Relational and First-Order Decision-Theoretic Planning: Deterministic Planning Supplement

Scott Sanner

Department of Computer Science

University of Toronto

ssanner@cs.toronto.edu

April 25, 2004

Abstract

This document is intended as a supplement to the paper *Relational and First-Order Decision Theoretic Planning: Foundations and Future Directions.* This document covers the background for the deterministic restriction of decision-theoretic planning, focussing specifically on ideas that relate the enumerated, propositional, relational, and first-order deterministic planning paradigms.

Contents

1 Decision-theoretic planning

Fully-observable decision-theoretic planning is the problem of finding an optimal sequence of actions that maximize expected reward assuming that the state can be fully observed and that the transition dynamics are known. We can generally define such a decision-theoretic planning problem[1] as a tuple: $\langle S, A, T, R \rangle$ where: S is a finite state space, A is a finite set of actions, T is a transition distribution $T : S \times A \times S \rightarrow [0, 1]$ where $T(s, a, \cdot)$ is a probability distribution over S for any $s \in S$ and $a \in A$, and R is a bounded reward function $R : S \times A \rightarrow \mathbb{R}$. One can think of a reward defined over states and actions as incorporating the cost of performing the action from the given state. In certain cases, we can decompose the reward function as the following: $R(s, a) = StateReward(s) - Cost(s, a)$ where $StateReward$ is the immediate reward for being in state s and $Cost$ is the cost of executing action a in state s.

Our goal in decision-theoretic planning is to find a plan (i.e., a sequence of actions) that maximizes the infinite horizon, discounted reward criterion: $E_\pi[\sum_{t=0}^{\infty} \gamma^t \cdot r^t | s_0]$ where r^t is a reward obtained at time t, γ is a discount factor, π is the plan being executed, and s_0 is the initial starting state.

Given the domain and policy representations, we can define a planner P as a system that accepts as arguments a problem domain $\langle S, A, T, R \rangle$ and returns an optimal plan π^* that maximizes the reward criteria. As we introduce each planning formalism in this paper, we will define the specific domain and policy representation within this framework in order to make the assumptions of each framework mathematically explicit.

2 Deterministic planning

First we consider the task of deterministic planning and how to cast it as a special case of the previously described decision theoretic planning framework. While there are a number of ways to define deterministic planning, we choose to focus on problems with a single goal since this is consistent with the majority of the literature. For this representation, S is a finite state space, A is a finite set of actions, and T is a *deterministic* transition function $T : S \times A \rightarrow S$ where $T(s, a)$ maps from $s \in S$ and $a \in A$ to a successor state $s' \in S$, and $R : S \times A \rightarrow \mathbb{R}$ is the standard reward function for a state and action.

In most deterministic planning systems, the goal is to find a plan that represents the lowest-cost path from an initial state to a goal state. We use the $Cost$ and $StateReward$ decomposition for $R(s, a) \rightarrow \mathbb{R}$ as defined previously. Here the $Cost$ function is simply the finite cost of taking an action from a state,

[1] This is actually the definition of a Markov devision process (MDP) [16]. We do not discuss the full stochastic MDP theory at this point because it is unnecessary for the following section on deterministic planning.

and *StateReward* is defined using the value of the goal v:

$$StateReward(s) = \begin{cases} 0 & \text{if } goal(s) \\ v & \text{if } \neg goal(s) \end{cases}$$

These planning problems are usually finite so we must make one slight modification to fit them into the infinite horizon reward framework: We modify the domain so that all goal states transition to a non-goal absorbing state with zero action cost. We then assign $\gamma = 1$ to ensure that the value of a policy is exactly the sum of the accumulated actions costs and goal reward. Recall that the reward is zero for every non-goal state so it can be easily shown that the reward is finitely bounded in this case. Any planner that returns an optimal policy with respect to this modified domain will find the optimal lowest-cost policy with respect to the original finite horizon problem.

Alternately, if we simply want a shortest-path policy (where all actions have the same cost) then we cam modify the above approach slightly. We use the same assumptions as above, but set all action costs to zero and ensure that $0 < \gamma \leq 1$. Any policy maximizing expected discounted reward in this case will return a shortest-path policy.

Now, for a policy representation, we have two options. If the initial state is unknown, then a planner needs to return an optimal policy $\pi^*(s) \rightarrow a$ that returns the optimal action to perform from any state. However, a more common use of deterministic planners is to find an optimal plan from a given initial state. Since the state transitions are deterministic, we only need return the action sequence leading from the initial state to the goal state. Consequently, we can simply return a plan $\pi^*_{s_0}(s) = \{a_1, \ldots, a_n\}$ that specifies this optimal sequence of actions to take from the given initial state.

We next proceed to specify the specific $\langle S, A, T, R \rangle$ representation and solution algorithms for each of the following successively more powerful deterministic planning formalisms. Since traditional deterministic planning often only deals with single goal problems, we will use a running example drawn from the blocks world domain. This domain is shown in figure 1. Note that all of the following formalisms usually solve either the shortest-path or in some cases the lowest-cost planning problem.

2.1 Flat search

Flat state space search is the simplest of all planning formalisms. In this case, we enumerate both our states: $S = \{s_1, \ldots, s_n\}$ and our actions: $A = \{a_1, \ldots, a_n\}$. The transition, action cost, and state-based reward functions are simply defined over this state and action space.

For example, if we allow for three states $S = \{s_1, s_2, s_3\}$ in our blocks world domain and we have two blocks b_1 and b_2, then we could make the state and action assignments given in Figure 2.1.

Given this domain definition, we can execute a number of algorithms to perform optimal planning for both the lowest-cost (every action has an associated cost) and shortest-path (all actions have the same cost) reward criteria. Some of these are described below:

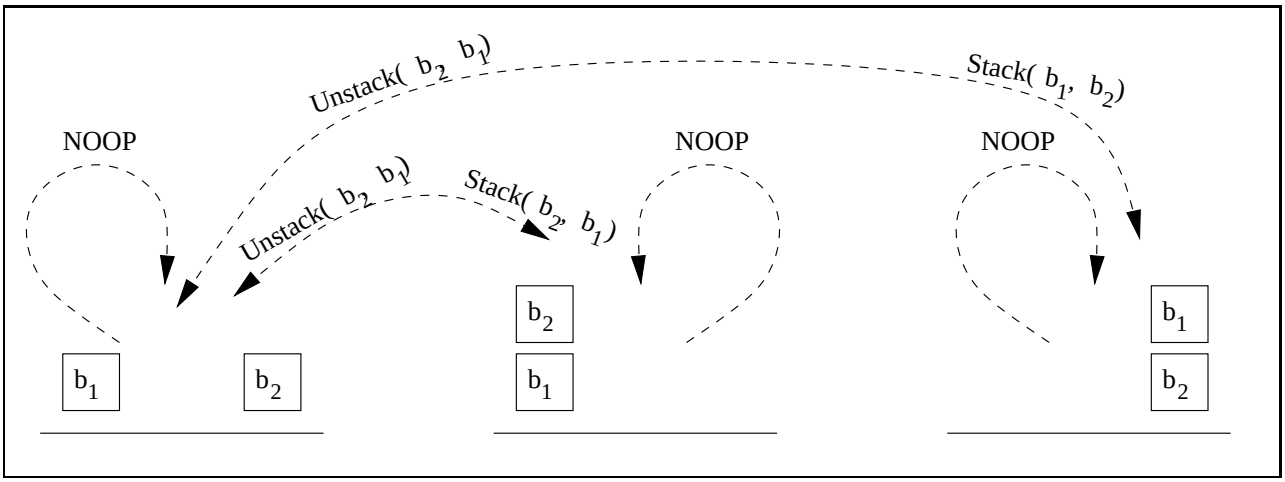

Figure 1: Since most deterministic planners focus on single goal problems, we will use a running example from the blocks world that exemplifies this type of problem. In this domain, one has a number of blocks that can be stacked and unstacked from each other and the goal is to find a plan that constructs the goal state configuration given an initial state configuration. This diagram shows the legal state configurations and action transitions for a small problem instance. For simplicity of exposition, we assume that any block not on another block is implicitly on the table (i.e. flat surface).

State	**Flat Encoding**	**Description**
1	s_1	b_1 and b_2 are both clear
2	s_2	b_1 is on b_2
3	s_3	b_2 is on b_1

Action	**Flat Encoding**	**Description**	**Preconditions**	**Transition Function**
1	a_1	Do nothing (NOOP)	Legal from all states	Transition to same state
2	a_2	Stack b_1 on top of b_2	Legal from state s_1	Transition to state s_2
3	a_3	Unstack b_1 from b_2	Legal from state s_2	Transition to state s_1
4	a_4	Stack b_2 on top of b_1	Legal from state s_1	Transition to state s_3
5	a_5	Unstack b_2 from b_1	Legal from state s_3	Transition to state s_1

Figure 2: A sample flat representation of a blocks world planning problem with three states $S = \{s_1, s_2, s_3\}$ and five actions $A = \{a_1, a_2, a_3, a_4, a_5\}$.

- *Priority-first search*: In this paradigm we maintain a queue that prioritizes the frontier of search states (states that are one action/edge away from the current set of visited states): We initialize the priority queue with our initial state, always search the highest priority frontier state next, and

maintain the priority-queue of frontier state values according to our reward definition. If, for every visited state, we maintain the lowest cost state/action pair that placed it on the frontier queue, we can derive the optimal lowest-cost or shortest-path policy from the first goal state reached. These ideas were introduced by Dijkstra in the context of shortest path search in a weighted graph [4].

- *A^* search*: In this case, we execute the above algorithm but use a heuristic priority value in place of the actual future reward, i.e. let the priority $P(s)$ be defined as $P(s) = G(s) + H(s)$ where $G(s)$ is the current reward estimate for s and $H(s)$ is an admissable heuristic predicting the future reward accumulated on the way to the goal. We say that $H(s)$ is admissable if it underestimates the maximum future reward. So long as $H(s)$ is admissable, this algorithm is guaranteed to find the optimal path to the goal, and will often do so much more quickly on account of the heuristic guidance. This algorithm was introduced by Nilsson [7].

- *Iterative deepening A^**: This algorithm is a variant of A* that uses the iterative deepening strategy instead of the breath-first priority search. This algorithm starts at depth 1, and performs an exhaustive depth-first search. If the search finds a solution, this is returned, otherwise it increases the depth by one and repeats the search depth-first search. It continues doing this until the goal is found. This algorithm can be shown to have the same asymptotic time complexity and optimality guarantees as A* while requiring only an amount of space linear in the search depth. It was first introduced by Korf [11].

In addition to the above algorithms, it should also be mentioned that since we have a model of the transition dyanmics, we can perform the same algorithms in reverse by swapping the initial and goal states. This search is known as goal regression.

2.2 Propositional planning

One problem with the flat domain representation is that there is often much more information in the states than is apparent from a flat encoding. For example, in the blocks world domain, an action on two blocks does not affect the state of any other block. Yet, in our transition encodings, we need to specify the effect of an action for every distinct block configuration and action. This representation is exponential in the number of blocks. There is no efficient way to specify that only part of a state is affected by an action in the flat representation - we simply don't have access to this state information in a flat representation.

A more compact representation would rely on a propositional encoding of state. In this case, we assume our state S is defined by n state variables $S = \{S_1 \times \cdots \times S_n\}$ where each state variable takes a value assignment from a finite domain. For example, we could define a state for our blocks world problem as $S = \{On\text{-}b1\text{-}b2, On\text{-}b2\text{-}b1\}$. In this example, each state variable would have a boolean domain

State	Propositional Encoding
1	$\{On\text{-}b1\text{-}b2 = \text{F}, On\text{-}b2\text{-}b1 = \text{F}\}$
2	$\{On\text{-}b1\text{-}b2 = \text{F}, On\text{-}b2\text{-}b1 = \text{T}\}$
3	$\{On\text{-}b1\text{-}b2 = \text{T}, On\text{-}b2\text{-}b1 = \text{F}\}$
4	$\{On\text{-}b1\text{-}b2 = \text{T}, On\text{-}b2\text{-}b1 = \text{T}\}$

Action	Encoding	Description	Preconditions	Transition Effect (unchanged if not specified)
1	a_1	Do nothing (NOOP)	$\top$	$\emptyset$
2	a_2	Stack b_1 on top of b_2	$\neg On\text{-}b1\text{-}b2 \wedge \neg On\text{-}b2\text{-}b1$	$On\text{-}b1\text{-}b2 = \text{T}$
3	a_3	Unstack b_1 from b_2	$On\text{-}b1\text{-}b2 \wedge \neg On\text{-}b2\text{-}b1$	$On\text{-}b1\text{-}b2 = \text{F}$
4	a_4	Stack b_2 on top of b_1	$\neg On\text{-}b1\text{-}b2 \wedge \neg On\text{-}b2\text{-}b1$	$On\text{-}b2\text{-}b1 = \text{T}$
5	a_5	Unstack b_2 from b_1	$\neg On\text{-}b1\text{-}b2 \wedge On\text{-}b2\text{-}b1$	$On\text{-}b2\text{-}b1 = \text{F}$

Figure 3: A sample propositional representation of a blocks world planning problem. Note that the fourth state should be unreachable and the transition semantics enforce this for any starting state other than the fourth state. We will discuss more complex data structures for propositional transition encoding in future sections. For now, the semantics of the transition function should be readily apparent. Note that as more blocks are added and the number of state propositions combinatorically expands, there is no need to change the transition function effects for these five actions. This clearly offers substantial representational savings over the flat encoding where each additional block would combinatorically explode the transition table.

corresponding to whether it was true or not - presumably, it would be impossible to reach a state where both were true.

Now we can compactly represent transitions and rewards as decision trees, ADDs [3], or any other compact data structure for propositional encoding. See Figure 3 for a propositional encoding of this domain.

While the actual state space is exponential in the length of the number of state variables, the encoding of the transition and reward functions are often much smaller than this on account of independence between variables. While solving this representation has the same worst-case computational bounds as for flat search, it is often the case that one can do deterministic planning without expanding the entire state space since many states are unreachable. In this case, having a formalism that does not require one to expand the entire state space is of utmost importance for achieving efficiency in search. The same algorithms used for flat search can be applied here but the states can be represented compactly in the propositional form. Unfortunately, we still need to enumerate each action since we have do not have an efficient propositional encoding for them.

2.3 Relational planning

In the paradigm of relational planning, state properties and actions are specified using relations over domain objects. When a relational planning model is grounded for a particular instantiation of domain objects, one attains a propositional model of the domain. STRIPS and PDDL are two of the most popular languages for relational planning representations so we cover these in the next section.

2.3.1 STRIPS and PDDL

STRIPS is one of the most commonly used planning languages and was introduced by Fikes and Nilsson [5]. Since the STRIPS syntax has been modified over the years, we cover a slightly enhanced but standardized STRIPS-like syntax known as PDDL (planning domain description language). The original PDDL syntax and semantics was created by McDermott et al. [14] and revised for version 2.1 by Long and Fox [6]. PDDL's most important departure from STRIPS is the addition of both universal and conditional effects that allow an action effect to modify *any* relations meeting some condition (e.g., a paint action that paints *all* objects *at* some location).

A PDDL domain consists of the following:

- *State Representation* States are described using relations over objects from a finite domain. Each ground atom (e.g., $On(b_1, b_2)$) represents a boolean proposition. To make this specification compact, a state is specified by stating only the $true$ ground atoms, any unspecified ground atoms are assumed $false$.

- *Action Representation* An action is specified by its preconditions and its effects (represented as add and delete lists in STRIPS or non-negated and negated effects in PDDL). Following is an example of the stack action expressed in PDDL (its intent should be clear from the syntax):

```
(:action stack
 :parameters (?a ?b)
 :precondition (and (forall (?c) (not (on-top-of ?c ?a)
                    (forall (?c) (not (on-top-of ?c ?b))))))
 :effect       (and (forall (?c)
                        (when (on-top-of ?a ?c)
                              (not (on-top-of ?a ?c))))
                        (on ?a ?b)))
```

- *Problem Representation* A planning problem is represented by a domain instantiation, an initial state and a goal formula. The initial state is represented by a set of true ground atoms. The goal representation in PDDL can be any closed first-order formula.

Example Domain & Problem Description	
Relation Schema:	$\{On(a,b)\}$
Domain Instantiation:	$\Delta = \{b_1, b_2\}$
Example Initial State:	$\{On(b_1, b_2)\}$
Example Goal:	$\neg\exists a, b On(a,b)$

State	Ground Relation/Proposition Assignment	STRIPS/PDDL Representation
1	$On(b_1,b_1) = \text{F}$, $On(b_1,b_2) = \text{F}$, $On(b_2,b_1) = \text{F}$, $On(b_2,b_2) = \text{F}$	$\emptyset$
2	$On(b_1,b_1) = \text{F}$, $On(b_1,b_2) = \text{F}$, $On(b_2,b_1) = \text{F}$, $On(b_2,b_2) = \text{T}$	$\{On(b_2,b_2)\}$
3	$On(b_1,b_1) = \text{F}$, $On(b_1,b_2) = \text{F}$, $On(b_2,b_1) = \text{T}$, $On(b_2,b_2) = \text{F}$	$\{On(b_2,b_1)\}$
4	$On(b_1,b_1) = \text{F}$, $On(b_1,b_2) = \text{F}$, $On(b_2,b_1) = \text{T}$, $On(b_2,b_2) = \text{T}$	$\{On(b_2,b_1), On(b_2,b_2)\}$
5	$On(b_1,b_1) = \text{F}$, $On(b_1,b_2) = \text{T}$, $On(b_2,b_1) = \text{F}$, $On(b_2,b_2) = \text{F}$	$\{On(b_1,b_2)\}$
6	$On(b_1,b_1) = \text{F}$, $On(b_1,b_2) = \text{T}$, $On(b_2,b_1) = \text{F}$, $On(b_2,b_2) = \text{T}$	$\{On(b_1,b_2), On(b_2,b_2)\}$
7	$On(b_1,b_1) = \text{F}$, $On(b_1,b_2) = \text{T}$, $On(b_2,b_1) = \text{T}$, $On(b_2,b_2) = \text{F}$	$\{On(b_1,b_2), On(b_2,b_1)\}$
8	$On(b_1,b_1) = \text{F}$, $On(b_1,b_2) = \text{T}$, $On(b_2,b_1) = \text{T}$, $On(b_2,b_2) = \text{T}$	$\{On(b_1,b_2), On(b_2,b_1), On(b_2,b_2)\}$
9	$On(b_1,b_1) = \text{T}$, $On(b_1,b_2) = \text{F}$, $On(b_2,b_1) = \text{F}$, $On(b_2,b_2) = \text{F}$	$\{On(b_1,b_1)\}$
10	$On(b_1,b_1) = \text{T}$, $On(b_1,b_2) = \text{F}$, $On(b_2,b_1) = \text{F}$, $On(b_2,b_2) = \text{T}$	$\{On(b_1,b_1), On(b_2,b_2)\}$
11	$On(b_1,b_1) = \text{T}$, $On(b_1,b_2) = \text{F}$, $On(b_2,b_1) = \text{T}$, $On(b_2,b_2) = \text{F}$	$\{On(b_1,b_1), On(b_2,b_1)\}$
12	$On(b_1,b_1) = \text{T}$, $On(b_1,b_2) = \text{F}$, $On(b_2,b_1) = \text{T}$, $On(b_2,b_2) = \text{T}$	$\{On(b_1,b_1), On(b_2,b_1), On(b_2,b_2)\}$
13	$On(b_1,b_1) = \text{T}$, $On(b_1,b_2) = \text{T}$, $On(b_2,b_1) = \text{F}$, $On(b_2,b_2) = \text{F}$	$\{On(b_1,b_1), On(b_1,b_2)\}$
14	$On(b_1,b_1) = \text{T}$, $On(b_1,b_2) = \text{T}$, $On(b_2,b_1) = \text{F}$, $On(b_2,b_2) = \text{T}$	$\{On(b_1,b_1), On(b_1,b_2), On(b_2,b_2)\}$
15	$On(b_1,b_1) = \text{T}$, $On(b_1,b_2) = \text{T}$, $On(b_2,b_1) = \text{T}$, $On(b_2,b_2) = \text{F}$	$\{On(b_1,b_1), On(b_1,b_2), On(b_2,b_1)\}$
16	$On(b_1,b_1) = \text{T}$, $On(b_1,b_2) = \text{T}$, $On(b_2,b_1) = \text{T}$, $On(b_2,b_2) = \text{T}$	$\{On(b_1,b_1), On(b_1,b_2), On(b_2,b_1), On(b_2,b_2)\}$

Action	Schema	Preconditions	Action Effects (Conditional effects indicated by $\rightarrow$)
NOOP	$NOOP()$	$\top$	$\emptyset$
Stack	$Stack(a,b)$	$(\neg\exists c\, On(c,a)) \wedge (\neg\exists c\, On(c,b))$	$\{On(a,b),\ (\forall c\, On(a,c) \rightarrow \neg On(a,c))\}$
Unstack	$Unstack(a,b)$	$(\neg\exists c\, On(c,a))$	$\{\neg On(a,b)\}$

Figure 4: A sample relational encoding of a blocks world domain and problem. This state table represents all sixteen states for a blocks world domain with relation schema $On(a,b)$ and domain objects $\Delta = \{b_1, b_2\}$. We denote that a ground relation is true by the symbol T and that a ground relation is false by the symbol F. Note that a lot of these states are non-sensical given our natural intuitions about the blocks world, e.g. $\{On(b_1,b_1)\}$ or $\{On(b_1,b_2), On(b_2,b_1)\}$. However, there is no inherent property of STRIPS that restricts relation assignments to legal configurations - such constraints must be enforced by avoiding illegal initial states and ensuring that action definitions prevent these states being reached from legal states. Note that while the state and action space can be quite large, they were instantiated from a very small schema - one that holds for any domain instantiation.

Figure 4 contains an example relational domain and problem.

2.3.2 Other PDDL enhancements

In addition to the PDDL language elements already discussed, PDDL version 2.1 [6] includes a few additional enhancements over basic STRIPS:

- Numeric functional fluents
- Explicit representation of time and duration
- Specification of plan metrics (reward) as part of problem instances (i.e. a problem instance might specify that a reward is number of boxes in a location minus the remaining liters of fuel in a truck's gas tank)

2.3.3 Relational planning approaches

In an overview of planning systems, Weld [18] outlines a number of approaches to relational planning domains:

- *Progression - Forward-Search*: In this approach the planner starts at the initial world state and progresses the state through legal actions in an attempt to satisfy the goal state. This search can be breadth-first but is more often depth-first using a potentially heuristic ordering on action selection and employing some form of backtracking. Backtracking can be done when the goal is found to be unachievable, at a fixed-depth, or based on a heuristic. This approach is sound and complete under a breadth-first or iterative-deepening search strategy. Note that these methods can be augmented by A* search to heuristically guide it while preserving soundness and completeness.
- *Goal Regression - Backward-Chaining*: In this approach the planner regresses from the goal state in an attempt to find a state that is subsumed by the initial state. On each regression step, the planner selects an action that achieves some element of the goal state and does not remove any other elements of the goal state. Then the state is updated as follows:

  $$RegressedState = Preconditions(Action) \cup (CurrentState - PosEffects(Action))$$

  This approach is sound and complete under a breadth-first or iterative-deepening search strategy. And again, note that these methods can be augmented by A* search to heuristically guide it while preserving soundness and completeness.
- *Plan-space search*: In this approach, the planner searches not in the space of world states but rather in the space of plans. That is, at each search step, it determines some way to modify a plan (e.g. adding or deleting an action at some point in the plan) and then checks if this modification

yielded a successful plan. This approach is clearly sound, and is complete if it can be shown that the entire plan space is searched.

2.3.4 Relational planning systems

There have been literally hundreds of planners for STRIPS and PDDL but a few have gained notoriety in recent years for their superior performance versus other state-of-the-art planners. We briefly cover these systems here:

- *Partial-Order Planning - SNLP and UCPOP*: We briefly start with partial-order planning (POP) although we do not cover this topic in-depth since it has been superceded in performance by a number of recent non-POP systems. In short, however, partial-order planners attempt to search in the plan space by finding a partial-order of plan actions that achieved the goal given the initial state conditions. Most of the work in these systems relied on techniques for maintaining a partial order of plans and ensuring that these did not conflict. Once a non-conflicting partial-order plan is found that achieves all goal conditions from the initial state, it can be reduced to any consistent total-order to yield a successful plan. SNLP was a POP system for STRIPS developed by McAllester et al. [12] and a UCPOP [15] was a POP system for STRIPS domains with universal and conditional effects (i.e. a subset of the PDDL language above).
- *Graphplan*: This planning system by Blum et al. [1] relied on a mutual exclusion analysis of the Plan Graph to make a very efficient goal regression planner. Following is a brief outline of this system:
  - Use a Plan Graph to record mutual exclusions between propositions and and concurrent action executions at a given time step. These exclusions are inferred by an analysis of action properties, such as effect interference and competing needs.
  - One nice property of the Plan Graph is that if a successful plan exists at time t, it will be a subgraph of the Plan Graph
  - Once the system has built the Plan Graph to a given level where none of the goal propositions are mutually exclusive, it then proceeds to do regression search from the goal.
  - Since the Plan Graph efficiently encodes mutual exclusion information, this information can be used to prune the regression search. Additionally, memoization of unachievable goal sets is used to further increase the efficieny of goal regression search.

  At the time of it's introduction, GraphPlan outperformed a number of contemporary planners and many subsequent systems have been based on the ideas introduced by GraphPlan.

- *HSP*: This system was a heuristic forward-search planner by Geffner and Bonet [2]. The approach of this planner is as follows:

  - Take a planning domain P specified in a STRIPS-like language and relax it to a domain P^+ by removing the delete list.
  - Using P^+, efficiently derive a heuristic, but not necessarily exact, distance from a given state.
  - Use this heuristic value of a state to guide a forward search planner.

- *FFPlan*: FFPlan [8] is a system combining aspects of both HSP and GraphPlan that placed first in the domain-independent systems track of the 2002 International Planning Competition. Following is a brief description of the FFPlan algorithm:

  - Use a GraphPlan style algorithm on P^+ to derive an exact distance to the goal.
  - Use breadth-first search from every intermediate state to find an action sequence to a strictly-better state (using the Graph Plan derived P^+ distance to goal as an estimate.

  Recent research [9, 10] has shown that the topology of many planning problems makes the P^+ derived heuristic a good distance estimate to the goal. Exceptions to these types of planning problems are the Blocks World domains which tend to have many local minima due to delete list effects that are not well-modeled but the P^+ heuristic.

2.4 First-order planning

While the STRIPS and PDDL relational planning representations provide a template for any domain instantiation, many planning algorithms that use these representations cannot be applied when the domain instantiation is unknown. In order to plan for *all* domain instantiations, including those with infinitely many objects, it is often easier to convert the STRIPS and PDDL domains to a first-order formalism where there exist well-defined algorithms to handle these cases.

2.4.1 Situation calculus

The situation calculus is a first-order language for axiomatizing dynamic worlds [13]. It's basic language elements consist of actions, situations and fluents:

- *Actions*: Actions are first-order terms consisting of an action function symbol and arguments. For example, an action for placing b_2 on b_1 in the running blocks world example would be given by the term $stack(b_2, b_1)$.

- *Situations*: A situation is a first order term denoting a specific state. The initial situation is usually denoted as s_0 and subsequent situations resulting from action executions are obtained from the $do(a, s)$ function that represents the situation resulting from the execution of action a in state s. For example, the situation resulting from stacking b_2 on b_1 in the initial situation, and then stacking b_3 on b_2 is given by the term $do(stack(b_3, b_2), do(stack(b_2, b_1), s_0))$.

- *Fluents*: A fluent is a relation whose truth value varies from situation to situation. A fluent is simply a relation whose last argument is a situation term. For example, given a correct domain axiomatization and initial state s_0 where b_2 is not on b_1 and both blocks have nothing on them, then $On(b_2, b_1, s_0)$ is false while $On(b_2, b_1, do(stack(b_2, b_1), s_0))$ is true. We do not consider functional fluents here for simplicity of exposition although all of the following ideas still apply in this case.

Domain axiomatization

Now that we've defined the basic domain, we need to describe how one may go about axiomatizing a domain theory. In order to do so, we must first consider how to describe the effects and non-effects of actions. We can first start off by describing effect axioms which characterize how fluents change as a result of actions.

- *Positive Effect Axioms*: Following is an example of a positive effect axiom stating that a successful $stack(b_1, b_2)$ operation from situation s results in $On(b_1, b_2, do(stack(b_1, b_2), s))$.

$$\forall a, b_1, b_2, s. a = stack(b_1, b_2) \wedge (\neg\exists b_3 On(b_3, b_1, s)) \wedge (\neg\exists b_3 On(b_3, b_2, s)) \supset On(b_1, b_2, do(a, s))$$

- *Negative Effect Axioms*: Following are two examples of negative effect axioms. The first states that a successful $stack(b_1, b_3)$ operation from situation s results in $\neg On(b_1, b_2, do(stack(b_1, b_3), s))$ for *any* b_2 that b_1 was previously on. The second states that a successful $unstack(b_1, b_2)$ operation from situation s results in $\neg On(b_1, b_2, do(stack(b_1, b_2), s))$.

$$\forall a, b_1, b_2, s. (\exists b_3 a = stack(b_1, b_3) \wedge On(b_1, b_2, s)) \supset \neg On(b_1, b_2, do(a, s))$$

$$\forall a, b_1, b_2, s. a = unstack(b_1, b_2) \wedge (\neg\exists b_3 On(b_3, b_1, s)) \supset \neg On(b_1, b_2, do(a, s))$$

This takes care of specifying *what changes*, however we have not provided any axioms for specifying *what does not change*, i.e., the so-called *Frame Axioms*. Obviously, if we want to prove anything useful in our theory, we have to specify Frame Axioms. Otherwise, we would *never* be able to infer the properties of a successor or predecessor state for an action as simple as a $NOOP$. However, specifying exactly what does not change in a *compact* manner has been an extremely difficult problem to solve for the situation calculus.

Without covering the various proposals for solutions to the *Frame Problem* (i.e., how to compactly characterizing what fluents change or do not change as the result of an action), we jump directly to Reiter's [17] default solution. In this solution, one must specify all positive and negative effects for a fluent. We use the following normal form for positive effect axioms:

$$\gamma_F^+(\vec{x}, a, s) \supset F(\vec{x}, do(a, s))$$

And we use the following normal form for negative effect axioms:

$$\gamma_F^-(\vec{x}, a, s) \supset \neg F(\vec{x}, do(a, s))$$

So, from our previous specification of positive and negative effects for $On(b_1, b_2, s)$, we have the following positive and negative effect axioms:

$$\begin{aligned}
\gamma_{On}^+(b_1, b_2, a, s) &= a = stack(b_1, b_2) \wedge (\neg\exists b_3 On(b_3, b_1, s)) \wedge (\neg\exists b_3 On(b_3, b_2, s)) \\
\gamma_{On}^-(b_1, b_2, a, s) &= (\exists b_4 a = stack(b_1, b_4) \wedge On(b_1, b_2, s) \wedge (\neg\exists b_3 On(b_3, b_1, s)) \wedge (\neg\exists b_3 On(b_3, b_2, s))) \\
&\quad \vee\ a = unstack(b_1, b_2) \wedge (\neg\exists b_3 On(b_3, b_1, s))
\end{aligned}$$

From this normal form, and explanation closure axioms stating that these are the only effects that hold, Reiter showed that we can build successor state axioms that compactly encode both the effect and frame axioms for a fluent. The format of the successor state axiom for a fluent F is as follows:

$$\begin{aligned}
F(\vec{x}, do(a, s)) &\equiv \Phi_F(\vec{x}, a, s) \\
&\equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s) \qquad (1)
\end{aligned}$$

So, for our running example, the equivalent successor state axiom for $On(b_1, b_2, do(a, s))$ is given in Figure 5. Intuitively, this successor state axiom for $On(b_1, b_2, do(a, s))$ states that a block b_1 is on another block b_2 in a successor state if $b1$ was explicitly stacked on b_2 *or* if b_1 was already on b_2 and b_1 was not stacked on another block *and* b_1 was not explicitly unstacked from b_2.

This very intuitive definition of a successor state axiom makes a translation from STRIPS/PDDL very easy. The algorithm in Figure 6 automates this translation.

Regression definition

The regression of a formula ψ through an action a is a formula ψ' that represents the conditions that must hold prior to performing a, if ψ holds after performing a. The successor state axioms lend themselves to a very natural definition of regression; specifically, if we want to regress a fluent $F(\vec{x}, do(a, s))$ through an action, we need only substitute that action for a in the successor state axiom and replace the fluent with its equivalent pre-action formula $\Phi_F(\vec{x}, a, s)$. In general, we can inductively define the regression for all first-order formulae as follows:

Example Domain & Problem Description	
Fluent Schema:	$\{On(b_1, b_2, s)\}$
Action Schema:	$\{noop(), stack(b_3, b_4), unstack(b_3, b_4)\}$
Domain Instantiation:	None required
Example Initial State:	$\{\exists b_1, b_2\ On(b_1, b_2, s_0)\}$
Example Goal:	$\neg\exists b_1, b_2 On(b_1, b_2, s)$

Successor State Axiom

$$
\begin{aligned}
On(b_1, b_2, do(a, s)) \equiv \quad & a = stack(b_1, b_2) \wedge (\neg\exists b_3 On(b_3, b_1, s)) \wedge (\neg\exists b_3 On(b_3, b_2, s)) \\
\vee \ & On(b_1, b_2, s) \\
& \wedge \ \neg((\exists b_4 a = stack(b_1, b_4) \wedge On(b_1, b_2, s)) \\
& \qquad \wedge (\neg\exists b_3 On(b_3, b_1, s)) \wedge (\neg\exists b_3 On(b_3, b_2, s))) \\
& \wedge \ \neg((a = unstack(b_1, b_2) \wedge (\neg\exists b_3 On(b_3, b_1, s))))
\end{aligned}
$$

Figure 5: A sample first-order sitation-calculus encoding of a blocks world domain and a sample problem. Here we show the final successor state axiom built from the positive and negative effects derived from the PDDL representation of this problem. We only have one successor state axiom because we only have one fluent $On(b_1, b_2)$. Note the compactness of this representation in comparison to all previous deterministic planning formalisms.

- $Regr(F(\vec{x}, do(a, s))) = \Phi_F(\vec{x}, a, s)$
- $Regr(\neg\psi) = \neg Regr(\psi)$
- $Regr(\psi_1 \wedge \psi_2) = Regr(\psi_1) \wedge Regr(\psi_2)$
- $Regr((\exists x)\psi) = (\exists x) Regr(\psi)$

Using the unique names assumption for actions and these regression rules, we can now perform regression on any statement. For example, if we know $\exists x_1, x_2\ On(x_1, x_2, s)$ then we can let $s = do(stack(x_3, x_4), s')$ and see what conditions must have held true for this statement in s' if we regress

1. For every relation in the STRIPS/PDDL domain, add a situation parameter to it's argument list to create a fluent head. Specify variable bindings for each of the fluent head's non-situation arguments and $do(a, s)$ for the situation argument (e.g. $On(a, b) \rightarrow$ Fluent head: $On(v_1, v_2, do(a, s))$).

2. Initialize an empty set of positive and negative effects for each fluent (e.g. $\gamma^+_{On} = \gamma^-_{On} = \emptyset$).

3. For each action $act(va_1, \ldots, va_n)$, analyze its effects list:

   - If an effect mentions a non-negated fluent:
     - Find the most general unifier (mgu) of the action effect (minus universal quantifiers) and fluent head.
     - Substitute the fluent head variables from this mgu in the action preconditions.
     - If the effect is conditional, also conjoin the unified condition with the preconditions.
     - Conjoin these unified preconditions with $a = act(va_1, \ldots, va_n)$ where act represents this action.
     - Existentially quantify any non-action, non-situation variables remaining in this formula.
     - Add the resulting formula to the positive effects (γ^+_F) for the fluent head.
   - Do the same for a negated fluent except add the resulting formula to the negative effects (γ^-_F) for the fluent head.

4. Given the positive and negative effect sets for each fluent, build a successor state axiom for that fluent using equation 1.

Figure 6: An automated algorithm for translating from STRIPS/PDDL to the situation calculus using Reiter's successor state axioms. To get an intuitive understanding for this algorithm, observe how the positive and negative effects for $On(a, b)$ were derived from the relational PDDL model in the running example.

it through the $stack(x_3, x_4)$ action. Our result:

$$
\begin{aligned}
&Regr(\exists x_1, x_2\ On(x_1, x_2, do(stack(x_3, x_4), s'))) \\
&= \exists x_1, x_2\ Regr(On(x_1, x_2, do(stack(x_3, x_4), s'))) \\
&= \exists x_1, x_2\ \Phi_{On}(x_1, x_2, do(stack(x_3, x_4), s')) \\
&= \exists x_1, x_2\ On(x_1, x_2, s') \\
&\qquad \wedge \neg((x_3 = x_1) \wedge (\exists b_4 = x_4) \wedge On(x_1, x_2, s) \wedge (\neg\exists b_3 On(b_3, x_1, s)) \wedge (\neg\exists b_3 On(b_3, x_2, s)))) \\
&\quad \vee (x_3 = x_1) \wedge (x_4 = x_2) \wedge (\neg\exists b_3 On(b_3, x_1, s')) \wedge (\neg\exists b_3 On(b_3, x_2, s'))
\end{aligned}
$$

Although this is a little hard to read at first, what it says is very intuitive: If some block x_1 was on some block x_2 after the operation $stack(x_3, x_4)$, then either this same state held in the previous situation *and* a stack operation did not move x_1 to another block *or* the blocks we stacked were actually x_1 and x_2 and both were clear before the stack operation.

Planning approaches

With these regression definitions, we now have very straightforward algorithms for performing both forward and regression search in a first-order domain:

- *Forward Search*: If we have a situation calculus description of an initial state and goal then we need only postulate actions $a_1, \ldots, a_n$ and determine if the regression of the goal formula through the postulated action sequence $do(a_n, do(\ldots(do(a_1))))$ is satisfied by the initial state. For a complete forward-search algorithm, we simply need to ensure a breadth-first or iterative-deepening approach to augmenting the action sequence. However, this approach is generally inefficient in practice without more search guidance due to the combinatorial explosion of action sequences. This bottleneck is the motivation for GOLOG that will be discussed in a following section.

- *Regression Search*: Since the successor state axioms are directly amenable to regression, another planning method would be to do goal regression, postulating actions to regress through and checking if the initial state implies the regressed state. This method might provide a slightly more efficient way of searching all action sequences (especially under an iterative-deepening strategy) by backtracking to previously regressed states without having to start regression from the goal on each additional search step.

We end our coverage of the situation calculus on one final note which was the motivation for taking this approach to planning. With the translation from a STRIPS or PDDL planning domain specification to the situation calculus, we can now plan without having a fixed domain instantiation. We can even plan in domains with infinitely many objects! And we can now plan with our initial and goal states defined using any logical description, including disjunction. Of course, with such expressive power in a planning formalism, there is no guarantee that search will be tractable. But, there is some reason to believe that planning in many first-order domains may be tractable and furthermore, and the ability to abstract over actions using existential quantification means that this formalism may outperform many planners in the context of large numbers of domain objects.

However, most importantly, we now have a full first-order axiomatization of many common planning domains. This formalism serves as the groundwork for a number of additional generalizations. (See *Relational and First-Order Decision Theoretic Planning: Foundations and Future Directions* for some of this work.)

3 Concluding remarks

We have concluded a basic overview of deterministic planning in a decision-theoretic framework. All of these ideas will generalize to more expressive frameworks and thus it is important to understand these representations and the ways in which the various algorithms exploit them for efficient planning. The paper *Relational and First-Order Decision Theoretic Planning: Foundations and Future Directions* will generalize these representations to stochastic domains and introduce the additional notions of program constraints and hierarchy.

References

[1] Avrim L. Blum and Merrick L. Furst. Fast planning through graph analysis. In *IJCAI 95*, pages 1636–1642, Montreal, 1995.

[2] Blai Bonet and Hector Geffner. Heuristic search planner 2.0. *The AI Magazine*, 22(1):77–80, 2001.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[4] Thomas H. Cormen, Charles E. Lierson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[5] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *AI Journal*, 2:189–208, 1971.

[6] M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains., 2001.

[7] P. E. Hart, N. J. Nilsson, , and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[8] J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search, 2001.

[9] Jörg Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 453–458, Seattle, Washington, USA, August 2001.

[10] Jörg Hoffmann. Local search topology in planning benchmarks: A theoretical analysis. In *Proceedings of the 6th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-02)*, Toulouse, France, April 2002. 379-387.

[11] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[12] David McAllester and David Rosenblitt. Systematic nonlinear planning. In *AAAI 91*, pages 634–639, Anaheim, 1991.

[13] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., 1968, pages 410-417.

[14] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL—The planning domain definition language, 1998.

[15] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for adl. In *KR 92*, pages 103–114, Cambridge, MA, 1992.

[16] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming.* Wiley, New York, 1994.

[17] Raymond Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation (Papers in Honor of John McCarthy)*, pages 359–380. Academic Press, San Diego, 1991.

[18] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, Winter 1994:27–61, 1994.