Online appendices for the paper

# *Domain-Independent Cost-Optimal Planning in ASP*

published in Theory and Practice of Logic Programming

David Spies, Jia-Huai You, Ryan Hayward

*University of Alberta, Edmonton, Canada*

*submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003*

## Appendix A Actions Happen As Soon As Possible

Here, we give the rule that says that actions always happen as soon as it is possible, but we must be careful. There are quite a few things which might prevent an action from occurring any sooner. If we leave any out, we risk rendering the problem unsolvable. For an action to be able to occur at the previous time-step, its preconditions must hold at the previous time-step, its delete-effects should not be used at the previous time-step, and its used add-effects should not be deleted at the previous time-step. There are a few other conditions which at first appear to be necessary (such as its preconditions must not be deleted at the previous time-step), but upon further consideration you may notice that all of these are redundant if our goal is specifically to prevent the action from occurring *at the current time-step*. We must borrow our definition of `deleted/2` from the modified encoding of rule 4 in Section 2 of the paper (see **Encoding Reduction** of that section) and additionally add a similar definition for `used/2`.

```
deleted(F,K) :- happens(A,K); del(A F).
used(F,K) :- happens(A,K); pre(A,F); not preserving(A).

:- happens(A,K); K > 0; not preserving(A);
   holds(F,K-1) : pre(A,F); not used(F,K-1) : del(A,F);
   not deleted(F,K-1) : add(A,F), holds(F,K).
```

How does this defeat the 100-switch-scenario? Remember that the 100 switches exponentially increased the plan length because the planner may choose to flip some switches but not others to achieve one state, but then flip those other switches later to achieve an alternative state.

For every switch this rule boils down to, "if we want to flip switch $i$ at time $t$, then we must also flip switch $i$ at time $t-1$ as well". Otherwise the solution fails this rule since the switch flip *could have occurred* one action sooner.

Under this rule, we have made it impossible to achieve more than 100 unnecessary states within the 100-switch problem. At each step where we do not make progress somewhere else, we must choose at least one switch to stop flipping (if we toggle the exact same set of switches as in the last step, we revert to the same overall state as two steps earlier which is forbidden by the layered 'make-progress' rule).

More generally, one can see that wherever a planning problem has multiple independent parts, this rule forces all the parts to proceed independently and not stall needlessly. However, the rule still has some gaps.

- Even adding a linear number of unnecessary steps is suboptimal. All the switches are independent so we really should not be adding more than one layer regardless of how many switches there are.
- The switch scenario is contrived to make our solution look better than it is. One can easily see that by using three-state switches rather than two-state switches (where each state is reachable from the other two), it is still possible to construct exponential-length plans even with this restriction in place. This is because we can still reach an exponential number of states while continually changing every switch at every time-step.

The above issues are addressed by Definition 3.2 of the main paper where we give the 'make-progress' notion which is used by the stepless planner.

## Appendix B Proofs

### *B.1 Correctness of Mutex Action Rules*

We show that the following handle all mutex action constraints that we care about strictly via unit propagation (labels are added for reference):

```
1. used_preserved(F,K) :- happens(A,K); pre(A,F); not del(A,F).
2. deleted_unused(F,K) :- happens(A,K); del(A,F); not pre(A,F).
3. :- {used_preserved(F,K); deleted_unused(F,K);
       happens(A,K) : pre(A,F), del(A,F)} > 1; valid_at(F,K).

4. deleted(F,K) :- happens(A,K); del(A,F).
5. :- holds(F,K); deleted(F,K-1).
```

given the existing rules:

```
6. holds(F,K) :- happens(A,K); pre(A,F).
7. :- holds(F,K); holds(G,K); mutex(F,G).
```

*Proof*
Suppose $A$ and $B$ are mutex actions because $A$ deletes fluent $F$ and $B$ has $F$ as a precondition. The proof is based on a case analysis.

*Case 1*. $A$ does not have $F$ as a precondition and $B$ does not delete $F$. If we select $happens(A,K)$ then by unit propagation we have $deleted\_unused(F,K)$ (2) and then $\neg used\_preserved(F,K)$ (3) and then $\neg happens(B,K)$ (1). We can write this as:

$$happens(A,K) \Rightarrow_2 deleted\_unused(F,K) \Rightarrow_3 \neg used\_preserved(F,K) \Rightarrow_1 \neg happens(B,K)$$

*Case 2*. $A$ has $F$ as a precondition and $B$ does not delete $F$. Then,

$$happens(A,K) \Rightarrow_3 \neg used\_preserved(F,K) \Rightarrow_1 \neg happens(B,K)$$

*Case 3*. $A$ does not have $F$ as a precondition, $B$ deletes $F$. Then,

$$happens(A,K) \Rightarrow_2 deleted_unused(F,K) \Rightarrow_3 \neg happens(B,K)$$

*Case 4*. $A$ has $F$ as a precondition, $B$ deletes $F$. Then,

$$happens(A,K) \Rightarrow_3 \neg happens(B,K)$$

In the case of conflicting effects ($A$ adds $F$, $B$ deletes $F$), there's only a conflict when the conflicted fluent "holds". So in fact we actually only care about a unit propagation mutex between $holds(F,K+1)$ and $happens(B,K)$. $A$ is not relevant (as mentioned in the Section 2 of the main paper, this is where our mutex rules differ from those of (Blum and Furst 1997)). So

$$holds(F,K+1) \Rightarrow_5 \neg deleted(F,K) \Rightarrow_4 \neg happens(B,K)$$

Conversely:

$$happens(B,K) \Rightarrow_4 deleted(F,K) \Rightarrow_5 \neg holds(F,K+1)$$

Finally, suppose $A$ and $B$ have mutex preconditions ($pre(A,F)$; $pre(B,G)$; $mutex(F,G)$). Then (again by unit propagation),

$$happens(A,K) \Rightarrow_6 holds(F,K) \Rightarrow_7 \neg holds(G,K) \Rightarrow_6 \neg happens(B,K)$$

Thus no explicit mutex action rules are needed beyond this. Thanks to unit propagation, the effect is the same. □

### *B.2 Soundness and Completeness of Variant-II Solver*

The make-progress rule here refers to the layered 'make-progress' rule.

*Lemma Appendix B.1*
Given a planning problem solution (plan) $P$, we can find a solution $P^*$ which satisfies the 'make-progress' rule such that $C(P^*) \leq C(P)$ ($C(P)$ is the sum action cost of plan $P$) and $n(P^*) \leq n(P)$ ($n(P)$ is makespan of plan $P$).

*Proof*
If $P$ satisfies the 'make-progress' rule, then $P^* = P$ and we're done. Otherwise there exists a pair of layers $l_1$ and $l_2$ such that $S_P(l_2) \subset S_P(l_1)$ ($S_P(l)$ is the set of fluents which hold at layer $l$ in $P$). We can reduce $P$ by "removing" all the layers between $l_1$ (exclusive) and $l_2$ (inclusive) along with any actions that occur on those layers. Repeat this process until no such pair of layers exists. Since every iteration removes at least one layer and $P$ has a finite number of layers, it follows that this will eventually terminate and the resulting plan $P^*$ will have no such pair and thus satisfy the 'make-progress' rule. □

*Corollary Appendix B.1*
If a planning problem $Q$ is solvable, then it has an optimal solution which "makes progress" according to the rule.

*Lemma Appendix B.2*
Given a plan $P^*$ which satisfies the 'make-progress' rule, any prefix of that plan $P_{..k}$ also satisfies the 'make-progress' rule.

*Proof*
This is trivial: If there exists no pair of layers in $P^*$ with some property, then of course there exists no pair of layers with that property in any prefix of $P^*$. □

*Lemma Appendix B.3*
Given a planning problem $Q$ with solution $P$, the delete-free relaxation of $Q$ has a solution $P^*$, the set of actions that occur in $P$ (ordered according to the first time they are taken in $P$). It follows that $C(P^*) \leq C(P)$ (note: this is not strictly equal since actions in $P$ may be taken more than once and incur their cost every time).

*Proof*
Also trivial: If a precondition or goal is satisfied at some layer in $P$, then it must also be satisfied by that time in the delete-free relaxation since all the same actions have occurred. □

*Theorem Appendix B.2*
(Variant-II Soundness Theorem) If $Q$ has an optimal solution $P$ with cost $C$ and makespan $n$, then for any $k \leq n$, at makespan $k$ the Variant-II solver will find a relaxed plan with cost $\leq C$.

*Proof*
This can be established by constructing a solution at makespan $k$ with cost $C(P,k) \leq C$. By Corollary Appendix B.1, we may assume WLOG that $P$ makes progress. First, set the 'subgoal' fluents to be $S_P(k)$. The fluents and actions in the normal part of the program match $P$ exactly. By Lemma Appendix B.2 these will satisfy the 'make-progress' rule. Finally, the suffix layer is solved by the set of actions that occur in the suffix $P_{k..}$ which solves the delete-free relaxation by Lemma Appendix B.3. $C(P,k)$ here is the sum of two parts; the solution to the prefix $P_{..k}$ which is the same as $P_{..k}$ (and therefore has the same cost as $P_{..k}$), and the relaxed solution to the suffix $P_{k..}$, which by Lemma Appendix B.3 is no greater than in $C(P_{k..})$ so $C(P,k) \leq C$. It follows that the *optimal solution* at makespan $k$ is at most $C(P,k)$ (and so is transitively $\leq C$). □

*Theorem Appendix B.3*
(Variant-II Completeness Theorem) If a planning problem $Q$ has no solution, the Variant-II solver will eventually produce an UNSAT instance.

*Proof*
A plan which makes progress cannot encounter the same state twice and there are a finite number of possible states. This means that the length of a plan which makes progress is bounded by the number of possible states. Thus, for a sufficiently-large makespan, the make-progress rule is unsatisfiable. □

### *B.3 Soundness and Completeness of Stepless Planner*

*Theorem Appendix B.4*
(Stepless Soundness Theorem) All plan costs produced by the stepless planner are lower bounds on the cost of the true optimal plan.

*Proof*
*Case 1.* There are sufficient occurrences of fluents and actions to construct the optimal plan: In this case, these occurrences constitute a solution so the minimal solution to this instance will have a cost which is no greater.

Otherwise pick an arbitrary sequentialization of the optimal plan. Now we have two cases:

*Case 2.* The first missing occurrence in the plan is an action occurrence. In this case, consider the plan cut where all actions up to this point occur (since we have enough occurrences) and the state at this cut form the subgoals. By Lemma Appendix B.3 again, we can put the remainder of the plan into the suffix layer. The missing action occurrence will be a starting action and that action will also be saturated so the saturation rules are satisfied.

*Case 3.* The first missing occurrence in the plan is a fluent occurrence. In that case, use the plan cut up to (but not including) the action occurrence which *adds* this fluent occurrence (putting the remainder of the plan including the adding action into the suffix layer by Lemma Appendix B.3). This action will be a starting action so the added fluent will be a saturated starting fluent which also satisfies the saturation requirement. □

The 'make progress' rule below refers to the stepless 'make-progress' rule.

*Lemma Appendix B.4*
Given a collection of action occurrences in a plan $P$, they may be ordered such that for each consecutive pair there is an $(s,t)$-cut which puts the first one on the $s$-side and the other one on the $t$-side.

*Proof*
Pick a serialization of $P$. Order the actions according to the sub-order in that serialization. Place the cuts anywhere between them. □

*Lemma Appendix B.5*
For any action $A$ there exists a (finite) count $k$ such that the stepless planner will not add more than $k$ occurrences of $A$.

*Proof*
In order to add another occurrence of $A$, it must be the case that $A$ is saturated in some plan that makes progress. This means that $k$ occurrences of $A$ are used in the plan. Order the occurrences of $A$ as $(A_0, A_1, A_2, ..., A_k)$ such that there exists a cut between each consecutive pair (by Lemma Appendix B.4; also because of our symmetry-breaking rule this can be the natural ordering by action index). It follows by the 'make-progress' rule that the state at each of the cuts must be distinct from the state at any other cut. Thus, $k$ cannot exceed the number of possible states (which is finite). □

*Corollary Appendix B.5*
For any fluent $F$ there exists a finite count $k$ such that the stepless planner will not add more than $k$ occurrences of $F$.

*Proof*
When $F$ is saturated, each occurrence must be caused by some action occurrence. Since Lemma Appendix B.5 bounds the number of action occurrences which a progress-making plan can have, it follows that the number of fluent occurrences is also bounded. □

*Theorem Appendix B.6*
(Stepless Completeness Theorem) The stepless planner will eventually find the solution if it exists or produce an UNSAT instance if it doesn't.

*Proof*
In the proof of Theorem Appendix B.4 we show that if a plan exists, then there will always be a solution to any stepless instance constructed from that problem (either the plan itself if there are enough occurrences, or a partial plan with a suffix layer since some fluent or action does not have enough occurrences and can therefore be saturated). Lemma Appendix B.5 and Corollary Appendix B.5 together ensure that the process of alternately solving instances and then including any saturated fluents or actions will eventually halt (since there are at most a finite number of actions and fluents that can be included before finding a progress-making plan which saturates something becomes impossible). □

## Appendix C Delete-Free Planning

Recall that delete-free planning can be modeled as a graph problem: Given a directed bipartite graph $G = (X, Y, E)$ with weights on $X$ and a goal set $Y_F \subseteq Y$, find a minimum *acyclic* subgraph $G^* = (X^*, Y^*, E^*)$ such that

1. $Y_F \subseteq Y^*$
2. If $x \in X^*$ and $(y, x) \in E$, then $y \in Y^*$ and $(y, x) \in E^*$
3. For all $y \in Y^*$, $E^*$ contains at least one edge $(x, y)$ (and $x \in X^*$).

Recall its connection to delete-free planning: $X$ is the set of actions, and $Y$ is the set of fluents, the $(x, y)$ edges are add-effects and the $(y, x)$ edges are preconditions. $Y_F$ is the goal set and the initial set has been removed (together with all corresponding preconditions) from the graph. Rule 1 means the goal fluents must be true. Rule 2 means an action implies its preconditions. Rule 3 means every fluent must have a causing action. The graph must be acyclic to ensure the actions can occur in some order. The entire problem of a delete-free planning problem can be encoded in a single "one-shot" ASP program. This is possible because there is no incentive to ever take an action or cause a fluent more than once. As soon as any fluent is true, it is permanently true.

Let us write an ASP program to solve the problem of delete-free planning. Here we do not worry about makespan; thanks to the NP-ness of delete-free planning, we can solve this problem all in one go. Note that we can trivially add an extra rule to make our plans cost-optimal. (The code below is a complete ASP program: run it on the problem and get an optimal solution.)

```
holds(F) :- init(F).
{happens(A)} :- holds(F) : pre(A,F); action(A).
holds(F) :- add(A,F); happens(A).
:- goal(F); not holds(F).
:~ happens(A); cost(A,C).[C,A]
```

We have again encountered a five-line program which, magnificently, does everything. It handily encodes the problem of delete-free planning. To be supported, an action's preconditions must hold independently of that action itself and a fluent's causing action must not require that fluent.

However, we have lost something by encoding planning "from the ground up". Earlier, we mentioned how the state-space for solving a planning problem was reduced when we started

from the goal, and built support up backwards. That is, an action should only happen if something needs it. Let us fix that.

If we build up the plan backwards, we must be careful to ensure that the actions can happen in some order. As such, we need to explicitly include atoms whose only purpose is to ensure supportedness.

```
% Delete-free planning
holds(F) :- goal(F).
{happens(A) : add(A,F)} >= 1 :- holds(F), not init(F).
holds(F) :- pre(A,F); happens(A).

supportFluent(F) :- init(F); holds(F).
supportAct(A) :- supportFluent(F) : pre(A,F), holds(F); happens(A).
supportFluent(F) :- supportAct(A); happens(A); add(A,F); holds(F).

:- holds(F); not supportFluent(F).
:~ happens(A); cost(A,C).[C,A]
```

Now the first three rules encompass neededness. We add actions and fluents in working backwards from the goal until we encounter the initial fluents. Meanwhile the second three rules indicate whether an action or fluent is supported. Together, with the restriction that all the fluents must be supported, these guarantee a correct plan. Essentially, for an action or fluent to occur, it now must have support both from the bottom and from the top.

## Appendix D Stepless Planner with Suffix Layer

Requires an external program to detect which fluents and actions are saturated each time the suffix layer is used and feed in more occurrences.

```
is(fluentOcc(F,1)) :- fluent(F).
is(actOcc(A,1)) :- action(A).
is(fluentOcc(F,0)) :- init(F).


% ======================== Problem Description ========================

% Helper function to recognize subsequent occurrences of fluent/action.
nextOcc(fluentOcc(F,0),fluentOcc(F,1)) :- fluent(F).
nextOcc(fluentOcc(F,M),fluentOcc(F,M+1)) :- is(fluentOcc(F,M)).
nextOcc(actOcc(A,N),actOcc(A,N+1)) :- is(actOcc(A,N)).

% Fluent occurrence which is not initial (M > 0) must have exactly one
% causing action
{causes(actOcc(A,N),fluentOcc(F,M)) : add(A,F), is(actOcc(A,N))}=1 :-
  holds(fluentOcc(F,M)); M > 0.
% If an action causes a fluent, it happens.
happens(AO) :- causes(AO,_).
```

```
% An action cannot cause more than one occurrence of the same fluent.
:- {causes(AO,fluentOcc(F,M))} > 1; is(AO); fluent(F).

% For each precondition an action occurrence has, some occurrence of
% that fluent must permit it.
{permits(fluentOcc(F,M),actOcc(A,N)) : is(fluentOcc(F,M))}=1 :-
  happens(actOcc(A,N)); pre(A,F).
% A fluent occurrence which permits an action must hold.
holds(FO) :- permits(FO,_).
% A fluent which is used to satisfy a subgoal condition "permits" it.
% For each subgoal condition, exactly one occurrence of that fluent
% permits it.
{permits(fluentOcc(F,M),subgoal(F)) : is(fluentOcc(F,M))}=1 :-
  subgoal(F).
% A fluent which permits a subgoal condition cannot be deleted.
:- deleted(FO); permits(FO,subgoal(_)).

% An occurrence of an action deletes an occurrence of a fluent if
% it permits it and that action has the fluent as a delete effect.
deletes(actOcc(A,N),fluentOcc(F,M)) :-
  permits(fluentOcc(F,M),actOcc(A,N)); del(A,F).
% No fluent may be deleted by more than one action.
:- {deletes(_, FO)} > 1; is(FO).

% An action which deletes a fluent, but doesn't have it as a precondition
% follows some occurrence of that fluent. Can possibly follow occurrence
% index 0 even if the fluent is not an initial fluent (indicating this
% action occurs before any occurrence of that fluent).
{follows(actOcc(A,N),fluentOcc(F,M)) : holds(fluentOcc(F,M));
  follows(actOcc(A,N),fluentOcc(F,0))}=1 :-
  del(A,F); not pre(A,F); happens(actOcc(A,N)).

% Fluent occurrences 0 which aren't initial fluents count as "deleted".
deleted(fluentOcc(F,0)) :- fluent(F); not init(F).
% A fluent is deleted if something deletes it.
deleted(FO) :- deletes(_, FO).
% A fluent is deleted if something follows it.
deleted(FO) :- follows(_, FO).

% Weak constraint charging the cost of an action occurrence.
:~ happens(actOcc(A,N)); cost(A,V).[V,A,N]

% An occurrence of a fluent doesn't hold if its previous occurrence
% doesn't hold.
:- holds(fluentOcc(F,M+1)); not holds(fluentOcc(F,M));
   is(fluentOcc(F,M)); M > 0.
```

```
% An occurrence of an action doesn't happen if its previous occurrence
% didn't happen.
:- happens(BO); not happens(AO); nextOcc(AO,BO).


% ======================= Plan Event Graph ============================

% Events in the graph; these will be grouped into vertices
event(start(FO)) :- holds(FO).
event(end(FO)) :- holds(FO).
event(end(fluentOcc(F,0))) :- fluent(F).
event(AO) :- happens(AO).
% subgoals are events
event(subgoal(F)) :- subgoal(F).

% Triggering actions
% The start of a fluent by its causing action.
actionTriggers(AO,start(FO)) :- causes(AO,FO).
% The end of a fluent by its deleting action.
actionTriggers(AO,end(FO)) :- deletes(AO,FO).

% Vertices
% If no action triggers an event, then it gets a vertex by itself.
vertex(V) :- event(V); not actionTriggers(A,V) : is(A).
% Otherwise it belongs to the vertex for its trigger action.
inVertex(E,V) :- actionTriggers(V,E).
% Every event which is the name of a vertex belongs to that vertex.
inVertex(V,V) :- vertex(V).

% Graph edges
% A fluent ends after it starts
edge(start(FO),end(FO)) :- holds(FO).
% If a fluent permits an action, then the action happens after
% the start of the fluent
edge(start(FO),AO) :- permits(FO,AO).
% If a fluent permits an action but the action doesn't delete the
% fluent, then the action happens before the end of the fluent.
edge(AO,end(FO)) :- permits(FO,AO); not deletes(AO,FO).

% An action happens after the fluent it follows
edge(end(FO),AO) :- follows(AO,FO).
% but before the next occurrence
edge(AO,start(GO)) :- follows(AO,FO); nextOcc(FO,GO); holds(GO).
% The start of the next occurrence of a fluent happens after the
% end of the previous occurrence
edge(end(FO),start(GO)) :- holds(GO); nextOcc(FO,GO).
```

```
% The next occurrence of an action happens after the previous
% occurrence
edge(AO,BO) :- happens(AO); happens(BO); nextOcc(AO,BO).

% And now we use stable models to assert that the graph is acyclic; sup(X)
% indicates that X has acyclic support going back to the root of the graph.

% The input for a given event has support if all events joined
% by any incoming edge have support.
sup(in(E)) :- sup(D) : edge(D,E); event(E).
% A vertex has support if all of its events' inputs have support.
sup(V) :- sup(in(E)) : inVertex(E,V); vertex(V).
% An event has support if its vertex has support.
sup(E) :- sup(V); inVertex(E,V).
% Every vertex must have support.
:- vertex(V); not sup(V).


% ====================== Strong Minimality ============================

% A counterexample to strong minimality consists of two cuts, cut1 and cut2.
cut(cut1; cut2).

% For each vertex V and each cut C, V is on either the s-side or
% the t-side of V. Note this rule is disjunctive.
onSideOf(V,s,C) | onSideOf(V,t,C) :- vertex(V); cut(C).
% An event belongs to the cut side of its vertex.
onSideOf(E,X,C) :- inVertex(E,V); onSideOf(V,X,C).
% Any subgoal is always on the t-side of cut2.
onSideOf(subgoal(F),t,cut2) :- subgoal(F).
% If there's a directed edge from D to E, but D is on the t-side
% and E is on the s-side, this is not a cut (invalidating this
% counterexample to strong minimality).
not_counterexample :- edge(D,E); onSideOf(D,t,C); onSideOf(E,s,C).
% If a fluent starts on the s-side of cut2 and ends on the t-side,
% then it "holds over" cut2.
holdsOver(FO,cut2) :-
  onSideOf(start(FO),s,cut2); onSideOf(end(FO),t,cut2).
% Similarly if it starts and ends on the same side of cut1, then it
% doesn't hold over cut1.
not_holdsOver(FO,cut1) :-
  onSideOf(start(FO),X,cut1); onSideOf(end(FO),X,cut1).
% Action occurrence AO is not between cut1 and cut2 if it's on the
% s-side of cut1 or the t-side of cut2.
not_betweenCuts(AO) :- onSideOf(AO,s,cut1).
not_betweenCuts(AO) :- onSideOf(AO,t,cut2).
```

```
% If no action occurs between the two cuts, then this is not a counterexample.
not_counterexample :- not_betweenCuts(AO) : happens(AO).
% If there exists a fluent for which some occurrence holds over cut2,
% but no occurrence holds over cut1, then this is not a counterexample.
not_counterexample :-
  holdsOver(fluentOcc(F,_),cut2);
  not_holdsOver(fluentOcc(F,M),cut1) : holds(fluentOcc(F,M)).

% There should be no counterexample (sorry for the triple negative).
:- not not_counterexample.
% If this is not a counterexample, all atoms must hold.
onSideOf(V,s,C) :- vertex(V); cut(C); not_counterexample.
onSideOf(V,t,C) :- vertex(V); cut(C); not_counterexample.
```

To see why this works, imagine that we find a plan which satisfies these rules. Consider the candidate model which includes the atom *not_counterexample*. Because all the rules here are strictly positive, the last two rules force all the others to hold. *Any* other solution is a strict subset. Therefore if some *other* solution exists which does not include the *not_counterexample* atom, then a model including it would be rejected for not being minimal. It follows that the *only* models which include *not_counterexample* (and satisfy the triple-negative rule) are those for which no counterexample exists.

```
% ======================== Suffix Layer ==============================

% All goal fluents hold in the suffix layer.
suffix(holds(F)) :- goal(F).
% If a fluent holds in the suffix layer, either some action causes it
% or it is a subgoal.
{subgoal(F); suffix(causes(A,F)) : add(A,F)} = 1 :- suffix(holds(F)).
% If an action causes a fluent in the suffix, it happens.
suffix(happens(A)) :- suffix(causes(A,_)).
% If an action occurs in the suffix layer, then all of its
% preconditions hold in ths suffix layer
suffix(holds(F)) :- suffix(happens(A)); pre(A,F).

% If any action happens in the suffix layer, then we are using it.
useSuffix :- suffix(happens(_)).

% A fluent is supported in the suffix if it's a subgoal
suffix(sup(holds(F))) :- subgoal(F).
% An action is supported in the suffix if all of its preconditions are
suffix(sup(happens(A))) :-
  suffix(sup(holds(F))) : pre(A,F); suffix(happens(A)).
% A fluent is supported in the suffix if its causing action is
suffix(sup(holds(F))) :- suffix(sup(happens(A))); suffix(causes(A,F)).
```

```
% No action happens in the suffix without support
:- suffix(happens(A)); not suffix(sup(happens(A))).
% No fluent holds in the suffix without support
:- suffix(holds(F)); not suffix(sup(holds(F))).

% Actions that happen in the suffix layer impose their cost.
:~ suffix(happens(A)); cost(A,V).[V,A,suffix]
% Very weak preference to avoid using the suffix layer.
:~ useSuffix.[1@-1]

% ========================== Saturated ================================

% A fluent is saturated if all occurrences of it hold (besides the 0th).
saturated(fluent(F)) :-
  holds(fluentOcc(F,M)) : is(fluentOcc(F,M)),M>0; fluent(F).
% An action is saturated if all occurrences of it happen.
saturated(action(A)) :-
  happens(actOcc(A,N)) : is(actOcc(A,N)); action(A).

% If an action happens in the suffix layer and all of its preconditions
% are subgoals, we designate it a "starting" action.
suffix(start(action(A))) :- subgoal(F) : pre(A,F); suffix(happens(A)).
% Any fluent caused by a starting action is designated a "starting" fluent.
suffix(start(fluent(F))) :- suffix(start(action(A))); suffix(causes(A,F)).

% Guarantees that some starting action or fluent will be saturated.
:- useSuffix; not saturated(X) : suffix(start(X)).

% ======================================================================
#show causes/2.  #show deletes/2.  #show happens/1.
#show holds/1.  #show permits/2. #show follows/2.
#show suffix(happens(A)) : suffix(happens(A)).
```

## Appendix E An Example of Stepless Planning: Bridge Crossing

We will use a modified version of the bridge-crossing problem from (Eiter et al. 2003).

In the original problem, we have four people, Joe, Jack, William, and Averell, needing to cross a bridge in the middle of the night. The bridge is unstable, so at most two people can cross at a time. The four only have a single lantern between them and since there are planks missing it is unsafe to cross unless someone in your party is carrying the lantern. In the original problem, it takes Joe 1 minute to run across, Jack 2 minutes, William 5 minutes and Averell 10. When two people cross together they must go at the slower speed of the two. What's the fastest all four can get across considering that after each crossing somebody needs to cross back carrying the lantern?

In our version we'll add two more people Jill and Candice for a total of six people. Jill takes 3

minutes to cross and Candice takes 20 (the original problem doesn't make for a very interesting example of stepless planning).

We can now phrase the problem as follows:

```
person(joe;jack;jill;william;averell;candice)
side(side_a;side_b)
crossing_time(joe,1).
crossing_time(jack,2).
crossing_time(jill,3)
crossing_time(william,5).
crossing_time(averell,10).
crossing_time(candice,20).

fluent(lantern_at(S)) :- side(S).
fluent(at(P,S)) :- person(P); side(S).

init(at(P,side_a)) :- person(P).
init(lantern_at(side_a)).
goal(at(P,side_b)) :- person(P).

action(cross_alone(P,FROM,TO)) :-
  person(P); side(FROM); side(TO); FROM != TO.
pre(cross_alone(P,FROM,TO),at(P,FROM)) :-
  action(cross_alone(P,FROM,TO)).
pre(cross_alone(P,FROM,TO),lantern_at(FROM)) :-
  action(cross_alone(P,FROM,TO)).
add(cross_alone(P,FROM,TO),at(P,TO)) :-
  action(cross_alone(P,FROM,TO)).
add(cross_alone(P,FROM,TO),lantern_at(TO)) :-
  action(cross_alone(P,FROM,TO)).
del(cross_alone(P,FROM,TO),at(P,FROM)) :-
  action(cross_alone(P,FROM,TO)).
del(cross_alone(P,FROM,TO),lantern_at(FROM)) :-
  action(cross_alone(P,FROM,TO)).
cost(cross_alone(P,FROM,TO),C) :-
  action(cross_alone(P,FROM,TO)); crossing_time(P,C).

action(cross_together(P_SLOW,P_FAST,FROM,TO)) :-
  side(FROM); side(TO); FROM != TO;
  crossing_time(P_SLOW,T1); crossing_time(P_FAST,T2); T2 < T1.
pre(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_SLOW,FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
pre(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_FAST,FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
pre(cross_together(P_SLOW,P_FAST,FROM,TO),lantern_at(FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
```

```
add(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_SLOW,TO)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
add(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_FAST,TO)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
add(cross_together(P_SLOW,P_FAST,FROM,TO),lantern_at(TO)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
del(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_SLOW,FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
del(cross_together(P_SLOW,P_FAST,FROM,TO),at(P_FAST,FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
del(cross_together(P_SLOW,P_FAST,FROM,TO),lantern_at(FROM)):-
  action(cross_together(P_SLOW,P_FAST,FROM,TO)).
cost(cross_alone(P_SLOW,P_FAST,FROM,TO),C) :-
  action(cross_alone(P_SLOW,P_FAST,FROM,TO)); crossing_time(P_SLOW,C).
```

Let's run the stepless solver on this. On the first iteration we input one occurrence of every fluent and every action as well as a bonus zero'th occurrence of each initial fluent.

```
is(fluentOcc(F,1)) :- fluent(F).
is(actOcc(A,1)) :- action(A).
is(fluentOcc(F,0)) :- init(F).
```

It gives back a directed graph of action and fluent dependencies. After toplogically sorting the graph and throwing out everything that isn't an action we have the plan:

```
cross_together(jack,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
suffix cross_together(candice,averell,side_a,side_b)
suffix cross_alone(joe,side_a,side_b)
suffix cross_together(william,jill,side_a,side_b)
cost: 29
```

In the suffix layer when Candice and Averell cross from *side_a* to *side_b*, the fluent $lantern\_at(side\_a)$ is not deleted (because the suffix layer encodes the delete-free relaxation of the problem), so this is still considered to be achieved when Joe, and then William and Jill cross. Nobody needs to bring the lantern back for them. The use of the suffix layer is allowed because there isn't a second occurrence of the fluent $at(joe(side_b))$, but this is a starting fluent (all 3 suffix actions are starting actions since they do not depend on each other). Since the suffix layer was used, we add a second occurrence of each of the fluents and actions which were saturated by this plan:

```
Adding:
is(fluentOcc(at(joe,side_a),2)).
is(fluentOcc(lantern_at(side_a),2)).
is(fluentOcc(lantern_at(side_b),2)).
is(fluentOcc(at(joe,side_b),2)).
is(fluentOcc(at(jack,side_b),2)).
is(actOcc(cross_together(jack,joe,side_a,side_b),2)).
is(actOcc(cross_alone(joe,side_b,side_a),2)).
```

and run it again:

```
cross_together(william,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(jill,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
suffix cross_together(candice,averell,side_a,side_b)
suffix cross_together(jack,joe,side_a,side_b)
cost: 32
```

This time we start by having William and Joe cross together and then Joe carries the lantern back, crosses with Jill and carries it back again. In the suffix layer, Candice and Averell cross together while Jack and Joe cross together (each pair making use of the same undeleted lantern). Again the suffix layer occurs because we don't have enough occurrences of $at(joe(side\_b))$.

Interestingly, a cheaper solution seems to have been skipped. Namely the plan which is identical to the cost-29 plan, but with Joe running across and running back first for a total cost of 31.

This is because such a plan fails to make progress. We can produce two cuts, namely the one at the start of the plan and the one after Joe crosses back the first time and see that no new fluents hold between the two cuts. The rules enforcing strong minimality will reject this plan.

Add another occurrence of each saturated item

```
Adding:
is(fluentOcc(at(joe,side_a),3)).
is(fluentOcc(lantern_at(side_a),3)).
is(fluentOcc(lantern_at(side_b),3)).
is(fluentOcc(at(joe,side_b),3)).
is(fluentOcc(at(jill,side_b),2)).
is(fluentOcc(at(william,side_b),2)).
is(actOcc(cross_together(jill,joe,side_a,side_b),2)).
is(actOcc(cross_together(william,joe,side_a,side_b),2)).
is(actOcc(cross_alone(joe,side_b,side_a),3)).
```

and again:

```
cross_together(william,jack,side_a,side_b)
cross_alone(jack,side_b,side_a)
cross_together(jill,jack,side_a,side_b)
suffix cross_alone(jack,side_b,side_a)
suffix cross_alone(joe,side_a,side_b)
suffix cross_together(candice,averell,side_a,side_b)
cost: 33
```

Here we have William and Jack crossing together. Then Jack crosses back alone. Jill and Jack cross together, and now Jack *would* cross back alone again taking the lantern, but there are only two occurrences of the action $cross\_alone(jack, side\_b, side\_a)$ in our bag so instead we move into the suffix layer. In the suffix layer he carries the lantern back, but because of the delete relaxation, we don't lose the fluent $at(jack, side\_b)$ so he doesn't need to cross back again. Candice and Averell use the lantern to cross as does Joe by himself.

The rest of the output from the stepless solver follows:

```
Adding:
is(fluentOcc(at(jack,side_a),2)).
is(fluentOcc(at(jack,side_b),3)).
is(actOcc(cross_together(jill,jack,side_a,side_b),2)).
is(actOcc(cross_together(william,jack,side_a,side_b),2)).
is(actOcc(cross_alone(jack,side_b,side_a),2)).

cross_together(jill,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(william,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(jack,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
suffix cross_alone(joe,side_a,side_b)
suffix cross_together(candice,averell,side_a,side_b)
cost: 34

Adding:
is(fluentOcc(at(joe,side_a),4)).
is(fluentOcc(lantern_at(side_a),4)).
is(fluentOcc(lantern_at(side_b),4)).
is(fluentOcc(at(joe,side_b),4)).
is(actOcc(cross_alone(joe,side_b,side_a),4)).

cross_together(jill,jack,side_a,side_b)
cross_alone(jill,side_b,side_a)
cross_together(william,jill,side_a,side_b)
suffix cross_alone(jill,side_b,side_a)
suffix cross_alone(joe,side_a,side_b)
suffix cross_together(candice,averell,side_a,side_b)
cost: 35

Adding:
is(fluentOcc(at(jill,side_a),2)).
is(fluentOcc(at(jill,side_b),3)).
is(actOcc(cross_together(william,jill,side_a,side_b),2)).
is(actOcc(cross_alone(jill,side_b,side_a),2)).

cross_together(jack,joe,side_a,side_b)
cross_alone(jack,side_b,side_a)
cross_together(jill,jack,side_a,side_b)
cross_alone(jack,side_b,side_a)
cross_together(william,jack,side_a,side_b)
suffix cross_alone(jack,side_b,side_a)
suffix cross_together(candice,averell,side_a,side_b)
cost: 36
```

```
Adding:
is(fluentOcc(at(jack,side_a),3)).
is(fluentOcc(at(jack,side_b),4)).
is(actOcc(cross_alone(jack,side_b,side_a),3)).

cross_together(jack,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(jill,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(candice,averell,side_a,side_b)
cross_alone(jack,side_b,side_a)
cross_together(william,joe,side_a,side_b)
cross_alone(joe,side_b,side_a)
cross_together(jack,joe,side_a,side_b)
cost: 37
```

In the last one, the suffix layer is not used so we're done. No other plans need be searched.

## References


BLUM, A. L. AND FURST, M. L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence 90,* 1, 281–300.

EITER, T., FABER, W., LEONE, N., PFEIFER, G., AND POLLERES, A. 2003. Answer set planning under action costs. *Journal of Artificial Intelligence Research 19*, 25–71.