPC 2008.

Representation

The cornerstone representation we build on is the *state-variable representation* for classical planning (Bäckström & Nebel, 1995). The domain of a state-variable contains facts about the world such that no two facts from one domain can be true at any given time. A state of the world can then be defined as an n-tuple of values of n state-variables. To capture the evolution of the state-variable in time we only need to keep the changes of its value, which is the role of *temporal databases*. *Resources* in general describe broad range of world properties. To take advantage of existing techniques for resource reasoning, we use different representation for each resource type (and also a resource-specific solver). The temporal databases and resources interfere with each other through shared *temporal reasoning*.

Temporal Reasoning

Temporal reasoning is managed as a Simple Temporal Problem (STP) (Dechter, 2003). We incrementally maintain a Simple Temporal Network (STN) in its minimal form. Formally, STN = (X,C) consists of a set of time points X and a set of binary constraints C between the time points in X. A binary constraint $[a,b]$ for a pair of time points x_1, x_2 determines that x_1 occurs at least a and at most b time units before x_2. An update of the STN is a triple $(x_1,x_2,[a',b'])$ and we say that it is a consistent update if $\max(a',a) \leq \min(b',b)$, where $[a,b]$ is the minimal constraint between x_1 and x_2 in the STN.

The upside of maintaining a minimal network is mainly in the constant time detection of inconsistent updates, possibility to solve resource sub-problems upon a smaller sub-network (taking only a subset of time points) and

constant access to lower bounds on time between helpful time points (e.g. the lower bound on makespan).

The downside is the need to perform expensive propagation of transitive closure, which also generates many unhelpful constraints. Using symmetry of the constraints and implicit constraints we can reduce the number of stored constraints to $(n^2 - n)/2$, where n is the number of time points. Further we can omit any time point that becomes redundant during the planning; once the constraint between any two time points reduces to [0,0], we can safely say, that one of the time points is unnecessary.

Temporal Databases

Our approach is similar to chronicles in the IxTeT system (M. Ghallab, 1994). For each state variable we use a single temporal database that consists of a partially ordered sequence of changes and requests, where a *change* represents the change of the state variable's value and the *request* represents a request on the state variable to keep certain value for a period of time.

Formally, for a state-variable with domain D, *change* is a quadruple $(x_s,x_e,v_{initial},v_{final})$, where x_s, x_e are time points, $v_{initial}$, $v_{final} \in D$, and *request* is a triple (x_s,x_e,v), where x_s, x_e are time points and $v \in D$. We say that the temporal database is consistent, if any two consecutive changes share the inner value and any request between those two changes shares their inner value as well.

The partial ordering of the changes and requests consists of total ordering of the changes, which is constructed as a result of strong decisions of the search algorithm, and partially ordered requests. Figure 1 illustrates an example of the temporal database.

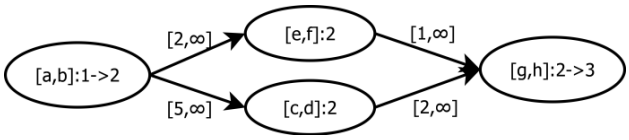

Figure 1. Illustration contains two changes (1→2 and 2→3) and two requests on the value 2. Time points are represented as letters a-h. The labels of arcs represent constraints from the underlying temporal network (e.g. b happens at least 2 and at most ∞ time units before e), which also determine the total ordering of the changes. The temporal relations between requests are unimportant with regard to the temporal database.

Single temporal database for a state variable can be conceptually seen as a *timeline*, a structure known in context of planners RAX-PS (Jonsson, Morris, Muscettola, Rajan, & Smith, 2000) and EUROPA (Frank & Jonsson, 2001), recently also forming a base of systems Timeline-based Representation Framework (Cesta & Fratini, 2008) and Constraint Network on Timelines (Verfaillie & Prelet, 2008). In terms of our temporal databases, timelines contain solely the requests on values, while the function of changes is provided by various approaches, often formulated as a CSP. By totally ordering the changes in the databases we sacrifice some flexibility of the final plan in favour of the planning system performance; the ordering of the changes is what makes our temporal databases different from recent partial order causal link planners such as VHPOP (Younes & Simmons, 2003)

Resources

Though *resources* can be modelled via state-variables, we approach the modelling of resources separately by creating for each planning problem a set of *resource instances*, where each instance corresponds to a single resource appearing in the problem. By itself the resource instance is a set of *resource events*, which take different forms based on the type of resource the instance is representing. In Filuta we have modelled well known unary resources, discrete resources and reservoirs (Laborie, 2001).

Unary Resource corresponds to a single machine that can support only one activity at any given time. An instance of the unary resource is a set of resource events, where each event consists of a pair of time points that represent the start and the end of the event.

Discrete Resource corresponds to a pool of multiple uniform machines. An instance of the discrete resource is a set of resource events, where each event is defined as a triple (x_s,x_e,rq), where x_s, x_e are time points, and $rq \in N$ represents the number of required machines. Each resource instance has a fixed capacity.

Reservoir is a resource that can be consumed and produced and consumption and production events may not happen in tandem. An instance of the reservoir resource is a set of events, where each event is defined as a pair (x,e), where x is a time point and $e \in Z$ is a relative change of the resource level; $e < 0$ represents consumption and $e > 0$ represents production. Each instance has fixed capacity.

Actions

Actions are grounded temporal operators that describe changes of the state-variables' values, requests on values of the state-variables, and resource events on the resource instances. Each action includes temporal parameters representing the start and the end of the action; *action instances* are derived from actions by instantiating their temporal parameters allowing multiple instances of a single action in the plan.

Formally, action is a sextuple (tp_s, tp_e, dur, CHs, RQs, REs), where tp_s and tp_e are time point parameters, dur $\in N$ is a duration of the action, CHs is a set of changes of the state-variables, RQs is a set of requests on the state-variables, and REs is a set of resource events (consumption/production) upon the resource instances.

For example we can imagine an action *load-truck3-package2-location1* that represents loading the package2 into the truck3 at the location1. The action takes 5 time units to execute, the truck has a limited capacity, loading a package requires a crane and the package2 requires 11 units of space. We further assume we have state-variable sv_p and sv_t, where sv_p represents the position of the package2 and sv_t represents the location of the truck3. The

corresponding action in our representation would be constructed as (x,y,5,{sv_p[x,y]:location1→truck3}, {sv_t[x,y]:location1},{crane[x,y],truck3-cap[y]:-11}), where sv_p[x,y]:location1→truck3 depicts the change of package position over time interval [x,y], sv_t[x,y]:location1 depicts the request on the location of truck3, crane[x,y] is an event for the unary resource instance representing the usage of the crane, and truck3-cap[y]:-11 depicts a consumption event upon the reservoir resource instance representing the space in truck3.

The knowledge of action duration is restrictive with regard to solving certain real-world domains. With minor extension of the system, we can go further and allow the actions' durations to be specified as an interval representing its minimal and maximal estimated duration, while still being able to efficiently manage temporal relations in the simple temporal network. Such extension would add more flexibility into the final plan.

Planning Problem

We define the *planning problem* as a quadruple (TDBs, RIs, Actions, Goals), where TDBs is a set of temporal databases, each corresponding to a single state-variable and containing the initial value of this variable, RIs is a set of resource instances, Actions is a set of actions and Goals is a set of goal values of state-variables.

The solution of the planning problem is a set of *action instances* allocated to time (a plan) such that the last values of the state-variables' temporal evolutions are the goal values, all temporal databases are consistent, underlying temporal network is consistent, all resource instances are consistent, and all changes, requests and resource events from the actions instances in the plan are settled in the corresponding temporal databases and resource instances.

The definition of planning problem does not consider intermediate goals; however the system can be extended to accommodate them. They can be either specified in the initial temporal databases (together with time points and temporal constraints in the initial temporal network), or we can include them into the set of Goals, which would further require some precedence constraint to distinguish intermediate and final goals attached to the same state variables.

Translation

The planner accepts planning problems defined in PDDL (typing, durative-actions and partially numeric-fluents). Since the numeric fluents are more general than the modelled resources, the only accepted numeric fluents are those that either represent modelled resource or disappear through grounding. The numeric fluents that represent resources are automatically translated into the planner's representation by checking grounded actions for increase and decrease effects, corresponding fluents and their numerical comparisons, and creating resource instances instead of the fluents.

Solving Approach

For a given planning problem we use a single STN, whose time points are used for temporal annotation of changes, requests and resource events in temporal databases and resource instances. The resource reasoning is realized by a resource manager, which keeps a least-commitment approach by maintaining the potential resource conflicts (overconsumptions and overproductions of a resource) as a CSP. Upon the state-variables we further build *domain transition graphs* (Jonsson & Bäckström, 1998).

Domain Transition Graphs

The domain transition graph (DTG) for a state-variable with domain D and a set of actions S is a directed multigraph (V,E), where V = D and an action from S represents arc $(v_i,v_j) \in E$ if and only if the action contains a change of the state-variable from v_i to v_j.

Having the domain transition graphs generated, we can look at the planning problem as a problem of finding paths from the initial node (which represents the initial value of the state-variable) to a goal value in each DTG (whose state-variable contains a goal value). However traversing a single arc in a domain transition graph represents adding the action into the plan. Such action then also represents traversing an arc in other domain transition graphs (for each change it contains), and the action may contain a request on certain value of another state-variable. To support these collateral transitions and requests, we need to traverse all other domain transition graphs to the point when the original transitions and requests do not violate consistency of the temporal databases, which is in principle the same problem as traversing the graph to satisfy a goal.

Since we construct DTGs in advance, we can also calculate shortest paths for them, and use the paths to guide the search algorithm. We calculate two types of shortest paths. T-P measures the length of the path in a graph as the minimal time needed to traverse the path (a sum of durations of actions traversed). OT-P measures the minimal number of arcs traversed, while less time demanding paths are preferred.

Resource Manager

For each category of resources we built an incremental solver. The input of the solver is an STN, a resource instance (a set of events), and one new event for this instance. The solver determines whatever the new event may cause an overproduction or overconsumption conflict in the resource instance, and if the conflict can be prevented by updating the temporal network with an appropriate set of new constraints – *resolvers*. The output of the solver is defined as a set SR = $\{S_1,\ldots,S_n\}$, where S_i is a set of resolvers – updates of the temporal network that prevent a single resource conflict. To prevent a resource conflict having the output of the solver, we have to choose from each set S_i (at least) one update, such that the set of chosen updates is consistent with the temporal network.

Trivial cases occur when SR = ∅, which indicates that no conflicts need to be resolved, and ∅ ∈ SR, indicating that some resource conflict cannot be resolved.

Planning problems generally contain multiple resource instances and the solvers together often produce multiple sets SR_1, …, SR_n. The formulation of the solvers output now becomes helpful as we can aggregate the outputs into one set $SR = SR_1 \cup \ldots \cup SR_n$. The purpose of the *resource manager* is to maintain the aggregated set SR of resolvers. The maintenance consists of removing updates inconsistent with given STN and checking the existence of solution (a selection of an update from each element of SR). Given an STN, to find a solution for SR we run a depth-first search in the space of possible choices of updates, while each choice is followed by realizing the update operation upon the STN and consequent removal of inconsistent updates. To improve efficiency, the resource manager works only with a sub-network of the STN (taking only the time points contained in the updates in SR).

In other words, the resource manager checks the existence of solution for a Disjunctive Temporal Problem (Stergiou & Koubarakis, 1998), which is incrementally built from the sub-network of the current STN and disjunctive constraints imposed by the resource-specific solvers.

We can see the resource manager as a coordinator of multiple resource-specific incremental solvers that are invoked only when a new resource event is introduced by an action, in which case the concerned solver produces a set of sets of temporal constraints, which are from that point on handled solely by the resource manager as a DTP.

The architecture of the resource manager is highly extensible; other resource models can be "plugged in" to support different types of resources. Current resource models include solvers for unary resources, discrete resources and two types of reservoirs (one supports only relative consumption/production events, the second one supports asymmetric events – e.g. resource is consumed relatively and produced absolutely, we can imagine an example of such resource as the fuel in a car that is consumed by driving the car (relative consumption) and the car is always refuelled to maximum capacity (absolute production).

Search Algorithm

In the Filuta system we adapted the plan-space planning approach (Ghallab, Nau, & Traverso, 2004), where the search space consists of states representing partially specified plans (note that the search state differs from the world state). For a planning problem (TDBs, RIs, Actions, Goals) we define the *initial state* as a quintuple s_0 = (STN, TDBs', RIs', SR, Plan), where TDBs' = TDBs, RIs' = RIs, SR = ∅, Plan = ∅ and STN is the *initial temporal network*.

The initial temporal network consists of a set of helpful time points. We first insert a pair of time points $x_{g\text{-}start}$ and $x_{g\text{-}end}$ and update the network by ($x_{g\text{-}start}$, $x_{g\text{-}end}$, [0,∞]); the time points represent global start and end of the world. Any further time point x inserted into the network is implicitly constrained by ($x_{g\text{-}start}$, x, [0,∞]). We further insert a time point $x_{i\text{-}end}$ for each $TDB_i \in$ TDBs and update the network by ($x_{i\text{-}end}$, $x_{g\text{-}end}$, [0,∞]) for each such time point; these time points represent local ends of the world upon the evolution of the corresponding state-variables. Whenever a request or a change is inserted into a TDB_i, the later time point x_e contained in the request or the change is constrained by (x_e, $x_{i\text{-}end}$, [0,∞]). The temporal relations between helpful timepoints are illustrated in Figure 2.

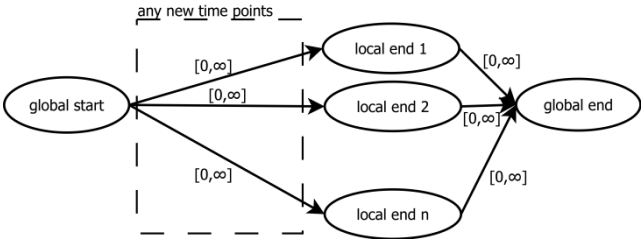

Figure 2. Illustration shows the initial configuration of helpful time points in the temporal network. Critical paths that are further propagated into the network allow estimating lower bound on makespan as the minimal value of constraint between global start and end, while constraints between global start and local ends provide heuristic estimates for "workload" upon evolutions of different state variables. Any new time points inserted into the network due to insertion of an action instance into the plan or creation of a goal request (temporally) fit between global start and (some) local end.

For a planning problem (TDBs, RIs, Actions, Goals) the *solution state* is such a state (STN', TDBs', RIs', SR, Plan) that the goals are satisfied (goal requests are the last in the temporal databases), STN' and TDBs' are consistent, and the set SR of resource resolvers has a solution (decided by the resource manager). The solution state is transformed into a solution of the planning problem by finding an optimal solution for SR upon STN' (the resource instances become consistent) and instantiating STN' starting with assignment $x_{g\text{-}start} \leftarrow 0$ and assigning the lowest possible value to all other time points. The Plan then contains a fully scheduled set of action instances that solve the planning problem.

The states of the search space evolve from the initial state s_0 by insertion of actions into the Plan, insertion of changes, requests and events of these actions into the corresponding temporal databases and resource instance, insertion of new time points (and constraints) into the temporal network (two time points per action instance), and insertion of *goal requests* into the temporal databases (a goal request is constructed from one new time point and the goal value of the state-variable). Solving the planning problem consists of finding a solution state that is reachable from the initial state.

State Evaluation

For a problem (TDBs, RIs, Actions, Goals) we denote the set of all possible search states as S. For a state s ∈ S we define *ms*(s) to be the smallest distance between $x_{g\text{-}start}$ and

$x_{g\text{-}end}$ in the corresponding STN (the lower bound for makespan), and *ft*(s) to be the sum of smallest distances between $x_{g\text{-}start}$ and $x_{i\text{-}end}$ for all end points in TDBs (the lower bound for the sum of times to achieve all last values in TDBs).

We define the state evaluation function *eval*: $S \rightarrow N \times N$ as *eval*(s) = (*ms*(s), *ft*(s)) and the goal of planner is to find a reachable solution state with the lexicographically minimal value of the *eval* function.

The state evaluation reflects simple empirical heuristic that it is better to choose less time demanding actions even if in the current context the estimate for makespan does not change; additionally the *ft* estimate supports "load balancing" among time requirements of the state variables' evolutions.

Search Procedures

The search algorithm divides into four interleaved search procedures *root_search*, *way_search*, *support_search* and *resource_search.* The input of all procedures is a state and the current upper bound, which can be an evaluation of the best state found so far, it can be given arbitrarily (makespan of the previous random restart), or it can be unknown (represented as (∞,∞)). The output of the procedures is a state, where a state = ∅ indicates that either all states in sub-tree were pruned (the lower bound exceeded the upper bound), or the sub-tree does not contain the intended partial solution. The interaction between search procedures is depicted in Figure 4.

For a problem (TDBs, RIs, Actions, Goals), the *root_search* (Algorithm 1) proceeds by picking a goal value of a state-variable from Goals that is not currently achieved (the last change in the corresponding TDB does not support the goal value), building a goal request (from a new time point in STN and the goal value) and calling the *way_search* to find a way in the corresponding DTG to support the goal requests (lines 05-09). The process is iterated until a solution state is found; the solution state is constructed incrementally as the first call of the *way_search* takes the initial state s_0 and returns state s_1, which is taken by the consecutive call of the *way_search* and so on (a goal request can be constructed multiple times for one goal value as *way_search* may invalidate a previously achieved goal). This is similar to STRIPS algorithm for classical planning.

The *way_search* (Algorithm 1) deals with the problem of finding a way in a domain transition graph from an *anchoring change* (for the goal request the anchoring change is the last change in the corresponding TDB, otherwise the anchoring change is provided by *support_search*) to a given *fact* that is either a change or a request. The *way_search* initially imposes new constraints into the current STN to improve the lower bound according to *eval*; the constraints represent the minimal time needed to traverse the path in DTG from the final value of the anchoring change to the initial value of the fact (we use the value of the shortest path T-P). To find the best path in the DTG (according to *eval*) *way_search* recursively performs a depth-first search in the DTG, where each arc traverse represents insertion of an action instance into the plan (an instance is created from the action representing the traversed arc and two new time points). The collateral transitions imposed by the inserted action instance are passed to *support_search* (line 27), whose output state is passed to the next step of the depth-first search. The search is guided by the shortest paths OT-P (the shorter paths are tried first). If the anchoring change is not the last change in TDB, the *way_search* also finds a way back (from the final value of the fact to the initial value of next change in TDB). In essence, the way_search procedure either extends the sequence of changes or adds a hitch as illustrated in Figure 3. The boolean parameter *jump* of the way_search procedure determines if the procedure should continue to finish a hitch (finding a path back to support the next change).

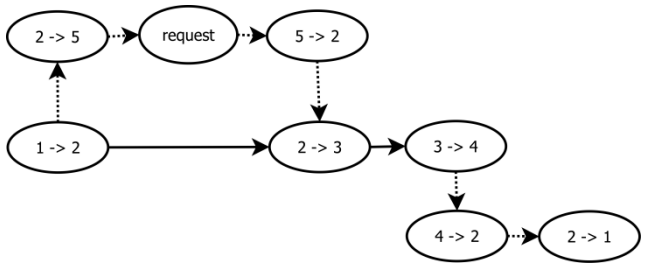

Figure 3. Illustration of an example where a sequence of changes (1→2; 2→3; 3→4) in a temporal database is extended once at the end (4→2; 2→1), and once by a hitch (2→5; 5→2) to support a request.

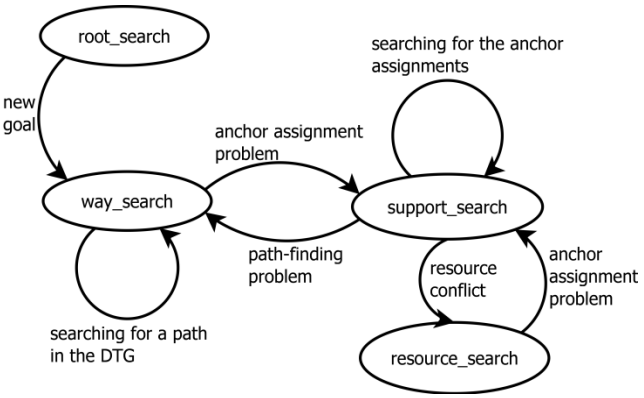

Figure 4. Illustration of the interactions between search procedures. The labelling of arcs shows for what purpose the procedures are called. The loops upon way_search and support_search represent recursive depth-first searches.

The task of *support_search* (Algorithm 1) is to find an anchoring change for each fact from a given set of facts (changes and requests that contain the time points propagated from the action instance) such that solving all the resulting path problems (finding the paths through *way_search*) produces the best state according to *eval*. The *support_search* performs a depth-first search in the space of possible assignments of the anchoring changes to the

facts. The search is guided by the *fewest-options-first* principle.

Algorithm 1. Search procedures root_search, way_search and support_search.

```
01  root_search(s0, goals, bound)
02    open_goals ← goals
03    s ← s0
04    while open_goals ≠ ∅
05      foreach goal ∈ open_goals
06        tp ← new time point in s.stn
07        change ← the latest change in s.TDB
08        request ← (goal, tp)
09        s ← way_search(s, change, request, bound, false)
10        if s = ∅ return ∅
11        update open_goals with s
12    return s
13
14  way_search(s, ch, fact, bound, jump)
15    if eval(s) > bound return ∅
16    if ch.v_final = fact.v_initial
17      if jump
18        ch' ← next change after fact in s.TDB
19        s ← way_search(s, fact, ch', bound, false)
20      return s
21    my_best ← ∅
22    foreach a ∈ aplicable_actions
23      bound ← min(eval(my_best), bound)
24      ai ← new action instance derived from a
25      s.Plan ← s.Plan ∪ {ai}
26      facts ← all changes and requests in ai
27      found ← support_search(s,facts,bound)
28      if found ≠ ∅
29        ch' ← next change in found.TDB added by ai
30        found ← way_search(found,ch',fact,bound,jump)
31        if found ≠ ∅ my_best ← found
32   return my_best
33
34  support_search(s, facts, bound)
35    if eval(s) > bound return ∅
36    my_best ← ∅
37    choose f ∈ facts
38      foreach ch ∈ suitable changes for fact
39        bound ← min(eval(my_best), bound)
40        if ch is the last in s.TDB
41          found ← way_search(s,ch, f,bound,false)
42        else
43          found ← way_search(s,ch,f,bound,true)
44        if found ≠ ∅
45          found ← support_search(found, facts\{f}, bound)
46          if found ≠ ∅ my_best ← found
47    return my_best
```

The *resource_search* procedure (Algorithm 2) is called whenever the set SR in the current state becomes inconsistent (the consistency check fails); this occurs mainly upon the insertion of a resource event into a resource instance. The *resource_search* identifies the inconsistent resource instance and systematically tries to extend the plan by an action that contains a helpful event for the inconsistent resource instance and the choice of the action is the best according to *eval*; for example the helpful event can be a production event for a reservoir instance, which was over consumed. Same as in *way_search*, the facts (changes and requests in the chosen action) are passed to *support_search*.

In essence, the search procedures branch on choices of actions in domain transition graphs (way_search), choices of temporal context of facts (anchor assignments in support_search) and choices of actions to resolve a resource conflict (resource_search), while the lower bound is carried in the simple temporal network (the lower bound comes from the propagation of critical paths T-P into the network). Then for each open goal, the algorithm tries to find combination of action instances and ordering constrains such that merging them with the current partial plan (rather than adding them to the end of plan) produces another partial plan that satisfies the goal and is the best according to the state evaluation.

Although the procedures way_search, support_search and resource_search perform complete depth-first searches, they are in global sense greedy as each of them considers only the locally optimal results of the other two.

Random Restarts

The described search algorithm assumes a given ordering of the goal values in the planning problem. We further extended the algorithm with random restarts (Algorithm 2) of the ordering of the goal values (we explore random permutations of the sequence of the goal values). The random restarts are helpful for tightening the upper bound for consecutive searches, which significantly improves pruning the search space. This is the same technique as used in Anytime Weighted A* introduced in (Hansen & Zhou, 2007).

Algorithm 2. Search procedures resource_search and RR (random restarts).

```
01  resource_search(s, bound)
02    AR ← actions that may resolve the resource conflict
03    my_best ← ∅
04    foreach a ∈ AR
05      bound ← min(eval(my_best),bound)
06      facts ← all changes and requests in ai
07      ai ← new action instance derived from a
08      s.Plan ← s.Plan ∪ {ai}
09      facts ← all changes and requests in ai
10      found ← support_search(s,facts,bound)
11      if found ≠ ∅ my_best ← found
12    return my_best
13
14  RR(s0, goals)
15    best ← ∅
16    while not stopped
17      s ← s0
18      next_perm ← permute_randomly(goals)
19      s ← root_search(s, next_perm, eval(best))
20      s ← postprocessing(s)
21      if s ≠ ∅ best ← s
```

A reason for using a simple depth-first search in way_search and support_search procedures (as opposed to A*) is based on the considerable memory requirements of the simple temporal network, which is a part of each search state and grows in $O(n^2)$, where n is the number of time points; the growth of the temporal network would directly impact the number of states that could be queued deeper in the search tree.

Further discussion of the algorithm can be found in (Dvořák, 2009).

Experimental Results

We implemented the Filuta system in Java and compared it with the best planners competing in the latest planning competition. In particular, to evaluate the efficiency of resource reasoning integration into planning we used three temporal planning domains with significant presence of resource reasoning: Openstacks, Elevators, and Transport from the deterministic temporal satisficing track of IPC 2008 (Helmert, Do, & Refanidis, 2008) and compared planning systems competing in this track, namely SGPlan6 (the winner), TFD (the runner-up), and Base-line planner proposed by the competition organizers.

Table 1. Makespan achieved by different planners for problems from the Elevators domain of IPC 2008; the last column shows runtime of Filuta system.

#	Base	SGPlan6	TFD	Filuta^{RR}	Filuta^{1}	Filuta^{1}(sec)
1	210	162	144	84	132	*0.031*
2	122	121	144	91	96	*0.001*
3	66	80	54	46	54	*0.016*
4	163	205	156	97	129	*0.047*
5	110	151	92	58	70	*0.031*
6	248	211	316	110	169	*0.062*
7	144	226	257	90	98	*0.156*
8	185	268	267	115	124	*0.047*
9	216	141	111	73	111	*0.094*
10	397	333	411	138	261	*0.297*
11	305	260	380	162	228	*0.125*
12	438	456	617	218	310	*0.361*
13	466	707	537	186	285	*0.578*
14	505	523	882	233	330	*0.751*
15	812	688		255	403	*1.375*
16	456	420		225	292	*1.453*
17	488	659	1074	290	414	*2.502*
18	788	751	1273	416	601	*3.532*
19	866	1425		539	906	*51.579*
20	628	841		342	410	*3.828*
21	629	757	674	184	236	*2.172*
22	400	570	419	244	280	*6.109*
23	477	796		279	397	*5.422*
24	475	939		209	345	*14.751*
25	776	1407		335	545	*21.907*
26	736	1043		387	464	*29.281*
27	868	1145		387	449	*47.109*
28	862	1607		433	471	*26.546*
29	877	1244		382	514	*73.625*
30	1237	1762		488	532	*78.485*

We used the same setting as during the competition, that is, each planner was given a 30-minutes time limit (we used 2.5 GHz Intel Dual-core CPU) and 2 GB memory per single problem. We run Filuta in two modes: Filuta^{1} uses a single-shot run so we present a runtime for this mode while Filuta^{RR} is using random restarts so it is running for all 30 minutes.

Table 1 compares the makespan achieved by different planners in the Elevators domain which is briefly described as a problem of planning movements of elevators for a set of passengers (the complete descriptions of the domains and the planning problem instances itself can be found in (Helmert, Do, & Refanidis, 2008)); it clearly demonstrates that Filuta generates plans of best quality.

Table 2. Makespan achieved by different planners for problems from the Transport domain of IPC 2008; the last column shows runtime of Filuta system.

#	Base	SGPlan6	TFD	Filuta^{RR}	Filuta^{1}	Filuta^{1}(sec)
1	52	52	52	52	52	*0.031*
2	217	217	241	126	173	*0.031*
3	243	432	669	189	295	*0.468*
4		845		256	405	*0.375*
5		359		242	335	*0.454*
6		965		256	423	*3.4693*
7				418	474	*18.828*
8				382	449	*127.66*
9				288	447	*18.406*
10				577	673	*150.73*
11	629	629	549	332	332	*0.001*
12	817	817	1009	490	490	*0.016*
13	1216	650	3383	386	420	*0.157*
14	2059			620	768	*5.016*
15		2249		807	973	*7.828*
16		1875		840	840	*1194.7*
17		3331		804	971	*43.828*
18				1194	1429	*207.34*
19				1341	1341	*1647.6*
20		6362				
21	113	113	161	69	69	*0.001*
22	238	238				
23	423	642				
24	1019	1116				
25	1404			201	201	*1.875*
26				234	241	*8.437*
27				244	364	*24.516*
28				308	348	*49.251*
29				307	380	*70.062*
30				362	394	*139.45*

Table 2 shows the results from the Transport domain which is briefly described as a problem of planning routes for a set of trucks that consume fuel and have limited capacity such that all packages of various sizes are delivered to their destinations. Filuta was able to solve 26 out of 30 problems with the smallest makespan among the competing systems; however it cannot solve instances 22-24, since they contain a "trap" for our subgoal-oriented approach (a truck gets stuck without enough fuel to reach its own goal destination). Also finding a plan for instance 20 took almost one hour.

The Openstacks domain differs significantly in the type of resources (single reservoir) and Filuta was able to solve

only 11 smaller problems out of 30, while for larger problems it exceeded the 30-minutes limit due to time consuming generation of resource resolvers. Nevertheless, for the solved problems Filuta found better plans than other planners. The Openstacks domain is a known NP-hard optimalization problem. While pure satisfaction planning for the domain is easy (runs in linear time), optimalization is the hard part. When compared with known near-optimal results for the first 11 instances, plans produced by Filuta are not worse by more than 15%.

The preliminary results from the other IPC temporal domains lacking resources show the dependency of Filuta on quality of the domain transition graphs; in other words, graphs with a few nodes and near-instant actions do not provide enough information to efficiently prune the search space (e.g. the smallest instances in Peg Solitaire domain take over 2 minutes of runtime). Additionally, since the root_search procedure does not backtrack over partial solutions, the false dead-ends may occur (this is also the case of the instances 22-24 in the Transport domain). Filuta does not yet implement cycle prevention in the root_search procedure, therefore solving problems that contain cycles of dependencies among state variables may lead to cycling of the planner (this wasn't the case of the three evaluated domains).

Conclusions and Future Work

The paper presents an integrated approach to solving planning problems with time and resource constraints. The proposed system Filuta exploits existing techniques for temporal reasoning, has a modular architecture to describe resource constraints, and uses domain transition diagrams to guide the search procedure. Experimental comparison showed that Filuta generates better plans for the temporal domains with significant resource constraints from the IPC2008 than the top competitors.

Most of the time during planning (about 96%) was spent by maintenance of the temporal network so novel incremental techniques for temporal reasoning may significantly improve runtime.

Resource reasoning in Filuta is still not fully exploiting the existing techniques from scheduling and for example the existing global constraints modelling resources may help there. The generation and aggregation of resolvers work efficiently in case of the Elevators and Transport domain; however the Openstacks domain contains only a single reservoir and imposes minimal number of constraints which leads to exponential growth of the number of resolvers.

From the planning side, subgoal-oriented approach brings problems with false dead-ends and cycling, although it turns out to be very efficient for the examined domains. While cycling can be prevented with some effort, the false dead-ends require addition of some advanced planning techniques and heuristics.

The current planning domains where Filuta can solve problems efficiently consist of problems with rich domain transition graphs and less dependencies among state variables, while resources (especially reservoirs) should not be undersubscribed; by itself, the number of resources in the problem does not have serious impact on the performance.

The efficiency comes mainly from the sub-goal oriented search algorithm, which exhibits interesting performance in domains with fewer dependencies among state variables, and from the constraint propagation into the simple temporal network from resource reasoning, pre-calculated critical paths and partial orderings in the temporal databases allowing early pruning and early detection of inconsistencies.

Acknowledgements

The research is supported by the Czech Science Foundation under the contract P103/10/1287.

References

Bäckström, C., & Nebel, B. (1995). Complexity results for SAS+ planning. *Computational Intelligence* , pp. 625-665.

Cesta, A., & Fratini, S. (2008). The Timeline Representation Framework as a Planning and Scheduling Software Development Environment. *The 27th Workshop of the UK PLANNING AND SCHEDULING Special Interest Group* .

Dechter, R. (2003). *Constraint Processing.* Elsevier, Morgan Kauffman Publishers.

Dvořák, F. (2009). *AI Planning with Time and Resource Constraints.* Master Thesis, Charles University in Prague, Faculty of Mathematics and Physics, Prague.

Frank, J., & Jonsson, A. (2001). *A Constraint-Based Planner with Attributes.*

Ghallab, M., Nau, D., & Traverso, P. (2004). *Automated Planning: Theory and Practice.* San Francisco: Morgan Kaufmann Publishers.

Hansen, E. A., & Zhou, R. (2007). Anytime heuristic search. *Journal of Artificial Intelligence Research* , pp. 267–297.

Helmert, M., Do, M., & Refanidis, I. (2008). Retrieved from International Planning Competition 2008 - Deterministic Part: http://ipc.informatik.uni-freiburg.de/

Jonsson, A. K., Morris, P. H., Muscettola, N., Rajan, K., & Smith, B. (2000). *Planning in interplanetary space: Theory and practice.*

Jonsson, P., & Bäckström, C. (1998). State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence* , pp. 100(1-2):125-176.

Laborie, P. (2001). Algorithm for propagating resource constraints in AI planning and scheduling: existing approaches and new results. *Proceedings of the European Conference on Planning* , pp. 205-216.

M. Ghallab, H. L. (1994). Representation and control in IxTeT, a temporal planner. *International Conference on AI Planning Systems*, (pp. 61-67).

Stergiou, K., & Koubarakis, M. (1998). Backtracking algorithms for disjunctions of temporal constraints. *15th National Conference on Artificial Intelligence*, (pp. 248-253).

Verfaillie, G., & Prelet, C. (2008). Using Contraint Network on Timelines to Model and Solve Planning and Scheduling Problems. *International Conference on Automated Planning and Scheduling 2008* , p. 272.

Younes, H. L., & Simmons, R. G. (2003). VHPOP: versatile heuristic partial order planner. *Journal of Artificial Intelligence Research* , pp. 405-430.