Towards Search Control via Dependency Graphs in Europa2

Sara Bernardini
London Knowledge Lab
University of London
23-29 Emerald Street, London WC1N 3QS
s.bernardini@lkl.ac.uk

David E. Smith
NASA Ames Research Center
Moffet Field, CA 94035–1000
david.smith@nasa.gov

Abstract

We develop domain-independent search control for NASA's Europa2 planning system based on the construction of Dependency Graphs, which succinctly represent the dependencies between the activities of a domain. This approach can be generalized to other temporal planners whose performance also relies on careful engineering of the domains and/or hand-coded domain-dependent search control information.

Introduction

Europa2 (Frank and Jónsson 2003) is a framework for solving planning and scheduling problems within the constraint-based temporal planning paradigm, which is characterized by an explicit notion of time, the use of state variables, and an emphasis on temporal constraint networks. It has been the core planning technology for several NASA missions including MAPGEN, the ground-based daily activity planner for the Mars Exploration Rover mission (Bresina et al. 2005). Although this planner has been successful in solving real-world complex problems, it is necessary to carefully tune the domain models and provide hand crafted control rules in order to obtain satisfactory performance. The process of engineering models and control information is generally time consuming, error-prone and can lead to models that are not very robust. It would therefore be useful to have powerful domain-independent heuristics for this planner.

The problem of relying on domain-dependent knowledge is not specific to Europa2, but is common to a number of large temporal planners developed for coping with real applications, such as ASPEN (Chien et al. 2000) and IxTET (Ghallab and Laruelle 1994). Since these planners share some significant features with Europa2, an investigation of domain-independent heuristics for Europa2 can be beneficial to them as well.

The *Activity Transition Graph* (ATG) technique presented in (Bernardini and Smith 2008a; 2007) represents a first step towards the introduction of automatically generated heuristic estimators for Europa2. This technique builds one transition graph for each state variable in the domain and then uses these graphs to compute distance estimates for choosing and resolving flaws within the Europa2 plan refinement mechanism. Although this method can provide good cost estimates of completing partial plans, the construction of ATGs relies on a rather strong assumption on the structure of the domains. The model must describe the full temporal unfolding of each state variable by specifying precedence relationships between its values. In particular, for each value v of a state variable $\mathcal{V}$, the model must specify what values of $\mathcal{V}$ can directly precede and follow v. If this assumption is not satisfied, the ATG for this variable will be disconnected, and cannot be used to extract distance estimates. Two types of domains are particularly unsuitable for ATG calculations: (a) Domains that look more like *scheduling* problems than planning problems, since they present many temporal and resource constraints between different state variables and few causal constraints involving single state variables; (b) *Hierarchical* domains, since their behavior is primarily specified via constraints between values of state variables that lay at different levels of abstraction. Unfortunately, many Europa2 domains used for practical applications (NASA 2009) fall in one of these two categories, and so ATG heuristics cannot be used on them.

In addition, the ATG technique as implemented in (Bernardini and Smith 2008a; 2007) does not produce accurate estimates on domains that present complex dependencies between different state variables. This is because it ignores such interactions and focuses instead on the evolution of the single state variables. In these domains, it is necessary to move from a local to a global strategy, able to reason on all the variables at once.

In this paper, we introduce a new technique for automatically generating heuristic guidance for Europa2 that overcomes these problems. Before illustrating the new technique, we give a brief overview of Europa2. Then, we describe the *Dependency Graphs* (DG) heuristic and conclude by discussing a preliminary evaluation and related work.

Overview of Europa2

Modelling Language: NDDL

In the modelling language NDDL, a *planning instance* I is a pair: $\mathrm{I} = (\mathcal{D}, \mathcal{P})$, where $\mathcal{D}$ is the *planning domain* and $\mathcal{P}$ is the *planning problem*.

A ***planning domain*** $\mathcal{D}$ is represented by:

- A set of ***timelines***: $\mathrm{T} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_n\}$, which are multi-valued state-variables capturing the evolution of a component or quantity over time.

- A set of mutually exclusive ***activities*** associated with each timeline $\mathcal{T}_i$: $\mathcal{A}ct[\mathcal{T}_i] = \{a_1(\vec{x}_1, \delta_1), \ldots, a_n(\vec{x}_n, \delta_n)\}$, where $\vec{x}_j$ is the vector of the parameters of the activity a_j and $\delta_j = [\delta_j^{min}, \delta_j^{max}]$ is a mathematical interval in $\mathbb{N}$ representing the *duration* of a_j.
- An ***evolution rule*** $\mathcal{R}[a]$ for each activity $a \in \mathcal{D}$ describing the temporal behavior of a. The rule $\mathcal{R}[a]$ is a conjunction of *compatibilities*: $\mathcal{R}[a] = \mathcal{K}_1[a] \wedge \ldots \wedge \mathcal{K}_n[a]$. A ***compatibility*** $\mathcal{K}[a]$ describes the relationship between the *master* activity $a(\vec{x}, \delta)$ and a number of *slave* activities $\mathcal{S}l[a, \mathcal{K}] = \{a_1(\vec{y}_1, \delta_1), \ldots, a_k(\vec{y}_k, \delta_k)\}$ (also called a's *subgoals* [1]). The slaves can be associated with the same timeline of a or with different timelines. Formally, a compatibility assumes the following form:

$$\mathcal{K}[a] : \bigwedge_{i=0}^{n}(g_i \Rightarrow \bigwedge_{j:a_j \in \mathcal{S}l[a,\mathcal{K}]}(t_j \wedge \bigwedge_{k=0}^{m} p_{jk}))$$

  which is composed of:
  - ***Guard constraints***: a constraint g_i is an atomic formula $\gamma = c_i$, where γ is a fixed variable called *guard* and c_i is a constant. It specifies when each conjunct of $\mathcal{K}[a]$ applies. Different values of the guard γ correspond to alternative temporal behaviors of a. If there is only one possible behavior for a, $\mathcal{K}[a]$ has only one conjunct and the guard constraint is empty. In this case, the compatibility is called *simple*, otherwise *disjunctive*.
  - ***Temporal constraints***: a set of constraints t_j specifying the temporal relations between the master activity a and its slaves. Given a slave a_j, a temporal constraint t_j looks like: `temp_rel` a_j, meaning that a and a_j should satisfy the relation `temp_rel` , which is an interval-based temporal relation in the following set: `meets`, `contains`, `before`, `starts`, `equals`, `parallels`, `starts_before_end`, `starts_during`, `starts_before`, `starts_after`, `contains_start` and all their inverses. These relations are derived from the thirteen temporal relations defined by (Allen 1983).
  - ***Codesignation constraints***: a set of constraints p_{jk} specifying restrictions over the possible instantiations of the parameters of the activities participating in a temporal constraint. Given the master $a(\vec{x}, \delta)$ and a slave $a_j(\vec{y}_j, \delta_j)$, a codesignation constraint p_{jk} is of the form x_m `rel` y_{jl}, where $x_m \in \vec{x}$, $y_{jl} \in \vec{y}_j$ and `rel` $\in \{=, \neq\}$. We assume that $p_{j0} = \top$.

  A ***planning problem*** $\mathcal{P}$ is a pair $\mathcal{P} = \{\mathrm{H}, \mathcal{I}\}$:
- $\mathrm{H} \in \mathbb{N}$ is the end of the ***planning horizon***. In Europa2 time is discretized into integers and the behavior of the system is planned over the temporal window $[0, \mathrm{H}]$.
- $\mathbb{I}$ is the ***initial configuration*** represented by a set of activities placed on their corresponding timelines. For each activity a_i in $\mathbb{I}$, it is possible either to specify the exact position of a_i on the timeline or to leave a_i floating on the timeline between the origin and the horizon. Note that $\mathcal{I}$ includes both the initial state and the goal state as they are defined in classical planning, and also it may contains intervals at other positions on the timelines.

For Europa2, the planning problem is to completely populate all the timelines with activities so that there are no gaps and all evolution rules are respected.

Example: *Crew Planning* domain.
As an illustration of an NDDL model organized hierarchically, consider a simplified version of the problem of planning the daily routine of the crew of the International Space Station (NASA 2009). Each member of the crew has the following associated timelines: *FastingWindow*, *CrewMember* and *CrewPlanner*. The timeline *FastingWindow* simply represents whether the crew member is fasting or eating and so has two associated activities, *Fasting()* and *NotFasting()*. The timeline *CrewMember* represents the activities that a crew member can perform, which are the following: *Sleep(480)*, *PreSleep(120)*, *PostSleep(180)*, *Meal(60)*, *Exercise(60)*, *PayloadActivity(id,60,120)*, *PowerActivity(id,60,120)* and *ChangeFilter(60)*. The timeline *CrewPlanner* represents the daily routines of one crew member. It simply contains a sequence of activities *DailyRoutine(1440)* over the entire planning horizon. A daily routine prescribes that every day a crew member must at least sleep, eat, exercise and execute the "post sleep" activities. These constraints concerning the activity *DailyRoutine(1440)* are expressed via the following evolution rule:

$\mathcal{R}$[*DailyRoutine()*] =(`meets` *DailyRoutine()* $\wedge$
`contains` *CrewMember.Exercise()* $\wedge$
`contains` *CrewMember.Meal()* $\wedge$
`contains` *CrewMember.Sleep()* $\wedge$
`contains` *CrewMember.PostSleep()*)

Similar temporal/resource constraints regulate the behaviors of the other activities.

Note that this domain is hierarchical in nature: *DailyRoutine()* represents the top level of the hierarchy while the activities on *CrewMember* represent lower level details. Along the same line, we can observe that the evolution rule of *DailyRoutine()* shown above is similar to an HTN method.

Search Algorithm

Europa2's planning algorithm is an instance of *plan refinement search*; given a domain $\mathcal{D}$ and a problem $\mathcal{P}$, the algorithm starts from the initial configuration $\mathcal{I}$ and incrementally refines it by adding and ordering activities to the timelines and binding variables until a final consistent configuration is found. However, while standard POCL planning proceeds strictly backward from the goal to the initial state on the basis of the causal information carried by the operator specification, the search algorithm in Europa2 works *bidirectionally*. This is because: 1) compatibilities specify general temporal relations between activities, and 2) activities in the initial configuration can take place at any time point of the planning horizon.

A ***partial plan*** Π consists of three elements:

[1] The notion of *subgoal* in Europa2 is different from the one in POCL. In Europa2 a subgoal can precede, overlap or follow its master, whereas in POCL a subgoal always precedes its master.

- For each timeline $\mathcal{T} \in \mathcal{D}$, a set of activities $\mathcal{A}ct_{\Pi} = \{t_1, t_2, \ldots, t_n\}$, which are not necessarily contiguous over time (actions in POCL).
- A ***temporal network*** $\mathcal{N}_{\Pi}$ representing all the start and end times of the activities in the plan and the constraints between them (ordering constraints in POCL).
- A set of ***flaws*** $\mathcal{F}_{\Pi} = \{f_1, f_2, \ldots, f_m\}$, where a flaw is an indication of a potential inconsistency in the partial plan. There are three types of flaws:
  - ***Open condition flaws***: They arise when applicable compatibilities are applied, triggering activities as slaves of masters that are already in the plan Π. Those slaves, called *free activities*, are enforced to be in the plan, but they are not yet associated with any timeline.
  - ***Ordering flaws***: They arise anytime an activity is placed on a timeline and an ordering is required for the activity with respect to the other activities already on that timeline.
  - ***Unbound variable flaws***: They arise when variables that have not yet been instantiated appear in the plan Π. There are two kinds of unbound variables: parameters of activities that are already in the plan and guards of applicable temporal constraints.

Refining a partial plan means to pick a flaw and resolve it. The process terminates when the set of flaws is empty. Each kind of flaw is resolved in a different way.

- *Resolvers for open condition flaws*

  Flaws for free activities can be resolved in two ways:
  - ***Merging*** A free activity is merged with a matching activity already in the plan. The operation of merging does not result in the addition of any new flaws to the current plan.
  - ***Activation*** We introduce a new activity a in the current plan associating it with the proper timeline, but without choosing a specific time slot for it. The compatibilities associated with a are applied and the slaves generated by those compatibilities are introduced as free activities. This results in both an ordering flaw, corresponding to the just activated activity, and a number of open condition flaws, corresponding to the added slaves.
- *Resolvers for ordering flaws*

  After the activation of an activity a, a temporal slot for placing a over its corresponding timeline is chosen among the slots that are compatible with its temporal constraints and maintain the temporal network consistency.
- *Resolver for unbound variable flaws*

  Unbound variable flaws are resolved by specifying a value in the domain of the variable. If the variable is a guard, the binding causes the introduction in the current plan of the slave activities associated with the chosen value.

The basic algorithm in Europa2 is a *depth-first search* characterized by *flaw selection*, *flaw resolution* and *constraint propagation* steps. Flaw selection identifies which flaw to resolve next, whereas flaw resolution deals with resolving a flaw by subsequently trying all the resolution options. Operations of plan refinement are interleaved with *constraint propagation* on the temporal network underlying the current partial plan to check its consistency. Note that this process does not always proceed strictly backward from the goal or forward from the initial state, but instead it may proceed bidirectionally.

Search Control for Europa2

In order to effectively speed up search in Europa2, we need a method of assessing the impact of each possible flaw resolution on the cost of completing a partial plan. To do that, we ground the domain model, parse the resulting grounded domain and store the information acquired during parsing in weighted graphs, called ***Dependency Graphs***. Such graphs are then used by the search component for attributing a cost to each resolver r_i of any flaw f. The resolver's cost is an estimate of the distance between the partial plan obtained by applying r_i and the final plan.

Building Dependency Graphs

A DG represents the *dependency relationships* between the activities of a domain, regardless of which timeline they belong to. More specifically, a dependency graph is an AND/OR graph whose nodes are partitioned into three subsets: activity nodes, which correspond to the activities in the domain, AND nodes and OR nodes, which together with the graph transitions represent the structure of the evolution rules of the domain. In a DG, each activity a is connected with all its subgoals $a_1 \ldots a_n$ – directly or via a path of non-activity nodes – and the transitions between a and $a_1 \ldots a_n$ are labelled with the corresponding temporal relationships. The cost assigned to a node a might be a duration or a resource usage/consumption depending on the type of the domain and problem under consideration.

More formally, given a domain $\mathcal{D}$, a ***Dependency Graph*** for $\mathcal{D}$ is a *directed graph* $\mathcal{G}^{\mathrm{D}}[\mathcal{D}] = \{\mathrm{V}, \mathrm{E}, \mathcal{C}_{\mathrm{v}}\}$. The set V is a set of vertices and is partitioned into three sub-sets: V^{Act} (*activity* vertices), V^{And} (*AND* vertices), and V^{Or} (*OR* vertices). The set $\mathrm{E} \subseteq \mathrm{V} \times \mathrm{V}$ is a set of directed edges. The function $\mathcal{C}_{\mathrm{v}}$ is a weight function that assigns a cost $\mathcal{C}_{\mathrm{v}}(v)$ to each activity vertex $v \in \mathrm{V}^{Act}$ in the graph. The graph $\mathcal{G}^{\mathrm{D}}[\mathcal{D}]$ is developed as follows (see Figure 1):

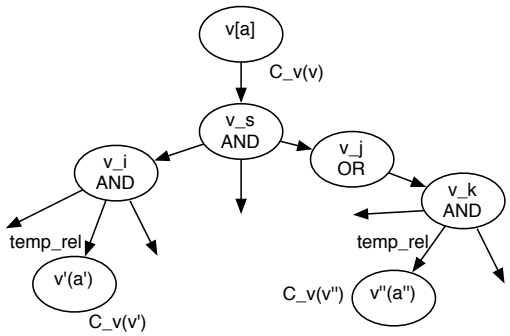

Figure 1: Building Dependency Graphs

1. We create an activity node $v \in \mathrm{V}^{Act}$ for each grounded activity a that belongs to $\mathcal{D}$ and label it with the cost $\mathcal{C}_{\mathsf{v}}(v)$ We indicate such a node as $v(a)$.
2. For each activity node $v \in \mathrm{V}^{Act}$, we create an AND node $v_s \in \mathrm{V}^{And}$ and add a directed edge from v to v_s.
3. For each simple compatibility $\mathcal{K}_i[a]$, we create an AND node $v_i \in \mathrm{V}^{And}$ and add a directed edge from v_s to v_i. For each temporal constraints $t_j = a$ `temp_rel` a' in $\mathcal{K}_i[a]$, we add a directed edge from the AND node v_i to the node $v(a')$ and annotate the edge with the temporal relation `temp_rel` between a and a'.
4. For each disjunctive compatibility $\mathcal{K}_j[a]$ controlled by a guard variable γ, we create an OR node $v_j \in \mathrm{V}^{Or}$ and add a directed edge from the AND node v_s to v_j. For each value c_k of γ, we create an AND node $v_k \in \mathrm{V}^{And}$ and add a directed edge from v_j to v_k. Finally, for each temporal constraint $t_i = a$ `temp_rel` a'' controlled by the value c_k, we add a directed edge from the AND node v_k to the node $v(a'')$ and annotate the edge with the temporal relation `temp_rel` between a and a''.

In summary, given an activity a, we associate with it an AND node that represents the conjunction between all its compatibilities. Then, we connect the AND node with all its subgoals. Disjunctive subgoals are grouped via OR nodes. In contrast to ATGs, we do not develop one dependency graph for each timeline, but just one dependency graph for the whole domain. When we handle domains with many activities and constraints, the graph may become very big and so difficult to manage. In such cases, we might want to develop the dependency graph by representing only activities that are reachable from the initial state and ignoring unreachable ones.

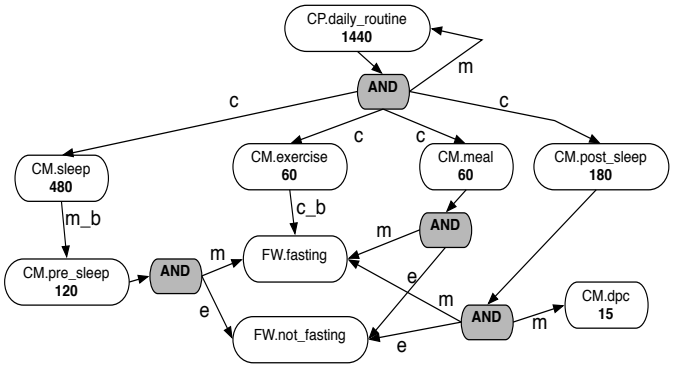

Figure 2: Fragment of DG for the *Crew Planning* domain

Exploring Dependency Graphs

Consider a domain $\mathcal{D}$ and a partial plan Π with an open condition or ordering flaw f, corresponding to a free activity a. Suppose that f has a possible resolution r. We define the cost of the resolution $\mathcal{C}_{res}(r)$ by using information extracted from $\mathcal{G}^{\mathrm{D}}[\mathcal{D}]$. In particular, we visit $\mathcal{G}^{\mathrm{D}}[\mathcal{D}]$ starting from the node corresponding to a and collect all the activities that are reachable from a and are not already in the plan in a set, which we call the *Dependency Set*, $\mathcal{DS}[a]$. These activities can be seen as a *relaxed plan* for a since they need to be introduced in the plan whenever a is in the plan. We assign to each activity in $\mathcal{DS}[a]$ a cost that depends on the current partial plan and the resolution r under consideration. By summing up these costs, we obtain the overall cost of the set $\mathcal{DS}[a]$, which gives us an estimate of the difficulty – in terms of subgoals to be achieved – of resolving f by applying r.

We now describe in detail how to obtain a dependency set for a. Given a dependency graph $\mathcal{G}^{\mathrm{D}} = \langle \mathrm{V}, \mathrm{E} \rangle$ and the activity node $v(a) \in \mathrm{V}^{Act}$, we extract from $\mathcal{G}^{\mathrm{D}}$ a subgraph $\mathcal{G}' = \langle \mathrm{V}', \mathrm{E}' \rangle$ such that $v \in \mathrm{V}'$ and $\mathcal{G}'$ is a *closed* graph. An AND/OR graph $\mathcal{G} = \langle \mathrm{V}, \mathrm{E} \rangle$ is *closed* if and only if:

- For each node $u \in \mathrm{V}$, if u is an AND node, then all its successors $\mathrm{S}(u)$ are in V.
- For each node $u \in \mathrm{V}$, if u is an OR node, then at least one of its successors in $\mathrm{S}(u)$ is in V. There are various strategies to choose u's successor, as we explain below.

The graph $\mathcal{G}'$ is called ***Candidate Subgoal Graph*** of v. We are interested only in *minimal* candidate subgoal graphs of $\mathcal{G}^{\mathrm{D}}$, where *minimal* means that there does not exist another candidate subgoal graph $\mathcal{G}''\langle \mathrm{V}'', \mathrm{E}'' \rangle$ of $\mathcal{G}^{\mathrm{D}}$ such that $\mathrm{V}'' \subseteq \mathrm{V}'$. In addition, if we define the cost of a candidate subgoal graph $\mathcal{G}'$ as the sum of the costs of the activities in $\mathcal{G}'$, we want the minimal candidate subgoal graph with the cheapest cost. We define a ***Subgoal Graph*** of v, $\mathcal{G}^{\mathsf{sg}}[v]$, as being a minimal candidate subgoal graph of v having the cheapest cost. Given $\mathcal{G}^{\mathsf{sg}}[v]$, the ***Dependency Set*** $\mathcal{DS}[a]$ is composed of the activities corresponding to the nodes of $\mathcal{G}^{\mathsf{sg}}[v]$:

$$\mathcal{DS}[a] = \{a_i \mid v(a_i) \in \mathrm{V} \wedge \mathcal{G}^{\mathsf{sg}}[v(a)] = \langle \mathrm{V}, \mathrm{E} \rangle\}$$

The activities in $\mathcal{DS}[a]$ are the minimal set of subgoals that will eventually be in a final plan Π_f, if Π_f contains a.

In our experience with Europa2 real-world domains, DGs have a limited number of OR nodes and so the extraction of a subgoal graph from them is feasible. However, when the number of OR nodes in a DG grows, the computation of a minimal closed subgraph might become infeasible, and we need to focus on finding just an approximation of it. In particular, instead of exploring all the successors of each OR node, we can adopt strategies to select and visit only subsets of them. Two independent approaches can be used:

- ***Weighting:*** it provides a way for choosing between different successors of an OR node. We assign a *weight* to each successor of any OR node. Then, when we encounter an OR node while visiting the graph, we sum the weighted costs of the successors. If we reach an OR node n such that all its successors have a weight of zero, we stop the visit of the subgraph rooted in n. There are a number of possible weighting policies:
  i. ***AND nodes only:*** We assign a weight of zero to all the successors of the OR nodes laying at the shallowest depth in $\mathcal{G}^{\mathrm{D}}$. In such a way, we collect a's subgoals that are dependent on a via AND nodes and ignore OR nodes altogether. Hence, the set $\mathcal{DS}[a]$ will

contain only the activities that are necessarily dependent on a and ignore disjunctive activities that might or might not be subgoals of a depending on the values assigned to the guard variables.

ii. ***Arbitrary choice:*** For each node OR in $\mathcal{G}^{\mathrm{D}}$, we arbitrarily assign 1 to one of its successors and zero to all the other successors.

iii. ***Probability mass distribution:*** We assign a weight of 1 to each OR node at the shallowest depth in $\mathcal{G}^{\mathrm{D}}$ and then split this weight equally among its successors. So, if an OR node has n branches, we assign to each successor a weight of $1/n$. Starting with the weight $1/n$, we repeat the same operation when the next OR node is encountered, so as to push the obtained probabilities down through the graph. The weight of a node represents the probability that the associated activity will be in the plan. The costs of the nodes are then attenuated by multiplying them by the probability.

- ***Discount Factor:*** it controls how deep we go in the exploration of a DG. We use a *discount factor* as in Markov Decision Processes to progressively reduce the contribution of the successor nodes, the deeper we go in the DG graph. For example, at depth 1, we can give full weight to the costs; at depth 2, we can weight the costs by a discount factor d; at depth 3, we can weight the costs by d^2, and so on. At some point the contributions become negligible and can be ignored.

Cost Estimatation through DGs

We now turn to the use of dependency sets to calculate cost resolution. Consider again a partial plan Π with flaw f, and a resolver r corresponding to the free activity a with an associated dependency set $\mathcal{DS}[a]$. Let Π' be the partial plan derived from Π after applying r. We define the cost of the resolution, $\mathcal{C}_{res}(r)$, as follows:

- *Merging* the activity a with some existing activity a' in Π:

$$\mathcal{C}_{res}(r) = 0$$

- *Placing* the activity a in a compatible empty slot s:

$$\mathcal{C}_{res}(r) = \sum_{a_i \in \mathcal{DS}[a]} \mathcal{C}ost_{\mathsf{sg}}(a_i)$$

where $\mathcal{C}ost_{\mathsf{sg}}(a_i)$ of an activity in $\mathcal{DS}[a]$ is defined as:

– if a_i can be merged with an activity a' in $\mathcal{A}ct_{\Pi'}$:

$$\mathcal{C}ost_{\mathsf{sg}}(a_i) = 0$$

– if a_i is compatible with a free activity a' in $\mathcal{F}_{\Pi}$:

$$\mathcal{C}ost_{\mathsf{sg}}(a_i) = 0$$

– if a_i can be activated and placed on its timeline:

$$\mathcal{C}ost_{\mathsf{sg}}(a_i) = \mathcal{C}_{\mathsf{v}}(a_i)$$

– otherwise:

$$\mathcal{C}ost_{\mathsf{sg}}(a_i) = +\inf$$

The first definition corresponds to the intuition that resolution through merging has no cost with respect to the subgoals that should be managed after its application. In fact, it does not modify the partial plan Π except for adding new temporal constraints. The second definition calculates the cost of activating a and placing it on the slot s by summing up the costs of the single subgoals that are dependent on a. So, this cost reflects the penalty incurred by adding a and all its subgoals to Π. The costs of a's subgoals depend on the new configuration of the timelines after the introduction of a in the plan Π. In particular, the cost of a single subgoal a_i in $\mathcal{DS}[a]$ is zero if an activity a'_i compatible with a_i is already in the plan Π' or it is in the set of flaws of Π, $\mathcal{F}_{\Pi}$. In these two cases, in fact, the cost of r does not depend on the cost of a_i since either the subgoal a_i has been already achieved (it is the plan) or it must be satisfied regardless from the application of the resolver r (it is already in the set of flaws). If these two alternatives do not apply, then the cost of a_i should be considered while calculating the cost of r. The cost of a_i corresponds to $\mathcal{C}_{\mathsf{v}}(a_i)$, and is set to infinite when a_i cannot possibly be placed in the current plan.

If $\mathcal{R}[f] = \{r_1, \ldots, r_k\}$ is the set of possible resolutions for a flaw f, we define the ***Cheapest Resolution*** as:

$$\mathrm{CR}(f) = min_{r_i \in \mathcal{R}[f]} \mathcal{C}_{res}(r_i)$$

By using the $\mathrm{CR}(f)$ for a flaw f, we prefer the resolver that is the cheapest in terms of the subgoals that need to be achieved because of its application. We prefer merging to other possibilities and, if merging is not possible, prefer slots that maximize the reuse of those fragments of the plan that are already in place, minimize the number of unsatisfied subgoals, and avoid unsuitable slots.

So far, we have focused on resolution costs for open condition and ordering flaws. However, we can use a similar mechanism to assign costs to unbound variable flaws as well. Choosing a resolver for an unbound variable flaw means choosing a value for a guard variable, which in turn corresponds to triggering one set of subgoals instead of another. In order to rank the different possible choices for a guard variable, we need to evaluate how difficult it is to satisfy the subgoals associated with that choice. Each subgoal corresponds to a free activity, whose cost can be estimated by using the technique that we have just presented. More formally, let us assign a cost to the value val_i of a guard variable γ. Suppose that the free activities $a_{i1}, \ldots, a_{in}$ are triggered by choosing the value val_i for γ. We calculate the dependency set $\mathcal{DS}[a_{ij}]$ for each activity a_{ij} and union these sets. The resulting set of subgoals is the dependency set associated with the value val_i: $\mathcal{DS}[val_i]$. The cost of binding γ to val_i is the following: $\mathcal{C}_{res}(val_i) = \sum_{a_{ij} \in \mathcal{DS}[val_i]} \mathcal{C}ost_{\mathsf{sg}}(a_{ij})$, where the cost $\mathcal{C}ost_{\mathsf{sg}}(a_{ij})$ for the activities in $\mathcal{DS}[val_i]$ is defined as above. Once we have the cost of each possible binding value val_i, we pick the value associated with the lowest cost and assign it to the guarded variable.

We can use the same concepts and mechanisms to assign selection costs to flaws in order to guide flaw selection. However, due to space limitation, we do not describe these techniques here.

Evaluation

The evaluation of domain-independent search control techniques for Europa2 is difficult due to the unavailability of a broad benchmark set of NDDL domains. We have conducted a preliminary evaluation of the DG technique on NASA's domains that are publicly available (NASA 2009). We focus here on the *Crew Planning* domain, a domain against which the ATG technique cannot be used. We show a simple example of how the DG heuristic can be helpful for controlling the search. The behavior of the timelines of the *Crew Planning* domain is not described by means of `meets` and `met_by` constraints involving activities belonging to the same timelines, but is specified hierarchically. The activity *DailyRoutine*, belonging to the high-level timeline *CrewPlanner*, induces the activities *Sleep(480)*, *PostSleep(180)*, *Meal(60)*, *Exercise(60)* on the lower-level timeline *CrewMember* and, in turn, these activities induce the activities *Fasting([1, +inf))* and *NotFasting([1, +inf))* on the lowest level timeline *FastingWindow*. Of course, the initial configuration $\mathcal{I}$ can contain activities at any hierarchical layer.

Figure 3 shows an intermediate partial plan Π for *Crew-Planning* and an open condition and ordering flaw f corresponding to the introduction in Π of the activity *Meal(60)*, slave of *DailyRoutine* via the constraint `contains`. Two slots, sl_1 and sl_2, satisfy the constraint and are possible resolvers of f. The activity *Meal(60)* has two compatibilities involving activities on the timeline *FastingWindow*:

$$\mathcal{R}[\textit{Meal()}] = (\texttt{meets}\ \textit{FastingWindow.Fasting()} \wedge \texttt{equals}\ \textit{FastingWindow.NotFasting()})$$

By exploring the DG graph of this domain (see Figure 2), we develop the dependency set of *Meal()*:

$$\mathcal{DS}[\textit{Meal()}] = \{\textit{Fasting()}, \textit{NotFasting()}\}$$

Considering the configuration of the timelines in Figure 3, we see that $\mathcal{C}_{res}(sl_2)$ is zero since we can merge the subgoals in $\mathcal{DS}[\textit{Meal()}]$ respectively with the activities *Fasting(id=6)* and *NotFasting(id=5)*, whereas the $\mathcal{C}_{res}(sl_1)$ is infinite since we cannot satisfy the subgoal *NotFasting()*. So, guided by the DG heuristic, we choose sl_2 and avoid choosing a resolver that would cause backtracking in one subsequent step.

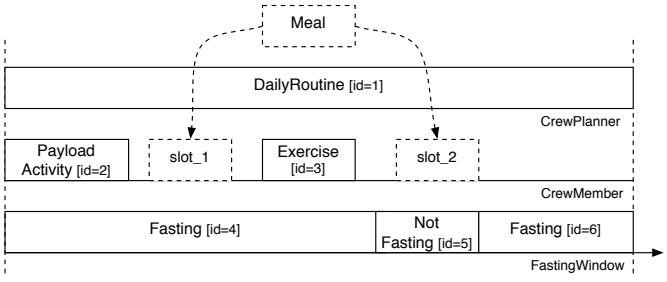

Figure 3: Timelines for the *Crew Planning* domain

We are setting up an extensive evaluation of the DG technique. We need to test how the DG estimates vary in correspondance with different exploration strategies of the dependency graph $\mathcal{G}^{\mathrm{D}}$. In particular, we have two independent dimensions along with to test the technique: (1) how deep we go in the exploration of the DG; (2) how we assign weight to different branches of an OR. Along the first dimension, we have different depth bounds and different discount factors to test. Along the other dimension, we have different weighting policies for OR branches: all zero weights, random choice, and probability mass distribution. We are also using domains developed for the International Planning Competition (IPC) translated from PDDL into NDDL (Bernardini and Smith 2008b). We believe that the DG technique will work equally well on them, but this remains to be proven. In this case, we can compare Europa2 augmented with the DG heuristics with state of the art temporal planners, such as CPT (Vidal and Geffner 2006). In addition, we intend to compare DG heuristics with ATG and hand coded heuristics. We believe that completing such an extensive experimental evaluation will lead to a broader understanding of what heuristics work best on different types of domains and ultimately will lead to identifying properties for classifying domains and automatically choosing the most suitable search control approach to apply.

Related Work

In the last ten years, there has been considerable work in the classical planning community on devising domain-independent heuristics. Generally, these techniques involve solving some relaxed form of the planning problem in order to obtain heuristic distance estimates, which are then used to guide search. Simple but effective ways to obtain relaxed problems are, for instance, ignoring PDDL operator delete lists or decomposing the goal set of atoms into smaller subsets (Bonet and Geffner 2001), (Haslum and Geffner 2000). The computation of some of those heuristics rely on the explicit construction of a *reachability graph* (Blum and Furst 1997), (Hoffmann, Porteous, and Sebastia 2004), while other methods perform *shortest path* calculations on a suitably defined atom space (Haslum and Geffner 2000). In addition to "distance-based" heuristics, other techniques have been proposed that work on multi-valued representations of planning tasks. Fast Downward (Helmert 2006), for example, extracts a heuristic function by constructing a *causal graph* of the domain and a *domain transition graph* for each state variable in the domain. The first graph represents the critical interactions between state variables, while the second graph describes the dependancies between the values of a single state variable.

There are several similarities between the abovementioned strategies developed for IPC planners and the method described here. For example, the DG technique applies an approach similar to ignoring delete lists since it assumes that an activity in the set of the free or activated activities can always be reused to resolve any new flaw corresponding to a compatible free activity. In addition, we can look at the DGs described here as a combination between the causal and domain transition graphs introduced by Helment

since DGs represent the dependencies between the values of all the state variables in the domain. Finally, it is worth noting that the application of the weighting policy "AND-nodes-only" for extracting dependency sets corresponds to the computation of *landmarks* for PDDL instances as described in (Hoffmann and Nebel 2001). Indeed, just as a landmark is a fact that needs to be achieved in any solution plan, a dependency set obtained by following AND nodes and ignoring OR nodes contains only those activities that will be necessarily part of any valid plan.

With regard to heuristics specifically devised for Europa2, in (Bernardini and Smith 2008a; 2007) we describe a method for controlling search based on the construction of *activity transition graph* (ATG). This technique requires that the domain model specifies for each activity a its `meets` and `met_by` constraints with activities taking place on the same timeline. Given a domain $\mathcal{D}$, for each timeline $\mathcal{T}$ in $\mathcal{D}$, we construct a graph describing the possible *transitions* between the activities that can be placed on $\mathcal{T}$. The nodes in the graph represent the activities in $\mathcal{A}ct[\mathcal{T}]$. The `meets` and `met_by` temporal constraints between these activities induce the transitions between the nodes. Each transition has a cost that identifies the *temporal distance* between the activities involved in the transition. Thus, an ATG for a timeline $\mathcal{T}$ represents the internal temporal evolution of $\mathcal{T}$. We calculate the cheapest path from any activity a_i to any other activity a_j by running an *all-pairs shortest path algorithm* on each graph and store the cost $\mathcal{C}ost_{\mathsf{sp}}(a_i, a_j)$ of each shortest path in a table. This table is pre-computed prior to beginning planning and then used for flaw resolution and flaw selection. In particular, given an open condition or ordering flaw f, if f can be resolved via merging, the cost of the resolution, $\mathcal{C}_{res}(r)$, is set to zero, since it does not modify the partial plan except for adding new temporal constraints. If f is resolved through activating an activity a and placing it in a slot s between a_i and a_{i+1}, which are already in the plan, $\mathcal{C}_{res}(r) = \mathcal{C}ost_{\mathsf{sp}}(a_i, a) + \mathcal{C}ost_{\mathsf{sp}}(a, a_{i+1}) - \mathcal{C}ost_{\mathsf{sp}}(a_i, a_{i+1})$. This cost estimates how well the activity a fits in the empty slot s on $\mathcal{T}$. If $\mathcal{R}[f] = \{r_1, \ldots, r_k\}$ is the set of resolutions for f, the *Cheapest Local Resolution* is defined as: $\mathrm{CLR}(f) = min_{r_i \in \mathcal{R}[f]} \mathcal{C}_{res}(r_i)$. By using the $\mathrm{CLR}(f)$, we prefer to place a in the slot where it causes the smallest increase in the net cost for the timeline $\mathcal{T}$. This provides a good estimate of cost since it prefers merging to other possibilities, and slots with low cost paths to higher cost paths.

Conclusions

We have presented a novel search control technique for Europa2 that can be used to guide both flaw selection and flaw resolution for domains that lack the strong causal structure of PDDL domains. This technique constructs a dependency graph that represents the dependencies between the activities in the domain model. The graph is then explored in order to gain information about the impact of different flaw resolutions on the current partial plan. The resolver with the lower impact is chosen to be applied. Although more experimental work is needed, this seems to be a viable method for guiding search on real-world domains whose structure does not allow us to use the simpler but less accurate ATG technique.

References

Allen, J. 1983. Maintaining knowledge about temporal intervals. *Communications of the ACM* 26(11):832–843.

Bernardini, S., and Smith, D. E. 2007. Developing domain-Independent search control for EUROPA2. In *Proc. of the Workshop on Heuristics for Domain-independent Planning, 17th Int. Conference on Automated Planning and Scheduling (ICAPS'07)*.

Bernardini, S., and Smith, D. E. 2008a. Automatically generated heuristic guidance for EUROPA2. In *Proc. of the 9th International Symposium on Artificial Intelligence, Robotics, and Automation for Space (iSAIRAS'08)*.

Bernardini, S., and Smith, D. E. 2008b. Translating PDDL2.2 into a constraint-based variable/value language. In *Proc. of the Workshop on Knowledge Engineering for Planning and Scheduling, 18th International Conference on Automated Planning and Scheduling (ICAPS'08)*.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90:281–300.

Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1-2). Special issue on Heuristic Search.

Bresina, J. L.; Jónsson, A.; Morris, P. H.; and Rajan, K. 2005. Mixed-initiative activity planning for Mars rovers. In *Proc. of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, 1709–1710.

Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; B.Smith; Fisher, F.; Barret, T.; Stebbins, G.; and Tran, D. 2000. ASPEN - Automated planning and scheduling for space missions operations. In *International Conference on Space Operations*.

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

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. of the Second International Conference on Artificial Intelligence Planning Systems (AIPS-94)*, 61–67. AAAI Press.

Haslum, P., and Geffner, H. 2000. Admissible heuristics for optimal planning. In *Proc. of the Fifth Int. Conference on Artificial Intelligence Planning and Scheduling (AIPS-00)*, 140–149.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered Landmarks in Planning. *Journal of Artificial Intelligence Research* 22:215–278.

NASA. 2009. Europa2 web page. `http://babelfish.arc.nasa.gov/trac/europa/`.

Vidal, V., and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence* 170 (3):298–335.