Efficient Temporal Planning Using Metastates

Amanda Coles* and **Andrew Coles*** and **J. Christopher Beck†**
* Department of Informatics, King's College London, UK.
† Department of Mechanical & Industrial Engineering, University of Toronto, Canada.
email: {amanda,andrew}.coles@kcl.ac.uk, jcb@mie.utoronto.ca

Abstract

When performing temporal planning as forward state-space search, effective state memoization is challenging. Whereas in classical planning, two states are equal if they have the same facts and variable values, in temporal planning this is not the case: as the plans that led to the two states are subject to temporal constraints, one might be extendable into at temporally valid plan, while the other might not. In this paper, we present an approach for reducing the state space explosion that arises due to having to keep many copies of the same 'classically' equal state – states that are classically equal are aggregated into metastates, and these are separated lazily only in the case of temporal inconsistency. Our evaluation shows that this approach, implemented in OPTIC and compared to existing state-of-the-art memoization techniques, improves performance across a range of temporal domains.

1 Introduction

Planning is fundamental to intelligent autonomous behavior and reasoning about time is essential to planning in many real-world domains. One of the most popular paradigms for planning is forward state-space search, and key to its success, is memoization. In classical planning if a given state (set of propositions and variable assignments) has been seen before, then it need not be explored again if it is reached by a different sequence of actions. This reasoning can easily be extended to a setting where actions have costs, by keeping only the lowest cost path to a given state.

Temporal planning brings with it further challenges for memoization (Coles and Coles 2016). Not only are the values of variables and propositions important, as in classical planning, but also the path taken to reach a state can determine whether the partial plan can be extended into a temporally valid plan to reach the goal. For example, if we took a longer path to reach a state, we might no longer be able to meet a deadline. In the worst case in temporal planning, this results in the need to keep different states for every possible path to every classically equal state. Coles and Coles (2016) made some progress on this issue with a technique to prune isomorphic plans and to identify special cases where achieving facts earlier can always be proven to be better. However, there are many cases where states that are not likely to be interestingly different still have to be considered as such.

In this paper we propose a radically different approach to dealing with the need to explore classically identical states in temporal planning. Our approach is based on the idea of *metastates*. Instead of inserting multiple copies of a classically identical state (which are reached by different paths) into the open list for search to explore, we maintain a single metastate that aggregates these. Each time a new state is generated, we either create a new metastate if it is classically unique; or if we have previously seen a classically identical state we add information to the existing metastate to record that there is an alternative path to the state (which may lead to different temporal constraints). Now, we search over the space of metastates, nominally expanding only one member from each. Since here we are interested in satisfycing planning we need only consider expanding the other members of a metastates, if its descendants are temporally inconsistent.

We empirically evaluate our approach on temporal planning domains and our results show a significant improvement in performance over the state-of-the-art in memoization for temporal planning.

2 Background

2.1 Problem Definition

A PDDL2.1 (Fox and Long 2003) planning problem is defined over a collection of propositions P, and a vector of numeric variables $\mathbf{v}$. These are manipulated and referred to by actions. The executability of actions is determined by their preconditions, conjunctions of *conditions*. A *condition* is either a single proposition $p \in P$, $\neg p$, or a numeric constraint over $\mathbf{v}$. We assume all such constraints are linear, and hence can be represented in the form $\mathbf{w}.\mathbf{v}\{>, \geq, <, \leq, =\}c$ where $\mathbf{w}$ is a vector of constants and c is a constant).

Each durative action A has three sets of preconditions: $\text{pre}_{\vdash} A$, $\text{pre}_{\leftrightarrow} A$, $\text{pre}_{\dashv} A$. These represent the conditions that must hold at its start, throughout its execution (invariants), and at the end, respectively. Instantaneous effects can occur at the start or end of A: $\text{eff}^{+}_{\vdash} A$ ($\text{eff}^{-}_{\vdash} A$) denote propositions added (resp. deleted) at the start; $\text{eff}^{num}_{\vdash} A$ denotes numeric effects. Similarly, $\text{eff}^{+}_{\dashv} A$, $\text{eff}^{-}_{\dashv}$ and $\text{eff}^{num}_{\dashv}$ record effects at the end. We assume numeric effects are of the form: $v\{\texttt{+=}, \texttt{-=}, \texttt{=}\}\mathbf{w}.\mathbf{v} + c$ ($v \in \mathbf{v}$). Semantically, the values instantaneous of effects become available small amount of time, ϵ, after they occur.

Finally, the action has a duration constraint: a conjunction of numeric constraints applied to a special variable dur_A de-

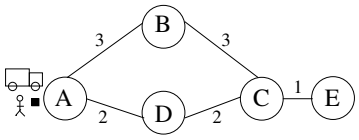

Figure 1: Example Driverlog Problem

noting its duration. As a special case, *instantaneous* actions have duration ϵ, and only one set of preconditions pre A and effects eff^+A, eff^-A, and $\text{eff}^{num}A$. A durative action A can be split into two instantaneous *snap*-actions, $A_\vdash$ and $A_\dashv$, representing the start and end of the action respectively, and a set of constraints (invariant and duration constraints). Action $A_\vdash$ has precondition $\text{pre}_\vdash A$ and effects $\text{eff}^+_\vdash A$, $\text{eff}^-_\vdash A$, $\text{eff}^{num}_\vdash A$. $A_\dashv$ is the analogous action for the end of A.

A solution to the problem is a *plan*: timestamped a sequence of actions with associated durations, that transforms the initial state I into one that satisfies the goal G. All pre/invariant conditions must satisfied at the time of/during execution and actions that have started must have finished.

2.2 Memoization in Temporal Planning

Forward search temporal planning begins from the initial state: a set propositions that are known to be true and assignments to the numeric variables in $\mathbf{v}$. At each state S during search the planner generates successor states, each S' corresponding to the application of a *logically applicable* snap action: one whose preconditions are satisfied in S, and whose effects do violate the invariant conditions $\text{pre}_\leftrightarrow A$ of any action A that has started but not yet ended. Searching in this way ensures all plans are logically valid (all preconditions are satisfied), but does not ensure they are temporally valid (respect the duration constraints of actions).

In this work we build on the planner OPTIC, which records temporal constraints in each state in the form of a Simple Temporal Problem (STP) (Dechter, Meiri, and Pearl 1991) or Mixed Integer Program (MIP), and updates these as search progresses. Because of these temporal constraints, state memoization – i.e. determining when two states are equivalent, to avoid redundant search – is more difficult than in classical planning. Suppose we have two states A and B, with the same facts, variable values, and executing actions, but different temporal constraints. While these may have the same *logically* applicable actions, applying the same action in each to yield A' and B' may yield different temporal constraints, such that those in A' are satisfied while those in B' are not. Thus, we cannot say that A and B are equal, as the search tree reachable under each may be different.

To understand why this might be, consider the Driverlog Shift problem depicted in Figure 1. The Driverlog Shift domain extends the Driverlog domain by adding a 'shift' action, that adds a predicate (available ?driver); then, all actions involving the driver (boarding, disembarking and driving the truck) have this predicate as an invariant, so must take place during the execution of shift. Let us assume that the duration of the shift action is 6 time units and it takes a nominal 0.1 time units to load and unload packages. This means that the plan to deliver the package to E (the goal) via B is not feasible as it will take too long; whereas going via D is feasible. Suppose in forward-search to reach this plan we encounter the state where the driver and package are in the truck at C, having followed the plan that drives via B, and memoize that we have seen the state with these facts true. Now, if we later reach this state again, having taken the path via D, classical memoization would prune the state, as it has the same facts as one that was seen before. In temporal planning this would render the problem unsolvable: the state reached via D, has an STN whose temporal constraints allow it to be extended into a solution plan; whereas the state via B, the only one we kept does not. In general, in temporal planning, we therefore cannot prune a state simply because it has the same facts as one we have already seen.

Coles and Coles (2016) took a step towards addressing this issue by testing for equality based on the *plan* to reach states by checking whether these plans are isomorphic. When OPTIC adds a new snap action to a plan, it is ordered only after other actions that are required for logical soundness, e.g. after the adder of its precondition. Suppose in our Driverlog example there are two packages *p* and *q* at A to be loaded onto the truck. OPTIC will generate two states, one resulting from applying load *p*, load *q* and the other, load *q*, load *p*. However, since the two load actions are independent they will not be ordered with respect to each other, so these plans are isomorphic partial orders. In terms of the planning problem isomorphic partial orders are effectively identical so by identifying such partial orders and keeping only one such state, the search space can be reduced.

This was effective at reducing the number of nodes expanded by search, but has the limitation that if the two plans contain different actions, they cannot be isomorphic: it can detect permutations of the same actions, but not interchanging different but effectively equivalent actions. It is this point we explore in this paper. For example, in our Driverlog Shift problem, two plans that start 'shift' then go from A to C – one going via B, the other via D – cannot be isomorphic as the 'move' actions are different, even if the truck in both is at C. We need to keep both for completeness when the duration of 'shift' is tight enough to preclude one of the plans from reaching a goal state; but if the duration of 'shift' is sufficiently long, it would not matter for the purposes of solving the problem which route was taken to C (although one might admit a better quality plan). But, regardless of the duration of 'shift', a temporal planner would have to keep both, and expand them as different states, blowing up the search tree; whereas a classical planner would keep only one.

While we implement our ideas in OPTIC, this is without loss of generality. Our work is applicable to other approaches to managing temporal constraints in forward planning, such as the decision epoch approach (Cushing et al. 2007) used in SAPA (Do and Kambhampati 2003) and TFD (Eyerich, Mattmüller, and Röger 2009) as these must still consider the queue of pending action ends when determining whether states are equal; as, again, the search tree reachable under two otherwise equal states may differ.

3 Metastate-Space Search

In this section we introduce the notion of metastates and explain the details of forward-search metastate space.

3.1 Preliminaries

A state is defined as follows:

Definition 3.1 — A state A state comprises:

- f – the facts that are true in the state.
- $\bar{v}$ – the values of each state variable.
- $P = [p_0..p_n]$ – a partial plan. Each p_i represents an instantaneous action, start snap action or end snap action.
- Q – a list of actions that started in P, but have not yet finished. For each $\langle a, i, i', d_{min}, d_{max}\rangle \in Q$:
  - a identifies the ground durative action;
  - i is its step index in the plan P;
  - d_{min}, d_{max} are the minimum/maximum duration of a, calculated based on the values of $\bar{v}$ in the state at step i.
- T – temporal constraints over the steps in the plan P.

In classical forward-search planning, two states S and S' can then be said to be equal if $S.f = S'.f$ and $S.\bar{v} = S'.\bar{v}$. The other parts of the tuple are irrelevant: Q is always empty, as all actions are instantaneous; T is unnecessary, as steps in P are totally ordered. Even if P is different in S and S', this has no bearing on further state expansion, that determines this is which preconditions are satisfied and since f and $\bar{v}$ are equal for each state the same preconditions must be satisfied in both. Thus, as the reachable search spaces under S and S' are identical, S' can be pruned if it is equal to another state S that has already been seen.

In temporal forward-search planning, we have to make three important distinctions:

- The list of actions Q matters: we can only end an action a – i.e. apply the snap-action $a_\dashv$, if $a \in Q$.
- Whenever an action has been applied, all the invariant conditions of the actions recorded in Q must be respected.
- T must be consistent. Otherwise, while the plan under construction may be logically consistent (all preconditions are satisfied), it may be temporally inconsistent – for instance, if T constrains a long durative action to be contained entirely within a shorter one.

The first two of these are logical in flavor: a straightforward inspection of Q will suitably restrict the applicable actions to respect invariant constraints and start–end planning semantics, without reference to temporal information *per se*. The last, however, poses an issue for state memoization. If two states S and S' have the same facts, the same variable values, and the same actions in Q, the same actions will be logically consistent extensions of S and S'. But, they might not be temporally consistent extensions of both S and S', due to their effect on the temporal constraints T. Thus, it is not completeness preserving to keep only one of S or S'.

The current state of the art (Coles and Coles 2016) addresses the memoization issue in two ways. First they prove that, for states in which no actions are executing if Q is empty (i.e. there are no open actions), and S and S' are both temporally consistent, it is completeness preserving to keep only one of them. Because search proceeds in a forwards direction, all subsequent plan steps will be ordered after those already in the plan; and as these constraints w.r.t. the existing plan steps are only ever minimum separation constraints (a later step is ordered 0 or ϵ after some existing step), any temporally consistent plan extension from S will also be consistent from S'. Second, for other all other states (with open actions), this observation no longer applies, in which case S and S' are both be kept unless their plans are isomorphic; i.e. exactly the same actions were applied in a different order, but leading to identical temporal constraints. This was shown to be effective compared to simpler alternative of keeping *all* states with open actions, but the requirement that S and S' must be reached by identical actions limits its pruning that can be achieved.

3.2 Strong and Weak Equality

The core limitation of standard memoization is that to preserve completeness, it must use strict state duplicate detection. As noted, this is particularly restrictive when comparing states with open actions. For instance, a state S is reached by moving from A to B to C and then starting an action a, is different to a state S' moving from A to D to C and then starting a – even if all the facts in the states are the same, a is an open action, so plan isomorphism is used. As the two plans contain different actions, they cannot be isomorphic, so both S and S' are kept.

In practice, we hypothesize that it is often sufficient to only expand one of S or S', as they are sufficiently similar, but we want to maintain completeness in the case where expanding the other was necessary. As a step towards this, we first define notions of *strong* and *weak* state equality. Strong equality is suitable for completeness-preserving memoization; weak equality is not, but defines the concept of two states being 'sufficiently similar'.

Ancillary to this, we define two helper functions, the number of times action a is executing in the queue Q of a state:

$$num\_exec(Q, a) = |\{i \mid \langle a, i, d_{\min}, d_{\max}\rangle \in Q\}|$$

...and a set of pairs each comprising an action, and the number of times that action is executing in a state:

$$exec(Q) = \{\langle a, num\_exec(Q, a)\rangle \mid num\_exec(Q, a) > 0\}$$

Recall that the semantics of PDDL 2.1 permit an action to self-overlap (Rintanen 2007) (i.e. a new instance of a can start before a previous instance of a has ended): this is why action a may be executing more than once in the same state.

We now define strong equality as follows:

Definition 3.2 — Strong equality
States $S = \langle f, \bar{v}, P, Q, T\rangle$ and $S' = \langle f', \bar{v}', P', Q', T'\rangle$, where T and T' are temporally consistent, are strongly equal *iff* $f = f'$, $\bar{v} = \bar{v}'$ and either (i) $Q = Q' = \emptyset$; or (ii) the partial plans P and P' are *isomorphic*.

The notation $S = S'$ denotes strong equality.

Note that if $S = S'$, and $S = S''$, then trivially, $S' = S''$: in the first case all queues must necessarily be empty; and in the second case, isomorphism is transitive.

Weak equality broadens this:

Definition 3.3 — Weak equality
States $S = \langle f, \bar{v}, P, Q, T\rangle$ and $S' = \langle f', \bar{v}', P', Q', T'\rangle$, where T and T' are temporally consistent, are weakly equal *iff* $f = f'$, $\bar{v} = \bar{v}'$, $exec(Q) = exec(Q')$.

The notation $S \approx S'$ denotes weak equality.

As with strong equality, weak equality is trivially transitive. Additionally, it is a strict relaxation of strong equality: $(S = S') \Rightarrow (S \approx S')$. For the two cases of strong equality:

- The first case is a restriction of weak equality: if $Q = Q' = \emptyset$ then $exec(Q) = exec(Q') = \emptyset$.
- In the second case, if P and P' are isomorphic they must contain the same actions; so if P contains $a_{\vdash}$ more times than $a_{\dashv}$ then so does P', so trivially $exec(Q) = exec(Q')$.

3.3 Metastates

We now make use of strong and weak equality to define a meta-state-space over which to search; and a search algorithm to do this. We define a meta-state as follows:

Definition 3.4 — Metastate
A metastate M is a tuple $\langle \Sigma, \Pi, \Gamma, \rho, ex, re, c\_re \rangle$, where:

- Σ is a list of member states $[\sigma_0..\sigma_n]$, that are pairwise weakly equal.
- Π is a list of parent metastates, and the action applied to reach M from the parent – $[\langle M_0, a_0 \rangle..\langle M_n, a_n \rangle]$.
- Γ, a list of child metastates and the action applied to reach them – $[\langle M_0, a_0 \rangle..\langle M_n, a_n \rangle]$.
- $q \in \{\bot, \top\}$ is a boolean flag set to $\top$ *iff* the metastate is queued for expansion.
- $ex \in \mathbb{Z}_0^+$, a counter of how many of Σ have been explicitly expanded.
- $re \in \{\bot, \top\}$ is a boolean flag set to $\top$ *iff* either M has not yet been expanded; or the most recent expansion of M was partial due to one or more successors being inconsistent according to the temporal constraints.
- $c\_re \in \mathbb{Z}_0^+$ counts how many of the children of M could in principle lead to a larger reachable search space if given an additional member.

In this definition, Σ records the states in the metastates, and Π and Γ define the structure of the metastate space: if there is an edge in the metastate space between two metastates M and M' labeled with the action a, then $\langle M', a \rangle \in M.\Gamma$ and $\langle M, a \rangle \in M'.\Pi$. The other entries in the tuple are bookkeeping information to support search, we will explain the meaning of these and how they are used later.

3.4 Searching with Metastates: Overview

We begin with a high-level overview of search:

a) A search queue of metastates is initialized to the metastate containing the initial state.

b) At each iteration, a metastate is popped from the search queue and expanded. The expansion of a metastate M expands one of its member states; specifically, $\sigma_{M.ex}$, the state in $M.\Sigma$ with index $M.ex$.

c) Search aims to expand each metastate only once; however, to ensure completeness there are two cases in which a metastate must be re-expanded:

  i If there was an action that was logically applicable in the last-expanded member of M, but which led to a temporally inconsistent child state (i.e. if $M.re$). It is necessary to consider other member states of M as they have different temporal constraints and hence may lead to a temporally consistent child.

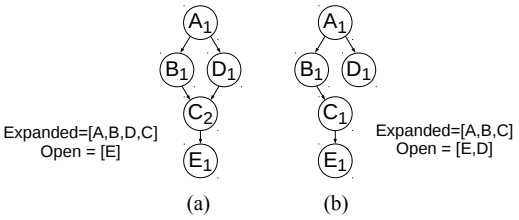

Figure 2: Example metastate spaces, with the order in which metastates were expanded, and their open lists. The letter denotes the truck location; the number is $|M.\Sigma|$.

  ii If one of the children of M, transitively, needs to be re-expanded. In this case, we must consider other member states of M as expanding them will add new members to the children of M, supporting their re-expansion.

d) Search attempts to avoid explicitly generating all the member states of a metastate. New members are only explicitly generated under one of two conditions:

  i If search reaches a state S that is weakly equal to the states in an existing metastate M, S is added to M.

  ii If a metastate needs to be re-expanded, c (i), but all the members of M have already been expanded, a traversal back via the parents of M ($M.\Pi$) is used to generate additional member states for M. If no new members can be found the c_re values of M's ancestors are incremented, so that if they acquire new members they are re-expanded as per c (ii).

To illustrate why re-expansion of metastates or generation of new metastate members is sometimes necessary, two example metastate spaces are depicted in Figure 2. These are based on the Driverlog Shift example (Figure 1) and the task is to reach location E and 'unload' a package. For simplicity we assume that in Metastate A (where search begins in this example) the package and driver are already in the truck and the 'shift' action has started. Further, we only show states reachable by applying move actions: the letter representing each metastate corresponds to the location of the truck. Each state-space demonstrates a different search scenario:

- In Figure 2(a), the metastate C has two members corresponding to the states reached by plans ABC and ADC. Metastate E, however, only has one member as when C was expanded, only its first member (reached via ABC) was used, generating one state in E for the plan ABCE. The metastate E is then popped from the open list and expanded. The 'unload' action needed here cannot be applied, due to the 'shift' action constraining how much time can pass: this is because the long route to E, via B, was taken. E must now be re-expanded (c (i)) but it has no other member states. Thus, a traversal back via the parents of E attempts to generate additional members for E (d (ii)). This traversal finds the second member state of C (reached via ADC); appends 'move CE' to the plan to yield another state weakly equal to the existing member of E; then enqueues E for re-expansion. This state will then be expanded and successfully reach the goal in time.
- In Figure 2(b), B has been expanded, but D has not. Thus, C has only one member, and when E is unsuccessfully

Algorithm 1: Memoize

Data: $memoized$, a set of all metastates generated; S a new state; $\langle MP, a\rangle$, the metastate parent of S and action applied to reach it
Result: $enqueue$, a list of metastates to subsequently enqueue

1 **if** $\exists M \in memoized \mid M.\sigma_0 \approx S$ **then**
2 $\quad$ **return** $AddMemberToMetaState(M, S, \langle MP, a\rangle)$;
3 **else**
4 $\quad M \leftarrow \langle [S], [\langle MP, a\rangle], [], \top, 0, \top, 0\rangle$;
5 $\quad memoized \leftarrow memoized \cup M$;
6 $\quad$ **return** $[M]$

Algorithm 2: AddMemberToMetaState

Data: M, a metastate; S, a new weakly-equal member state; $\langle MP, a\rangle$, the metastate parent of S and action applied to reach it
Result: $enqueue$, a list of zero or one metastates to subsequently enqueue

1 **if** $\exists S' \in M.\Sigma \mid S' = S$ **then return** [];
2 $M.\Sigma \leftarrow M.\Sigma + [S]$;
3 **if** $\langle MP, a\rangle \notin M.\Pi$ **then** $M.\Pi \leftarrow M.\Pi + [\langle MP, a\rangle]$;
4 **if** $M.q$ **then return** [];
5 **if** $M.re \vee M.c\_re > 0$ **then**
6 $\quad M.ex \leftarrow |M.\Sigma| - 1$; $M.q \leftarrow \top$; **return** $[M]$;
7 **return** []

expanded, a traversal back via its parents (d (ii)) cannot generate any additional members, as they all have only one member state. Hence, the c_re values in E's ancestors (C, B and A) are incremented to note that they have a child that needs to be re-expanded. When D is popped from the open list, the state generated (ADC) becomes an additional member of C (d (i)). C will then be put on the open list as $C.c\_re{>}0$ (c (ii)); C is re-expanded generating a new member for E (ADCE), putting E on the open list as $E.re{=}\top$ (c (i)); E is re-expanded, using this new member, leading to successful application of 'unload'.

3.5 Searching with Metastates: Algorithms

Algorithm 1 defines how state memoization is performed with reference to the metastate space. For each state S encountered in search, first a check is made at line 1 to see if there is an existing metastate whose members are weakly equal to S. If there is, AddMemberToMetaState is used to update the metastate. In the simpler case, if there is no such metastate, then a new one is created, containing just S.

Algorithm 2 deals with adding a new member S to a metastate M. At line 1 we check whether an existing member is strongly equal to S – if it is, S does not need to be kept. Otherwise, it is added to the members of the metastate.

As our aim is to avoid redundant expansion of the search space, when a new member has been added to a metastate, $M.q$ is then checked to see if M is already queued for expansion: if it is, it need not be queued again. Otherwise, if it is not queued, then the metastate is returned to be enqueued only if one of two conditions holds (e (i)/(ii) Section 3.4):

Algorithm 3: Search

Data: The initial state I
Result: A solution plan

1 $IM \leftarrow \langle [I], [], [], \top, 0, \top, 0\rangle$;
2 $memoized \leftarrow \{IM\}$;
3 $Q \leftarrow [IM]$;
4 **while** Q *is not empty* **do**
5 $\quad M \leftarrow$ pop the next metastate from Q;
6 $\quad S \leftarrow M.\sigma_{M.ex}$;
7 $\quad M.q \leftarrow \bot$; $M.ex \leftarrow M.ex + 1$;
8 $\quad enqueue \leftarrow []$;
9 $\quad$ **if** $M.c\_re > 0$ **then**
10 $\quad\quad$ **foreach** $\langle M_i, a_i\rangle \in M.\Gamma$ **do**
11 $\quad\quad\quad$ **if** $M_i.q$ **then continue**;
12 $\quad\quad\quad$ **if** $M_i.re \vee M_i.c\_re > 0$ **then**
13 $\quad\quad\quad\quad S_i \leftarrow$ apply a_i to S;
14 $\quad\quad\quad\quad$ **if** $S'.T$ *is temporally consistent* $\wedge\ \nexists S' \in M_i.\Sigma \mid S' = S_i$ **then**
15 $\quad\quad\quad\quad\quad M_i.\Sigma \leftarrow M_i.\Sigma + [S_i]$;
16 $\quad\quad\quad\quad\quad M_i.q \leftarrow \top$;
17 $\quad\quad\quad\quad\quad enqueue \leftarrow enqueue + [M_i]$;
18 $\quad\quad\quad\quad\quad M.c\_re \leftarrow M.c\_re - 1$;
19 $\quad$ **if** $M.re$ **then**
20 $\quad\quad M.re \leftarrow \bot$;
21 $\quad\quad$ **foreach** a *that is logically applicable in* S **do**
22 $\quad\quad\quad S' \leftarrow$ apply a in S;
23 $\quad\quad\quad$ **if** $S'.T$ *is temporally consistent* **then**
24 $\quad\quad\quad\quad$ **if** S' *is a goal state* **then return** S';
25 $\quad\quad\quad\quad enqueue \leftarrow enqueue + Memoize(memoized, S', \langle M, a\rangle)$;
26 $\quad\quad\quad$ **else** $M.re \leftarrow \top$;
27 $\quad$ **if** $M.re \vee (M.c\_re > 0)$ **then**
28 $\quad\quad$ **if** $M.ex = |M.\Sigma|$ **then** $M.\Sigma \leftarrow M.\Sigma + LookForAnotherMember(M, M, [])$;
29 $\quad\quad enqueue \leftarrow enqueue + [M]$;
30 $\quad\quad M.q \leftarrow \top$;
31 $\quad$ **foreach** $M' \in enqueue$ **do**
32 $\quad\quad$ **while** $M'.ex < |M'.\Sigma|$ **do**
33 $\quad\quad\quad hM' \leftarrow$ heuristic evaluation of $M'.\sigma_{M'.ex}$;
34 $\quad\quad\quad$ **if** $hM' \neq \infty$ **then**
35 $\quad\quad\quad\quad$ insert M' into Q with h-value hM';
36 $\quad\quad\quad\quad$ **break**;
37 $\quad\quad\quad M'.ex \leftarrow M'.ex + 1$;
38 $\quad\quad\quad$ **if** $M'.ex = |M'.\Sigma|$ **then** $M'.\sigma \leftarrow M'.\Sigma + LookForAnotherMember(M', M', [])$;
39 $\quad\quad$ **if** $M'.ex = |M'.\Sigma|$ **then**
40 $\quad\quad\quad M'.q \leftarrow \bot$;
41 $\quad\quad\quad IncrementParentCREValues(M')$;
42 **return** *problem unsolvable;*

- if $M.re$, then the last expansion of M – i.e. an expansion based on a pre-existing member state – did not lead to a temporally consistent child for each logically applicable action. In this case, expanding M again will derive

children from the new member state.

- if $M.c\_re$, then the last expansion of M *did* lead to a temporally consistent child for each logically applicable action, but one or more of these when expanded either *did not* lead to a temporally consistent child for each logically applicable action, or is the ancestor of such a metastate. In this case, because there is a new member for M, expanding it again will yield new members for its children.

Algorithm 3 presents search itself: a priority-queue-based search, using a heuristic function. We use WA* with W=5; and a temporal RPG heuristic, as in OPTIC.

At its core (lines 21– 26) is an ordinary forward-search expansion loop, generating one successor per applicable action; keeping it if it is returned by the $Memoized$ function (Algorithm 1). There are three possible reasons a metastate M could be on the open list. First, if either $M.re = \top$ – i.e. if the successors of the previous expansion of M were not all temporally consistent (Section 3.4 e (i)), or if M has not yet been expanded. The aforementioned forward-search expansion loop is used to expand the metastate in these first two cases. The final case is when c_re>0: (Section 3.4 f (ii)) one or more of M's children needs to be re-expanded but needs additional member states before this can occur; and by re-expanding M we will generate an additional member for this child. This is handled by a selective state-expansion, that only generates member states for the child metastates that need them (lines 9–18). If $M.re$=$\bot$, this selective loop alone is sufficient, as the re and c_re values in the child metastates record which need additional members: child metastates reached by the other applicable actions can be ignored.

Following expanding the current member S of the current metastate M, lines 27–30 determine whether, following expansion, M needs to be added to the open-list again for re-expansion. This occurs either because when M was expanded some of its successors were not temporally consistent (i.e. $M.re$ was changed back to true at line 26) (Section 3.4 e (i)) or because it has at least one child that still would benefit from an additional member state being generated (Section 3.4 e (ii)). The complication here is what to do in the case where there are no more members of M left for expansion, i.e. if $M.ex = |M.\Sigma|$ (Section 3.4 f (ii)). In this case, a helper function traverses back from M to its parents, to generate additional candidate members of M from the members of its parents. This is presented in Algorithm 4 – this recursively generates candidate plans for reaching M from a parent, and if one is found that is temporally consistent, and not strongly equal to an existing member of M, then it is returned to be added as a member to M. This corresponds to the case in Figure 2(a): when the metastate E is expanded for the first time, 'unload' could not be applied without violating the temporal constraints; thus, LookForAnotherMember would be used to look for another member state for E. When traversing back to its parent, C, and looping over its two members (ABC and ADC), extending the first (to yield ABCE) would duplicate the existing member of E, but extending the second (to yield ADCE) would produce a new member for E, supporting its re-expansion. Of course, LookForAnotherMember cannot always generate a new member in which case it returns an empty list.

Algorithm 4: LookForAnotherMember

Data: MC, a metastate; MT, a metastate; $tail$, the plan from MC to MT
Result: $enqueue$, a list of 0 or 1 metastates to subsequently enqueue

1 **foreach** $\langle M_i, a_i \rangle \in MC.\Pi$ **do**
2 **foreach** $S \in M_i.\Sigma$ **do**
3 $S' \leftarrow$ apply $([a_i]$+$tail)$ to S;
4 **if** $S'.T$ *is temporally consistent and* $\nexists S'' \in MT.\Sigma \mid S'' = S'$ **then**
5 $MT.\Sigma \leftarrow MT.\Sigma + [S']$;
6 $MT.q \leftarrow \top$;
7 **return** $[MT]$;
8 $rc \leftarrow LookForAnotherMember(M_i, MT, [a_i]$+$tail)$;
9 **if** $rc \neq \emptyset$ **then return** rc;
10 **return** [];

Algorithm 5: IncrementParentCREValues

Data: M, a metastate
1 **foreach** $\langle M_i, a_i \rangle \in M.\Pi$ **do**
2 **if** $M_i.c\_re = 0$ **then** $IncrementParentCREValues(M_i)$;
3 $M_i.c\_re \leftarrow M_i.c\_re + 1$;

The final loop in Algorithm 3 at lines 31–41 considers each metastate M' that is to be enqueued on Q. In the nominal case, a member state $M'.\sigma_{M'.ex}$ is heuristically evaluated, and its heuristic value is used when placing M in Q. There are two caveats though. First, if $M'.\sigma_{M'.ex}$ is a dead-end, one of the other members of M' (if there is one) may not be; hence, $M'.ex$ is incremented, to allow further heuristic evaluation. As this may exhaust the generated members of M', we see again a call to Algorithm 4 to attempt to generate another member state. Second, if no additional members can be generated, it may be that if search later finds another plan that reaches one of the parents of M', from which it is possible to generate another member of M'. To preserve completeness, it is important that this can happen. This is achieved through the use of the c_re values, as updated by Algorithm 5: these count how many of the immediate children of a metastate are awaiting an additional member. Referring back to earlier in Algorithm 3, lines 9–18 then use these values to selectively push additional member states down to their children until, in turn, the new member for M' is generated.

Finally, we sketch a proof for the completeness of our algorithm. Trivially, any goal states generated are returned (Algorithm 3 line 24). Thus, to be incomplete, there must be a state S from which applying a sequence of snap-actions $tail = [a_0..a_n]$ was the only way to reach a goal state; but S was never expanded. In the simple case, S is the first member of a metastate M, and all metastates are queued for expansion at least once; so S would have been expanded. Otherwise, S must have been added as a member of an existing metastate M. In this case, because applying $tail$ to S is posited as being the only way to reach a goal state, there must be a subsequence $tail\_sub = [a_0..a_i], i \leq n$ of the $tail$

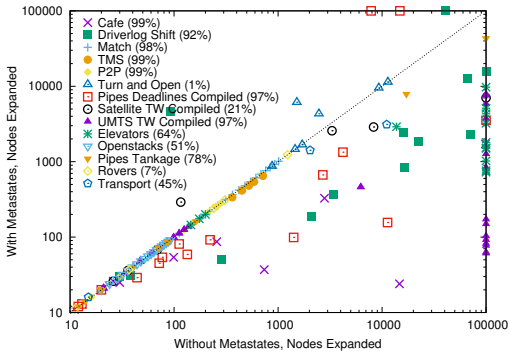

Figure 3: Scatterplot comparing nodes expanded

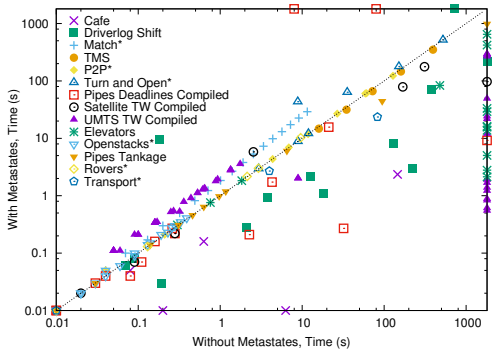

Figure 4: Scatterplot comparing time taken

snap actions that when applied to any other $S' \in M.\Sigma$ leads to an inconsistent state. Then, either (i) $S \in M.\Sigma$ when this occurred, so Algorithm 4 would be used to apply $tail\_sub$ to S; or (ii) $S \notin M.\Sigma$ when this occurred, so Algorithm 5 would increment $M.c\_re$ causing M (and hence S) to be expanded later, as soon as $S \in M.\Sigma$ (Algorithm 2 line 5). Thus, regardless of when S was added to $M.\Sigma$, $tail$ will be applied to it and hence the goal would be reached. ■

4 Evaluation

To evaluate our metastate search algorithm, we use as a control the state memoization techniques of Coles and Coles (2016) also implemented in OPTIC. In both cases we use WA* search (with W=5) guided by OPTIC's TRPG heuristic. For evaluation domains, we take International Planning Competition benchmarks, and temporally interesting domains from the literature. We exclude any domains where all actions are *compression safe*. In these domains, during search there will never be any states with open actions; hence, as discussed in Coles and Coles (2016), memoizing based on the classical planning notion of state equality is completeness preserving. Moreover, we would only have one member per metastate, as states with no open actions are strongly equal to other states with the same facts and variables, so our search would behave exactly as the control. This leaves 10 domains with required concurrency (Cushing et al. 2007) – Cafe and Driverlog Shift (Coles et al. 2009); P2P (Huang et al. 2009); TMS and Turn and Open (IPC2011); the compiled timewindows/deadlines variants of Pipes No-Tankage, Satellite and UMTS (IPC2004) – and 5 other IPC domains Rovers (2002); Pipes Tankage (2004); Transport, Elevators and Openstacks (2008).

An overview of the performance of our algorithm compared to the control is shown in Figures 3 and 4: where one configuration failed to solve a problem this is shown as 100,000 nodes or 1800 seconds. Nodes below the plotted line y=x indicate that using metastates improved the performance of the planner; those above that it was worsened. In total 194 problems were solved by at least one configuration, in 73 of these the planner using metastates expanded fewer nodes and only in 6 did it expand more. The only reason the planner without metastates might expand fewer nodes is because the expansion order of our search is not the same as unmodified WA*, and perturbing expansion order means that the control might happen to reach the goal first. The results for time taken to solve problems closely mirror those for nodes expanded, indicating the overheads of the bookkeeping required for metastates is minimal. In terms of coverage, using metastates we can solve 189 problem instances; without we solve only 164. A two-tailed Wilcoxon signed-rank test confirms statistical significance that using metastates out-performs the control, both in terms of nodes expanded and time taken to solve problems with $P > 0.99$.

Of our 14 domains we see a performance improvement in 9; while the performance in the other 5 remains stable. We can gain greater insights into this by examining the domains more closely. The numbers in brackets in Figure 3 show the percentage of states generated by the control planner that had open (currently executing) actions. Recall that it suffices to use weak equality in states without open actions; both planners exploit this. Therefore, if there are few states with open actions the potential of our technique to improve search is limited. This explains why we see relatively few gains in the Rovers and Turn And Open domains, but again, we also see no significant performance degradation, the number of states generated was identical to the control and both planners produced the same plan.

Perhaps surprisingly performance also remains the same on two of the domains with required concurrency, Match and P2P. In each of these domains states with open actions arise due to an envelope action (light match/serve file) inside which actions (mend fuse/download) must be fit. However, whilst the envelope action is open the heuristic easily guides search to perform the relevant activities and therefore very little search is required to solve these problems, and so potential gains are limited. Another domain with similar results is openstacks, as the benchmark set for this problem is trivial and requires very little to no search.

The remaining 9 domains where we see performance improvements are generally the more challenging domains and those in which the planner without metastates expands a large number of weakly equal states, when any one of these would allow it to reach a goal state. Many of these problems are solved an order of magnitude faster with metastates, thanks to the reduced size of the metastate search space. A final note on solution quality, the planners found plans with the same makespan in 154 of out of the 161 problems that were mutually solved, in the remaining 7 the solutions produced using metastates were slightly longer.

References

Coles, A. J., and Coles, A. I. 2016. Have I Been Here Before? State Memoisation in Temporal Planning. In *Proceedings of the Twenty Sixth International Conference on Automated Planning and Scheduling (ICAPS)*.

Coles, A.; Fox, M.; Halsey, K.; Long, D.; and Smith, A. 2009. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence* 173(1).

Cushing, W.; Kambhampati, S.; Mausam; and Weld, D. 2007. When is temporal planning *really* temporal planning? In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49.

Do, M. B., and Kambhampati, S. 2003. Sapa: Multi-objective Heuristic Metric Temporal Planner. *Journal of Artificial Intelligence Research (JAIR)* 20.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the Context-enhanced Additive Heuristic for Temporal and Numeric Planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Fox, M., and Long, D. 2003. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research (JAIR)* 20.

Hoffmann, J., and Edelkamp, S. 2005. The Deterministic Part of IPC-4: An Overview. *Journal of Artificial Intelligence Research (JAIR)* 24.

Huang, R.; Chen, Y.; and Zhang, W. 2009. An Optimal Temporally Expressive Planner: Initial Results and Application to P2P Network Optimization. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*.

Linares López, C.; Celorrio, S. J.; and Olaya, A. G. 2015. The deterministic part of the seventh international planning competition. *Artificial Intelligence* 223.

Long, D., and Fox, M. 2003. The 3rd International Planning Competition: Results and Analysis. *Journal of Artificial Intelligence Research (JAIR)* 20.

Rintanen, J. 2007. Complexity of concurrent temporal planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS)*, 280–287.