STAIRS 2014
U. Endriss and J. Leite (Eds.)

doi:10.3233/978-1-61499-421-3-41

Temporal Plan Quality Improvement and Repair using Local Search

Josef Bajada [1], Maria Fox and Derek Long
Department of Informatics, King's College London,
Strand, London WC2R 2LS, United Kingdom.
{josef.bajada, maria.fox, derek.long}@kcl.ac.uk

Abstract. This paper presents an approach to repair or improve the quality of plans which make use of temporal and numeric constructs. While current state-of-the-art temporal planners are biased towards minimising makespan, the focus of this approach is to maximise plan quality. Local search is used to explore the neighbourhood of an input seed plan and find valid plans of a better quality with respect to the specified cost function. Experiments show that this algorithm is effective to improve plans generated by other planners, or to perform plan repair when the problem definition changes during the execution of a plan.

Keywords. temporal planning, scheduling, optimisation, local search, plan repair

Introduction

Real world planning problems often need to take into account time and resources together with concurrency and exogenous events. PDDL 2.1 [1] and 2.2 [2] introduced the constructs necessary to model such problem domains, under the form of durative actions, numeric fluents and timed initial literals. Nevertheless, most of the state-of-the-art temporal planners struggle to cope with complex metric functions and concurrency. Most planners are biased to generate feasible plans that minimise plan length or makespan. However, in some problem domains it is preferable to generate plans that minimise a certain cost rather than plan duration. This is especially true for domains where plan execution is continual and the goal is to maintain some variables within specific bounds, or new goals are queued into the system during the plan's execution. One example is the demand-side electricity aggregator domain. In this case the system needs to find a plan, comprising of task-completing actions and load-shifting actions, within a planning horizon that features frequent electricity price fluctuations, with the objective of minimising wholesale electricity costs [3].

In this paper we present a domain-independent approach that generates high quality plans, in terms of some cost function. The proposed technique involves performing a local search on a provided input seed plan to explore its neighbourhood for better quality plans. This process can then be repeated until the time allocated for the algorithm has

[1]This research is funded by the UK Engineering and Physical Sciences Research Council (EPSRC) as part of the project entitled *The Autonomic Power System* (Grant Ref: EP/I031650/1)

elapsed. One can also utilise one of the existent temporal planners to generate a valid feasible plan, and use that as the input seed plan to find better quality plans. The proposed algorithm is also effective for plan repair, when exogenous events or changes in goals invalidate a plan.

1. Background

Local search is commonly used in various combinatorial problems, such as discrete optimisation, and it has also proved to be successful in classical planning. LPG [4] is one popular planner that uses this approach. An action graph that connects the initial state to the goal is found by adding or removing random actions from the problem's planning graph. The process is then repeated until the action graph becomes a solution graph, that is, until it has no flaws and the goal facts are present in the final state, making it a valid plan. While the original version of LPG caters only for propositional planning, it was later enhanced [5] to support some of the temporal constructs introduced in PDDL 2.1. Local search was also proposed as a solution for plan improvement in the context of classical planning [6]. In this case a neighbourhood graph of states is constructed from the states of the given seed plan, and the shortest plan that leads from the initial state to the goal is then extracted using Dijkstra's algorithm.

We propose to use local search to find plans in domains that not only require concurrent durative actions, but also have invariant conditions that are potentially mutually exclusive. Furthermore, our objective is to find plans of a high quality with respect to some metric. A problem's time-related constraints and characteristics, such as action durations and timed events, are used to build a planning time line. The respective plan violations at each time point are analysed and the respective metrics at each state are also calculated. This enables the algorithm to consider concurrent actions and also account for non-linear numeric effects. Most state-of-the-art temporal planners struggle with concurrent durative actions and non-linear numeric effects. While these planners are biased towards finding shorter plans, our approach can find plans of a better quality. Moreover, valid plans generated by these planners can be used as input seed plans for our algorithm, which will then search for a better plan in terms of some objective function. The input seed plan does not have to be valid, which also makes this algorithm useful for plan repair. If a plan becomes invalid due to changes in the environment, the new problem definition can be analysed in conjunction with the old plan to generate a new valid plan for the new version of the problem.

Local search algorithms comprise of two main components:

1. A **neighbourhood function**, which transforms an input state into a set of new but very similar states, with a very limited number of differences.
2. An **evaluation function**, which determines the states that are more desirable, according to some objective.

Using these two components new neighbours are generated, evaluated and explored according to the algorithm being used. Hill climbing algorithms incrementally choose neighbours that provide a better solution until no further improvements are possible. This carries the risk of getting stuck into a local minimum. Other flavours of local search algorithms include simulated annealing [7], which allows the exploration of inferior solutions with a certain probability, improving the chances of finding a global minimum.

2. Temporal Planning

Our notion of temporal planning follows the semantics of PDDL 2.1 [1], where actions have a duration, together with conditions and effects that are associated with the start, end or execution of the action. We also consider exogenous timed events, as defined in PDDL 2.2 [2]. The following definitions formulate the fundamental underpinnings of these semantics.

Definition 2.1. A temporal problem, $P = \langle A, I, L, G \rangle$, consists of a set of possible actions A, the initial state I, a set of timed initial literals L, and a goal condition G.

Definition 2.2. A temporal plan, $\pi = \{a_0, a_1, ..., a_k\}$, is defined as a list of durative actions. Each action a has a start time, denoted $start(a)$, and a duration, $dur(a)$.

The start time of a durative action is a positive rational number, indicating the time, after the start of the plan, when the action should commence. The duration is also a positive rational number. Multiple durative actions can be executed concurrently in a temporal plan.

Definition 2.3. A durative action, a, may have conditions that need to be satisfied just before it starts, referred to as $startCond(a)$, conditions that need to be satisfied just before it ends, referred to as $endCond(a)$, and invariant conditions that need to hold throughout the execution of the action, referred to as $inv(a)$.

Definition 2.4. A durative action, a, may have effects that are applied when the action starts, referred to as $startEff(a)$, and effects that are applied when the action ends, referred to as $endEff(a)$.

PDDL 2.1 [1] also defines continuous effects, to represent continuously changing values with respect to the time elapsed from the start of the action. This construct is not currently supported in the work presented here.

Timed initial literals (TILs) were introduced in PDDL 2.2 [2] to support predictable exogenous state-changing events that will occur at some predetermined time during the plan. A timed initial literal, l, has a time when it is predicted to take place, $time(l)$, and an associated proposition that will become *true* or *false*. This construct is useful to denote external changes in the environment, or time windows when certain activities can take place. For example, in the demand-side electricity aggregator domain, TILs are used to perform tariff switches. TILs can be seen as instantaneous actions without any preconditions that will take place at a predefined time.

Each durative action, a, can be translated into two *snap actions* [8], $a_{\vdash}$, corresponding to the start of the action, and $a_{\dashv}$, corresponding to the end of the action. Snap actions are essentially instantaneous actions where $pre(a_{\vdash}) = startCond(a)$, $pre(a_{\dashv}) = endCond(a)$, $eff(a_{\vdash}) = startEff(a)$ and $eff(a_{\dashv}) = endEff(a)$.

In order to avoid ambiguity in the application of action effects, a total order is enforced by introducing a minimal time separation ε between two successive actions, and only one snap action is allowed to be applied at a certain point in time. The sequence $E_\pi = \{e_1, e_2, ..., e_n\}$ corresponds to the state-changing activities (snap actions and TILs) of π, with $time(e_i)$ denoting the time when e_i will be executed.

Definition 2.5. A plan's state time-line $\Upsilon(\pi) = \{\langle 0, s_0\rangle, \langle t_1, s_1\rangle, \langle t_2, s_2\rangle, ..., \langle t_n, s_n\rangle\}$ is a sequence of pairs $\langle t, s\rangle$ where t is a rational number corresponding to the time from the start of the plan when the current state will become s. State s_0 is the initial state.

A period, p_i, denotes the time interval between two successive states on the time-line. These periods are used to identify possible insertion points where new actions could be added to the plan. The last period p_n is open-ended since it denotes the time interval that follows the final state s_n, thus allowing actions to be added to the end of the plan.

$$\forall \langle t_i, s_i\rangle \in \Upsilon(\pi), \text{ where } 0 \le i \le n : p_i = \begin{cases} \langle t_i, t_{i+1}\rangle, & \text{if } i < n \\ \langle t_i, \infty\rangle, & \text{if } i = n \end{cases} \tag{1}$$

The invariant conditions of a period p_i correspond to all the invariant conditions of the actions running concurrently throughout that period, as defined in Equation 2.

$$\begin{gathered} invp(p_i) = \bigcup_{a \in A_i} inv(a), \text{ where } 0 \le i < n \text{ and} \\ A_i = \{a | start(a) \le t_i < t_{i+1} \le start(a) + dur(a)\} \end{gathered} \tag{2}$$

3. The Neighbourhood of a Temporal Plan

We define a plan π' as the neighbour of a plan π, denoted $\pi' \in N(\pi)$, if π' can be obtained by either *adding* one applicable action at some point on the time-line, *removing* an existent action from the plan, or *moving* an action to start and end at a different time point on the plan's time-line. Each of these operations will naturally change the plan's state time-line and also the invariant conditions for each period.

3.1. Adding an Action

By adding a new durative action, a_{new}, to start within period p_i and end within period p_j, where $0 \le i \le j \le n$, two new states, s_s and s_e, will be added to the time-line, corresponding to the two snap actions of a_{new}. The state $s_s = startEff(a_{new})(s_i)$, reflects the start effects of a_{new} applied to state s_i. The state $s_e = endEff(a_{new})(s'_j)$, reflects the application of the end effects of the action, where s'_j is the new state obtained from applying all subsequent actions following s_s in sequence up till t_j. All the states s_x, where $i < x \le n$, following s_s on the time-line, need to be propagated and updated to s'_x, to account for the effects of the new action. All the states $\{s_0, ..., s_i\}$ will remain the same while the subsequent states will be updated. The new action will naturally run concurrently with any other actions scheduled to run during periods $\{p_i, ..., p_j\}$, making it possible to find solutions in cases where concurrency is required [9].

A durative action, a_{new}, is only eligible for addition to the plan's time-line at an arbitrary time point during period p_i if it satisfies the following compatibility criteria:

1. The state s_i satisfies the start conditions of a_{new}, that is $s_i \models startCond(a_{new})$.

2. The new state s_s satisfies all the invariant conditions during that period, including those of a_{new}, that is $s_s \models inv(a_{new}) \cup invp(p_i)$.
3. All the subsequent states of s_s up to and including s_e satisfy the invariant conditions of a_{new}, that is $\forall s \in \{s_s, s'_{i+1}, ..., s'_j, s_e\} : s \models inv(a_{new})$

This ensures that the new action is compliant with the plan. However, this does not mean that subsequent conditions or invariants associated with any periods p_j, where $j > i$, will not be violated by adding a_{new} at p_i. Nevertheless, invalid neighbouring plans can still lead to valid plans that are further away in the neighbourhood of the seed plan π. By considering a violated plan π', that can be repaired through further exploration, the algorithm improves its chances of escaping from local minima.

The set $\{p_j | i \leq j \wedge start(p_j) < end(p_i) + dur(a_{new}) \wedge end(p_j) > start(p_i) + dur(a_{new})\}$ (where $start(p)$ and $end(p)$ correspond to the start and end time-point of period p respectively) defines the possible candidate periods where a_{new} can end. Each possibility that also satisfies the above compatibility criteria can be used to obtain a valid neighbouring plan π'. The start time of a_{new} is then set to an arbitrary value that satisfies $max[start(p_i), start(p_j) - dur(a_{new})] < start(a_{new}) < min[end(p_i), end(p_j) - dur(a_{new})]$ and $\forall e \in E_\pi : start(a_{new}) \neq time(e) \neq start(a_{new}) + dur(a_{new})$.

3.2. Removing an Action

Any durative action, a_{del}, that is already in the plan, can be selected for removal. The two states, s_d and s_r, where $0 < d < r \leq n$, correspond to states obtained by applying the start and end effects of the action a_{del} respectively. By removing the action, these two states are removed from the plan's time-line, and the rest of the states that follow s_d are updated accordingly.

3.3. Moving an Action

Moving an action, a_{mov}, can be seen as a macro action that involves removing an action and adding it again at a different point on the time-line. The effect of this modification would be a change in the plan's total ordering of the actions rather than a change of the plan's set of actions. This is especially useful in domains where the order of the actions has an impact on the cost of the plan, or exogenous events change the action costs at specific time points. This move needs to satisfy the same compatibility criteria used for adding an action to be considered a valid operation.

Substituting an action with another one might intuitively also seem like a valid neighbourhood operation. However, the benefits of such an operation in a temporal context, where actions have different durations and action swapping can change the total ordering of state-changing events, need to be analysed further.

4. Evaluation of Neighbouring Temporal Plans

A temporal plan obtained through the neighbourhood function needs to be evaluated on two levels. Firstly, we need to determine how close the plan is to a valid solution. Secondly, we need to measure the cost of the plan, with respect to some cost function. In order not to get stuck in local minima, inferior plans to the current one are also evaluated and explored, with a certain probability. This is governed by a probability distribution that diminishes proportionally with the exploration *distance*, that is the number of nodes traversed from the best one. This means that immediate neighbours of the best plan have a higher probability of being accepted than ones further away in the neighbourhood graph. However, valid plans that have a better cost than the current best plan will always be accepted. This is intuitively similar to the approach used in simulated annealing [7], where a *temperature* value decreases with each iteration, and the probability of accepting a weaker solution is computed using a function of this *temperature*. However, in our case we reset the *distance* each time we restart searching again from the best plan, thus assigning a high acceptance probability to closer neighbours, irrespective of when they were discovered.

4.1. Computing a Plan's Validity

Let $\alpha_i = \{a | start(a) = t_i\}$ be the set of actions in a plan π starting at a time point t_i, and $\omega_i = \{a | start(a) + dur(a) = t_i\}$ be the set of actions ending at a time point t_i, where $0 < i \leq n$. Equation 3c defines the set of conditions that need to be satisfied at time point t_i, in terms of the start and end conditions of actions starting or ending at t_i, defined by Equations 3a and 3b respectively. $G(t_i)$ represents any goal conditions that need to be satisfied at t_i. Goals that are only required to be satisfied at the end of the plan and do not have any time constraints are included in the set $G(t_n)$.

$$startCondT(t_i) = \bigcup_{a \in \alpha_i} startCond(a) \tag{3a}$$

$$endCondT(t_i) = \bigcup_{a \in \omega_i} endCond(a) \tag{3b}$$

$$cond(t_i) = startCondT(t_i) \cup endCondT(t_i) \cup invp(p_i) \cup G(t_i) \tag{3c}$$

The conditions that are actually satisfied at t_i are those that are satisfied by the state s_{i-1}, defined as $\sigma(t_i) = \{c | c \in cond(t_i) \wedge s_{i-1} \models c\}$. Conversely, $\phi(t_i) = cond(t_i) \setminus \sigma(t_i)$ defines the set of conditions that are not satisfied at t_i. A plan is considered valid if $\forall t_i \in \{t_1, ..., t_n\} : \phi(t_i) = \emptyset$. If a plan is invalid, the number of violations $v(\pi)$ is calculated by accumulating all the unsatisfied conditions at a given time point t_i, excluding any conditions that were already unsatisfied at t_{i-1}. This helps to avoid inflating the violation count from a common condition that is required by more than one action or by the same action at more than one point on the time-line. This process is defined recursively through Equations 4a to 4d, where $0 < i \leq n$. $cond^+(t_i)$ represents the cumulative set of conditions that need to be satisfied at t_i together with any conditions that were not satisfied at t_{i-1}. Similarly, $\sigma^+(t_i)$ represents the cumulative conditions that are satisfied at t_i, including any conditions carried forward from previous states, and conversely $\phi^+(t_i)$ represents the cumulative set of conditions unsatisfied at t_i. We can then extract $\phi^*(t_i)$, the set of conditions that are introduced at t_i.

$$cond^+(t_i) = cond(t_i) \cup \phi^+(t_{i-1}) \tag{4a}$$

$$\sigma^+(t_i) = \{c | c \in cond^+(t_i) \wedge s_{i-1} \models c\} \tag{4b}$$

$$\phi^+(t_i) = cond^+(t_i) \setminus \sigma^+(t_i) \tag{4c}$$

$$\phi^*(t_i) = \phi(t_i) \setminus \phi^+(t_{i-1}) \tag{4d}$$

If a condition becomes satisfied at t_j, but becomes unsatisfied again at a later time t_k, (where $i < j < k$), and is needed by an action at or after t_k, it is counted twice. This is because at least two additional actions are needed to make the plan valid. Equation 5 defines the violation count $v(\pi)$ for a plan π. If $v(\pi) = 0$, the plan π is valid.

$$v(\pi) = \sum_{i=1}^{n} |\phi^*(t_i)| \tag{5}$$

4.2. Acceptance Probability Function

In order to promote the exploration of a broader neighbourhood we need a function that has a high probability of accepting plans that are close neighbours of the initial plan. On the other hand, in order to avoid exploring deep branches that do not lead to a solution, we need such a function to asymptotically decrease towards 0 in proportion to the exploration distance d, parametrised by the maximum distance m we want to explore. One candidate that fits these criteria is the sigmoid function defined in Equation 6.

$$g_m(d) = 1 - \left(\frac{1}{1 + e^{(m-d)}}\right) \tag{6}$$

We also want to increase the chances of accepting a good candidate at any exploration distance, depending on its *fitness* ratio with respect to the previous plan. This value is computed using the function $f(\pi', \pi)$, which compares two plans π' and π, as shown in Equation 7.

$$f(\pi', \pi) = \begin{cases} \dfrac{c(\pi') + 1}{c(\pi) + 1}, & \text{if } v(\pi) = v(\pi') = 0 \\[2ex] \dfrac{v(\pi') + 1}{v(\pi) + 1}, & \text{otherwise} \end{cases} \tag{7}$$

If both plans are valid, the actual cost function $c(\pi)$ is used, which is subject to the respective domain and problem instance. If one of the plans is invalid, the fitness ratio with respect to an adjacent plan is calculated using the number of violations in the two plans. In both cases 1 is added as a smoothing parameter. Equation 8 defines the acceptance probability function of exploring π' from its adjacent neighbouring plan π, with the current distance from the best plan π^* being d.

$$p_m(\pi', \pi, d) = g_m(d)^{f(\pi', \pi)} \tag{8}$$

5. Searching for Temporal Plans

The algorithm to search for a better temporal plan, starting from an initial seed plan π, involves exploring its neighbourhood until a better one is found. A plan π' is considered better than π, denoted by the relation $\prec_c$, if the plan π' has no violations, and either π is not a valid plan, or the cost of π is more than that of π'. Formally, $\pi' \prec_c \pi$ if $(v(\pi') = 0) \wedge (v(\pi) > 0 \vee c(\pi') < c(\pi))$. Given that the neighbourhood of a plan can be very large, a neighbour is generated randomly, with the search taking the form of a random walk guided by the acceptance probability function, p_m, as described in Algorithm 1. The set *visited* keeps track of the plans explored during one random walk to avoid cycles.

Algorithm 1 Local search for a better temporal plan

```
Require: Seed plan $\pi^*$, maximum distance $m$
 1: $\pi \leftarrow \pi^*$ ; $d \leftarrow 0$
 2: $visited \leftarrow \{\pi\}$
 3: while not($\pi \prec_c \pi^*$) and not($term$) do
 4:     if $d \geq m$ then
 5:         $\pi \leftarrow \pi^*$ ; $d \leftarrow 0$
 6:         $visited \leftarrow \{\pi\}$
 7:     end if
 8:     $\pi' \leftarrow$ select random plan from $N(\pi) \setminus visited$
 9:     if $(v(\pi') > 0)$ and $(v(\pi') < v(\pi^*))$ then
10:         $\pi^* \leftarrow \pi \leftarrow \pi'$ ; $d \leftarrow 0$
11:         $visited \leftarrow \{\pi\}$
12:     else if $\pi' \prec_c \pi$ then
13:         $\pi \leftarrow \pi'$ ; $d \leftarrow d + 1$
14:         $visited \leftarrow visited \cup \{\pi\}$
15:     else
16:         $r \leftarrow$ random double between 0 and 1
17:         if $r \leq p_m(\pi', \pi, d)$ then
18:             $\pi \leftarrow \pi'$ ; $d \leftarrow d + 1$
19:             $visited \leftarrow visited \cup \{\pi\}$
20:         end if
21:     end if
22: end while
23: return $\pi$
```

In its simplest form, this algorithm carries the risk of running indefinitely if no better plan is found, or if no valid solution actually exists. The terminating condition *term* determines whether the loop should stop or continue iterating, even if no better plan has been found. This terminating condition could depend on the total number of iterations, the time elapsed from the start of the search, or some other context-dependent condition. Once a better plan is found, this algorithm can be executed again using the new plan as the input seed plan π^*, in order to improve the plan quality further.

The initial seed plan can be an invalid one. This algorithm will compute the number of violations in the seed plan and try to find valid plans that do not have any violations. This makes it also suitable for plan repair, since it will exploit the plan structure of the initial plan to try to find a similar one that satisfies all the required conditions.

6. Preliminary Experiments

The proposed algorithm has been implemented within a temporal plan solver capable of parsing PDDL 2.2 domain and problem files, together with a seed plan for the problem. This solver performs a local search and outputs a new improved plan. If no seed plan is provided, it will try to find a feasible solution to the problem, which will then be used as the seed plan for subsequent iterations.

Some preliminary experiments have been performed with two temporal domains; the Transport domain, from the IPC 2008 competition, and the Aggregator domain, specifically designed to find solutions for managing demand-side electrical loads. Since the original version of the Transport domain was designed to minimise time, it has been slightly modified to also keep track of the total fuel used. The problem instances were also modified to make alternative cheaper or more expensive solutions also possible.

The aggregator domain represents a demand-side electricity aggregator that needs to schedule flexible load (such as dish-washing or EV charging), and also use storage devices to shift load to more preferable times of the day. Both the forecasted inflexible load and wholesale electricity prices fluctuate throughout the day and the goal is to find the best combination of actions that minimises the cost of inflexible and flexible load. The cost of the plan is calculated by adding the inflexible and flexible load components and multiplying the result with the energy cost at that time. This domain is particularly challenging for existent planners due to the fact that this cost depends on various variables that are changing over time rather than accumulating a monotonically increasing cost with each action.

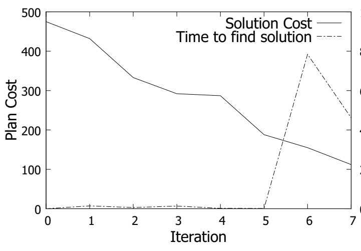
Table 1 shows an example of how an initial seed plan generated by the planner POPF [10] for the Aggregator domain was improved to minimise the costs by moving activities to cheaper periods and making use of batteries. Each line indicates a separate durative action in the plan. The number before the action indicates its start time, while the number in square brackets indicates its duration.

Table 1. Initial seed plan and improved plan for the Aggregator domain.

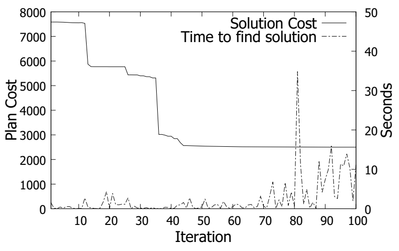
Seed Plan	`0.000: (start-metering) [1440.000]`
	`0.001: (perform wash-dishes-h3 wash-dishes-normal) [110.000]`
	`0.001: (perform wash-dishes-h2 wash-dishes-fast) [80.000]`
	`0.001: (perform wash-dishes-h1 wash-dishes-fast) [90.000]`

Improved Plan	`0.0: (start-metering) [1440.0]`
	`251.222: (charge battery-s1 charge-normal) [166.667]`
	`296.667: (perform wash-dishes-h2 wash-dishes-fast) [80.0]`
	`298.333: (charge battery-s2 charge-normal) [66.667]`
	`308.944: (perform wash-dishes-h3 wash-dishes-normal) [110.0]`
	`315.0: (perform wash-dishes-h1 wash-dishes-fast) [90.0]`
	`626.667: (discharge battery-s1 discharge-fast) [83.333]`
	`630.0: (discharge battery-s2 discharge-fast) [44.444]`

Figure 1a shows the plan improvements when running the proposed algorithm on a problem instance of the Transport domain with 5 cities, 2 trucks and 2 packages. Figure 1b shows the plan improvements for the Aggregator domain with 10 household tasks, 10 electricity storage units and 6 tariff switches over 24 hours. One should keep in mind that the potential plan improvement is naturally problem dependent and Figure 1 only demonstrates that for temporal numeric planning problems that have a broad solution space this approach is capable of improving plan quality.



(a) Transport domain



(b) Aggregator domain

Figure 1. Improvements in plan costs with respect to the number of iterations.

7. Conclusions and Future Work

We have presented an algorithm that improves the quality of plans for temporal planning problems with numeric properties. While current state-of-the-art temporal planners are very capable of finding feasible plans, the proposed algorithm is able to exploit the structure of such plans to find better solutions. Preliminary experiments have been performed using two temporal domains and it has been demonstrated that this algorithm can improve plan quality, albeit more effective in problems with a broad solution space.

Future work includes incorporating techniques that shorten the makespan of a plan when it does not have any impact on the plan cost and inter-period optimisation to find the best schedule of a sequence of actions. Support for additional PDDL 2.1 constructs such as continuous effects and duration inequalities is also being investigated, together with further experiments with other domains using temporal and numeric characteristics.

References

[1] M. Fox and D. Long, "PDDL2.1: An extension to PDDL for expressing temporal planning domains," *Journal of Artificial Intelligence Research*, vol. 20, pp. 61–124, 2003.

[2] S. Edelkamp and J. Hoffmann, "PDDL2.2: The language for the classical part of the 4th international planning competition," Tech. Rep. 195, 2004.

[3] J. Bajada, M. Fox, and D. Long, "Challenges in Temporal Planning for Aggregate Load Management of Household Electricity Demand," in *31st Workshop of the UK Planning & Scheduling Special Interest Group (PlanSIG)*, 2014.

[4] A. Gerevini, A. Saetti, and I. Serina, "Planning Through Stochastic Local Search and Temporal Action Graphs in LPG.," *Journal of Artificial Intelligence Research*, vol. 20, pp. 239–290, 2003.

[5] A. E. Gerevini, A. Saetti, and I. Serina, "Temporal Planning with Problems Requiring Concurrency through Action Graphs and Local Search," in *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS 2010)*, no. Icaps, pp. 226–229, 2010.

[6] H. Nakhost and M. Martin, "Action Elimination and Plan Neighborhood Graph Search : Two Algorithms for Plan Improvement," in *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pp. 121–128, 2010.

[7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing.," *Science (New York, N.Y.)*, vol. 220, pp. 671–80, May 1983.

[8] A. Coles, M. Fox, D. Long, and A. Smith, "Planning with Problems Requiring Temporal Coordination," in *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pp. 892–897, 2008.

[9] W. Cushing, S. Kambhampati, Mausam, and D. S. Weld, "When is Temporal Planning Really Temporal?," in *20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, 2007.

[10] A. Coles, M. Fox, and D. Long, "POPF2: a Forward-Chaining Partial Order Planner," *The 2011 International Planning Competition*, pp. 65–70, 2011.