AI Planning with Time and Resource Constraints

Filip Dvořák, Roman Barták

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

Abstract. Planning deals with the problem of finding a partially ordered sequence of actions (plan) that transfers the world from some initial state to a desired state. Causal relations between actions play a critical role here. Introduction of explicit time and resources into planning is an important step towards modeling 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 with time and resources from the International Planning Competition 2008 and compare our results with the competition participants.

Keywords: automated planning, temporal constraints, resource reasoning.

1 Introduction

Planning is an abstract, explicit deliberation process that chooses and organizes actions by anticipating their expected outcomes. Real world planning problems usually differ from each other significantly; various approaches were taken dealing with path and motion planning, perception planning, navigation planning, manipulation planning, communication planning or different branches of social and economic planning. These approaches often rely on their own domain representations and problem specific techniques limiting their reusability and transferability to other branches of planning problems. Finding a common ground for representing and solving planning problems has always been a challenge as the representation of various real-world features and emphasis on different aspects of problem are required and domain-independent planning is generally a PSPACE-complete problem [1].

AI Planning [2] deals with finding a set of actions that transfer an initial state of the world into a goal state; such set is then called a *plan*. Actions in a plan may be required to be partially ordered, totally ordered or scheduled in time. Usually, the goal state is only partially defined and the number of possible plans is infinite, unless either a bound is a natural part of the planning problem, or we bound the problem arbitrarily. Abstraction is a natural part of planning problem formulation; we omit details of the world properties that are not relevant for the problem. Resources in planning are another form of abstraction, where the omitted detail is the uniqueness, e.g., the exact seat a passenger took in car may not be relevant, if we are only interested in the number of available seats. Historically, resources have been considered a domain of scheduling, in which they were extensively studied. While planning is concerned in finding a set of actions needed to achieve a goal, the scheduling problem consists of finding time and resource allocation for a given set of actions [3]. Solving many real world problems naturally requires both planning and scheduling; however separation of both processes may not always be efficient, e.g., the problem with many valid plans and a few valid schedules would require many iterations of the planning process. The problem of such sequential model is that planning itself is not informed enough about how a plan should be shaped and structured to satisfy the constraints later enforced in the scheduling process. Hence, it seems more appropriate to integrate planning and scheduling, for example by assuming the scheduling constraints such as limited time and resources during the planning process. This approach is advocated in this paper where we present a planning system called Filuta that handles time and resource constraints in addition to traditional causal relations from planning.

In this paper we focus on fully observable, deterministic temporal planning with resources [2]. It means that world states are completely known to the planner (fully observable), the effects of planning actions are unique (deterministic) and the actions have some duration, can overlap in time and require resources for execution. 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 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.

In the paper we will first describe the formal representation of the planning problem consisting of temporal databases modeling evolution of state variables and resource models. Then we will show how to solve the problem by integrating search decisions with maintaining consistency of temporal databases and resource models. Finally, we will experimentally compare Filuta with the state-of-the-art planners.

2 Representation

The cornerstone representation we build on is the *state-variable representation* for classical planning [4]. The State-variable is a variable whose domain contains facts about the world such that no two facts from the domain can be true at any given time. For example, the state variable describes a position of robot where possible locations determine the values of the state variable. A state of the world can then be defined as an *n*-tuple of values of *n* state-variables. Since we need to represent time explicitly, we represent the states of the world through a set of functions $\{sv_1,\ldots,sv_n\}$, where each function maps time to the domain of the state variable. For purposes of our planner we rely on the structure of time as modeled by the set of natural numbers $\mathbb{N}$. The functions capturing the evolution of state variables in time are piecewise-constant, hence to represent them we solely need to keep the changes of their value in time, which is the role of *temporal databases*.

Though *resources* can be modeled via state variables, we approach the modeling 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 category of resource the instance is representing (see below).

Actions in our representation are grounded temporal operators that contain changes of the state variables' value (effects), requests on the value of the state variable (preconditions) and resource events on the resource instances (preconditions and effects).

We define a *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 scheduled *action instances* (a plan) such that the last values of the state variables' temporal evolutions are the goal values (we do not consider intermediate goals), all temporal databases are 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. Note that action may appear as several action instances in the plan.

In the following subsection we will describe the Simple Temporal Problem and the Simple Temporal Network used for temporal reasoning. We will further define representations of temporal databases, resources and actions in our planning system.

2.1 Simple Temporal Problem

Simple temporal problem [5], shortly STP, is built upon constraint satisfaction problem formalism [6]. Formally, STP is a kind of CSP, where $X = \{x_1, \ldots, x_n\}$ is a set of CSP-variables, also known as time points, whose domains are $\mathbb{N}$ (in our case), and $C = \{c_1, \ldots, c_m\}$ is a set of unary and binary constraints, where each constraint c_i is represented by an interval $[a_i, b_i]$. A unary constraint c_j upon variable x_i restricts the domain of a variable to an interval; it represents the relation $a_j \leq x_i \leq b_j$. A binary constraint c_k upon (x_i, x_j) restricts the permissible values for the distance $x_j - x_i$; it represents the relation $a_k \leq x_j - x_i \leq b_k$. Although we have defined unary constraints, we can directly omit them, since they can be transformed into binary constraints by relating the concerned time points to some reference time point, e.g., "the beginning of the world". Having only binary constraints, we can depict a STP instance as a directed graph, whose nodes represent the time points and arcs between nodes are annotated by the corresponding intervals. Such graph is called a Simple temporal network (STN). We say that the STN instance is consistent if and only if there exists such instantiation of the time points that all the binary constraints are satisfied.

We further define two intuitive operations upon pairs of constraints: composition $c_{ij} \cdot c_{jk} = [a_{ij} + a_{jk}, b_{ij} + b_{jk}]$, and intersection $c_{ij} \cap c'_{ij} = [\max\{a_{ij}, a'_{ij}\}, \min\{b_{ij}, b'_{ij}\}]$.

Together we can define a transitive closure operation as $c_{ij} \leftarrow c_{ij} \cap (c_{ik} \cdot c_{kj})$. The propagation of transitive closure upon consistent STN tightens constraints to its *minimal form* [5]; such network is called minimal. Having a minimal network (X,C) we define an update operation $up(x_i,x_j,c)$ as $c_{ij} \leftarrow c \cap c_{ij}$, where x_i, $x_j \in X$, c is a new constraint of form $a \leq x_j - x_i \leq b$, and we say that the operation is consistent iff $c \cap c_{ij} \neq \emptyset$; intuitively, for $up(x_i,x_j,c)$, where c = [a,b], the application of $up(x_i,x_j,c)$ operation upon a STN says that x_i happens at least a and at most b time units before x_j. Although computing the minimal network takes generally $O(n^3)$ operations, where n is the number of time points, we can minimize the network, whose minimality was invalidated by a single consistent update, in $O(n^2)$ using IFPC algorithm [7].

While STN allows us to represent quantitative temporal relations, we need to represent qualitative relations as well. We say that x_i *happens possibly before or at the same time as* x_j iff $up(x_i,x_j,[0,\infty])$ is consistent; we denote it as $PBE(x_i,x_j)$. We say that x_i *happens necessarily before or at the same time as* x_j iff $up(x_j,x_i,[1,\infty])$ is inconsistent; we denote it as $NBE(x_i,x_j)$.

The upside of maintaining a minimal network is mainly in the constant time detection of inconsistent updates, possibility to solve certain 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 $n^2/2 - n$. Further we can omit any time points that become 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.

2.2 Temporal Databases

The purpose of temporal database is to store information on how a state variable evolves in time. Since the time evolution of a state variable is a piecewise constant function, we can express and store the time evolution of a state variable as a set of *changes* of the state variable's value. Additionally we need to represent *requests* on a state variable to keep certain value for a period of time. Using qualitative temporal relations we have defined for a simple temporal network, we define changes and requests for a state variable with domain D and a minimal temporal network (X,C) as follows: *change* is a quadruple $(x_s,x_e,v_{initial},v_{final})$, where x_s, $x_e \in X$, $NBE(x_s,x_e)$,$v_{initial}$, $v_{final} \in D$, and *request* is a triple (x_s,x_e,v), where x_s, $x_e \in X$, $NBE(x_s,x_e)$, $v \in D$.

For a minimal temporal network (X,C) we define the temporal database TDB for a state variable to be a sequence $(ch_1,R_1,...,ch_n,R_n)$, where ch_i is a change, $R_i = \{(x_{s1}, x_{e1}, v_1), ..., (x_{sm}, x_{em}, v_m)\}$ is a set of requests, and all time points contained in an element of the sequence happen necessarily before or at the same time as all the time points contained in any consecutive element. We say that a temporal database TDB is consistent if and only if $\forall ch_i$, $R_i \in TDB$ $\forall (x_{sj},x_{ej},v_j) \in R_i$: $v_{final-i} = v_j = v_{initial-i+1}$. In other words, consistency tells us that any two consecutive changes must share the inner value and any request between those two changes must share the value of the state variable as well.

Other principles and concepts of temporal databases can be found in [2]. Our approach differs by distinguishing *temporal expressions* to changes and requests, keeping them as a sequence instead of a more general set, and using a temporal database for each state-variable instead of managing binding constraints, which are, in our case, strong decisions of the search algorithm (section 3).

2.3 Resources

The concept of resources is similar to temporal databases in sense that we only need to capture discrete changes in the evolution of the resource level. However, contrary to temporal database, the changes of the resource level are frequently coupled (borrowing a resource means consumption at the beginning of action and production at the end of action). Since the behavior of a resource varies depending on the represented properties of the world, we distinguish resources into categories and define compact representations for each category; in our planning system we have so far defined category for a single-capacity reusable resource (also known as a unary resource), a multi-capacity reusable resource (also known as a discrete resource), and a multi-capacity replenishable resource (also known as a reservoir).

- **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 $rq \in \mathbb{N}$ 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 $e \in \mathbb{Z}$ is a relative change of the resource level; $e < 0$ represents consumption and $e > 0$ represents production. Each instance has fixed capacity.

2.4 Actions

Based on definitions of temporal network, temporal database and resource instance, we define an action. An action is a sextuple $A = (tp_s, tp_e, dur, CHs, RQs, REs)$, where

- tp_s and tp_e are time point parameters; upon the introduction of the action into a plan we associate them with the time points from the temporal network.
- $dur \in N$ is a duration of the action,
- CHs is a set of changes of the state variables' value,
- RQs is a set of requests on the state variables,
- REs is a set of resource events upon the resource instance.

Once the action becomes instantiated with the time points, we call it an *action instance.* The instantiation of action by a pair of time points propagates these time points into the changes, requests and events contained in the action. The changes and requests can occur at the beginning, at the end, or over the duration of the action. The propagation into events depends on the resource instance the event belongs to, e.g., an event for a unary resource instance occurs over the duration of the action. To demonstrate the definition, 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.

Our concept of action takes the middle ground in context of similar approaches to temporal planning. In [2] authors define a more general concept of temporal operator, where the parameters include the planning atoms, e.g. there would exist only one action *load* parameterized by trucks, packages and locations. In [8] authors take opposite direction, removing the temporal parameters of the action and efficiently modeling temporal planning problems as a CSP, which in other hand implies that an action cannot occur multiple times and the approach is not suitable for problems with large numbers of grounded actions.

3 Solving Approach

The cornerstone of our planning system is the simple temporal network, whose time points are used for temporal annotation of changes and requests in the temporal databases and temporal annotations of the events in the resource instances. The resource reasoning upon the resource instances is realized by a *resource manager*, which keeps a least-commitment approach (not deciding unless necessary) 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* [9].

The domain transition graph (DTG) for a state-variable with a domain D and a set of actions S is a directed multigraph (V,E), where V = D and an action from S represents an arc $(v_i,v_j) \in E$ if and only if it contains a change of the concerned 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 represented 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 help to guide the search algorithm. We calculate two types of shortest paths. One type measures the length of the path in a graph as the minimal time needed to traverse this path (a sum of durations of actions traversed); we denote these paths as T-P. The second type measures the minimal number of arcs traversed, while less time demanding paths are preferred; we denote these paths as OT-P.

Generated DTGs also allow us to simplify the planning problem through detection of additional unary resources. If we find such DTG (V,E) that all arcs in E are loops upon one node, we remove such DTG, the corresponding temporal database and the state-variable, and we create a new instance of the unary resource and substitute the changes for resource events in the actions that represented the looping arcs.

In the following subsections we describe the resource manager and the search algorithm in more details.

3.1 Resource Manager

Our representation of resources distinguishes resources into several categories. For each category we build 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*. Since we would like to keep all the options of preventing the conflict (because strong early decision can cut us from good plans), we build the solver to encode all the options as an output. 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 one update, such that the set of chosen updates is consistent with the temporal network. Trivial cases occur when SR = $\emptyset$, which indicates that no conflicts need to be resolved, and $\emptyset \in$ SR, indicating that some resource conflict cannot be resolved (which may imply a backtrack point for the search algorithm).

We shall demonstrate the above-mentioned principle using an example for an instance of unary resource. Assuming we have an STN, an instance {(a,b), (c,d)} and a new event (e,f), where a-f are time points, the solver for the unary resource produces a set $\{\{f < a,\ b < e\},\ \{f < c,\ d < e\}\}$, where $x < y$ represents an update operation up(x,y,[0,∞]) (a resolver that enforces NBE(x,y) upon the STN). The intuitive meaning of the produced set and the semantic of the solver is that since only one event (an action) may occupy the unary resource instance (a machine) at any time, the new event must necessarily happen before or after any other event. By choosing an update from each set of updates we temporally separate the new event from all other events (separations between the other events is an assumption of the incremental solver). Due to space limitations we do not describe solvers for other resource types; the reader may find their concepts in [2], [10] and [11].

Planning problems generally contain multiple resource instances and the solvers together often produce multiple (non-trivial) sets $SR_1, \ldots, SR_n$. 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 including outputs of solvers into SR, 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 updates in SR). The efficiency can also be improved by remembering the last found solution, relaxing the frequency of solution checking and incorporating CSP techniques (the described problem can be seen as a meta-CSP).

3.2 Search algorithm

In the Filuta system we adapted the plan-space planning approach [2] 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 = $\emptyset$, Plan = $\emptyset$ and STN is the *initial temporal network*.

The initial temporal network is constructed from 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 up($x_{g\text{-start}}$, $x_{g\text{-end}}$, $[0,\infty]$); the time points represent global start and end of the world. Any further time point x inserted into the network is implicitly constrained by up($x_{g\text{-start}}$, x, $[0,\infty]$). We further insert a time point $x_{i\text{-end}}$ for each $TDB_i \in$ TDBs and update the network with up($x_{i\text{-end}}$, $x_{g\text{-end}}$, $[0,\infty]$) for each such time point; these time points represent local ends of the world upon the evolution of the corresponding temporal databases. 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 up(x_e, $x_{i\text{-end}}$, $[0,\infty]$).

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 makespan of the plan is then defined as the value of the time point that represents the end-time of the latest action in the plan (the same value as in the time point representing “the global end of the world”).

The states of the search 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 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 path from the initial state to some solution state, or more precisely finding a solution state that is reachable from the initial state.

For a problem (TDBs, RIs, Actions, Goals) we denote the set of all possible search states as S. For a state $s \in S$ 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 goal values). We define the state evaluation function *eval*: $S \rightarrow \mathbb{N} \times \mathbb{N}$ as *eval*(s) = (*ms*(s), *ft*(s)) and the goal of planner is to find a reachable solution state with the minimal value of the *eval* function (using ordering $(a,b) < (c,d) \leftrightarrow a < c$ or $(a = c$ and $b < d)$).

The search algorithm divides into 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 (random restarts), or it can be unknown (represented as (∞,∞)). The output of the procedures is a state, where a state = $\emptyset$ 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.

For a problem (TDBs, RIs, Actions, Goals), the *root_search* 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. 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 the *way_search* may invalidate a previously achieved goal). This is similar to STRIPS algorithm for classical planning [2].

The *way_search* procedure takes as an additional input a problem of finding a way in a domain transition graph, which consists of an anchoring change from the current state and a *fact* that is either a change or a request (for a goal request the anchoring change is the last change in the corresponding TDB, otherwise the choice of the anchoring change is a decision made in the *support_search*). The *way_search* initially imposes new constraints into the current STN, improving the lower bound; 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 pre-calculated for the DTG). 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 involves a call of the *support_search*, 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). Traversing an arc also includes insertion of the action's resource events into the resource instance (invoking a resource manager), while the changes and requests from the action are passed to *support_search*. 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).

The additional input of the *support_search* is a set of facts (changes and requests that contain the time points propagated from the action instance). The task upon the *support_search* is to find an anchor change for each fact 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 anchor changes to the facts. The search is guided by the *fewest-options-first* principle, which is particularly efficient, since by assigning a anchor change to a fact, we may also significantly reduce the number of possible assignments of anchor changes to other facts; this is caused by the updates of the STN that preserve the consistency of the TDB and shared time points among the facts (we can imagine the principle as narrowing a temporal window that represents the possible temporal positions of an action instance).

The *resource_search* procedure is called whenever the set SR in the current state becomes inconsistent (we are not able to prevent all the resource conflicts); 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 overconsumed. For each choice of the action the contained facts are passed to the *support_search*.

The described search algorithm assumes a given ordering of the goal values in the planning problem (we achieve the goal values in a sequence). We further extend the algorithm with random restarts of the ordering of the goal values (we explore random permutations of the sequence of the goal values). The random restarts are also helpful for tightening the upper bound for consecutive searches, which significantly improves pruning the search space. We formulate the algorithm extended with the random restarts as an anytime algorithm. Due to space limitations we do not include the pseudo code; the reader may find it in [11].

4 Experimental Results

To evaluate the above described planning system, we implemented Filuta in Java and compared it with the best planners competing in the latest planning competition. In particular, we used three planning domains: Openstacks, Elevators, and Transport from the deterministic temporal satisfaction track of International Planning Competition 2008 [12] 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. Due to space reasons we present here only the results for 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 [12]). 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 and it clearly demonstrates that Filuta generates plans of best quality. Similar results were achieved for the Transport domain where Filuta was able to solve 25 out of 30 problems (the largest number among the competing systems). All plans generated by Filuta^{RR} for the solved problems have the smallest makespan among the competing system. The Openstacks domain differs significantly in the type of resources (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 (the Openstacks domain forms an NP-complete optimalization problem, which together with the least-commitment of the resource reasoning causes exponential grow of the runtime). Nevertheless, for the solved problems Filuta found plans better than other planners.

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

Problem	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*

The experimental evaluation was mainly focused on temporal planning problems with significant presence of resource reasoning, which determined the three chosen domains that also represented the initial motivation examples for the development of the planning system. The preliminary results from the other IPC temporal domains lacking resources show the dependency of Filuta on quality of the domain transition graphs, e.g. graphs with a few nodes and near-instant actions do not provide enough information to efficiently prune the search space.

5 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 (other types of resources can be easily added), and uses domain transition diagrams to guide the search procedure. Experimental comparison showed that Filuta generates better plans than existing state-of-the-art planners. The experiments also showed the bottlenecks of such integrated systems. Most of the time during planning was spent by maintenance of the temporal network so novel incremental techniques for temporal reasoning may significantly improve runtime. Also resource reasoning in Filuta is still not fully exploiting the existing techniques from scheduling and for example the existing global constraints modeling resources may help there. Similarly, advanced planning techniques such as finding landmarks or using more advanced heuristics may be added.

Acknowledgments. The research is supported by the Czech Science Foundation under the contract 201/07/0205.

References

1. Ero, K., Nau, D., Subrahmanian, V.: Complexity, decidability and undecidability results for domain-independent planning. Artificial Intelligence 76, 65-88 (1995)
2. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers, San Francisco (2004)
3. Baptiste, P., Pape, C., Nuijten, W.: Constraint-based Scheduling: Applying Constraint Programming to Scheduling Problems Second Printing edn. Kluwer Academic Publishers (2001)
4. Jonsson, P., Bäckström, C.: State-variable planning under structural restrictions: Algorithms and complexity. Artificial Intelligence, 100(1-2):125-176 (1998)
5. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artificial Intelligence, 49:91-95 (1991)
6. Dechter, R.: Constraint Processing. Elsevier, Morgan Kauffman Publishers (2003)
7. Planken, L. R.: New Algorithms for the Simple Temporal Problem. Master thesis, Faculty EEMCS, Delft University of Technology, Delft, the Netherlands (2008)
8. Vidal, V., Geffner, H.: Branching and pruning: An optimal Temporal POCL Planner based on Contraint Programming. Artificial Intelligence, 298-335 (2006)
9. Bäckström, C., Nebel, B.: Complexity results for SAS+ planning. Computational Intelligence, 625-665 (1995)
10. Laborie, P.: Algorithm for propagating resource constraints in AI planning and scheduling: existing approaches and new results. Proceedings of the European Conference on Planning, 205-216 (2001)
11. Dvořák, F.: AI Planning with Time and Resource Constraints. Master Thesis, Charles University in Prague, Faculty of Mathematics and Physics, Prague (2009)
12. Helmert, M., Do, M., Refanidis, I. In: International Planning Competition 2008 - Deterministic Part. (Accessed 2008) Available at: http://ipc.informatik.uni-freiburg.de/