Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2014-027 October 24, 2014

tBurton: A Divide and Conquer Temporal Planner

David Wang and Brian C. Williams

CSAIL

tBurton: A Divide and Conquer Temporal Planner

David Wang and **Brian Williams**
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract

Planning for and controlling a network of interacting devices requires a planner that accounts for the automatic timed transitions of devices while meeting deadlines and achieving durative goals. For example, a planner for an imaging satellite with a camera intolerant of exhaust would need to determine that opening a valve causes a chain reaction that ignites the engine, and thus needs to shield its camera. While planners exist that support deadlines and durative goals, currently, no planners can handle automatic timed transitions. We present tBurton, a temporal planner that supports these features while additionally producing a temporally least-commitment plan. tBurton uses a divide and conquer approach: dividing the problem using causal-graph decomposition and conquering each factor with heuristic forward search. The 'sub-plans' from each factor are unified in a conflict directed search, guided by the causal graph structure. We describe why tBurton is fast and efficient and present its efficacy on benchmarks from the International Planning Competition.

Introduction

Embedded machines are being composed into ever more complex networked systems, including earth observing systems and transportation networks. The complexity of these systems require automated coordination, but controlling these systems pose unique challenges: timed transitions – after turning a projector off a cool-down period must be obeyed before the projector will turn on again; periodic transitions – a satellite's orbit provides predictable time-windows over when it will be able to observe a phenomenon; required concurrency (Cushing et al. 2007), – a communication channel must be open for the duration of a transmission. Furthermore, a user may wish to specify when different states need to be achieved (time-evolved goals) and expect a plan that allows flexibility in execution time (a temporally least-commitment plan).

While there has been a long history of planners developed for these systems, no single planner supports this set of features. Model-based planners, such as Burton (Williams and Nayak 1997), the namesake of our planner, have exploited the causal structure of the problem in order to be fast and generative, but lack the ability to reason over time. Timeline based planners such as EUROPA (Frank and Jónsson 2003) and ASPEN (Chien et al. 2000) can express rich notions of metric time and resources, but have traditionally depended on domain-specific heuristics to efficiently guide backtracking search. Finally, metric-time heuristic planners (Coles et al. 2010; Benton, Coles, and Coles 2012; Röger, Eyerich, and Mattmüller 2008) have been developed that are domain-independent, fast and scalable, but lack support for important problem features such as timed and periodic transitions.

tBurton is a fast and efficient partial-order, temporal planner designed for networked devices. Our overall approach is divide and conquer, a.k.a factored planning (Amir and Engelhardt 2003), but we leverage insights from model, timeline, and heuristic-based planning. Like Burton, tBurton factors the planning problem into an acyclic causal-graph and uses this structure to impose a search ordering from child to parent. Associated with each factor is a timeline on which the plan will be constructed. Timelines help maintain locality of the causal information, thereby reducing the need for time-consuming threat-detection steps common in partial-order planning. To find a plan, we use a conflict directed search that leverages the efficiency of a heuristic-based sub-planner to completely populate the timeline of a factor, before regressing the sub-goals of that plan to the timeline of its parent.

The contributions of this paper are three fold: First, we introduce a planner for networked devices that support a set of features never before found in one planner. Second, we introduce a new approach to factored planned based on timeline-based regression and heuristic forward search. Third, we demonstrate the effectiveness of this approach on planning benchmarks.

We start by elaborating more on tBurton's approach in the context of prior work and introduce a running example. We then define our notation, before presenting the algorithms underlying tBurton. Finally, we close with an empirical validation on IPC benchmarks.

Background

Divide and conquer is the basic principal behind factored planning, but is only part of the story. A factored planner must decide how to factor (divide) the problem, how to plan for (conquer) each factor, *and* how to unify those plans.

Divide Key to tBurton's approach to factored planning is exploiting the benefits of causal-graph based factoring in partial-order, temporal planning.

tBurton inherits its causal reasoning strategy from namesake, Burton (Williams and Nayak 1997), a reactive, model-based planner demonstrated on the Deep Space-One spacecraft. Burton, exploited the near-DAG structure of its domain and grouped cyclicaly-dependent factors to maintain an acyclic causal graph. The causal graph could then be used to quickly determine a serialization of subgoals. Even though this strategy is not optimal in the general case, complexity analysis has shown it is difficult to do better in the domain independent case (Brafman and Domshlak 2006; 2003).

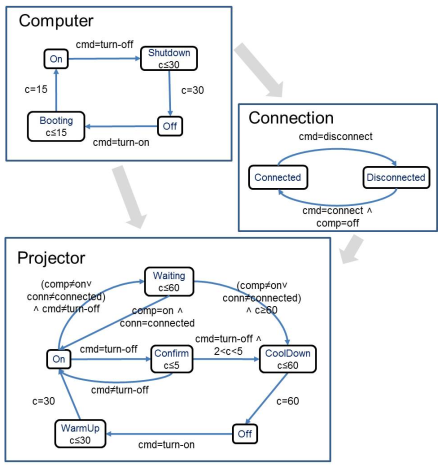
Despite the lack of optimality, the clustering of variables identified by an acyclic causal-graph has important ramifications. Goal-regression, partial-order planners (Penberthy and Weld 1992; Younes and Simmons 2003) traditionally suffer from computationally expensive threat detection and resolution, where interfering actions in a plan must be identified and ordered. Factoring inherently identifies the complicating shared variables, reducing the number of cases that must be checked for interference. Furthermore, threat resolution is equivalent to adding temporal constraints to order sub-plans during composition – reducing the number of threats under consideration also reduces the difficulty of temporal reasoning.

Conquer In order to plan for each factor, tBurton uses a heuristic-based temporal planner. Heuristic-based planners, and esp. heuristic forward search planners have scaled well (Röger, Eyerich, and Mattmüller 2008; Coles et al. 2010; Vidal 2011), and consistently won top places in the International Planning Competition. Using a heuristic forward search planner (henceforth sub-planner) allows tBurton to not only benefit from the state-of-the-art in planning, but degrade gracefully. Even if a problem has a fully connected causal-graph, and therefore admits only one factor, the planning time will be that of the sub-planner plus some of tBurton's processing overhead.

tBurton plans for each factor by first collecting and ordering all the goals for that factor along its timeline. The sub-planner is then used to populate the timeline by first planning from the initial state to the first goal, from that goal to the next, and so on. The problem presented to the sub-planner only contains variables relevant to that factor.

Unify Sub-plans are unified by regressing the subgoals required by a factor's plan, to parent factors. Early work in factored planning obviated the need for unification by planning bottom-up in the causal-graph. Plans were generated for each factor by recursively composing the plans of its children, treating them as macro-actions (Amir and Engelhardt 2003; Brafman and Domshlak 2006). This approach obviated the need for unifying plans at the cost of storing all plans. Subsequent work sought to reduce this memory overhead by using backtracking search through a hierachical clustering of factors called a dtree (Kelareva et al. 2007). While tBurton does not use a dtree, we do extend backtracking search with plan caching and conflict learning in order to more efficiently unify plans.

Projector Example A running example we will use in the remainder of this paper involves a computer, projector, and



- The guard always includes the clock constraint 0≤c<∞ unless otherwise noted.
- The clock variable is always reset, c:=0

Figure 1: A simple computer-projector system represented as Timed Concurrent Automata.

connection between them, which are needed to give a presentation. The computer exhibits boot-up and shutdown times. The projector exhibits similar warm-up and cool-down periods, but will also shutdown automatically when disconnected from the computer. Figure 1 depicts this example in tBurton's native, automata formulation.

Problem Formulation

The representation tBurton uses for the planning problem inherits from prior work in extending Concurrent Constraint Automata (CCA) based reasoning to time (Ingham 2003). It can be viewed as a variation on timed-automata theory (Alur and Dill 1994), where transitions are guarded by expressions in propositional state-logic rather than symbols from an alphabet.

Formally, the planning problem tBurton solves is the tuple, $\langle TCA, SP_{part} \rangle$, where TCA is a model of the system expressed as a set of interacting automata called *Timed Concurrent Automata*, and SP_{part} is our goal and initial state representation which captures the timing and duration of desired states as a *Partial State Plan*. We represent our plan as a Total State Plan SP_{total}, which is an elaboration of SP_{part} that contains no open goals, and expresses not only the control commands needed, but the resulting state evolution of the system under control.

Timed Concurrent Automata

Definition 1. a TCA, is a tuple $\langle L, \mathcal{C}, \mathcal{U}, \mathcal{A} \rangle$, where:

- $\mathcal{L}$ is a set of variables, $l \in \mathcal{L}$, with finite domain $D(l)$, representing locations within5 the automata. An assignment to a variable is the pair (l, v), $v \in D(l)$.
- $\mathcal{C}$ is a set of positive, real-valued clock variables. Each clock variable, $c \in \mathcal{C}$, represents a resettable counter that increments in real-time, at the same rate as all other clock variables. We allow the comparison of clock variables to real valued constants, $c\ op\ r$, where $op \in \{\leq, <, =, >, \geq\}$, $r \in \mathbb{R}$, and assignment of real-valued constants to clock variables $c := r$, but do not allow clock arithmetic.

- $\mathcal{U}$ is a set of control variables, $u \in \mathcal{U}$, with finite domain $D(u)$, representing commands with which users outside the TCA can control them.
- $\mathcal{A}$ is a set of timed-automata, $A \in \mathcal{A}$,.

A TCA can be thought of as a system, where the location and clock variables maintain internal state, while the control variables provide external inputs. The automata that compose the TCA are the devices of our system.

Definition 2. A single automaton, A is the 5-tuple $\langle l, c, u, \mathbb{T}, \mathbb{I} \rangle$.

- $l \in \mathcal{L}$ is a location variable, whose values represent the locations over which this automata transitions.
- $c \in \mathcal{C}$ is the unique clock variable for this automaton.
- $u \in \mathcal{U}$ is the unique control variable for this automaton.
- $\mathbb{T}$ is a transition function, that associates with a start and end location l_s, l_e a guard g and a reset value for clock $c := 0$. A guard is expressed in terms of propositional formulas with equality, φ, where: $\varphi ::= \texttt{true} \mid \texttt{false} \mid (l_o = v) \mid (u = v) \mid (c\ op\ r) \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$. The guard can be expressed only in terms of location variables not belonging to this automaton, $l_o \in \mathcal{L} \backslash l$, and the control and clock variable of this automaton. For brevity, we will sometimes use the expression $l \neq v$ in place of $\neg(l = v)$. The automaton is said to instantaneously transition from l_s to l_e and reset its clock variable when the guard is first evaluated true. More conveniently, we also define the related next-state function, $t(g, l_s) \rightarrow l_e$.
- $\mathbb{I}$ is a function that associates with each location an invariant, a clock comparison of the form $c < r$ or $c \leq r$ that bounds the maximum amount of time an automata can stay in that location.

In the projector example (figure 1), the projector has locations `Off`, `WarmUp`, `On`, `Waiting`, `Confirm`, and `CoolDown`. The transitions, represented by directed edges between the locations, are labeled by guards that are a function of other location variables, clock variables (denoted by `c`), and control variables (denoted by `cmd`). Invariants label some states, such as `WarmUp`, which allows us to encode the bounded-time requirement that the projector must transition from `WarmUp` to `On` in 30.

An automaton is *well formed* if there exists a unique next-state for all possible combination of assignments to location, control, and clock variables. With regards to the transitions, an automaton is said to be *deterministic* if for any l_s only the guard g of one transition can be true at any time. Figure 2 depicts the set of next-states from the 'Waiting' state of the well-formed, deterministic Projector automaton. Note that there is exactly one possible next-state for each combination of inputs, and a unique next-state at the upper-bound of the invariant. In order to produce plans with predictable behavior, tBurton must plan over TCA models consisting of well-formed, deterministic automata.

State Plans

To control TCA, we need to specify and reason about executions (runs) of the system in terms of state trajectories; we represent these as state plans. A state plan specifies both the desired (goals) and required (plan) evolution of the location and command variables through a set of timelines we refer to as a state *histories*. The temporal aspect of these histories and the relationship between events within them are expressed through simple temporal constraints (Dechter 2003). The flexibility afforded by simple temporal constraints allows the state plan to be a temporally least commitment specification, which is important in enabling plan executives (Levine 2012; Muscettola, Morris, and Tsamardinos 1998) to adapt to a range of disturbances without the need to replan.

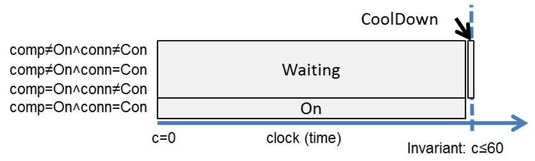


Figure 2: A graphical representation of the possible set of next-states from the Waiting state of the Projector automaton. a well-formed and deterministic automaton.

Definition 3. a History H is a tuple $\langle EV, EP \rangle$ where:
$EV = \{e_1, e_2, ..., e_n\}$, is a set of events e_i (that represent positive, real-valued time points), and
$EP = \{ep_1, ep_2, ..., ep_n\}$. is a set of episodes, where each episode, $ep = \langle e_i, e_j, lb, ub, sc \rangle$, associates with a temporal constraint $lb \leq e_j - e_i \leq ub$, a state constraint sc expressed as a propositional formula over location and control variables.

Definition 4. a State Plan SP is a tuple $\langle GH, \mathcal{VH}, \mathcal{J} \rangle$ where:

- GH is a *goal history*, a type of history in which episodes represent the *desired* evolution of location and control variables.
- $\mathcal{VH}$ is a set of value histories, $VH \in \mathcal{VH}$. Each *value history*, VH, is a type of history that represents the planned evolution of locations or control variables. The state constraints in a value history are restricted to contain only variable assignments and only assignments to the variables to which the value history is associated. As such, a value history represents the trace of the corresponding variables.
- $\mathcal{J}$ is a justification, a type of episode with a state constraint of value `true`, which is used to relate the temporal occurrence of events in goal histories to events in value histories.

As with many partial-order planners, tBurton searches over a space of partial plans, starting from the partial state-plan SP_{part} and elaborating the plan until a valid total plan SP_{total}, in which all goals are closed, is reached. State plans allow tBurton to not only keep track of the plan (through value histories), but also keep track of why the plan was created (goal histories). This is useful for post-planning validation, but also allows tBurton to keep track of open-goals during the planning process.

Semantics of TCAs and State Plans

Until now, we have used the words *run*, *trace*, *closed*, *valid*, and *least-commitment* in a general sense. We now return to formalize these definitions, starting with the semantics of $TCAs$ and their relation to state plans.

The *run* of a single timed-automaton, A_a, can be described by its timed state trajectory, $S_a = ((l_{a0}, 0), (l_{a1}, t_{a1})..., (l_{am}, t_{am}))$, a sequence of pairs describing the location and time at which the automaton first entered each state. l_{a0} is the initial assignment to the location variable of A_a. We say a run $((l_{ai}, t_{ai}), (l_{aj}, t_{aj}))$ is *legal* if two conditions are met. First, if l_{ai} has invariant guard $c < r$, then $t_{aj} - t_{ai} < r$ must be true. Second, if there exists a transition between locations l_{ai} to l_{aj}, guarded by g, the first time g became true in the interval from t_{ai} to t_{aj} must be at time t_{aj}. A *trace* in timed automata theory is usually defined as the timed-trajectory of symbols, a word. Relative to TCAs, this corresponds to the timed trajectory of the guard variables. For $TCAs$, where automata asynchronously guard each other through their location variables, a trace and run are almost identical, with the exception that a trace for a TCA would also include the timed state trajectory of control variables used in guards. A trace for a TCA therefore captures the evolution of all the variables.

Definition 5. closed. A goal-episode with constraint sc_g is considered trivially closed, if sc_g evaluates to true, and trivially un-closable, if it evaluates to false. Otherwise, a goal-episode is considered *closed* if there is a co-occurring episode in a value-history whose constraint entails the goal's constraint. Formally, a goal-episode $\langle e_{gs}, e_{ge}, lb_g, ub_g, sc_g\rangle$ is closed by a value-episode $\langle e_{vs}, e_{ve}, lb_v, ub_v, sc_v\rangle$, if $sc_v \models sc_g$, and the events are constrained such that $e_{vs} = e_{gs}$ and $e_{ve} = e_{ge}$. A goal appearing in the goal-history which is not closed is *open*.

Definition 6. valid. In general, SP_{total} is a *valid* plan for the problem $\langle TCA, SP_{part}\rangle$, if SP_{total} has no *open* goal-episodes, *closes* all the goal-episodes in SP_{part}, and has value-histories that both contain the value history of SP_{part} and is a legal trace of the TCA.

For tBurton, we help ensure SP_{total} is a valid plan by introducing two additional requirements. First, we require that SP_{part} contains an episode in the value history of each location variable, l, whose end event must precede the start event of any goal on l, thus providing a complete 'initial state'. Second we require that SP_{part} be a subgraph of SP_{total}. These two additional requirements allow us to simplify the definition of valid: SP_{total} is a valid plan if it has no open goals and is a trace of the TCA.

To ensure SP_{total} is temporally, *least commitment*, the state plan must be consistent, complete, and minimal with respect to the planning problem. A valid state plan is already *consistent* and *complete* in that it expresses legal behavior for the model and closes all goals. We consider a state plan to be *minimal* if relaxing any of the episodes (decreasing the lower-bound, or increasing upper-bound) in the histories admit traces of the TCA that are also legal. Note, that by the definition of the state plan, tBurton cannot return plans containing either disjunctive temporal or disjunctive state constraints.

tBurton Planner

The tBurton planner consists of several algorithms, but the fundamental approach is to plan in a factored space by performing regression over histories. Practically, this search involves: 1. Deciding which factor to plan for first. This scopes the remaining decisions by selecting the value history we must populate and the goal-episodes we need to close. 2. Choosing how to order the goal-episodes that factor can close 3. Choosing a set of value-episodes that should be used to close those goal-episodes. 4. Choosing the plans that should be used to achieve the value-episodes. 5. And finally, extracting the subgoals of the value history (guards) required to make the value-history a legal trace of the automata and adding corresponding goal-episodes to the goal history. These steps repeat until a plan is found, or no choices are available.

As a partial order planner, tBurton searches over variations of the state plan. Since we use the causal graph to define a search order, and subgoal extraction requires no search, tBurton only has three different choices with which to modify SP_{part}.:

1. **Select a goal ordering.** Since actions are not reversible and reachability checking is hard, the order in which goals are achieved matters. tBurton must impose a total ordering on the goals involving the location of a single automaton. Recall that since an automaton can have no concurrent transitions, a total order does not restrict the space of possible plans for any automaton. Relative to SP_{part}, imposing a total order involves adding episodes to the goal history of the form $ep = \langle e_i, e_j, 0, \infty, true\rangle$, for events e_i and e_j that must be ordered.
2. **Select a value to close a goal.** Since goals can have constraints expressed as propositional state-logic, it is possible we may need to achieve disjunctive subgoals. In this case, tBurton must select a value that entails the goal.

   To properly close the goal (definition 5), tBurton must also represent this value selection as an episode added to the value history of the appropriate automata or control variable, and introduce justifications to ensure the .
3. **Select a sub-plan to achieve a value.** The sub-plan tBurton must select need only consider the transitions in a single automaton, A. Therefore, the sub-plan must be selected based on two sequential episodes, ep_s ep_g, in the value history of A (which will be the initial state and goal for the sub-plan), and the set-bound temporal constraint that separates them. The method tBurton uses to select this sub-plan can be any blackbox, but we will use a heuristic forward search, temporal planner. To properly add this sub-plan to SP_{part}, tBurton must add the plan to the value history and introduce any new goals this plan requires of parent automata.

Adding the plan to the value history is a straightforward process of representing each action in the plan as a sequential chain of episodes $ep_1, ep_2, ..., ep_n$ reaching from the end event of ep_s to the start event of ep_g. Introducing goals is a bit trickier. We must introduce subgoals for all episodes in the value history corresponding to the sub-plan, ep_i, as well as ep_g (for which we did not introduce goals as a result of the previous type of plan-space action, selecting a value to close a goal). The purpose of these goals is to ensure the parent automata and control variables have traces consistent with the trace (value-history) of A. There are two types of goals we can introduce: One is a maintenance related goal, that ensures A is not forced by any variables it depends on to transition early out of ep_i; The other expresses the goal required to effect a transition.

To formalize the modifications to SP_{part}, define $ep1 = \langle e_{s1}, e_{e1}, lb_1, ub_1, sc_1 \rangle$ as a generic placeholder for either episode ep_i or ep_g in the value history of A, for which we need to introduce goals. We similarly define $ep2$ as the episode that immediately follows $ep1$. Since $ep1$ (and similarly for $ep2$) is in the value history, we know from definition 4 that sc_1 can only involve an assignment to l, the location variable of A. Let's also identify the set of transition functions of the form, $\mathcal{T} = \langle l_s, l_e, g, c := 0 \rangle$ [def. 2], for which sc_1 is the assignment $l = l_s$.

The maintenance goal requires the following additions: A new goal episode for each $\mathcal{T}$ where sc_2 is not the assignment $l = l_e$, of the form $ep_{new} = \langle e_{s1}, e_{e1}, 0, \infty, \neg g \rangle$. The goal required to affect a transition requires both a new goal episode and a justification. We add one goal episode for $\mathcal{T}$ where sc_2 is the assignment $l = l_e$, of the form $ep_{new} = \langle e_{e1}, e_{new}, 0, \infty, g \rangle$. We also add a justification of the form $J = \langle e_{e1}, e_{new}, 0, \infty \rangle$. These last two additions ensure that as soon as the guard of the transition is satisfied, A will transition to its next state.

Algorithm Specifics At this point we have defined many of the underlying details of the tBurton algorithm. In order to maintain search state, the algorithm uses a queue to keep track of the partial plans, SP_{part} that it needs to explore. For simplicity, one can assume this queue is FIFO, although in practice, a heuristic could be used to sort the queue. To make search more efficient, we make two additional modifications to the SP_{part} we store on the queue. First, we annotate SP_{part} with which of the three choices it needs to make next. The second involves the use of Incremental Temporal Order (ITO). When tBurton needs to select a goal ordering for a given partial plan, it could populate the queue with partial plans representing all the variations on goal ordering. Since storing all of these partial plans would be very memory intensive, we add to the partial plan a data structure from ITO, which allows us to store one partial plan, and poll it for the next temporally consistent variation in goal ordering.

Unifying a Goal History

One of the choices that tBurton needs to make is how to order the goals required of an automaton. More specifically, given a set of goal-episodes, we want to add temporal constraints relating the events of those goal-episodes, so any goal-episodes that co-occur have logically consistent guards. By computing the total order over all the goal-episodes we can discover any inconsistencies, where, for example two goal-episodes could not possibly be achieved. In practice, computing the total order of goal-episodes is faster than computing a plan, so we can discover inconsistencies inherent in the set of goal-episodes faster than we can discover that no plan exists to get from one goal-episode to the next.

Our incremental total order algorithm builds upon work which traverses a tree to enumerate all total orders given a partial order (Ono and Nakano 2005). We modify their algorithm into an incremental one consisting of a data structure maintaining theposition in the tree, an initialization function, `init`, and an incremental function, `next`, which returns the next total order.

Algorithm 1: ITO($ITOdata, SP_{part}, l$)

```
Input: The ITO data structure $ITOdata$, Partial State Plan $SP_{part}$, and the
       variable for which we want to unify the goal histories.
Output: Partial State Plan $SP_{part}$ with the a total order over the goal episodes
        of l.
while True do
    //return the total order over all events in $SP_{part}$;
    total-order = ITOdata.next();
    if total-order = null then
        return null;
    started-goal-episodes = $\emptyset$;
    for each event in total-order do
        ep = get_episode_started_by_event($SP_{part}$,event);
        if (ep != null) and (ep.guard.contains(l)) then
            for each episode ep2 in started_goal_episodes do
                if $guard(ep) \wedge guard(ep2) = false$ then
                    continue;
                else
                    started_goal_episodes = started_goal_episodes $\cup$ ep;
        ep = event_ends_an_episode($SP_{part}$,event));
        if (ep != null) and (ep.guard.contains(event)) then
            started_goal_episodes = started_goal_episodes $\setminus$ ep;
    // induce the total order in the goal episodes of the partial plan.;
    for each sequential pair of events {e1, e2} in total-order do
        $SP_{part}$.add-episode(e1,e2,0,$\infty$);
    If ITC($SP_{part}$) is true, return $SP_{part}$. Otherwise, continue.
```

Our ITO algorithm is similarly divided into two pieces. The `initITO` algorithm, not shown, creates the ITO data structure by calling `init` on SP_{part}, treating the episodes as partial orders. `ITO`, as given in algorithm 1, can then be called repeatedly to enumerate the next temporally consistent goal ordering.

Causal Graph Synthesis and Temporal Consistency

We use existing algorithms for Causal Graph Synthesis and Incremental Temporal Consistency (ITC), so we briefly motivate their purpose and use.

The Causal Graph Synthesis (CGS) algorithm is based on the algorithm used for Burton (Williams and Nayak 1997), and is simple to describe. Given a TCA, CGS checks the dependencies of each automaton in TCA and creates a causal graph. If the causal graph contains cycles, the cycles are collapsed and the product of the participating automata are used to create a new compound automata. Finally, each automaton in the causal graph is numbered sequentially in a depth-first traversal of the causal graph starting from 1 at a leaf. The numbering imposes a search order (with 1 being first), which removes the need to choose which factor to plan for next.

The ITC algorithm is used to check whether the partial plan SP_{part} is temporally consistent, or that the temporal constraints in goal, value histories, and justifications are satisfied. Since tBurton will perform this check many times with small variations to the plan, it is important this check be done quickly. For this purpose we use the incremental temporal consistency algorithm defined in (Shu, Effinger, and Williams 2005).

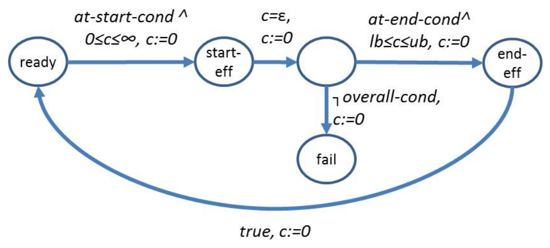


Figure 3: An example TCA automaton for a generic PDDL grounded action.

Mapping PDDL to TCA

Even though tBurton reasons over TCAs, it is still a capable PDDL planner. In order to run tBurton on PDDL problems, we developed a PDDL 2.1 (without numeric fluents) to TCA translator. Here, we provide only a sketch of this translator.

In order to maintain required concurrency, the translator first uses temporal invariant synthesis (Bernardini and Smith 2008) to compute a set of invariants. An instance of an invariant identifies a set of ground predicates for which at most one can be true at any given time. We select a subset of these invariant instances that provide a covering of the reachable state-space, and encode each invariant instance into an automaton. Each possible grounding of the invariant instance becomes a location in the automata.

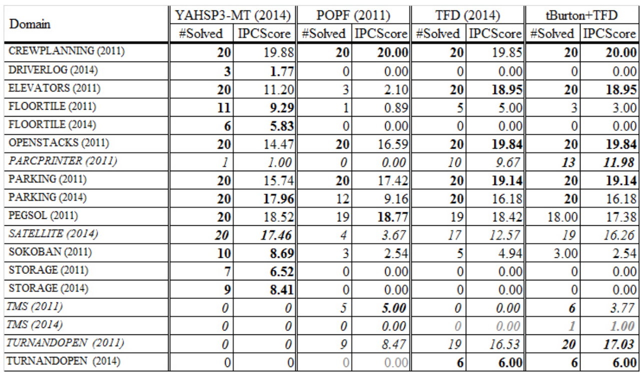
Each ground durative action is also translated into an automaton (Figure 4). Three of the transitions are guarded by conditions from the corresponding PDDL action, translated into propositional state logic over location variables. Another transition uses ϵ to denote a small-amount of time to pass for the start-effects of the action to take effect prior to checking for the invariant condition. A fifth transition is used to reset to the action.

Finally, the transitions of each invariant-instance based automata is labeled with a disjunction of the states of the ground-action automata that affect its transition.

Inherent in this mapping is an assumption that PDDL durative actions will not self-overlap in a valid plan. Planning with self-overlapping durative actions in PDDL is known to be EXPSPACE, without is PSPACE (Rintanen 2007). This suggests that tBurton may solve a simpler problem, but the actual complexity of TCA planning with autonomous transitions has yet to be addressed. In the meantime, if the number of duplicate actions can be known ahead of time, they can be added as additional ground-action automata.

Results

We benchmarked tBurton on a combination of IPC 2011 and IPC 2014 domains. Temporal Fast Downward (TFD) from IPC 2014 was used as an 'off-the-shelf' sub-planner for tBurton because it was straight-forward to translate $TCAs$ to the SAS representation used by TFD (Helmert 2006). For comparison, we also benchmarked against YAHSP3-MT from IPC 2014, POPF2 from IPC 2011, and TFD from IPC 2014, the winner or runners up in the 2011 and 2014 temporal satisficing track (Table 1). Each row represents a domain. Columns are grouped by planner and show the number of problems solved (out of 20 for each domain) and the IPC score (out of 20, based on minimizing make-span). Rows



Domain	YAHSP3-MT (2014)		POPF (2011)		TFD (2014)		tBurton+TFD	
	#Solved	IPCScore	#Solved	IPCScore	#Solved	IPCScore	#Solved	IPCScore
CREWPLANNING (2011)	**20**	19.88	**20**	**20.00**	**20**	19.85	**20**	**20.00**
DRIVERLOG (2014)	**3**	**1.77**	0	0.00	0	0.00	0	0.00
ELEVATORS (2011)	**20**	11.20	3	2.10	**20**	**18.95**	**20**	**18.95**
FLOORTILE (2011)	**11**	**9.29**	1	0.89	5	5.00	3	3.00
FLOORTILE (2014)	**6**	**5.83**	0	0.00	0	0.00	0	0.00
OPENSTACKS (2011)	**20**	14.47	**20**	16.59	**20**	**19.84**	**20**	**19.84**
PARCPRINTER (2011)	*1*	*1.00*	*0*	*0.00*	*10*	*9.67*	***13***	***11.98***
PARKING (2011)	**20**	15.74	**20**	17.42	**20**	**19.14**	**20**	**19.14**
PARKING (2014)	**20**	**17.96**	12	9.16	**20**	16.18	**20**	16.18
PEGSOL (2011)	**20**	18.52	19	**18.77**	19	18.42	18.00	17.38
SATELLITE (2014)	***20***	***17.46***	*4*	*3.67*	*17*	*12.57*	*19*	*16.26*
SOKOBAN (2011)	**10**	**8.69**	3	2.54	5	4.94	3.00	2.54
STORAGE (2011)	**7**	**6.52**	0	0.00	0	0.00	0	0.00
STORAGE (2014)	**9**	**8.41**	0	0.00	0	0.00	0	0.00
TMS (2011)	*0*	*0*	*5*	***5.00***	*0*	*0.00*	***6***	*3.77*
TMS (2014)	*0*	*0*	*0*	*0.00*	*0*	*0.00*	*1*	*1.00*
TURNANDOPEN (2011)	*0*	*0*	*9*	*8.47*	*19*	*16.53*	***20***	***17.03***
TURNANDOPEN (2014)	0	0	0	0.00	**6**	**6.00**	**6**	**6.00**

Figure 4: Benchmark results on IPC domains

with interesting results between TFD and tBurton are italicized, and the best scores in each domain are bolded. The tests were run with scripts from IPC 201 and were limited to 6GB memory and 30 minute runtime.

In general, tBurton is capable of solving approximately the same number of problems as TFD with the same quality, but for problems which admit some acyclic causal factoring (parcprinter, satellite, and TMS), tBurton performs particularly well. On domains which have no factoring, tBurton's high-level search provides no benefit and thus degrades to using the sub-planner. This often resulted in tBurton receiving the same score as its subplanner, TFD (elevators, parking, openstack). While the same score often occurs when TFD is already sufficiently fast enough to solve the problem, there are a few domains where tBurton's processing overhead imposes a penalty (floortile, sokoban). A nice benefit of using tBurton is that it is capable of solving problems with required concurrency. For TMS, tBurton is able to consistently solve more problems than the other planners. However, the IPCScore of tBurton is lower than POPF, perhaps because the goal-ordering strategy used by tBurton does not inherently minimize the make-span like the temporal relaxed planning graph used by POPF.

While bench-marking on existing PDDL domains is a useful comparison, it is worth noting that these problems do not fully demonstrate tBurton's capabilities. In particular, none of these domains have temporally extended goals, and all actions have a single duration value, instead of an interval. We look forward to more comprehensive testing with both existing PDDL domains and developing our own benchmarks.

Conclusion

This paper presents tBurton, a planner that uses a novel combination of causal-graph factoring, timeline-based regression, and heuristic forward search to plan for networked devices. It is capable of supporting a set of problem features not found before in one planner, and is capable of doing so competitively. Furthermore, tBurton can easily benefit from advancements in state-of-the art planning by replacing its sub-planner. The planning problem tBurton solves assumes devices with controllable durations, but we often have little control over the duration of those transitions. In future work we plan to add support for uncontrollable durations.

References

Alur, R., and Dill, D. 1994. A theory of timed automata. *Theoretical computer science* 126(2):183–235.

Amir, E., and Engelhardt, B. 2003. Factored planning. In *IJCAI*, volume 3, 929–935. Citeseer.

Benton, J.; Coles, A. J.; and Coles, A. 2012. Temporal planning with preferences and time-dependent continuous costs. In *ICAPS*, volume 77, 78.

Bernardini, S., and Smith, D. 2008. Translating pddl2. 2. into a constraint-based variable/value language. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling, 18th International Conference on Automated Planning and Scheduling (ICAPS08)*.

Brafman, R. I., and Domshlak, C. 2003. Structure and complexity in planning with unary operators. *J. Artif. Intell. Res.(JAIR)* 18:315–349.

Brafman, R. I., and Domshlak, C. 2006. Factored planning: How, when, and when not. In *AAAI*, volume 6, 809–814.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; et al. 2000. Aspen–automated planning and scheduling for space mission operations. In *Space Ops*, 1–10.

Coles, A. J.; Coles, A.; Fox, M.; and Long, D. 2010. Forward-chaining partial-order planning. In *ICAPS*, 42–49.

Cushing, W.; Kambhampati, S.; Weld, D. S.; et al. 2007. When is temporal planning really temporal? In *Proceedings of the 20th international joint conference on Artifical intelligence*, 1852–1859. Morgan Kaufmann Publishers Inc.

Dechter, R. 2003. *Constraint processing*. Morgan Kaufmann.

Frank, J., and Jónsson, A. 2003. Constraint-based attribute and interval planning. *Constraints* 8(4):339–364.

Helmert, M. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26(1):191–246.

Ingham, M. D. 2003. *Timed model-based programming: Executable specifications for robust mission-critical sequences*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Kelareva, E.; Buffet, O.; Huang, J.; and Thiébaux, S. 2007. Factored planning using decomposition trees. In *IJCAI*, 1942–1947.

Levine, S. J. 2012. *Monitoring the execution of temporal plans for robotic systems*. Ph.D. Dissertation, Massachusetts Institute of Technology.

Muscettola, N.; Morris, P.; and Tsamardinos, I. 1998. Reformulating temporal plans for efficient execution. In *In Principles of Knowledge Representation and Reasoning*. Citeseer.

Ono, A., and Nakano, S.-i. 2005. Constant time generation of linear extensions. In *Fundamentals of Computation Theory*, 445–453. Springer.

Penberthy, J., and Weld, D. 1992. Ucpop: A sound, complete, partial order planner for adl. In *proceedings of the third international conference on knowledge representation and reasoning*, 103–114. Citeseer.

Rintanen, J. 2007. Complexity of concurrent temporal planning. In *ICAPS*, 280–287.

Röger, G.; Eyerich, P.; and Mattmüller, R. 2008. Tfd: A numeric temporal extension to fast downward. ipc 2008 short pape rs.

Shu, I.; Effinger, R.; and Williams, B. 2005. Enabling fast flexible planning through incremental temporal reasoning with conflict extraction. *Proce. ICAPS-05, Monterey*.

Vidal, V. 2011. Yahsp2: Keep it simple, stupid. *The 2011 International Planning Competition* 83.

Williams, B., and Nayak, P. 1997. A reactive planner for a model-based executive. In *International Joint Conference on Artificial Intelligence*, volume 15, 1178–1185. LAWRENCE ERLBAUM ASSOCIATES LTD.

Younes, H. L., and Simmons, R. G. 2003. Vhpop: Versatile heuristic partial order planner. *J. Artif. Intell. Res.(JAIR)* 20:405–430.