Extracting Plans From Plans

David Pattison and **Derek Long**

Department of Computer and Information Science
University of Strathclyde, Glasgow G1 1XH, UK
{david.pattison, derek.long}@cis.strath.ac.uk

Abstract

Propositional *plans* of all forms often display a certain level of concurrency which can be exploited by *scheduling* the plan. This reveals the earliest times at which each action can be applied whilst still achieving the goal and shortening the plan duration. However, the output of this scheduling process is simply a set of timestamped actions, losing implicit information present in the original plan such as the causal links between actions and states.

In this paper we present PIMP (Plans Inside Multi-threaded Plans), a domain-independent algorithm which can schedule a plan whilst retaining the knowledge inherent in a traditional plan. We exploit this using the concept of *threads* to detect individual, concurrent and interleaved plans and discuss the benefits of these *thread-scheduled* plans and their possible applications.

1   Introduction

The field of Planning is concerned with the production of a series of ordered actions which transform an initial state into one which contains a set of required goal facts. These *plans* can often be *scheduled* to minimise the usage of certain resources and overall plan length by finding the earliest time at which each action is applicable. While the resulting set of discrete timestamped actions is useful, it loses information present in the original plan structure.

Consider a problem in which an astronaut and autonomous rover must collect rock and soil samples from two locations and analyse them at the lander. One plan would have both separate and take samples of each resource individually before returning to the lander, with the rover having a shorter plan length. Now consider that the domain has other resource locations which are of interest, but are not required to be sampled as part of the original plan and furthermore are unknown at the time of plan construction. Given a series of traditionally-scheduled discrete actions, detecting and exploiting *opportunities* like these is difficult. Yet it is clear to a human there are two separate plans being executed – one for the astronaut and one for the rover. If these *plan threads* are encoded and known to each of the executing agents (the astronaut and rover), then they can deduce their own *sub-state* of the overall state which in turn enables previously undetectable plan exploitation such as performing real-time plan modification. In the case of an autonomous agent such as the rover, this can allow the exploitation of further research opportunities, while adhering to the original plans time constraints. Furthermore, if the original plan requires *synchronisation* of threads, such that two threads must complete in order for the goal to be met or an action to become applicable, the agent will know which facts must hold at the synchronisation point, something which can be taken into account when amending a thread.

The production of a *thread graph* such as that in Figure 2, which contains all threads also has applications in autonomous assistance. For example, in the above problem the original plan may be for the astronaut to take both samples. When combined with a plan/goal recognition system (Kautz 1987), it becomes possible to detect which sample the astronaut is moving to first, at which point the rover can decide to execute the thread which achieves the second goal. Conversely, in an adversarial environment such as a real-time strategy (RTS) game the use of a recognition engine would allow the detection of possible weak points in an opponent's plan, such as breaking the conditions required at a synchronisation point.

In this paper we present the PIMP algorithm, which produces a series of plan *threads* from an unscheduled plan. Each thread is a subset of the original plan and allows for parallel, discrete execution of steps while retaining explicit links between actions in the same manner as the original plan. Threads are then stored in a graph structure which enables detection of thread synchronisation and splitting points. Later, we present possible applications of the algorithm, primarily in the context of video games but also discuss other applications which have similar traits.

The structure of this paper is as follows: we first begin by defining the various components of the problem and any assumptions made. The algorithm itself is then described in detail such that replicating results should be possible. We then offer various application areas where these *thread-scheduled* plans would be of benefit, before discussing our work in the context of previous related work. Finally, we present our conclusions and propose future extensions to the algorithm and research.

2 Problem Definition

In order to construct plan threads and an associated thread-graph, several assumptions are made about the working domain. We first assume we are working with a *propositional* domain (such as the STRIPS formalism (Fikes and Nilsson 1971)), and are given an unscheduled plan P which is comprised of a set of n totally-ordered actions $\langle a_1, a_2, a_3 \ldots a_n \rangle$. These actions may form a **complete** or **partial** plan, but the first action must be applicable in the current working state. This allows us to construct plans in both an incremental and post-hoc manner, thus allowing threads to be generated **during** plan construction or observation. P must be unscheduled and purely propositional as PIMP will perform its own scheduling which may disrupt any resource constraints present in a scheduled plan.

We also assume access to the planning problem $\Pi = \{F_R, O, A, I, G\}$ from which this plan has been created, where F_R is the set of all *reachable facts*, O is the set of *objects* which exist in the domain, A is the set of *grounded actions*, I is the set of facts true in the initial state and G is the set of goal facts which must be true in the final state S_n.

Each action $a \in A$ is a tuple $\{a_{pre}, a_{add}, a_{del}, a_{obj}\}$, where a_{pre}, a_{add} and a_{del} are sets of facts corresponding to the preconditions, add effects and delete effects of the action, with the union of a_{add} and a_{pre} denoted as a_{eff}. a_{obj} is the set of all *objects* which appear as parameters in a_{pre}, a_{add} and a_{del}.

Finally, we assume that the problem domain has no *undetectable* mutually-exclusive facts (mutexes). This ensures that there is no possibility of destructive interactions between actions being executed on separate threads at the same time. This final assumption can of course be relaxed if the system has access to a complete (possibly hand-made) set of known mutually-exclusive facts. We detect mutexes and reachable facts using Helmert's work in SAS+ (Helmert 2009) and to generate a *causal graph* from which we extract *controller objects*.

Definition 1. *Causal Graph – The **causal graph** (CG) is a single digraph (V, E), where each vertex $v \in V$ is equivalent to an object in the domain. A directed edge (u, v) exists if $u \neq v$ and there exists an action $a \in A$ such that $\exists eff \in a_{eff}$ has v in its parameters, and $\exists f \in \{a_{pre} \cup a_{del}\}$, for which f has u as a parameter.*

Definition 2. *Controller Objects – A node v in the causal graph is a **controller object** iff $|v_{out}| > 0$ and $|v_{in}| = 0$, where v_{out} and v_{in} correspond to the number of outgoing and incoming edges for v. Controller objects exist only to modify other objects without being directly affected themselves.*

The presence of controller objects is often an indication that the final graph will be split into parallel, non-overlapping threads, with each thread using a single controller object. We make use of *controller objects* during the thread generation process, but note that not all domains will exhibit these as a property.

Finally, as we are working with a propositional model we have no explicit concept of **time**. While it is true that actions are scheduled to begin at times of the form t or $t+n$

where $t, n \in \mathbb{Z}$, strict adherence to these timestamps by the executing system is left to its discretion.

3 Extracting Threads from Plans

In order to detect the threads present in a plan, we must reconstruct the plan from its initial state to produce a *thread graph* – a structure which contains traditionally-scheduled timestamped actions, but crucially also keeps track of the *links* between consecutive actions. As we are interested in detecting and exploiting parallel action sequences, we use the concept of *plan heads* to encapsulate the n current states which exist as the graph is constructed. For example, if we find that actions a_1 and a_5 can be applied in I and are non-mutex, two new plan heads would be formed as a result of application.

Each plan head $h \in H$, is a tuple $\{S_h, \mathcal{H}, I_h, F_U, O_P\}$, where S_h is the current state known to h and $S_h \subseteq S$, with S being the overall current state $S = \bigcup S_h, \forall h \in H$. $\mathcal{H}$ is a further tuple $\{S_{prev}, A_{prev}, L_{prev}\}$, corresponding to lists representing the previous *states*, *actions* and *action links* encountered in the life of h. I_h corresponds to the state this plan head began in and F_U is a set of *unused facts* which have been added to S_h since I_h and not deleted or used as a precondition to any action applied to h. O_P is the union of all parameters of all actions applied in h, such that $O_P = \bigcup a_{obj}, \forall a \in A_{prev}$.

Both F_U and O_P are used to determine the *link* that exists between the previous state of h. The detection of these links is a harder problem than simply scheduling the plan, as it requires the ability to link possibly non-consecutive actions into a linear sub-plan.

3.1 Graph Construction

At the beginning of the graph construction process (see Algorithm 1), only one plan head will exist containing the initial state I. The rest of the graph will be populated by applying actions to this head's state, with the resulting new state or states becoming the new plan heads. If the plan contains duplicate actions $\langle a_1 \ldots a_n \rangle$, we enforce a constraint which prevents a_{t+1} from being applied before a_t.

Once the initial head has been created, the algorithm loops until all actions in the plan have all been inserted into the graph. If an action is applicable in one of the active plan head states, it is mapped to this and a new head is formed from its application at the end of the loop, with the previous iterations heads being discarded.

However, it is insufficient to simply link an action to the first plan head state in which it is applicable as this will inevitably lead to an inconsistent plan thread. Consider a domain in which two trucks are to deliver two packages to waypoint C. Trucks 1 and 2 start at waypoint A and B and their respective packages are already present at these locations. We are then presented with a plan in which each truck is loaded with its package and driven to location C.

If we construct a thread-graph in which each action is linked to the first plan head which meets its preconditions, we end up with a graph similar to Figure 1, as the system does not recognise that (`drive_truck truck2 wpb wpc driver2`) would be better applied in head 3. We
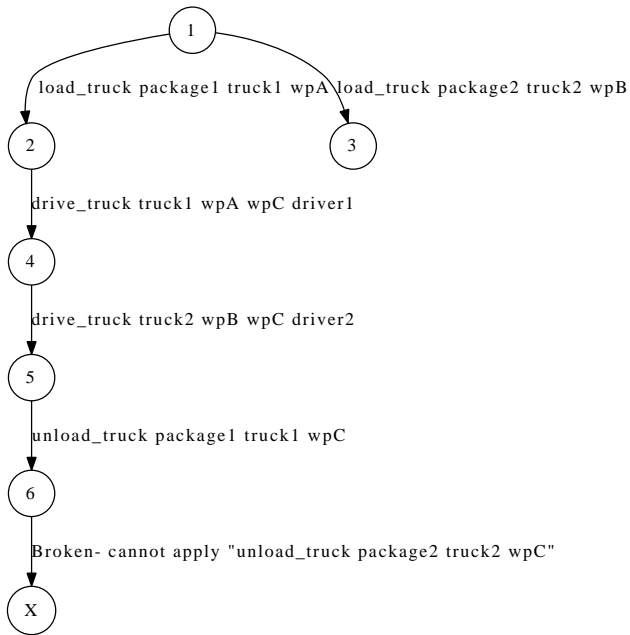
Figure 1: The result of applying actions to heads without any link inference.

solve the problem of actions being applicable in multiple heads through *link inference* – analysing the history of a plan head for a link with applicable actions.

Definition 3. *Links*

1. **Causal Link** – *A causal link exists between a plan head h and applicable action a iff $\exists f \in a_{pre}$ and $f \in F_U$. That is to say, if any of the actions applied within the lifetime of h added a fact which has not appeared as a precondition or delete effect to any succeeding action, then a causal link exists between the head and applicable action.*

2. **Controller Link** – *Given a domain which exhibits controller objects O_C, a controller link exists between a plan head h and applicable action a iff $\exists o \in O_C$ and $o \in O_P$. As controller objects tend to control entire threads from plan initialisation to completion, it is sufficient to infer that a link exists between a plan head and action if the controller object has been use previous to the currently applicable action.*

3. **Object Link** – *Object links are weaker forms of controller links which do not require a controller object to be present in O_P. Instead, a link is present iff $\exists o \in O_P$, $o \in a_{obj}$ and $o \notin O_C$. The number of object links between a plan head and n applicable actions is used to determine the precedence of the action.*

We prioritise links in the order $\langle$*causal, controller, object*$\rangle$, but also recognise there may be situations where no link exists between heads (as would be the case for all actions applicable in the head associated with I).

If we now apply link-inference to the previous example the output will be Figure 2, which correctly recognises the individual threads are connected by the appropriate links.
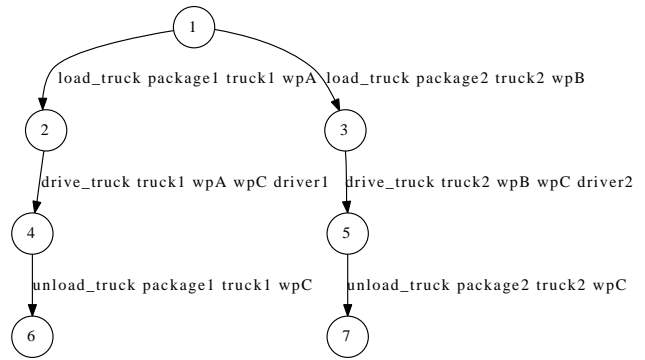

Figure 2: The correct thread-graph produced with link inference.

These links also act as a tie-breaker when an action is applicable in more than one head.

3.2 Mutex Detection

Naturally, it is unacceptable to simply allow every applicable action in a head to be inserted into the graph. To obtain a non-mutex set of applicable actions for each head, we filter actions based on type of mutex or their position in the original plan (see Algorithm 2);

Beyond the traditional GRAPHPLAN style mutexes (Blum and Furst 1995), we also detect a further "pause" mutex, in which the action both adds and deletes the same effect at the same time. These actions act as a semaphore on certain objects and must be detected in order to prevent otherwise non-mutex actions from being applied at the same timestep (regardless of whether they are being applied in the same head). An example of these can be seen in Figure 3, wherein communicate_soil_data cannot be applied in state 9, because communicate_rock_data both adds and deletes the effect (channel_free lander) which is also a precondition of communicate_soil_data. The former is applied in state 9 because it appears first in the original plan.

We also introduce a further test for mutex actions to cover the following situation. Consider another DRIVERLOG problem which has a plan head where the following 4 ordered actions are applicable: (unload p1 t l1), (load p2 t l1), (drive t l1 l2) and (disembark d t l1), where *p1* and *p2* are packages, *t* is a truck, *d* is a driver and *l1* and *l2* are locations.

PIMP first checks for mutexes between unload and the other actions, and discovers that drive deletes one of its preconditions. drive is then added to a *delayed* list and the algorithm progresses to load, which has no mutex actions as drive has been removed from the list of actions to be considered. This leaves only disembark which, being the last action checked, naturally has no actions to be mutex with, and so is added to the applicable list.

While this may see reasonable, by allowing disembark to be applied at timestep t we have prevented the algorithm from completing successfully, because the effects of disembark block drive from ever being applicable. To resolve this rare situation, we introduce a further test on all

delayed actions which determines if the action currently being checked for mutexes a_{ti} deletes any of the preconditions of a delayed action. If this delayed action also precedes a_{ti} in the original plan we also add a_{ti} to the *delayed* set, as it is conceivable that its application could prevent preceding actions from being applicable in any head.

3.3 Dead Head Propagation

Until now we have only considered plans which contain threads ending at the same time. However, it is often the case that one plan head will terminate long before others, thus becoming a *dead* head. That is, none of the remaining unscheduled actions can be applied to the current state.

This presents a dilemma if a longer thread which started at the same time as another shorter thread later relies on a subset of facts achieved by the shorter thread. For instance, in Figure 4 one of the threads terminates at sub-state 10 containing facts required in state 11, which prevents further thread-graph generation. We resolve this by *propagating* the facts from the dead head through all live heads which are connected by a common root in the graph (see Algorithm 2). This common root is the first node found by regressing back up through the graph from the dead head which also has a live head as a child node – other unconnected live heads are not affected.

These connected live heads have their plan history $\mathcal{H}$ updated to reflect the actions executed in the dead thread, and also have their *unused facts* F_U and *used objects* O_P updated. Due to the live head now containing all aspects of the dead head we must restart the iteration without assigning any previously applicable actions, as the addition of new facts may allow other actions to become applicable.

As a special case, if **all** heads are found to be dead but there are still unscheduled actions remaining, we must merge at least two dead heads into one live head. For instance, Figure 3 shows a plan for a modified ROVERS domain in which rovers must link-up together to provide enough power for communication to occur. Both rovers acquire samples for their respective goals then move to `waypoint4` where they link together. However, in order for the `synchronise` action to be applicable, both threads must be merged into a single state, because they individually do not contain the literal needed. This union of heads is performed by considering all possible combinations of currently live heads and choosing that which has the fewest heads. This guarantees that the next unscheduled action is applicable and execution can continue.

Once the thread-graph has been constructed fully it becomes possible to detect the earliest and latest possible start times of each action by performing a breadth-first search from the root node. Individual threads are then extracted by moving back through the graph from both live **and** dead heads. A thread terminates when it reaches the root node, or one of its actions requires a merge between two other threads. The set of timestamped threads can then be passed onto a dispatcher for execution at the appropriate time.
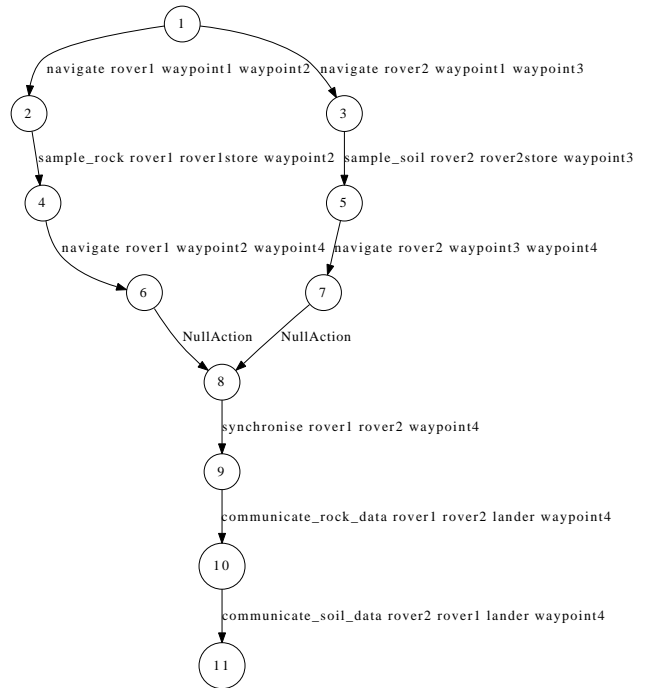

Figure 3: Synchronisation of two dead threads in order to allow further actions to be applicable.

4 Applications

We now present several possible situations where integration of the PIMP algorithm into a standard Planning system would be beneficial.

4.1 Opportunities

Once the original plan has been split into threads, each thread can be executed individually within its allocated time bounds. In a scenario where two threads must meet at a synchronisation point in order for a third thread to start, it is often the case that one of the threads is shorter than another. Given that each thread is aware of the facts which will be true in its final state and at each step in its execution, it becomes possible to insert other plan steps at any point during the thread. For example, consider the following example from the SETTLERS domain in the 3rd International Planning Competition (Long and Fox 2003). We are given a plan in which two units of stone and wood must be moved from locations A and B respectively to location C by two separate vehicles, at which point an ironworks will be constructed. The threads produced by PIMP will correspond to the actions for *cart1* and *cart2*, as seen in Figure 4. The final action is applied to the first thread, as both resources will be present due to dead-head propagation from the shorter thread.

If we assume that the synchronisation point has a strict start time, *cart2* is left with two timesteps to spare. If the executing system can look for *opportunities* it can recognise that this would allow for a cabin to be built at location E, thus increasing overall resources whilst still adhering to the synchronisation deadline.

Algorithm 1 Thread-Graph Generation

Require: plan steps $\langle a_1, a_2, ...a_n \rangle$
  $heads = \{\}$ {set of current plan heads}
  $dead = \{\}$ {set of heads which are of no further use}
  $heads \leftarrow I$ {add default plan head using I}
  $plan \leftarrow$ plan steps
  while $!empty(plan)$ **do**
    {Dictionary $(action \Rightarrow head)$}
    $applicable = getApplicableActions(heads, plan)$
    if $|applicable| == 0$ **then**
      {find the minimum set of heads that must be merged
      in order to apply the next unscheduled action}
      $next = plan.peek()$
      $minHeads = minimumUnion(heads, next)$
      $heads = mergeHeads(minimalHeads)$
      continue
    end if
    {Dictionary $(action \Rightarrow head)$}
    $nonMutex = getNonMutexActions(applicable)$
    $newDead = heads \setminus nonMutex.values$
    if $|newDead| > 0$ **then**
      $heads = propagateHeads(dead, nonMutex)$
      $dead \leftarrow newDead$
      continue
    end if
    $heads = \{\}$
    $applied = \{\}$
    for all action/head mapping $m \in nonMutex$ **do**
      {apply action to head to create new head}
      $newHead = apply(m.key, m.value)$
      $heads \leftarrow newHead$
      $applied \leftarrow m.key$
    end for
    $plan.removeAll(applied)$
  end while

Algorithm 2 Mutex Filtering

Require: $unfiltered$ - mapping of applicable actions to
  heads sorted on action number
  $filtered = \{\}$ {map of non-mutex actions to plan heads}
  $delayed = \{\}$ {set of actions which will not be applied}
  $paused = \{\}$ {subset of $delayed$ for paused actions}
  for all $\langle action, head \rangle$ $a \in unfiltered$ **do**
    if $a \in delayed$ **then**
      continue
    end if
    for all $actions$ $d \in delayed$ **do**
      {Check mutex and if d precedes a in original plan}
      if $(mutexType(a, d) == AdeleteBpc) \wedge (d_t < a_t)$
      then
        $delayed \leftarrow a$
        continue
      end if
    end for
    {Only consider actions which are after a in plan}
    for all $\langle action, head \rangle$ $b \in tail(a, unfiltered)$ **do**
      if $b \in delayed$ **then**
        break
      end if
      $mutexType = getMutex(a, b)$
      if $mutexType == ApauseB$ **then**
        $paused \leftarrow b$
        $delayed \leftarrow b$
        break
      end if
      {if a precedes b, delay a, otherwise delay b}
      if $a_t < b_t$ **then**
        $delayed \leftarrow b$
      else
        $delayed \leftarrow a$
      end if
    end for
    $filtered \leftarrow a$
  end for

4.2 Plan Robustness

Given a thread-graph representation of a plan and the realisation that one of the threads executing has failed in some way, it is possible to *replan* only the failing portion of the thread-graph. This prevents the need to produce another full plan, something that is often costly and may produce a sub-optimal version of the previous plan.

As we know the start and end states of the broken thread T_B, we can produce a minimal subset of the original problem domain in which the initial state is the state prior to the action which has failed a_f and the goal is either the overall goal achieved by the thread; the final thread state; or the facts required by the successor thread enabled by T_B. The range of objects available to the replanner is simply the inclusive union of those used by the actions succeeding a_f. Should this subset of the original domain fail to produce a plan, we can change the initial state to reflect the overall world-state at the failure point, and the range of available objects to be the original domain set.

4.3 Assistive Execution

The advantages of a thread-based plan execution architecture apply particularly to situations which integrate *autonomous* agents with a *human* agent. In a traditional real-time strategy (RTS) game scenario featuring unit/base construction and resource gathering, each player will have their own plan as to how to achieve their goal which they would normally endeavour to execute manually. However while these are an essential part of the overall plan, these tasks are often trivial and can distract the player from other issues, such as the problem of mining resources while preparing for a large-scale onslaught. If the game features an autonomous *lieutenant*, these tasks can be transparently passed onto it. This can be performed by the lieutenant being made explicitly aware of the user's plan or by performing *goal recognition* on the user's current actions. For instance, if the lieutenant determines that the user is carrying out thread *x*, it can choose to execute parallel thread *y* without bothering the

Algorithm 3 Dead Head Propagation

Require: deadHeads, liveHeads
$newHeads = \{\}$
for all $d \in deadHeads$ **do**
  $common = findCommonRoot(d, liveHeads)$
  for all $c \in common$ **do**
    $newHead = mergeHeads(d, common)$
    $newHeads \leftarrow newHead$
  end for
end for
return $newHeads$

user, thus streamlining the plan execution while increasing the perception of intelligence.

4.4 Adversarial Planning

The previous application details how to apply PIMP in a co-operative, assistive environment, but the same principles hold for *adversarial* agents too. If we again take a standard RTS game with a human player but assume that the opponent can form its own plans (such as a human or planner-equipped *bot*), then it becomes possible to **recognise** and **prevent** the execution of their plan. For example, if the goal recognition system suggests that the opponent is trying to destroy the player's vehicle factory, it is possible to infer their possible plan which can then be converted into a *thread-graph*. It then becomes trivial to detect the weakpoints in their plan execution and prevent the appropriate steps being achieved. Of course, this can also be applied in the context of the computer attempting to prevent the player's plan execution.

5 Related Work

To the best of our knowledge there has been no prior work in constructing explicitly linked sub-plans from a fully-formed **or** incremental plan. At first glance it may seem that previous work in *scheduling* (Smith, Frank, and Jónsson 2000) may be of use, however as we have no numeric or temporal resources to share amongst threads it becomes simpler to assume STRIPS plans which only require that mutually-exclusive facts be known.

Perhaps the closest analogy would be that of Hierarchical Task Network (HTN) planning (Nau, Ghallab, and Traverso 2004), in which plans are computed from high level tasks which decompose into other tasks and primitive actions. Indeed, the work of Wissing (2007) uses an HTN planner to create parallel plans during the execution phase. However, they do not provide a guarantee of non-overlapping plans[1], instead requiring the HTN architect to take this into account during design.

The construction of sub-plans **during** the planning process has existed in some form or another for several years. The authors of GRAPHPLAN (Blum and Furst 1995) recognised that such a system would be able to optimally detect the minimal set of threads required to achieve a goal-set, but did not export these individual sub-plans at the

---
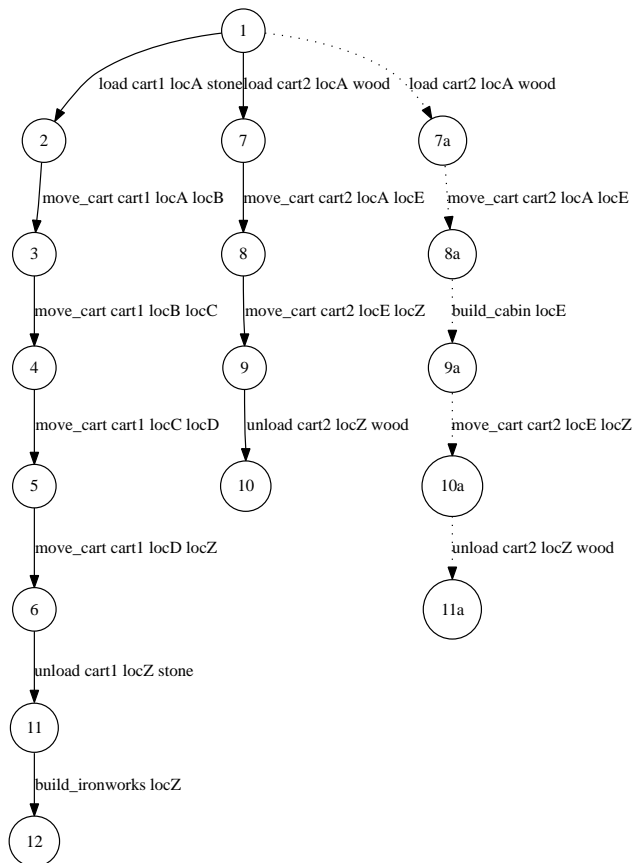[1]This is analogous to *threads* in the context of this work.

Figure 4: The thread graph for the SETTLERS example. The altered thread for *cart2* is shown with dotted edges.

end of search. Furthermore, the reality of generating a full *plan graph* until stabilisation is beyond the scope of most non-trivial problems due to the state-space explosion problem. However, this principle of *goal decomposition* during plan construction has been seen to be successful in other *forward-chaining* planners such as SGPLAN (Chen, Wah, and Hsu 2006) and those using the ADHG heuristic (Coles et al. 2008). Yet, the output of these planners remains a single list of actions, with all parallel knowledge discarded.

Elsewhere, *partial-order planning* (POP) has also seen a history of detecting and exploiting parallelism in plans. Knoblock (1994) identifies the class of problems where concurrency is possible in POP, while our notion of being able to perform concurrent execution without an explicit representation of time is reflected in the work of Boutilier and Brafman (2001). Their modifications of UCPOP algorithm (Penberthy and Weld 1992) allow multi-agent planning problems to be solved in a concurrent manner similar to that of a single-agent by modifying the STRIPS representation of actions to include a set of facts which must not hold in parallel with the chosen action. They also introduce new flaw resolution techniques to guarantee against destructive-interactions between concurrent actions.

This work is also related to earlier research

exploring the problem of lifting structure out of plans (Veloso, Perez, and Carbonell 1990; Bäckström 1998). Here the problem investigated was to lift partial ordered plans out of sequential plans with a minimal set of ordering constraints. The problem of finding a global minimum set of constraints is known to be hard (Bäckström 1998), but good heuristic approaches can achieve very good results (Veloso, Perez, and Carbonell 1990). However, this differs from the work reported here in two important respects. Firstly, the extraction of minimally ordered structures within plans is a different problem from finding the threads of activity that we consider here (although there is clearly a close relationship) and, secondly, the previous work considers the problem starting with the entire plan. In contrast, in this work we extract structure incrementally. This is an extremely important difference, since it allows the current work to be used in an on-line context to analyse plans as they unfold, step by step, and to identify the structure within them before they are complete.

6 Conclusion and Future Work

We have presented the PIMP algorithm, which enables the production of plan threads for parallel execution from an unscheduled plan featuring concurrent actions. We have detailed the algorithm itself and put forward several applications of planning which would benefit from knowledge of threading.

The next step in the evolution of the algorithm is to adapt the thread-graph produced to explicitly link dead heads with the actions or threads they enable. For instance, in Figure 4 there would be an explicit arc from state 10 to 11, thus simplifying the process of detecting thread synchronisation points which can then be used by other systems such as adversarial agents.

Introduction of an explicit and accurate representation of time and numbers would allow aspects of PDDL such as *timed initial literals* (Edelkamp and Hoffmann 2004) to be encoded directly into the thread-graph and respective threads and also aid in plan modification. Integration of PIMP with a large multi-threaded application such as an RTS game will provide a concrete example of the technology and enable rapid integration of the extensions detailed in the previous section.

Finally, it may be interesting to take the principle of *action links* presented in this paper and apply it to action selection during plan construction. In particular, POP has in the past shown to be a viable avenue for such links, with *causal links* being used to refine action selection (Penberthy and Weld 1992; Younes and Simmons 2003). Given the advancements in planning, the integration of the link detection strategies presented in this paper may be beneficial to the flaw resolution process.

References

[Bäckström 1998] Bäckström, C. 1998. Computational aspects of reordering plans. *J. AI Res.* 9:99–137.

[Blum and Furst 1995] Blum, A. L., and Furst, M. L. 1995. Fast planning through planning graph analysis. *Artificial Intelligence* 90:1636–1642.

[Boutilier and Brafman 2001] Boutilier, C., and Brafman, R. I. 2001. Partial-order planning with concurrent interacting actions. *Journal of Artificial Intelligence Research* 14:105–136.

[Chen, Wah, and Hsu 2006] Chen, Y.; Wah, B. W.; and Hsu, C.-W. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *J. AI Res.* 26:323–369.

[Coles et al. 2008] Coles, A.; Fox, M.; Long, D.; and Smith, A. 2008. Additive-disjunctive heuristics for optimal planning. In *Proc. Int. Conf. on Automated Planning and Scheduling*, 44–51.

[Edelkamp and Hoffmann 2004] Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the Classical part of the 4th International Planning Competition. Technical Report 195.

[Fikes and Nilsson 1971] Fikes, R., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. In *Proc. Int. Joint Conf. on AI*, 608–620.

[Helmert 2009] Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence* 173(5-6):503–535.

[Kautz 1987] Kautz, H. A. 1987. *A formal theory of plan recognition*. Ph.D. Dissertation, University of Rochester.

[Knoblock 1994] Knoblock, C. A. 1994. Generating parallel execution plans with a partial-order planner. In *In Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, 98–103.

[Long and Fox 2003] Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *J. AI Res.* 20:1–59.

[Nau, Ghallab, and Traverso 2004] Nau, D.; Ghallab, M.; and Traverso, P. 2004. *Automated Planning: Theory & Practice*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

[Penberthy and Weld 1992] Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A sound, complete, partial order planner for ADL. 103–114. Morgan Kaufmann.

[Smith, Frank, and Jónsson 2000] Smith, D. E.; Frank, J.; and Jónsson, A. K. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review* 15:2000.

[Veloso, Perez, and Carbonell 1990] Veloso, M. M.; Perez, M. A.; and Carbonell, J. G. 1990. Nonlinear planning with parallel resource allocation. In *In Proceedings of the DARPA Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 207–212. Morgan Kaufmann.

[Wissing 2007] Wissing, G. 2007. Multi-agent planning using HTN and GOAP. Master's thesis, LuleåUniversity of Technology.

[Younes and Simmons 2003] Younes, H. L. S., and Simmons, R. G. 2003. VHPOP: Versatile heuristic partial order planner. *J. AI Res.* 20:405.