Review of Classical Planning & HTN Planning
The Planning Problem

- Given:
  - A characterization of an Initial State
  - A characterization of a (set of) Goal State(s)
  - A characterization of possible actions

- Synthesize:
  - A sequence of actions that when executed in the Initial State transitions the world to a Goal State
Applications

Military Logistics

Robots

Games

Manufacturing

Autonomous Spacecraft

More comprehensive treatment of applications in the next lecture
## Dimensions of Planning

<table>
<thead>
<tr>
<th></th>
<th>Simple</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State Scope</strong></td>
<td>Finite</td>
<td>Non-finite</td>
</tr>
<tr>
<td><strong>Action Determinism</strong></td>
<td>Deterministic</td>
<td>Nondeterministic</td>
</tr>
<tr>
<td><strong>Action Duration</strong></td>
<td>Instantaneous</td>
<td>Durative</td>
</tr>
<tr>
<td><strong>World Observability</strong></td>
<td>Full</td>
<td>Partial</td>
</tr>
<tr>
<td><strong>World Dynamics</strong></td>
<td>Static</td>
<td>Exogenous events</td>
</tr>
<tr>
<td><strong>Goal Attainment</strong></td>
<td>Full</td>
<td>Partial</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>No time points</td>
<td>Rich model of time</td>
</tr>
</tbody>
</table>

**Classical Planning Problem**
Background in Planning from CS221

- Planning is a search problem over states that can be structured (i.e., expressed as logical formulas)
- Classical planning algorithms
  - Progression, regression, plan space search
- Efficient algorithms based on planning graphs (Graph Plan)
- Use of heuristics
  - For example, A*

- For more details, refer to CS221 lecture notes
  - [http://www.stanford.edu/class/cs221/notes/cs221-lecture2.ppt](http://www.stanford.edu/class/cs221/notes/cs221-lecture2.ppt)
  - [http://www.stanford.edu/class/cs221/notes/cs221-lecture9.ppt](http://www.stanford.edu/class/cs221/notes/cs221-lecture9.ppt)
  
or
  Chapter on Planning in Russell & Norvig textbook
Goals for this Lecture

• Make a connection to the situation calculus representation that we covered during the last lecture
• Build on what you learned in CS221
  – Refresh some basic concepts as per the B&L textbook
    • Primarily aimed to keep consistency in the course material
  – Discuss application of classical planning to video games
  – Introduce FF – a state of the art planning algorithm that uses classical planning techniques + graph plan + heuristics
• Cover in-depth a knowledge-based planning technique
  – Hierarchical Task Network Planning
Using the situation calculus

The situation calculus can be used to represent what is known about the current state of the world and the available actions. The planning problem can then be formulated as follows:

Given a formula $Goal(s)$, find a sequence of actions $a$ such that

$$KB \models Goal(do(a, S_0)) \land Legal(do(a, S_0))$$

where $do(\langle a_1, \ldots, a_n \rangle, S_0)$ is an abbreviation for

$$do(a_n, do(a_{n-1}, \ldots, do(a_2, do(a_1, S_0)) \ldots))$$

and where $Legal(\langle a_1, \ldots, a_n \rangle, S_0)$ is an abbreviation for

$$Poss(a_1, S_0) \land Poss(a_2, do(a_1, S_0)) \land \ldots \land Poss(a_n, do(\langle a_1, \ldots, a_{n-1} \rangle, S_0))$$

So: given a goal formula, we want a sequence of actions such that

- the goal formula holds in the situation that results from executing the actions, and
- it is possible to execute each action in the appropriate situation
Planning by answer extraction

Having formulated planning in this way, we can use Resolution with answer extraction to find a sequence of actions:

\[ KB \models \exists s. \text{Goal}(s) \land \text{Legal}(s) \]

The textbook has a detailed example on how it can be done.

Modern planning algorithms use techniques that are customized to planning.
STRIPS

- Stanford Research Institute Problem Solver
  - Fikes & Nilsson, 1971

- Term used generally to refer to classical planning formulations

- Original STRIPS planner performed an incomplete form of backward chaining
The STRIPS representation

STRIPS is an alternative representation to the pure situation calculus for planning.

from work on a robot called Shaky at SRI International in the 60’s.

In STRIPS, we do not represent histories of the world, as in the situation calculus.

Instead, we deal with a single world state at a time, represented by a database of ground atomic wffs (e.g., In(robot,room1)).

This is like the database of facts used in procedural representations and the working memory of production systems.

Similarly, we do not represent actions as part of the world model (cannot reason about them directly), as in the situation calculus.

Instead, actions are represented by operators that syntactically transform world models.

An operator takes a DB and transforms it to a new DB.
STRIPS operators

Operators have pre- and post-conditions

- precondition = formulas that need to be true at start
- “delete list” = formulas to be removed from DB
- “add list” = formulas to be added to DB

Example: PushThru(o,d,r₁,r₂)

“the robot pushes object o through door d from room r₁ to room r₂”

- precondition: InRoom(robot,r₁), InRoom(o,r₁), Connects(d,r₁,r₂)
- delete list: InRoom(robot,r₁), InRoom(o,r₁)
- add list: InRoom(robot,r₂), InRoom(o,r₂)

STRIPS problem space = initial world model, DB₀ (list of ground atoms)
set of operators (with preconds and effects)
goal statement (list of atoms)
desired plan: sequence of ground operators
STRIPS Example

In addition to PushThru, consider

GoThru\((d, r_1, r_2)\):

- **precondition**: \(\text{InRoom(robot, } r_1)\), \(\text{Connects}(d, r_1, r_2)\)
- **delete list**: \(\text{InRoom(robot, } r_1)\)
- **add list**: \(\text{InRoom(robot, } r_2)\)

\[\begin{array}{c}
\text{ROOM}_1 \\
\text{ROBOT} \\
\text{DOOR}_1 \\
\text{BOX}_1 \\
\text{ROOM}_2 \\
\text{DOOR}_2 \\
\text{ROOM}_3
\end{array}\]

\(DB_0:\)

- \(\text{InRoom(robot, } \text{room}_1)\)
- \(\text{InRoom(box}_1, \text{room}_2)\)
- \(\text{Connects}(\text{door}_1, \text{room}_1, \text{room}_2)\)
- \(\text{Box(box}_1)\)
- \(\text{Connects}(\text{door}_2, \text{room}_2, \text{room}_3)\)  
  
  ... 

**Goal**: \[\text{[ Box}(x) \land \text{InRoom}(x, \text{room}_1) \text{ ]}\]
Progressive planning

Here is one procedure for planning with a STRIPS like representation:

Input: a world model and a goal
Output: a plan or fail. (ignoring variables)

ProgPlan[DB, Goal] =
  If Goal is satisfied in DB, then return empty plan
  For each operator $o$ such that precond($o$) is satisfied in the current DB:
    Let DB$'$ = DB + addlist($o$) - delist($o$)
    Let plan = ProgPlan[DB$'$, Goal]
    If plan $\neq$ fail, then return [act($o$); plan]
  End for
  Return fail

This depth-first planner searches forward from the given $DB_0$ for a sequence of operators that eventually satisfies the goal

DB$'$ is the progressed world state
Regressive planning

Here is another procedure for planning with a STRIPS like representation:

**Input**: a world model and a goal (ignoring variables)

**Output**: a plan or fail.

```
RegrPlan[DB,Goal] =
    If Goal is satisfied in DB, then return empty plan
    For each operator \( o \) such that \( \text{dellist}(o) \cap \text{Goal} = \{\} \):
        Let \( \text{Goal}' = \text{Goal} + \text{precond}(o) - \text{addlist}(o) \)
        Let plan = RegrPlan[DB,\text{Goal}']
        If plan \neq \text{fail}, then return [plan ; \text{act}(o)]
    End for
    Return fail
```

This depth-first planner searches backward for a sequence of operators that will reduce the goal to something satisfied in \( DB_0 \)

\( \text{Goal}' \) is the regressed goal
Computational aspects

Even without variables, STRIPS planning is NP-hard.

Many methods have been proposed to avoid redundant search
e.g. partial-order planners, macro operators

Heuristics, graph plan
Application of STRIPS Planning to a Game

- F.E.A.R. (short for First Encounter Assault Recon) is a horror-themed first-person shooter developed by Monolith Productions
  - Gamespot’s Best AI Award in 2005
  - Ranked 2nd in the list of most influential AI games
    - http://aigamedev.com/open/highlights/top-ai-games/
- Technical overview available at
  - http://web.media.mit.edu/~jorkin/gdc2006_orkin_jeff_fear.zip
Design Philosophy behind F.E.A.R

- Designer’s job is: Create environments that allow AI to showcase their behaviors.
- Designer’s job is NOT: Script behavior of individual AI
  - The behavior is a function of the plans that AI comes up with based on its goals and actions available to it

Adapted from Jeff Orkin
Actions Available to Key Characters

<table>
<thead>
<tr>
<th>Action</th>
<th>Action</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 AI/Actions/Attack</td>
<td>1 AI/Actions/Attack</td>
<td>1 AI/Actions/Animate</td>
</tr>
<tr>
<td>2 AI/Actions/AttackCrouch</td>
<td>2 AI/Actions/InspectDisturbance</td>
<td>2 AI/Actions/Idle</td>
</tr>
<tr>
<td>3 AI/Actions/SuppressionFire</td>
<td>3 AI/Actions/LookAtDisturbance</td>
<td>3 AI/Actions/GotoNode</td>
</tr>
<tr>
<td>4 AI/Actions/SuppressionFireFrom</td>
<td>4 AI/Actions/SurveyArea</td>
<td>4 AI/Actions/UseSmartObjectNodeMounted</td>
</tr>
<tr>
<td>Cover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 AI/Actions/FlushOutWithGrenade</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 AI/Actions/AttackFromCover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 AI/Actions/BlindFireFromCover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 AI/Actions/AttackGrenadeFromCover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 AI/Actions/AttackFromView</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 AI/Actions/DrawWeapon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 AI/Actions/HolsterWeapon</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 AI/Actions/ReloadCrouch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 AI/Actions/ReloadCovered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14 AI/Actions/InspectDisturbance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 AI/Actions/LookAtDisturbance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 AI/Actions/SurveyArea</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17 AI/Actions/DodgeRoll</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18 AI/Actions/DodgeShuffle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19 AI/Actions/DodgeCovered</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 AI/Actions/Uncover</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21 AI/Actions/AttackMelee</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Adapted from Jeff Orkin
Benefits of Planning

• Separation of Goals and Actions
• Layering of behavior
• Dynamic Problem Solving
  – Example video clips

Adapted from Jeff Orkin
Fast Forward Planning Algorithm

- Proposed by Hoffman and Nebel
- Winner of the 2000 planning competition
- Approach
  - Compile problem into grounded STRIPS
  - Perform Enforced-Hill-Climbing (EHC) until either solved or no further progress can be made.
    - Sound, not complete
  - Perform Best-First-Search
    - Sound, complete.
Using FF in the context of a Game

As part of HW3, Iceblox will be provided as an example use of FF. We will use FF planner to solve three configurations of Sokoban.
Goals for this Lecture

• Make a connection to the situation calculus representation that we covered during the last lecture
• Build on what you learned in CS221
  – Refresh some basic concepts as per the B&L textbook
    • Primarily aimed to keep consistency in the course material
  – Discuss application of classical planning to video games
  – Introduce FF – a state of the art planning algorithm that uses classical planning techniques + graph plan + heuristics
• Cover in-depth a knowledge-based planning technique
  – Hierarchical Task Network Planning
Motivation

- We may already have an idea how to go about solving problems in a planning domain
- Example: travel to a destination that’s far away:
  - Domain-independent planner:
    - many combinations of vehicles and routes
  - Experienced human: small number of “recipes”
    - e.g., flying:
      1. buy ticket from local airport to remote airport
      2. travel to local airport
      3. fly to remote airport
      4. travel to final destination

- How to enable planning systems to make use of such recipes?
HTN Planning

- Problem reduction
  - *Tasks* (activities) rather than goals
  - *Methods* to decompose tasks into subtasks
  - Enforce constraints
    » E.g., taxi not good for long distances
  - Backtrack if necessary
**HTN Planning**

- **Problem reduction**
  - *Tasks* (activities) rather than goals
  - *Methods* to decompose tasks into subtasks
  - Enforce constraints
    - E.g., taxi not good for long distances
  - Backtrack if necessary

Dana Nau: Lecture slides for Automated Planning
Licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License: http://creativecommons.org/licenses/by-nc-sa/2.0/
HTN Planning

- HTN planners may be domain-specific
- Or they may be domain-configurable
  - Domain-independent planning engine
  - Domain description that defines not only the operators, but also the methods
  - Problem description
    » domain description, initial state, initial task network
Simple Task Network (STN) Planning

- A special case of HTN planning
- States and operators
  - The same as in classical planning
- Task: an expression of the form \( t(u_1, \ldots, u_n) \)
  - \( t \) is a task symbol, and each \( u_i \) is a term
  - Two kinds of task symbols (and tasks):
    - **primitive**: tasks that we know how to execute directly
      - task symbol is an operator name
    - **nonprimitive**: tasks that must be decomposed into subtasks
      - use methods (next slide)
Methods

- Totally ordered method: a 4-tuple
  \[ m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m)) \]
  
  - \text{name}(m): an expression of the form \( n(x_1, \ldots, x_n) \)
    
    \[ x_1, \ldots, x_n \text{ are parameters - variable symbols} \]
  
  - \text{task}(m): a nonprimitive task
  
  - \text{precond}(m): preconditions (_literals)
  
  - \text{subtasks}(m): a sequence of tasks \( \langle t_1, \ldots, t_k \rangle \)

\[ \text{air-travel}(x, y) \]

\[ \text{task: } \text{travel}(x, y) \]

\[ \text{precond: } \text{long-distance}(x, y) \]

\[ \text{subtasks: } \langle \text{buy-ticket}(a(x), a(y)), \text{travel}(x, a(x)), \text{fly}(a(x), a(y)), \text{travel}(a(y), y) \rangle \]
Methods (Continued)

- Partially ordered method: a 4-tuple
  \[ m = (\text{name}(m), \text{task}(m), \text{precond}(m), \text{subtasks}(m)) \]
  - \text{name}(m): an expression of the form \( n(x_1, \ldots, x_n) \)
    - \( x_1, \ldots, x_n \) are parameters - variable symbols
  - \text{task}(m): a nonprimitive task
  - \text{precond}(m): preconditions (literals)
  - \text{subtasks}(m): a partially ordered set of tasks \( \{t_1, \ldots, t_k\} \)

\[
\begin{align*}
\text{air-travel}(x,y) \\
\text{task:} & \quad \text{travel}(x,y) \\
\text{precond:} & \quad \text{long-distance}(x,y) \\
\text{network:} & \quad u_1 = \text{buy-ticket}(a(x), a(y)), u_2 = \text{travel}(x, a(x)), u_3 = \text{fly}(a(x), a(y)) \\
& \quad u_4 = \text{travel}(a(y), y), \quad \{(u_1, u_3), (u_2, u_3), (u_3, u_4)\}
\end{align*}
\]
Domains, Problems, Solutions

- STN planning domain: methods, operators
- STN planning problem: methods, operators, initial state, task list
- Total-order STN planning domain and planning problem:
  - Same as above except that all methods are totally ordered

- Solution: any executable plan that can be generated by recursively applying
  - methods to non-primitive tasks
  - operators to primitive tasks

Dana Nau: Lecture slides for Automated Planning
Licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License: http://creativecommons.org/licenses/by-nc-sa/2.0/
Example

- Suppose we want to move three stacks of containers in a way that preserves the order of the containers.
A way to move each stack:

- first move the containers from $p$ to an intermediate pile $r$
- then move them from $r$ to $q$
Partial-Order Formulation

take-and-put\((c, k, l_1, l_2, p_1, p_2, x_1, x_2)\):
  task: \(\text{move-topmost-container}(p_1, p_2)\)
  precond: \(\text{top}(c, p_1), \text{on}(c, x_1)\); true if \(p_1\) is not empty
  attached\((p_1, l_1)\), \(\text{belong}(k, l_1)\); bind \(l_1\) and \(k\)
  attached\((p_2, l_2)\), \(\text{top}(x_2, p_2)\); bind \(l_2\) and \(x_2\)
  subtasks: \(\langle \text{take}(k, l_1, c, x_1, p_1), \text{put}(k, l_2, c, x_2, p_2) \rangle\)

recursive-move\((p, q, c, x)\):
  task: \(\text{move-stack}(p, q)\)
  precond: \(\text{top}(c, p), \text{on}(c, x)\); true if \(p\) is not empty
  subtasks: \(\langle \text{move-topmost-container}(p, q), \text{move-stack}(p, q) \rangle\)
    ;; the second subtask recursively moves the rest of the stack

do-nothing\((p, q)\)
  task: \(\text{move-stack}(p, q)\)
  precond: \(\text{top}(\text{pallet}, p)\); true if \(p\) is empty
  subtasks: \(\langle \rangle\); no subtasks, because we are done

move-each-twice()
  task: \(\text{move-all-stacks}()\)
  precond: ; no preconditions
  network: ; move each stack twice:
    \(u_1 = \text{move-stack}(p1a, p1b)\), \(u_2 = \text{move-stack}(p1b, p1c)\),
    \(u_3 = \text{move-stack}(p2a, p2b)\), \(u_4 = \text{move-stack}(p2b, p2c)\),
    \(u_5 = \text{move-stack}(p3a, p3b)\), \(u_6 = \text{move-stack}(p3b, p3c)\),
    \(\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}\)
take-and-put\((c, k, l_1, l_2, p_1, p_2, x_1, x_2)\):
  task: \(\text{move-topmost-container}(p_1, p_2)\)
  precond: \(\text{top}(c, p_1), \text{on}(c, x_1)\); true if \(p_1\) is not empty
  attached\((p_1, l_1)\), belong\((k, l_1)\); bind \(l_1\) and \(k\)
  attached\((p_2, l_2)\), top\((x_2, p_2)\); bind \(l_2\) and \(x_2\)
  subtasks: \(\langle \text{take}(k, l_1, c, x_1, p_1), \text{put}(k, l_2, c, x_2, p_2)\rangle\)

recursive-move\((p, q, c, x)\):
  task: \(\text{move-stack}(p, q)\)
  precond: \(\text{top}(c, p), \text{on}(c, x)\); true if \(p\) is not empty
  subtasks: \(\langle \text{move-topmost-container}(p, q), \text{move-stack}(p, q)\rangle\)
    ;; the second subtask recursively moves the rest of the stack

do-nothing\((p, q)\)
  task: \(\text{move-stack}(p, q)\)
  precond: \(\text{top}(\text{pallet}, p)\); true if \(p\) is empty
  subtasks: \(\langle \rangle\); no subtasks, because we are done

move-each-twice()
  task: \(\text{move-all-stacks()}\)
  precond: ; no preconditions
  subtasks: ; move each stack twice:
    \(\langle \text{move-stack}(p_1a, p_1b), \text{move-stack}(p_1b, p_1c),\)
    \(\text{move-stack}(p_2a, p_2b), \text{move-stack}(p_2b, p_2c),\)
    \(\text{move-stack}(p_3a, p_3b), \text{move-stack}(p_3b, p_3c)\rangle\)
Solving Total-Order STN Planning Problems

$$\text{TFD}(s, (t_1, \ldots, t_k), O, M)$$
if $k = 0$ then return $\langle \rangle$ (i.e., the empty plan)
if $t_1$ is primitive then
  $$\text{active} \leftarrow \{(a, \sigma) \mid a \text{ is a ground instance of an operator in } O,$$
  $$\sigma \text{ is a substitution such that } a \text{ is relevant for } \sigma(t_1),$$
  $$\text{and } a \text{ is applicable to } s\}$$
if $\text{active} = \emptyset$ then return failure
nondeterministically choose any $(a, \sigma) \in \text{active}$
$$\pi \leftarrow \text{TFD}(\gamma(s, a), \sigma((t_2, \ldots, t_k)), O, M)$$
if $\pi = \text{failure}$ then return failure
else return $a. \pi$
else if $t_1$ is nonprimitive then
  $$\text{active} \leftarrow \{m \mid m \text{ is a ground instance of a method in } M,$$
  $$\sigma \text{ is a substitution such that } m \text{ is relevant for } \sigma(t_1),$$
  $$\text{and } m \text{ is applicable to } s\}$$
if $\text{active} = \emptyset$ then return failure
nondeterministically choose any $(m, \sigma) \in \text{active}$
$$w \leftarrow \text{subtasks}(m). \sigma((t_2, \ldots, t_k))$$
return $\text{TFD}(s, w, O, M)$
Comparison to Forward and Backward Search

- In state-space planning, must choose whether to search forward or backward

- In HTN planning, there are two choices to make about direction:
  - forward or backward
  - up or down

- TFD goes down and forward
Comparison to 
Forward and Backward Search

- Like a backward search, 
  TFD is goal-directed
  - Goals correspond to tasks

- Like a forward search, it generates actions in the same order in which they’ll be executed
- Whenever we want to plan the next task
  - we’ve already planned everything that comes before it
  - Thus, we know the current state of the world
Limitation of Ordered-Task Planning

- TFD requires totally ordered methods

- Can’t interleave subtasks of different tasks
- Sometimes this makes things awkward
  - Need to write methods that reason globally instead of locally
Partially Ordered Methods

- With partially ordered methods, the subtasks can be interleaved

```
get-both(p,q)
```

```
get(p)
get(q)
```

```
walk(a,b) stay-at(b) pickup(p) pickup(q) walk(b,a) stay-at(a)
```

- Fits many planning domains better
- Requires a more complicated planning algorithm
Algorithm for Partial-Order STNs

\[ \text{PFD}(s, w, O, M) \]
if \( w = \emptyset \) then return the empty plan
nondeterministically choose any \( u \in w \) that has no predecessors in \( w \)
if \( t_u \) is a primitive task then
\[ \text{active} \leftarrow \{(a, \sigma) \mid a \text{ is a ground instance of an operator in } O, \right. \]
\[ \left. \quad \sigma \text{ is a substitution such that } \text{name}(a) = \sigma(t_u), \right. \]
\[ \left. \quad \text{and } a \text{ is applicable to } s \} \]
if \( \text{active} = \emptyset \) then return failure
nondeterministically choose any \( (a, \sigma) \in \text{active} \)
\[ \pi \leftarrow \text{PFD}(\gamma(s, a), \sigma(w - \{u\}), O, M) \]
if \( \pi = \text{failure} \) then return failure
else return \( a. \pi \)

else
\[ \text{active} \leftarrow \{(m, \sigma) \mid m \text{ is a ground instance of a method in } M, \right. \]
\[ \left. \quad \sigma \text{ is a substitution such that } \text{name}(m) = \sigma(t_u), \right. \]
\[ \left. \quad \text{and } m \text{ is applicable to } s \} \]
if \( \text{active} = \emptyset \) then return failure
nondeterministically choose any \( (m, \sigma) \in \text{active} \)
nondeterministically choose any task network \( w' \in \delta(w, u, m, \sigma) \)
return \( \text{PFD}(s, w', O, M) \)
Algorithm for Partial-Order STNs

If \( w = \emptyset \) then return the empty plan nondeterministically choose any \( u \in w \) that has no predecessors in \( w \)

- Intuitively, \( w \) is a partially ordered set of tasks \( \{t_1, t_2, \ldots\} \)
  - But \( w \) may contain a task more than once
    - e.g., travel from UMD to LAAS twice
  - The mathematical definition of a set doesn’t allow this

- Define \( W \) as a partially ordered set of task nodes \( \{u_1, u_2, \ldots\} \)
  - Each task node \( u \) corresponds to a task \( t_u \)

In my explanations, I talk about \( t \) and ignore \( u \)

Else

\[
\begin{align*}
\text{active} & \leftarrow \{(m, \sigma) \mid m \text{ is a ground instance of a method in } M, \\
& \sigma \text{ is a substitution such that name}(m) = \sigma(t_u), \\
& \text{and } m \text{ is applicable to } s\} \\
\text{if } \text{active} = \emptyset \text{ then return failure} \\
\text{nondeterministically choose any } (m, \sigma) \in \text{active} \\
\text{nondeterministically choose any task network } w' \in \delta(w, u, m, \sigma) \\
\text{return}(\text{PFD}(s, w', O, M))
\end{align*}
\]
Algorithm for Partial-Order STNs

\[ PFD(s, w, O, M) \]

if \( w = \emptyset \) then return the empty plan
nondeterministically choose any \( u \in w \) that has no predecessors in \( w \)
if \( t_u \) is a primitive task then
\[
\begin{align*}
\text{active} & \leftarrow \{ (a, \sigma) \mid a \text{ is a ground instance of an operator in } O, \\
& \quad \sigma \text{ is a substitution such that } \text{name}(a) = \sigma(t_u), \\
& \quad \text{and } a \text{ is applicable to } s \} \\
\end{align*}
\]
if \( \text{active} = \emptyset \) then return failure
nondeterministically choose any \((a, \sigma) \in \text{active})
\[ \pi \leftarrow PFD(\gamma(s, a), \sigma(w - \{u\}), O, M) \]
if \( \pi = \text{failure} \) then return failure
else return \( a. \pi \)
else
\[
\begin{align*}
\text{active} & \leftarrow \{ (m, \sigma) \mid m \text{ is a ground instance of a method in } M, \\
& \quad \sigma \text{ is a substitution such that } \text{name}(m) = \sigma(t_u), \\
& \quad \text{and } m \text{ is applicable to } s \} \\
\end{align*}
\]
if \( \text{active} = \emptyset \) then return failure
nondeterministically choose any \((m, \sigma) \in \text{active})
nondeterministically choose any task network \( w' \in \delta(w, u, m, \sigma) \)
return \( \text{PFD}(s, w', O, M) \)
Algorithm for Partial-Order STNs

\( \delta(w, u, m, \sigma) \) has a complicated definition in the book. Here’s what it means:

- We nondeterministically selected \( t_1 \) as the task to do first
- Must do \( t_1 \)'s first subtask before the first subtask of every \( t_i \neq t_1 \)
- Insert ordering constraints to ensure that this happens
Comparison to Classical Planning

STN planning is strictly more expressive than classical planning

- Any classical planning problem can be translated into an ordered-task-planning problem in polynomial time
- Several ways to do this. One is roughly as follows:
  - For each goal or precondition $e$, create a task $t_e$
  - For each operator $o$ and effect $e$, create a method $m_{o,e}$
    - Task: $t_e$
    - Subtasks: $t_{c1}, t_{c2}, \ldots, t_{cn}$, $o$, where $c_1, c_2, \ldots, c_n$ are the preconditions of $o$
    - Partial-ordering constraints: each $t_{ci}$ precedes $o$

- (I left out some details, such as how to handle deleted-condition interactions)
Comparison to Classical Planning (cont.)

- Some STN planning problems aren’t expressible in classical planning
- Example:
  - Two STN methods:
    - No arguments
    - No preconditions
  - Two operators, \(a\) and \(b\)
    - Again, no arguments and no preconditions
  - Initial state is empty, initial task is \(t\)
  - Set of solutions is \(\{a^n b^n \mid n > 0\}\)
  - No classical planning problem has this set of solutions
    - The state-transition system is a finite-state automaton
    - No finite-state automaton can recognize \(\{a^n b^n \mid n > 0\}\)
- Can even express undecidable problems using STNs
SHOP2

- SHOP2: implementation of PFD-like algorithm + generalizations
  - Won one of the top four awards in the AIPS-2002 Planning Competition
  - Freeware, open source
HTN Planning

- HTN planning is even more general
  - Can have constraints associated with tasks and methods
    - Things that must be true before, during, or afterwards
  - Some algorithms use causal links and threats like those in PSP
Domain-Configurable Planners Compared to Classical Planners

- Disadvantage: writing a knowledge base can be more complicated than just writing classical operators
- Advantage: can encode “recipes” as collections of methods and operators
  - Express things that can’t be expressed in classical planning
  - Specify standard ways of solving problems
    - Otherwise, the planning system would have to derive these again and again from “first principles,” every time it solves a problem
    - Can speed up planning by many orders of magnitude (e.g., polynomial time versus exponential time)
Goals for this Lecture

- Make a connection to the situation calculus representation that we covered during the last lecture
- Build on what you learned in CS221
  - Refresh some basic concepts as per the B&L textbook
    » Primarily aimed to keep consistency in the course material
  - Discuss application of classical planning to video games
  - Introduce FF – a state of the art planning algorithm that uses classical planning techniques + graph plan + heuristics
- Cover in-depth a knowledge-based planning technique
  - Hierarchical Task Network Planning
Readings

• Required
  – Chapter 15 in B&L Textbook
  – Chapter 11 of Automated Planning by Ghallab, Nau and Traverso

• Optional
  – Olivier Bartheye and Eric Jacopin: A PDDL-Based Planning Architecture to Support Arcade Game Playing