CPPlanner: A Temporal Planning System using Critical Paths

Tien Ba Dinh and Barbara Smith
University of Huddersfield, UK
t.b.dinh@hud.ac.uk, b.m.smith@hud.ac.uk

Introduction

The last few years have seen a lot of significant improvements in AI planning. One of the most well-known advances is the Graphplan approach (Blum & Furst 1997), which has been the basis of many planners, such as GP-CSP (Do & Kambhampati 2001a), TGP (Smith & Weld 1999), TPSYS (Garrido, Fox, & Long 2002), STAN (Fox & Long 2001), LPGP (Fox & Long 2002)... In TGP and TPSYS, it has been extended to deal with temporal planning domains to find the optimal solution.

In this paper, we describe our planning system, CPPlanner, which is based on the Graphplan approach to find the optimal solution in terms of time (*makespan*) for temporal planning domains. Unlike TGP and TPSYS, CPPlanner deals with richer temporal planning domains.

In TGP, actions are assumed to have effects only at the end of the execution and the conditions hold throughout the execution. In TPSYS, PDDL 2.1 (Fox & D.Long 2001a) is used and actions can have effects at the beginning and at the end of the execution. In CPPlanner, we have extended the action representation to deal with effects which can happen anywhere during the execution. With this extension, the mutex relations are built up as constraints in the graph expansion phase and they are re-checked in the solution extraction.

The solution extraction phase uses "critical paths" to prune irrelevant branches in the search tree earlier. At first, it will get all the paths from the initial propositions leading to the goal propositions at the examining timepoint. It considers them as the "critical path" candidates. Then, each time, the planner will choose one of the candidates to be the critical path for the solution extraction phase. Because all the actions of the critical path are chosen before actually performing the search, other actions and propositions are quickly eliminated if they are mutex.

Action representation

With the ambition of solving real world problems, our action representation is mainly influenced by the PDDL+ (Fox & D.Long 2001b) and the representation of the Sapa planner (Do & Kambhampati 2001b). Unlike TGP and TPSYS, it has been extended to deal with actions that can have effects at any time during the execution. Each action A has a duration Dur_A, a starting time Start_A, and an ending time End_A (= Start_A + Dur_A). The duration Dur_A can be statically defined for a domain, or dynamically calculated at the execution of the action. For example, the action *Fly* has duration calculated based on the distance between the two places at execution time. Action A has a set of conditions Cond_A, which consists of a conjunction of tuples $\langle \text{Cond}_{A_i}, \text{d}_{A_i} \rangle$ ($\langle$condition, duration$\rangle$). d_{A_i} for a certain Cond_{A_i} means that the Cond_{A_i} needs to be true from the starting time of the action A until (Start_A + d_{A_i}). Similarly, action A has a set of effects called Eff_A, which consists of tuples $\langle \text{Eff}_{A_j}, \text{d}_{A_j} \rangle$ ($\langle$effect, time$\rangle$). d_{A_j} for a certain Eff_{A_j} means that the Eff_{A_j} happens at the time (Start_A + d_{A_j}).
Therefore, an action A is presented as $\{\text{Dur}_A, \text{Cond}_A, \text{Eff}_A\}$ in which:

- Dur_A: the duration of the action A. ($\text{Dur}_A > 0$).
- Cond_A = $\{\langle \text{Cond}_{A_1}, \text{d}_{A_1} \rangle, ..., \langle \text{Cond}_{A_k}, \text{d}_{A_k} \rangle\}$ with $\forall i \in [1, k] : 0 \leq d_{A_i} \leq Dur_A$.
- Eff_A = $\{\langle \text{Eff}_{A_1}, \text{d}_{A_1} \rangle, ..., \langle \text{Eff}_{A_h}, \text{d}_{A_h} \rangle\}$ with $\forall i \in [1, h] : 0 \leq d_{A_i} \leq Dur_A$.

Graph expansion

In AI planning, given the initial state, the goal state and a set of actions, a solution of the problem is a sequence of actions which takes the initial state to the goal state, i.e. a plan. In CPPlanner, we assume that the actions in the solution must complete and the *makespan* is the ending time of the latest action in the plan even if the goal state has been achieved as a result of effects taking place before the end of the action. With this assumption, the expansion phase is much more complicated than if plans can rely on uncompleted actions. In the expansion, the planner has to wait for the action to finish before taking its effects as conditions for other actions. Furthermore, when an action is applied, its starting time need not be the ending time of another action, but the maximum of the *timestamps* of its conditions.

Starting from the first stage of the graph, in which propositions are in the initial state with *timestamps* 0, the graph is advanced in time step by step. The main idea is that at an *examining timepoint t*, we will advance the graph to

the next timepoint by choosing the next action, say *A*, which has the earliest ending time of all possible actions. After choosing *A*, all of its effects are added to the graph with their corresponding *timestamps* and the *examining timepoint* *t* is advanced to *t* + Dur_A. This process is repeated until all the goals appear. At this time, the solution extraction is tried to look for a solution. If a solution extraction fails, the process will be performed again to move to the next possible *timepoint*. Otherwise, the algorithm is terminated.

In the expansion phase of TGP, since an action only has effects at the end of its execution, the examining timepoint *t*, at which at least one action completes, is also the time when new effects appear (the effects of the completed actions). Therefore, when all possible actions at the examining timepoint *t* are applied, *t* is used as the starting time of these possible actions. However, in CPPlanner, the starting times of the possible actions are more complex, because the effects of actions can happen anywhere during the execution. We cannot use the examining time *t* as the starting time for the possible actions, because some of these actions may happen earlier than *t*. Instead, the starting time for each possible action is calculated by the maximum of the *timestamps* of its conditions. This leads to the fact that propositions will be stored in the list *Props* with new *timestamps* if they appear again.

For example, in TGP, it is very simple that if we are applying a possible action *A* at the examining time *t*, the starting time of *A* will be *t* and the effects will be attached with the *timestamp* = *t* + Dur_A. However, in CPPlanner, because actions can have effects at any time during the execution, an action *X* may take some intermediate effects of action *Y* as its conditions and start while action *Y* is still under way. Note that at the examining time *t*, *Y* has finishes already. So when we apply the action *X*, if *X* has several conditions, the starting time of *X* will be calculated as $\max\{timestamp_{Cond_X}\}$, not the examining time t. Note that this value may less than *t*. The *timestamps* of *X*'s effects are calculated based on this starting time.

Mutex relations

In TGP, actions are assumed to have effects only at the end and preconditions must be hold through out the execution of actions. In TPSYS, actions have effects only at the beginning or at the end of their execution. As a consequence of this, the mutex relations introduced by TGP or TPSYS are calculated once in the expansion phase to know whether proposition-proposition, proposition-action, or action-action are mutex. These relations can be used again in the solution extraction phase. For example, the mutex relation of action *A* and action *B* is calculated as true. It means that they cannot overlap with each other in the result plan regardless the time that these actions start. Therefore, in the solution extraction phase, when choosing actions and propositions for the plan, if the mutex checking is required, the planner just looks into the mutex relation that has been calculated in the expansion phase earlier. However, in

// Put all initial props attached with
// timestamp 0 into the Queue
PropsQueue = $\{< p_1, 0 >, < p_2, 0 >, ..., < p_n, 0 >\}$
// Set the examining timepoint 0
t = 0
Loop
  while *PropsQueue* $\neq \emptyset$
    CurProp = the first prop in the *PropsQueue*
    Remove the first proposition from *PropsQueue*
    Create mutex relations for *CurProp* with
    other *Props* and *Actions*
    PossibleActions={all actions having *CurProp* as one
    of their conditions, and the others from Props}
    Apply *PossibleActions*
    TmpProps ← {Attach their effects with *timestamps*}
    // Add *CurProp* to the Graph
    Props ← *CurProp*;
  end {while}
  // move to the next possible timepoint where at
  // least one action completes
  t = timestamp of the next ending effect in the *TmpProps*
  // add the completed actions to the Graph
  Actions ← {new completed actions after moving
  to the next possible timepoint}
  PropsQueue ← all props in *TmpProps*
  belonging to new completed actions at *t*
  Remove these propositions in *TmpProps*
  If goals $\subseteq$ {*Props* $\cup$ *PropsQueue* }and pairwise
  nonmutex, do solution extraction.
  If (solution extraction succeeds)
    terminate the algorithm.
End {loop}

Table 1: Graph expansion algorithm

our planner, because actions may have effects during their execution, the mutex relations depend on the time when actions start or propositions become true. Therefore, in the solution extraction phase, when actions and propositions are assigned different *timestamps* from the expansion phase, the mutex relations have to be re-checked to know whether they are mutex.

The mutex relations of our planner are described as follows:

- **Proposition - proposition:** propositions p and q are mutex if (1) they are negations or (2) all actions supporting q are mutex with p and vice versa.
- **Action - proposition:** action A and proposition p are mutex if one of these holds
  - (p and q are mutex) $\wedge$ (q $\in$ Cond$_A$) $\wedge$ (p is true at t with t $\in$ [Start$_A$, Start$_A$+ d$_{A_q}$]).
  - (p and q are mutex) $\wedge$ (q $\in$ Eff$_A$) $\wedge$ (p is true at t with t $\in$ [Start$_A$ + d$_{A_q}$, End$_A$]).
- **Action - action:** action A, B are mutex if one of these holds
  - (p and q are mutex) $\wedge$ (p $\in$ Cond$_A$) $\wedge$ (q $\in$ Cond$_B$) $\wedge$ (Start$_A$ + d$_{A_p}$ $\geq$ Start$_B$)
  - (p and q are mutex) $\wedge$ (p $\in$ Cond$_A$) $\wedge$ (q $\in$ Eff$_B$) $\wedge$ (d$_{A_p}$ $\geq$ d$_{B_q}$)
  - (p and q are mutex) $\wedge$ (p $\in$ Eff$_A$)$\wedge$ (q $\in$ Eff$_B$) $\wedge$ (End$_B$ $\geq$ Start$_A$ + d$_{A_p}$)

We store the mutex relation as constraints between nodes in the graph. There are two types of mutex. One is the static mutex which we can know after reading the planning domain. It is always mutex regardless of the time, e.g. negations of propositions. The other is the dynamic one which depends on the time of propositions or actions. Therefore, when checking whether this proposition or action is mutex with another one, we will know immediately if it is static mutex. Otherwise, we have to check the mutex constraint between them at the examining time.

Solution extraction

When we have expanded the planning graph to an examining timepoint t_G where all the goals appear, we try to find an executable plan which will achieve all the goals at time t_G - this is the solution extraction phase. In the planning graph, a proposition-action path is a path with sequence of proposition and action nodes. Each action node in the path has the previous proposition node as one of its conditions and the next proposition node as one of its effects. The path starts with a proposition node and also ends with a proposition node. In the expansion phase, when the graph is expanded to time t_G, there is at least one proposition-action path which starts from the propositions in the initial state to the propositions attached with the timestamp t_G. We trace the graph to get all of these paths and consider them as "critical path" candidates. When we do the solution extraction, we choose a path from these candidates and use it as the critical path of our search. All the actions along this path will be chosen before we actually do the solution extraction search. With the chosen actions, other actions or propositions will be quickly eliminated in the search later on if they are mutex.

```
// Attach the goal propositions with timestamp t_G
Goals = {< p_1, t_G >, < p_2, t_G >, ..., < p_n, t_G >}
Check if Goals is a subset of initial state,
      stop the algorithm and print the solution.
Set t = t_G.
Candidates = proposition-action paths leading to t_G
while Candidates ≠ ∅
   CriticalPath ← get one from Candidates
   Delete it from Candidates
   ActionPlan = all actions of the critical path
                with their timestamps
   Remove propositions supported by the critical path
   from Goals
   Add all conditions of actions in the critical path to
   Goals with their timestamps
   // Note: excluding the conditions are effects of other
   // actions in the critical path
   While can choose (NextAction = one of the actions
   which supports p such that p∈Goals, and doesn't
   mutex with ActionPlan and Goals).
      Slide NextAction as late as possible, but its ending
      time not exceeding t.
      Add NextAction into the ActionPlan.
      Delete its effects in the Goals.
      Add its conditions to the Goals with timestamps.
      If the timestamp of any propositions < 0,
      then fail and try another NextAction
      Checking if Goals is a subset of initial state,
      print the solution and terminate the algorithm.
      Update the time t = ending time of the chosen action
   End {while}
End {while}
If cannot find a solution, graph expansion phase is
run again by the Loop
```

Table 2: Solution extraction algorithm

In this algorithm, we call t_G the examining timepoint where we are looking for a solution. Firstly, to expand to t_G, the graph expansion phase has known actions finished at the time t_G. Based on the graph, we trace back to get all proposition-action paths leading to t_G (because perhaps several actions finish at t_G). Note that we only trace proposition-action paths of sub-goals which have the timestamp t_G, because there are other subgoals which have the timestamp less than t_G that we failed to find a solution earlier. We consider these proposition-action paths as critical-path candidates. CPPlanner then takes one by one from the candidates to look for a solution. With the critical path, some actions will be added to the *ActionPlan* before the backtracking search for solution extraction is actually performed. It will help to prune irrelevant branches in the

search tree earlier by stopping other propositions or actions which are mutex with *ActionPlan* from being selected.

The backtracking process of the solution extraction is also different from the one used by TGP and TPSYS. In TGP or TPSYS, the goals are dequeued one by one following a predefined order. For each goal, each of the possible supporting actions is tried. The new subgoals which are preconditions of the action will be added to the list. This work continues until a solution is found or all possible cases have been explored. With this approach, in order to check all possible cases, the algorithm has to try all possible actions as well as their possible start times. It will lead to some redundant search because they cannot use any bound for the next step (next proposition) in searching. Besides, with this searching process, it is very difficult to apply other techniques like conflict-directed backjumping to improve the performance. In CPPlanner, we are using the examining time t to act as a bound for the next choice of action in order to avoid some symmetry in searching for solutions. It means that chosen actions have a chronological order in their ending times. The earlier it is chosen in the searching process, the later ending time it has. With this backtracking process, it can be easily developed to apply the conflict directed backjumping which is described in more detail in the next section.

Discussion and further development

The performance of Graphplan-based planning systems depends much on the solution extraction phase. In order to prune more irrelevant branches in the search tree, we will apply conflict-directed backjumping (Prosser 1993) (Kambhampati 2000). In our solution extraction, because we find actions one by one supporting the current sub-goal propositions, we can store the conflict list for the k^{th} action, called as level k. For example, we are looking for the k^{th} action in the extraction phase. We try action a_i and see whether it is mutex with any previous chosen propositions and actions, if we find out that it is mutex with another action a_x that we have chosen in an earlier level, we will store the level of a_x in the conflict-list for the level k. Then, we will try another action a_j. Also, if this action a_j is mutex with another action, say a_y, we add its level to the conflict-list. At this time, supposing that all the actions have been tried, we will sort the conflict list and backtrack to the previous level appearing next in the list. This will help us to jump directly back to the source of the conflict instead of wasting time looking around.

Conclusion

We have described our Graphplan-based temporal planning system, CPPlanner, to find an optimal solution for temporal planning domains. Unlike other Graphplan-based planning systems, it has richer action representation in which the effects can happen anywhere. The solution extraction phase uses the critical paths provided by the expansion phase to prune irrelevant search branches earlier. The searching process can be easily extended with the conflict-directed backjumping in the future. We have implemented and tested it on some temporal planning domains. However, because of the richer action representation, the mutex constraints are checked again in the solution extraction. Therefore, CPPlanner has poorer performance than TGP and TPSYS. We are going to improve the algorithm so that the solution extract phase can re-use some mutex relations between actions which only have effects at the beginning or at the end of their execution. The performance of the planner will be improved because of not wasting time re-checking the mutex relations for all actions in the solution extraction.

References

Blum, A. L., and Furst, M. L. 1997. Fast planning through Planning Graph Analysis. *Artificial Intelligence* 90:281–300.

Do, M. B., and Kambhampati, S. 2001a. Planning as Constraint Satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence* 132:151–182.

Do, M. B., and Kambhampati, S. 2001b. Sapa: A Domain-Independent Heuristic Metric Temporal Planner. In *In Proceedings of European Conference on Planning*.

Fox, M., and D.Long. 2001a. PDDL2.1: An extension to PDDL for expressing temporal planning domains . In *Technical Report, Dept of Computer Science, University of Durham*.

Fox, M., and D.Long. 2001b. PDDL+: An extension to PDDL2.1 for modelling planning domains with continuous time-dependent effects. In *Technical Report, Dept of Computer Science, University of Durham.*

Fox, M., and Long, D. 2001. Hybrid STAN: Identifying and Managing Combinatorial Optimisation Sub-problems in Planning. In *In Proceedings of IJCAI*.

Fox, M., and Long, D. 2002. *Fast Temporal Planning in a Graphplan Framework*. www.dur.ac.uk/computer.science/research/stanstuff/planpage.html.

Garrido, A.; Fox, M.; and Long, D. 2002. Temporal Planning with PDDL2.1. In *In Proceeding of ECAI'02*.

Kambhampati, S. 2000. Planning Graph as a (dynamic) CSP: Exploiting EBL, DDB, and other CSP search techniques in Graphplan. *Artificial Intelligence Research* 12:1–34.

Prosser, P. 1993. Domain filtering can degrade intelligent backtracking search. In *In Proceedings of IJCAI*.

Smith, D., and Weld, D. 1999. Temporal Planning with Mutual Exclusion Reasoning. In *In Proceedings of IJCAI*.