Creating Programs on State: Through Activity Planning

David Wang
MIT CSAIL
mers.csail.mit.edu

Contributions:
Brian Williams
Maria Fox

February 22nd, 2016
The Firefighting Scenario

Objective: Put out all the fires using UAV1, avoid no-fly zones.
The Firefighting Scenario

Objective: Put out all the fires using UAV1, avoid no-fly zones.
**Traditional Solution:**

Specify each activity *(the usual programmatic way)*

```java
class Main{
    UAV uav1;
    Lake lake1;
    Lake lake2;
    Fire fire1;
    Fire fire2;
    // constructor
    Main (){ }
        uav1 = new UAV();
        uav1.location= base_1_location;
        uav1.flying = no;
        uav1.loaded = no;
        lake1 = new Lake();
        lake1.location = lake_1_location;
        lake2 = new Lake();
        lake2.location = lake_2_location;
        fire1 = new Fire();
        fire1.location = fire_1_location;
        fire1 = high;
        fire2 = new Fire();
        fire2.location = fire_2_location;
        fire2 = high;
    } // "main" method
    method run() {
        sequence{
            uav1.takeoff();
            uav1.fly(base_1_location, lake_2_location);
            uav1.load_water(lake2);
            uav1.fly(lake_2_location, fire_2_location);
            uav1.drop_water_high_altitude(fire2);
            ... <13 additional activities> ...
            uav1.land();
        }
    }
}
```

**Specify each activity (the usual programmatic way):**

These are the actions the UAV can take.

A program that specifies the exact sequence of activities.
State-based Solution:
Specify the desired states, let the computer plan the activities.

```java
class Main{
    UAV uav1;
    Lake lake1;
    Lake lake2;
    Fire fire1;
    Fire fire2;
    // constructor
    Main (){;
        uav1 = new UAV();
        uav1.location= base_1_location;
        uav1.flying = no;
        uav1.loaded = no;
        lake1 = new Lake();
        lake1.location = lake_1_location;
        lake2 = new Lake();
        lake2.location = lake_2_location;
        fire1 = new Fire();
        fire1.location = fire_1_location;
        fire1 = high;
        fire2 = new Fire();
        fire2.location = fire_2_location;
        fire2 = high;
    }
    // "main" method
    method run() { 
        sequence{
            (fire1 == out);
            (fire2 == out);
            (uav1.flying == no &&
             uav1.location == base_1_location);
        }
    }
}
```

These are the actions the UAV can take.

```java
class UAV {
    Roadmap location;
    Boolean flying;
    Boolean loaded;
    
    primitive method takeoff()
    flying == no => flying == yes;
    
    primitive method land()
    flying == yes => flying == no;
    
    primitive method load_water(Lake lakespot)
    ((flying == yes) && (loaded == no)
    && (lakespot.location == location)) => loaded == yes;
    
    primitive method drop_water_high_altiture(Fire firespot)
    ((flying == yes) && (loaded == yes)
    && (firespot.location == location) && (firespot == high)
    => ((loaded == no) && (firespot == medium));
    
    primitive method drop_water_low_altiture(Fire firespot)
    ((flying == yes) && (loaded == yes)
    && (firespot.location == location) && (firespot == medium)
    => ((loaded == no) && (firespot == out));
    
    #MOTION_PRIMITIVES(location, fly, flying==yes)
}
```

A program that specifies the desired states.
Activity Planning

initial state:

goals:

operators:

plan:
class UAV {
    Roadmap location;
    Boolean flying;
    Boolean loaded;

    primitive method takeoff()
        flying == no => flying == yes;

    primitive method land()
        flying == yes => flying == no;

    primitive method load_water(Lake lakespot)
        ((flying == yes) && (loaded == no)
            && (lakespot.location == location)) => loaded == yes;

    primitive method drop_water_high_altitude(Fire firespot)
        ((flying == yes) && (loaded == yes)
            && (firespot.location == location) && (firespot == high))
            => ((loaded == no) && (firespot == medium));

    primitive method drop_water_low_altitude(Fire firespot)
        ((flying == yes) && (loaded == yes)
            && (firespot.location == location) && (firespot == medium))
            => ((loaded == no) && (firespot == out));

    #MOTION_PRIMITIVES(location, fly, flying==yes)
}

class Main{
    UAV uav1;
    Lake lake1;
    Lake lake2;
    Fire fire1;
    Fire fire2;

    // constructor
    Main()
    {
        uav1 = new UAV();
        uav1.location= base_1_location;
        uav1.flying = no;
        uav1.loaded = no;

        lake1 = new Lake();
        lake1.location = lake_1_location;

        lake2 = new Lake();
        lake2.location = lake_2_location;

        fire1 = new Fire();
        fire1.location = fire_1_location;
        fire1 = high;

        fire2 = new Fire();
        fire2.location = fire_2_location;
        fire2 = high;
    }

    // "main" method
    method run() {
        sequence{
            uav1.takeoff();
            uav1.fly(base_1_location,lake_2_location);
            uav1.load_water(lake2);
            uav1.fly(lake_2_location,fire_2_location);
            uav1.drop_water_high_altitude(fire2);
            ... <13 additional activities> ...
            uav1.land();
        }
    }
}
Outline

• Programming on State with Activity Planning

  Classic Planning Problem

• Planning as Heuristic Forward Search (Fast Forward Planner)
  – Enforced Hill Climbing
  – Fast Forward Heuristic

• Planning with Time (Crikey 3 Planner)
  – Temporal Planning Problem
  – Temporal Relaxed Plan Graph
Plan Representation

Many ways of expressing planning problems.

All include:

Inputs:

• initial state – a set of facts about the world
• goal – subset of facts that must appear in the goal state.
• actions – a set of named precondition and effect pairs.

Outputs:

• plan – a schedule of actions (i.e. a sequence or list of actions).
“Classic” Representation (PDDL)

Action Model:

**Objects** (things):

**Predicates** (used to create true or false statements about the world, “facts”):

(On A, B)

(Clear A)

**Actions** (used to change truth of predicates):

(Clear A)

(Clear B)

Stack

preconditions

effects

Initial: (Clear A) (Clear B) (Clear C)

Goal: (On A, B) (On B, C)
“Classic” Planning Actions

**Preconditions** – a conjunction of statements that must be true **before** the action is applied.

**Actions** (used to change truth of predicates):

(Clear $A$)
(Clear $B$)

---

**Effects** – a conjunction of statements that must be true **after** the action is applied.

“**Delete**” Effects – statements that must NOT be true.

“**Add**” Effects – statements that must be true.
Automata Representation

Action/model:

Concurrent Constraint Automata (like a state-machine):

Initial:  
Goal:  

Note: This is a very simple example, there are usually many automata, and guards on the transitions.

Algorithms exist to map between the two representations.
What is the difference between Path and Activity Planning?

For a Path Planner:

State = a location

(identified by name)

Operator = a weighted edge

The start and end states uniquely identify where this operator can be applied.

State Space = a map
Formulating Activity Planning as Search

Search needs a **State Space**, constructed from **States** and **Operators**:

**State** = a set of facts
- I’m hungry
- I want cake
- I have flour
- I have sugar

(identified by the set of statements)

**Operator** = an action
- **Bake cake**
  - I have flour
  - I have sugar
  - I have milk
  - I have eggs

  **Precondition**: Statements that must be a subset of the starting state.
  - I have flour
  - I do NOT have flour.
  - I do NOT have sugar.
  - I do NOT have milk.
  - I do NOT have eggs.

  **Effect**: Statements that will be true in ending state.
  - I have cake
  - I do NOT have flour.
  - I do NOT have sugar.
  - I do NOT have milk.
  - I do NOT have eggs.

**State Space** = states reachable given the available actions.

- **Unlock Bike**
  - have flour
  - have eggs
  - hungry
  - want cake
  - locked bike

- **Shopping Run**
  - have flour
  - have eggs
  - hungry
  - want cake
  - unlocked bike

- **Bake Cake**
  - have flour
  - have eggs
  - have sugar
  - have milk
  - hungry
  - want cake
  - unlocked bike

- **Eat Cake**
  - have cake
  - hungry
  - want cake
  - unlocked bike

- **Steal Picnic Basket**
  - have flour
  - have eggs
  - have sugar
  - have milk
  - hungry
  - want cake
  - locked bike

- **Lock Bike**
  - full
  - locked bike

- **Eat Cake**
  - full
  - unlocked bike

- **Bake Cake**
  - have cake
  - hungry
  - want cake
  - locked bike

2/22/2016 Activity Planning
Activity Planning as Search

• Providing a “map” will all possible actions is too large and time-consuming.
• Instead, we provide a set of actions...

\[
\text{Actions} = \{ \text{Unlock Bike, Lock Bike, Shopping Run, Steal Picnic Basket, Bake Cake, Eat Cake} \}
\]

and expect the planner to build the “map” as needed.
How Hard is Activity Planning?

The “map” is usually not provided in activity planning, but we can imagine how hard the planning problem is relative to depth first search.

Complexity of Depth First Search: $O(b^d)$

Planning Problem with:
- 10 actions
- 10 statements = 1024 possible states (not necessarily all reachable)

Scenario 1: Let's assume our expected plan is 10 actions long
  if $b = 10$ actions, $d = 10$ states
  $b^d = 10,000,000,000$

Scenario 2: A few actions are applied over and over again in different orders, visiting all possible states.
  if $b = 10$ actions, $d = 1024$ states
  $b^d > \text{atoms in the universe (}3.0 \times 10^{23})$
Activity Planning Search Strategies

The order we search for actions matters a lot . . .

– **Forward search** – start at beginning; ‘simulate’ forward, with all states grounded.
  • Heuristic Forward Search* (Enforced Hill Climbing)

– **Goal-regression search** – start with goals; ask “what actions are needed to achieve each goal?”

– **Constraint Satisfaction** – encode as constraint problem; solver exploits tightest constraints.

* Very popular right now.
Forward Search

Initial State ($s_0$)

- Action 1: $s_1^0$ → $s_2^0$
- Action 2: $s_1^1$ → $s_2^1$
- Action 3: $s_1^2$ → $s_2^2$

Time

Goal

Action 4...

2/22/2016 Activity Planning
Goal-Regression Search

Initial State ($s_0$)

action2

... action2

$S_t^0$ action1

$S_t^1$ action4

$S_t^2$ action2

goal
Outline

• Programming on State with Activity Planning
• Classic Planning Problem

Planning as Heuristic Forward Search (Fast Forward Planner)
  – Enforced Hill Climbing Search
  – Fast Forward Heuristic

• Planning with Time (Crikey 3 Planner)
  – Temporal Planning Problem
  – Temporal Relaxed Plan Graph
Enforced Hill-Climbing Search
(i.e., greedy without backup)

Basic Enforced Hill-Climbing Algorithm
Start with the initial state.
If the state is not the goal:
1. Identify applicable actions.
2. Obtain heuristic estimate of the value of the next state for each action considered.
3. Select action that transitions to a state with better heuristic value than the current state.
4. Move to the better state.
5. Append action to plan head and repeat.
(Never backtrack over any choice.)

Legend: \( s, h(s) \)

Time
Used by FF (Hoffmann, IJCAI, 2000), FastDownward (Helmert, JAIR, 2006) and many others.
Enforced Hill-Climbing Search
(i.e., greedy without backup)

Basic Enforced Hill-Climbing Algorithm
Start with the initial state.
If the state is not the goal:
1. Identify applicable actions.
2. Obtain heuristic estimate of the value of the next state for each action considered.
3. Select action that transitions to a state with better heuristic value than the current state.
4. Move to the better state.
5. Append action to plan head and repeat.
(Never backtrack over any choice.)

Legend: \(s, h(s)\)

Time
Used by FF (Hoffmann, IJCAI, 2000), FastDownward (Helmert, JAIR, 2006) and many others.
Enforced Hill-Climbing Search
(i.e., greedy with-out backup)

Basic Enforced Hill-Climbing Algorithm
Start with the initial state.
If the state is not the goal:
1. Identify applicable actions.
2. Obtain heuristic estimate of the value of the next state for each action considered.
3. Select action that transitions to a state with better heuristic value than the current state.
4. Move to the better state.
5. Append action to plan head and repeat.
   *(Never backtrack over any choice.)*

Legend: \( s, h(s) \)

Time
Used by FF (Hoffmann, IJCAI, 2000), FastDownward (Helmert, JAIR, 2006) and many others.
Enforced Hill-Climbing Search
(i.e., greedy without backup)

Basic Enforced Hill-Climbing Algorithm
Start with the initial state.
If the state is not the goal:
1. Identify applicable actions.
2. Obtain heuristic estimate of the value of the next state for each action considered.
3. Select action that transitions to a state with better heuristic value than the current state.
4. Move to the better state.
5. Append action to plan head and repeat. (Never backtrack over any choice.)

Legend: $s, h(s)$

Time Used by FF (Hoffmann, IJCAI, 2000), FastDownward (Helmert, JAIR, 2006) and many others.

2/22/2016 Activity Planning
Enforced Hill-Climbing Search
(i.e., greedy without backup)

Basic Enforced Hill-Climbing Algorithm
Start with the initial state.
If the state is not the goal:
1. Identify applicable actions.
2. Obtain heuristic estimate of the value of the next state for each action considered.
3. Select action that transitions to a state with better heuristic value than the current state.
4. Move to the better state.
5. Append action to plan head and repeat.
(Never backtrack over any choice.)

Legend: $s, h(s)$

Time Used by FF (Hoffmann, IJCAI, 2000), FastDownward (Helmert, JAIR, 2006) and many others.
Enforced Hill-Climbing Search
(i.e., greedy without backup)

Basic Enforced Hill-Climbing Algorithm
Start with the initial state.
If the state is not the goal:
1. Identify applicable actions.
2. Obtain heuristic estimate of the value of the next state for each action considered.
3. Select action that transitions to a state with better heuristic value than the current state.
4. Move to the better state.
5. Append action to plan head and repeat.
(Never backtrack over any choice.)

Legend: $s, h(s)$

Time
Used by FF (Hoffmann, IJCAI, 2000), FastDownward (Helmert, JAIR, 2006) and many others.
Enforced Hill-Climbing Search
(i.e., greedy without backup)

Basic Enforced Hill-Climbing Algorithm
Start with the initial state.
If the state is not the goal:
1. Identify applicable actions.
2. Obtain heuristic estimate of the next state for each action considered.
3. Select action that transitions to a state with better heuristic value than the current state.
4. Move to the better state.
5. Append action to plan head and repeat.
   (Never backtrack over any choice.)

Legend: \(s, h(s)\)

Resulting Plan: \(\{a_2, a_6\}\)

Time
Used by FF (Hoffmann, IJCAI, 2000), FastDownward (Helmert, JAIR, 2006) and many others.

Done!

Search finishes when the current state contains the goal. The actions along the path form the plan.
Enforced Hill-Climbing (EHC) Pseudo Code

The basic Enforced Hill-Climbing algorithm, shown before, is conceptually easy to understand but hides interesting details.

```plaintext
open_list = [initial_state];
best_heuristic = heuristic value of initial_state;
while open_list not empty do
    current_state = pop state from head of open_list;
    successors = the list of states reachable from current_state;
    while successors is not empty do
        next_state = remove a state from successors;
        h = heuristic value of next_state;
        if next_state is a goal state then
            return next_state;
        end if
        if h better than best_heuristic then
            clear successors;
            clear open_list;
            best_heuristic = h;
        end if
        place next_state at back of open_list;
    end while
end while
Recover Plan (i.e. by using and walking backwards over parent pointers)
```

*EHC uses a queue to remember states to expand.*

*If there are no successor states with better heuristic value, EHC expands the current state in a breadth first manner.*

*The queue is cleared when a better state is found, effectively enforcing the search to only consider the successors of the best state.*
Planning as Enforced Hill-Climbing (cont.)

• Success depends on an informative heuristic.
  – Fast Forward uses Delete-Relaxation heuristic, which is informative for a large class of benchmark planning domains.

• Strategy is suboptimal.
  – Heuristic may over estimate.

• Strategy is incomplete.
  – Never backtracking means some parts of the search space are lost.

• If Enforced Hill-Climbing fails (ends without reaching a goal state), the Fast Forward planner switches to best-first search.
  – (e.g., Greedy search with Backup or A* search).
Where does this Heuristic come from?

• Numerous heuristics have emerged over 15 years.

• Many heuristics solve an easier, relaxed, problem by:
  – Ignoring information or constraints.
  – Being optimistic.

• The FF heuristic applies the previous fastest planner, Graph Plan, to a relaxed problem.
Fast Forward Heuristic, $h_{ff}(s)$

- **Observation:**
  - Actions *complicate* planning by “deleting” the *progress* made by other actions.
  - If actions can *never undo* effects, planning is *simple*.

- **Idea: Delete Relaxation**
  - Ignore “delete” effects of actions.
  - Generate simplified plan using relaxed actions.
  - The heuristic counts the actions in that simplified plan.

- **Example: The Farmer, Fox Goose and grain**
  - A farmer must use a boat to *move* a fox, goose, and bag of grain across a river two-at-a-time.
  - If left alone, the fox will *eat* the goose and the goose will *eat* the bag of grain.
  - What is the plan?

  - For the *relaxed heuristic*: ignore eating each other.

Simple Planning Problem

Actions:

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Add Effects</th>
<th>Delete Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(b, t, c)</td>
<td>BoxIn(b, c), TruckIn(t, c)</td>
<td>BoxOn(b, t)</td>
<td>BoxIn(b, c)</td>
</tr>
<tr>
<td>Unload(b, t, c)</td>
<td>BoxOn(b, t), TruckIn(t, c)</td>
<td>BoxIn(b, c)</td>
<td>BoxOn(b, t)</td>
</tr>
<tr>
<td>Drive(t, c, c’)</td>
<td>TruckIn(t, c)</td>
<td>TruckIn(t, c’)</td>
<td>TruckIn(t, c)</td>
</tr>
</tbody>
</table>
Simple Planning Problem

Problem: “Get the box to Paris”

Initial:

<table>
<thead>
<tr>
<th>Atom</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoxIn(b, c₁)</td>
<td>True</td>
</tr>
<tr>
<td>BoxIn(b, c₂)</td>
<td>False</td>
</tr>
<tr>
<td>BoxIn(b, Paris)</td>
<td>False</td>
</tr>
<tr>
<td>BoxOn(b, t)</td>
<td>True</td>
</tr>
<tr>
<td>TruckIn(t, c₁)</td>
<td>False</td>
</tr>
<tr>
<td>TruckIn(t, c₂)</td>
<td>True</td>
</tr>
<tr>
<td>TruckIn(t, Paris)</td>
<td>False</td>
</tr>
</tbody>
</table>

Goal:

<table>
<thead>
<tr>
<th>Atom</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoxIn(b, c₁)</td>
<td></td>
</tr>
<tr>
<td>BoxIn(b, c₂)</td>
<td></td>
</tr>
<tr>
<td>BoxIn(b, Paris)</td>
<td>True</td>
</tr>
<tr>
<td>BoxOn(b, t)</td>
<td></td>
</tr>
<tr>
<td>TruckIn(t, c₁)</td>
<td></td>
</tr>
<tr>
<td>TruckIn(t, c₂)</td>
<td></td>
</tr>
<tr>
<td>TruckIn(t, Paris)</td>
<td></td>
</tr>
</tbody>
</table>

* Indicates unassigned (don’t care)
Getting to Paris the **Correct** Way

Initial:

Goal:

### Original Actions:

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Add Effects</th>
<th>Delete Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Load(b, t, c)</strong></td>
<td>BoxIn(b, c), TruckIn(t, c)</td>
<td>BoxOn(b, t)</td>
<td>BoxIn(b, c)</td>
</tr>
<tr>
<td><strong>Unload(b, t, c)</strong></td>
<td>BoxOn(b, t), TruckIn(t, c)</td>
<td>BoxIn(b, c)</td>
<td>BoxOn(b, t)</td>
</tr>
<tr>
<td><strong>Drive(t, c, c')</strong></td>
<td>TruckIn(t, c)</td>
<td>TruckIn(t, c')</td>
<td>TruckIn(t, c)</td>
</tr>
</tbody>
</table>
Getting to Paris the \textit{Relaxed} Way

Simple Idea: \textbf{Ignore Delete Effects}

\textbf{Initial:}

- BoxIn(b, c₁)
- TruckIn(t, c₂)

\textbf{Goal:}

- BoxIn(b, c₁)
- BoxOn(b, t)
- Drive(t, c₂, c₁)
- TruckIn(t, Paris)

\textbf{Relaxed Actions:}

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Add Effects</th>
<th>Delete Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load(b, t, c)</td>
<td>BoxIn(b, c), TruckIn(t, c)</td>
<td>BoxOn(b, t)</td>
<td>BoxIn(b, c)</td>
</tr>
<tr>
<td>Unload(b, t, c)</td>
<td>BoxOn(b, t), TruckIn(t, c)</td>
<td>BoxIn(b, c)</td>
<td>BoxOn(b, t)</td>
</tr>
<tr>
<td>Drive(t, c, c')</td>
<td>TruckIn(t, c)</td>
<td>TruckIn(t, c')</td>
<td>TruckIn(t, c')</td>
</tr>
</tbody>
</table>
The Fast Forward Heuristic, in Practice

Enforced Hill Climbing: searches for the **correct** plan...

**Initial:**

```
BoxIn(b, c1)
TruckIn(t, c2)
Drive(t, c2, c1)
BoxIn(b, c1)
TruckIn(t, c1)
Drive(t, c2, Paris)
BoxIn(b, c1)
TruckIn(t, Paris)
```

**Goal:**

```
BoxIn(b, Paris)
```

For each possible next state, while the Fast Forward Heuristic: searches for the **relaxed** plan...
Enforced Hill Climbing searches for the correct plan...

For each possible next state,

while the Fast Forward Heuristic searches for the relaxed plan...

How do we efficiently find this relaxed plan?
Solution: Use a Relaxed Plan Graph
1. Create Relaxed Plan Graph that Encodes All Plans

Fact 1

BoxIn(b, c₁)
TruckIn(t, c₁)

Write down initial facts
1. Create Relaxed Plan Graph that Encodes All Plans

Fact 1
- BoxIn(b, c₁)
- TruckIn(t, c₁)

Action 1
- Noop
- Load(b, t, c₁)
- Noop
- Drive(t, c₁, c₂)
- Drive(t, c₁, Paris)

Write down initial facts
Write down all actions we can take
1. Create Relaxed Plan Graph that Encodes All Plans

**Fact 1**
- BoxIn(b, c₁)
- TruckIn(t, c₁)

**Action 1**
- Noop
- Load(b, t, c₁)
- Noop
- Drive(t, c₁, c₂)
- Drive(t, c₁, Paris)

**Fact 2**
- BoxIn(b, c₁)
- BoxOn(b, t)
- TruckIn(t, c₁)
- TruckIn(t, c₂)
- TruckIn(t, Paris)

**Fact 3**

---

Write down initial facts
Write down all actions we can take
Write down all facts that follow
1. Create Relaxed Plan Graph that Encodes All Plans

For clarity, no-ops are not explicitly shown in the activity layer, but the facts are carried forward from one layer to the next.

Write down initial facts
Write down all actions we can take
Write down all facts that follow

Repeat until all goals appear.
2. Extract Relaxed Plan

Find the set of actions for the relaxed plan by searching backward in the planning graph.

Recursively select actions that achieve goals
Fast Forward Heuristic

Simple Idea: Search, while maintaining a relaxed plan graph (ignore delete effects), \( h_{ff}(s) = \) the number of actions in the relaxed plan until the goal first appears.

Initial:

\[
\text{BoxIn}(b, c_1) \quad \text{TruckIn}(t, c_2) \\
\text{Drive}(t, c_2, c_1) \\
\text{BoxIn}(b, c_1) \quad \text{TruckIn}(t, c_1) \\
\text{Drive}(t, c_2, \text{Paris}) \\
\text{Load}(b, t, c_1) \\
\text{Drive}(t, c_1, c_2) \quad \text{TruckIn}(t, c_2) \\
\text{BoxIn}(b, c_1) \quad \text{TruckIn}(t, \text{Paris}) \\
\text{BoxIn}(b, \text{Paris}) \\
\boxed{\text{BoxIn}(b, \text{Paris})} \\
\text{Drive}(t, \text{Paris}, c_2) \\
\text{Drive}(t, c_1, \text{Paris}) \\
\text{Drive}(t, \text{Paris}, c_1) \\
\text{Drive}(t, c_1, \text{Paris}) \\
\mathbf{h_{ff}(s) = 3} \\
\]

Goal:

\[
\text{BoxIn}(b, \text{Paris}) \\
\text{Load}(b, t, c_1) \quad \text{Unload}(b, t, c_1) \\
\text{Unload}(b, t, c_1) \\
\text{Unload}(b, t, \text{Paris}) \\
\text{Drive}(t, c_7, c_1) \\
\text{Drive}(t, c_1, c_2) \\
\text{Drive}(t, c_1, \text{Paris}) \\
\text{Drive}(t, \text{Paris}, c_1) \\
\text{Drive}(t, \text{Paris}, c_2) \\
\mathbf{h_{ff}(s) = ?} \\
\]

Relaxed Plan Graph:
Fast Forward Heuristic

Simple Idea: Search, while maintaining a relaxed plan graph (ignore delete effects), \( h_{ff}(s) = \) the number of actions in the relaxed plan until the goal first appears.

Initial:

\[ h_{ff}(s) = 3 \]

Goal:

\[ h_{ff}(s) = 3 \]

In this simple example, both actions appear equally good.
Fast Forward Heuristic Summary

• Solve a simpler planning problem to aid the harder planning problem.

• # of actions in the relaxed plan found in a relaxed planning graph is the heuristic.

• Ignoring constraints makes it fast.

• Extendable beyond classical domains
  – Enhance plan graph with more constraint propagation
  – Similar extensions exist for temporal problems
Real-World Planning Problems

Simple (classic) planning is hard!
but, still lacks many real-world features ...

• **Classical** – discretized world, finite domains, single agent of change.
• **Numeric** – domain allows continuous values
  - x, y position
• **Temporal** – actions take time, goals can have deadlines
  - walking will take 5-10 minutes, I need to be there in 1 hour.
• **Resources** – a quantity that can be consumed or regulated (type of numeric domain)
  - fuel, battery, CPU usage
• **Optimality** – do we minimize or maximize a particular value
  - number of actions, time spent, fuel used, utility
• **Preferences** – express soft goals, preferred actions, or action ordering.
  - “I would like to visit my friends on the way to grandmothers house.”
• **Stochastic** – actions can have uncertain effects, uncertain durations.
  - driving will have a 99% of success of reaching your destination, but a 1% chance of an accident.
• **Multi-agent** – planning for multiple coordinating agents, or against an adversary.
  - multiple UAVs, or planning against cyber-attack.
Outline

• Programming on State with Activity Planning
• Classic Planning Problem
• Planning as Heuristic Forward Search (Fast Forward Planner)
  – Enforced Hill Climbing
  – Fast Forward Heuristic

Planning with Time (Crikey 3 Planner)
  – Temporal Planning Problem
  – Temporal Relaxed Plan Graph
Classical [, Instantaneous] Action

Instantaneous Action, $\text{IA} = \langle C, A, D \rangle$

Precondition, $C$

Effects:
• Add Effect, $A$
• Delete Effect, $D$

Time is discretized into “layers”, an action applies instantaneously at a particular layer index.
PDDL Durative Action

A Durative Action consists of:
- two instantaneous (aka “snap”) actions, and
- a condition that must hold during its execution, but must be applied atomically (all or nothing).

\[
\text{Durative Action, } \text{DA}= \langle C_S, C_O, C_E, A_S, A_E, D_S, D_E, lb, ub \rangle
\]

- **Duration**: \([lb, ub]\)
- **Conditions**:
  - At Start Condition, \(C_S\)
  - Overall Condition, \(C_O\)
  - At End Condition, \(C_E\)

- **Effects**:
  - At Start Add Effect, \(A_S\)
  - At Start Delete Effect, \(D_S\)
  - At End Add Effect, \(A_E\)
  - At End Delete Effect, \(D_E\)

\[
lb \leq t_{end} - t_{start} \leq ub
\]
Temporal Planning

Combination of Planning & Scheduling

- Planning – Deciding *what* to do.

- Scheduling – Deciding *when* to do it.
Strategies for Planning with Durative Actions

• **Compression**
  - Convert the Durative Action to Instantaneous Actions
    - $C = C_S \cup ((C_E \cup C_O) \setminus A_S)$ - union of conditions
    - $A = (A_S \setminus D_E) \cup A_E$ - union of add effects
    - $D = (D_S \setminus A_E) \cup D_E$ - union of delete effects
  - Plan using classical planner, expand and schedule at the end.
  - *Pro:* Allows the use of classical planners
  - *Cons:* Not as expressive

• **Snap Actions**
  - Convert the Durative Action to two Instantaneous Actions
  - Modify the Search the Enforce the Duration and Overall Condition
  - *Pro:* Builds on planning strategies developed for classical planners.
  - *Cons:* doubled number of actions

• **Automata**
  - We’ll talk about this next week.

*Note: There are many approaches and variations on those listed.*
Complications in Temporal Planning: Required Concurrency

Required Concurrency – a property of the temporal planning problem, when two actions must temporally overlap in any working plan. Therefore: Conditions/Effects must be considered at the same time as Duration.

Case 1. Action[s] Must Contain Another:

Operate Mine

\[ A_S = \text{Mine Open} \]

Dig

\[ C_O = \text{Mine Open} \]

\[ D_E = \neg \text{Mine Open} \]

Case 2. “Deadlines” force Actions to co-occur:

Initial State: RacelsOn

Player1 Act1

Player1 Act2

Player1 Act3

Player2 Act1

Player2 Act2

Amazing Race

\[ D_E = \neg \text{RacelsOn} \]
State Space of Crikey 3

Classical Planner State = Set of Facts

Crikey3 State = \( \langle F, E, T \rangle \)
- F – Set of Facts
- E – Set of Start Events in the form \( \langle \text{StartAct}, i, \text{min}, \text{max} \rangle \)
  - the action that has started
  - index indicating the ordering of events, and
  - the duration of the original durative action.
- T – Set of Temporal Constraints (A Simple Temporal Network)
Recall: Enforced Hill-Climbing Search

Crikey 3 uses the same basic Enforced Hill-Climbing algorithm, but with a more complex “successor” function than what we’ve seen so far.

Basic Enforced Hill-Climbing Algorithm
Start with the initial state.
If the state is not the goal:
1. Identify applicable actions.
2. Obtain heuristic estimate of the value of the next state for each action considered.
3. Select action that transitions to a state with better heuristic value than the current state.
4. Move to the better state.
5. Append action to plan head and repeat. (Never backtrack over any choice.)
Crikey 3’s Successor Function

Big Ideas

**Input:** Current State, \( S = \langle F, E, T \rangle \)

**Output:** Set of Successor States, \( S' = \langle F', E', T' \rangle \)

Recall: Crikey splits a durative action into two “snap” [instantaneous] actions: a start action and an end action.

The successor states can be found by applying all applicable start actions and end actions to the current state. (As in the classical case, this involves checking whether the preconditions of the snap action exist in \( F \), and then applying its effects to create the successor \( F' \), but there is also some bookkeeping for \( E \) and \( T \))

- Applying the start action is trivial.
- Applying an end action is more complicated. We must make sure the corresponding start action has already been executed, the durative action from which the end action was created has an overall condition that is consistent with all other actions being executed, and the temporal constraints are consistent.
Crikey 3’s Successor Function

Input: Current State, $S = \langle F, E, T \rangle$
Output: Set of Successor States, $S' = \langle F', E', T' \rangle$

• For each start action that could be applied to $S$, create $S'$ s.t.
  • $F' = \text{add/delete effects of start action from } F$
  • $E' = E \cup \langle \text{StartAct}, i, \text{min}, \text{max} \rangle$

• For each end action that could be applied to $S$
  • For each start action event, $e \in E$, that the end action closes, create $S'$ s.t.
    • $F' = \text{add/delete effects of end action from } F$
    • $E' = E \setminus e$
    • $T' = T \cup (e.\text{min} \leq \text{time(EndAct)} - \text{time(e.i)} \leq e.\text{max})$
  • Include $S'$ in the successor states if:
    • the overall condition of action is consistent with the started actions in $E$,
    • $T'$ is temporally consistent
Temporal Relaxed Planning Graph (TRPG)  
Big Ideas

**Input:** Current State, $S = \langle F, E, T \rangle$

**Output:** $R = a$ relaxed planning graph

In the Classical Plan Graph: fact and action layers are indexed by integers.

In the TRPG: layers are indexed by “real” time, starting with the current state $S$ at $t=0$.

We still build the plan graph in a “forward” in time, but how do we know when we should add a new pair of fact and action layers?

- Look at the lower-bound times of all started actions. Add a layer when the earliest action could end.
- If the earliest end time is 0, advance time by some small amount of time, $\epsilon$, just to make sure layers don’t overlap.
Temporal Relaxed Planning Graph (TRPG)

Note: We will keep track of:
• A fact “layer”, indexed by continuous time, start with facts \( F \).
• current time index, starts at 0
• The earliest time an action can end for each start action event in \( E \).

**Build the TRPG**

**Input:** Current State, \( S = (F, E, T) \)

**Output:** \( R = \) a relaxed planning graph

• For each possible action.
  • If the action has started in \( E \), set its earliest end to 0.
  • Else set it to infinity.
• While \( t < \infty \)
  • Create a new fact layer indexed at \( t + \varepsilon \), with all the facts of the previous layer.
  • Add effects of all end actions whose preconditions are met to the fact layer.
  • Add effects of all start actions whose preconditions are met to the fact layer, and update the earliest end time of any new actions.
  • If the fact layer has more facts, increment by \( t \) by \( \varepsilon \)
    • Otherwise, if all start actions have ended by now, return all fact layers.
    • If there are still start actions running, \( t = \) earliest of the end times.
Questions?
Appendix
Fast Forward Heuristic – Details
Limiting Children Evaluation

- Nodes typically have many children
- $h(s)$ might be slow to compute
- $h(s)$ may suggest "helpful" children to try first
Fast Forward Heuristic – Details

Helpful Actions

Problem: Evaluating the heuristic for all possible actions takes time!
Solution: Start with actions on the helpful actions list, before evaluating the rest.

Initial:  
Goal:

If we select this action, the next layer of actions in the relaxed plan graph are the associated helpful actions.
# Building the Helpful Action List

## Another Perspective....

<table>
<thead>
<tr>
<th>Possible Action</th>
<th>State</th>
<th>$h(s)$</th>
<th>Relaxed Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>$s_1$</td>
<td>$h(s_1) = 4$</td>
<td>${A'_1}, {A'_2, A'_3}, {A'_4}$</td>
</tr>
<tr>
<td>$A_2$</td>
<td>$s_2$</td>
<td>$h(s_2) = 3$</td>
<td>${A'_5, A'_6}$, ${A'_7}$</td>
</tr>
<tr>
<td>$A_3$</td>
<td>$s_3$</td>
<td>$h(s_3) = 4$</td>
<td>${A'<em>8, A'<em>9}, {A'</em>{10}}, {A'</em>{11}}$</td>
</tr>
</tbody>
</table>

*Action layer 1*

$A_2$ Minimizes $h(s)$

Helpful Actions List

<table>
<thead>
<tr>
<th>Possible Action</th>
<th>State</th>
<th>$h(s)$</th>
<th>Relaxed Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A'_5$</td>
<td>$s'_1$</td>
<td>$h(s'_1) = 2$</td>
<td>${A''_1, A''_2}$</td>
</tr>
<tr>
<td>$A'_6$</td>
<td>$s'_2$</td>
<td>$h(s'_2) = 3$</td>
<td>${A''_3}, {A''_4}, {A''_5}$</td>
</tr>
</tbody>
</table>

*Action layer 2*

$A'_5$ Minimizes $h(s)$

Helpful Actions List
### Fast Forward Heuristic – Details

#### How to assert a negative goal?

- What if goal state requires no truck in Paris?
  - Generate negative versions of each atom

#### Example

Given a state $S_0$:

<table>
<thead>
<tr>
<th>Atom</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TruckIn(t, c_1)</td>
<td>False</td>
</tr>
<tr>
<td>TruckIn(t, c_2)</td>
<td>True</td>
</tr>
<tr>
<td>TruckIn(t, Paris)</td>
<td>False</td>
</tr>
<tr>
<td>NoTruckIn(t, c_1)</td>
<td>True</td>
</tr>
<tr>
<td>NoTruckIn(t, c_2)</td>
<td>False</td>
</tr>
<tr>
<td>NoTruckIn(t, Paris)</td>
<td>True</td>
</tr>
</tbody>
</table>

To achieve a state $S_1$ that requires no truck in Paris, we can use the `Drive(t, c, c')` action.

<table>
<thead>
<tr>
<th>Action</th>
<th>Preconditions</th>
<th>Add Effect</th>
<th>Delete Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drive(t, c, c')</td>
<td>TruckIn(t, c)</td>
<td>TruckIn(t, c'), NoTruckIn(t, c)</td>
<td>TruckIn(t, c), NoTruckIn(t, c')</td>
</tr>
<tr>
<td>Drive'(t, c, c')</td>
<td>TruckIn(t, c)</td>
<td>TruckIn(t, c'), NoTruckIn(t, c)</td>
<td>--</td>
</tr>
</tbody>
</table>

**Relaxed Drive'**(t, c_2, Paris)
Planning Graph Intuition

• A plan graph is a compact way of representing the state-space.

• It collapses:
  – all the states 1 operation (action) away from the initial state into Fact Layer 2.
  – all the states 2 operations away from the initial state into Fact Layer 3.
  – etc...
Plan Extraction

- Step 5: Adding “Mutexes” – We add some realism back by marking facts and actions that could not possibly occur at the same time (mutual exclusions)
  - There are rules for how to compute mutexes, but they are not important for this lecture.

Mutexes: We can’t unload and drive at the same time, so we mark these actions as mutex. If some actions can’t happen at the same time, some effects also can’t appear at the same time.
Plan Extraction

- Step 6: Search – Find a path from each goal to the initial state that is free of mutexes in each layer
  - i.e. two facts in the same “layer” can only be included in the plan if there is not a mutex between them. The same goes for actions.
  - Search can be done via DFS, by following back-pointers.
**Problem:** Mutexes make plan extraction & generating action layers hard

What if...

- Remove the mutexes!
- Settle for suboptimal plan, instead of the optimal (shortest one).