From Classical Planning to Hierarchical Planning, From Modeling Problems to Solving Them.

Pascal Bercher

(Intelligent Systems and Foundations Clusters)

School of Computing
College of Systems and Society
The Australian National University

June 16, 2025

Australian National University

Australian Government
Australian Research Council

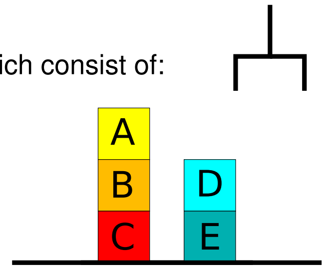
HTN Planning: Basics

Recap on Classical Planning

Recap on Classical Planning: Classical Planning

We consider *classical planning problems*, which consist of:

- All existing state variables V.
- An initial state $s_I \in 2^V$.
- A set of available actions A.
- A goal description $g \subseteq V$.



→ Find an action sequence (i.e., a *plan*) that transforms s_I into g.

For example, one of the available actions is:

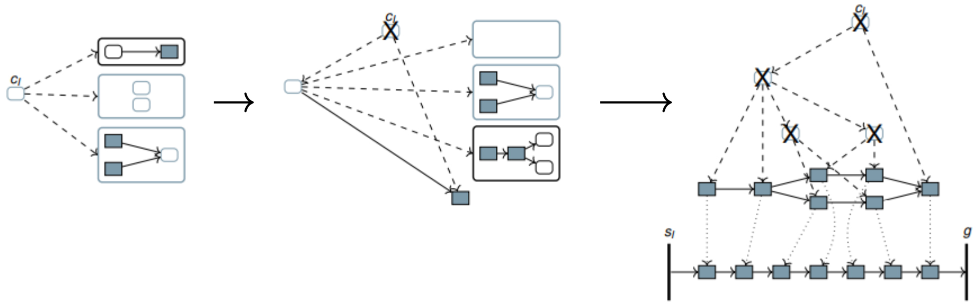
Preconditions	Action	Effects
gripperFree	**unstack** (?b1,?b2)	¬gripperFree
clear(?b1)		holding(?b1)
on(?b1,?b2)		¬on(?b1,?b2)
		¬clear(?b1)
		clear(?b2)

- For an action to be executable, all preconditions must hold.
- Actions change states by adding or deleting their effects.

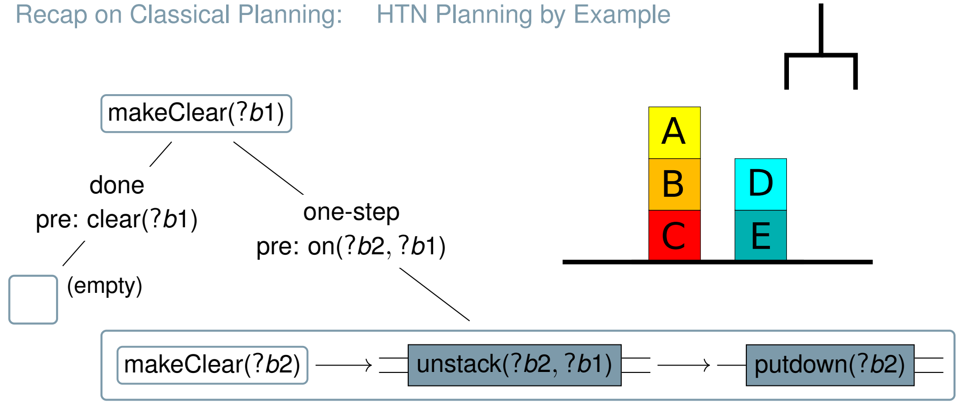
Recap on Classical Planning: Hierarchical Task Network (HTN) Planning

In HTN Planning,

- we do not plan for state-based variables, instead,
- we have initial *compound* tasks that need to be refined for which the model contains “methods”, the refinement rules.
- The solution is an executable, primitive task network (refinement).



Recap on Classical Planning: HTN Planning by Example



makeClear(C)
~~makeClear(C)~~ makeClear(B) unstack(B,C) putdown(B)
~~makeClear(C)~~ ~~makeClear(B)~~ makeClear(A) unstack(A,B) putdown(A) unstack(B,C) putdown(B)
~~makeClear(C)~~ ~~makeClear(B)~~ ~~makeClear(A)~~ unstack(A,B) putdown(A) unstack(B,C) putdown(B)

Note that in this model, we only need the predicates *clear(?b)* and *on(?b1,?b2)* – since the “logic” is encoded into the task hierarchy.

Recap: Formal Grammars

Recap: Formal Grammars: Definitions

Recap from Theoretical Computer Science:

A *context-free grammar* G is a tuple $\langle N, \Sigma, S, R\rangle$ where

- N is a finite set of *non-terminal symbols*,
- Σ, disjoint from N, is a finite set of *terminal symbols* (Σ is also called *alphabet*),
- $S \in N$ is the *start symbol*,
- $R \subseteq N \times (N \cup \Sigma)^*$ is a finite set of *production rules*.

Languages:

- A language L is any (possibly infinite) set of words (sequences of symbols). E.g., the sets $\emptyset$, $\{abc, \ldots, xyz\}$, and $\mathbb{N}$ are languages.
- The *language of a grammar*, $L(G) \subseteq \Sigma^*$, is the set of terminal words obtainable by refining S by only using production rules.

Recap: Formal Grammars: Example

- Let $G = \langle \{a, b\}, \{S, A, B\}, S, \{S \to aB, B \to Ab, A \to S, A \to \epsilon\} \rangle$, so we have:
  - Terminal symbols: $\{a, b\}$
  - Non-terminals: $\{S, A, B\}$
  - Start symbol: S
  - Production rules:
    - $S \to aB$
    - $B \to Ab$
    - $A \to S \mid \epsilon$
- Some example derivations:
  - $S \longrightarrow aB \longrightarrow aAb \longrightarrow ab$
  - $S \longrightarrow aB \longrightarrow aAb \longrightarrow aSb \longrightarrow \cdots \longrightarrow aabb$
- So, the language of G is $L(G) = \{a^n b^n \mid n \geq 1\}$

HTN Problem Definition

HTN Problem Definition: Based on Grammars/Languages

- Actions are defined by their name: $\delta : N_P \to 2^V \times 2^V \times 2^V$. Thus, solutions are (the same as) sequences of action names.
- Thus, any solution set $sol(\mathcal{P})$ is a language. Let:
  - $L_H(\mathcal{P}) = \{\bar{p} \mid \bar{p} \in L(G_P)$, where G_P is the induced grammar $\}$
  - $L_C(\mathcal{P}) = \{\bar{p} \mid \bar{p} \in sol(\mathcal{P}')$, where $\mathcal{P}'$ is the induced classical problem $\}$
- Now we can decompose the solution criteria:
  - L_H just looks at the word produced by the hierarchy,
  - L_C just looks at the executable words that produce the goal.
  - → $sol(\mathcal{P}) = L_H(\mathcal{P}) \cap L_C(\mathcal{P})$.

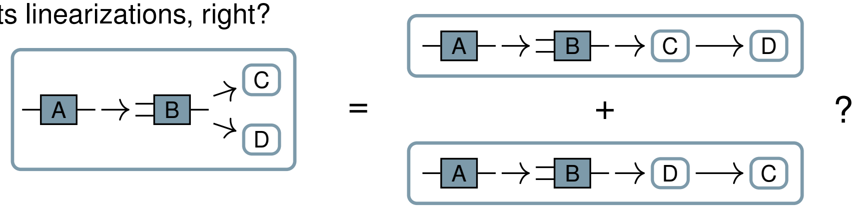
This observation gives a new/simplified view on HTN planning:

HTN planning = classical planning + grammar to filter solutions

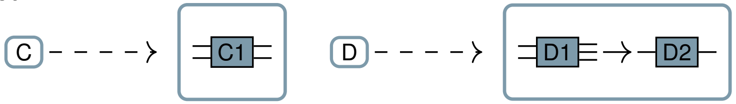
PO vs. TO

PO vs. TO: On the (Non?)equivalence of PO and TO HTN models

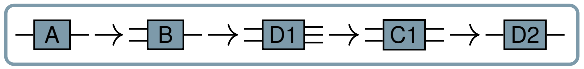
Each partially ordered task network is just a compact representation of its linearizations, right?



Let:



Can we create the following task network?

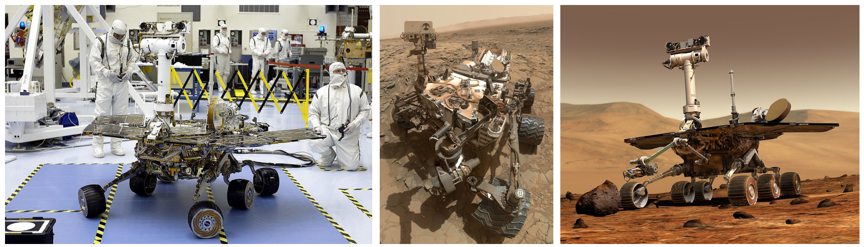


No! Not anymore...

Applications

Mars Rovers

Mars Rovers: Robotics (here: Mars Rovers Spirit and Opportunity)



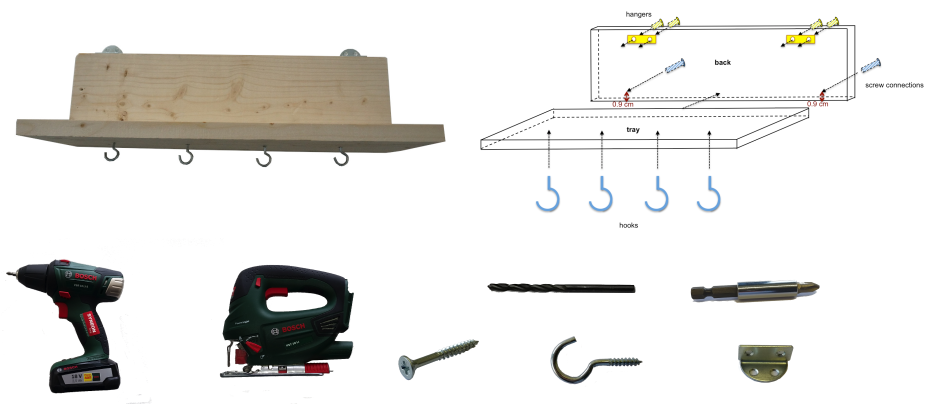
Source:
- left `https://commons.wikimedia.org/wiki/File:KSC-03PD-0786.jpg`
- middle `https://commons.wikimedia.org/wiki/File:Curiosity_Self-Portrait_at_%27Big_Sky%27_Drilling_Site.jpg`
- right `https://commons.wikimedia.org/wiki/File:NASA_Mars_Rover.jpg`

Copyright: public domain

Further reading:
- `https://www.nasa.gov/` and papers about *MAPGEN*.

DIY Assistant

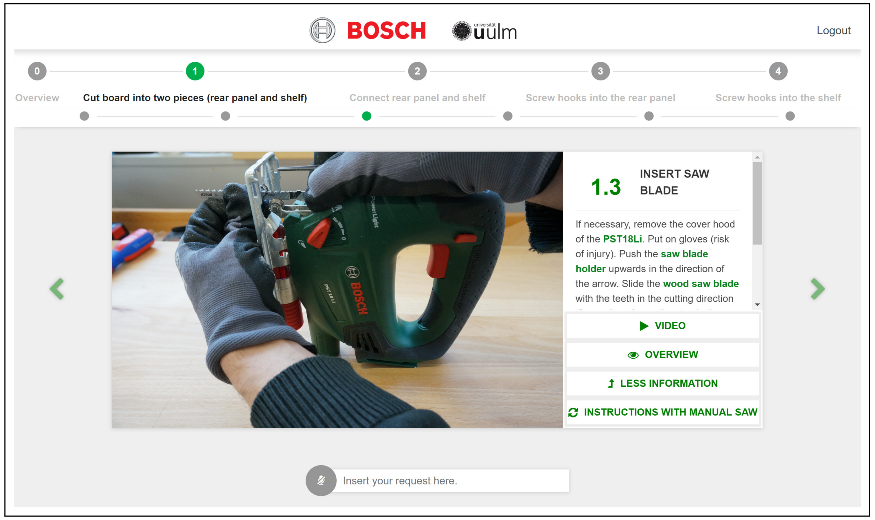
DIY Assistant: Overview



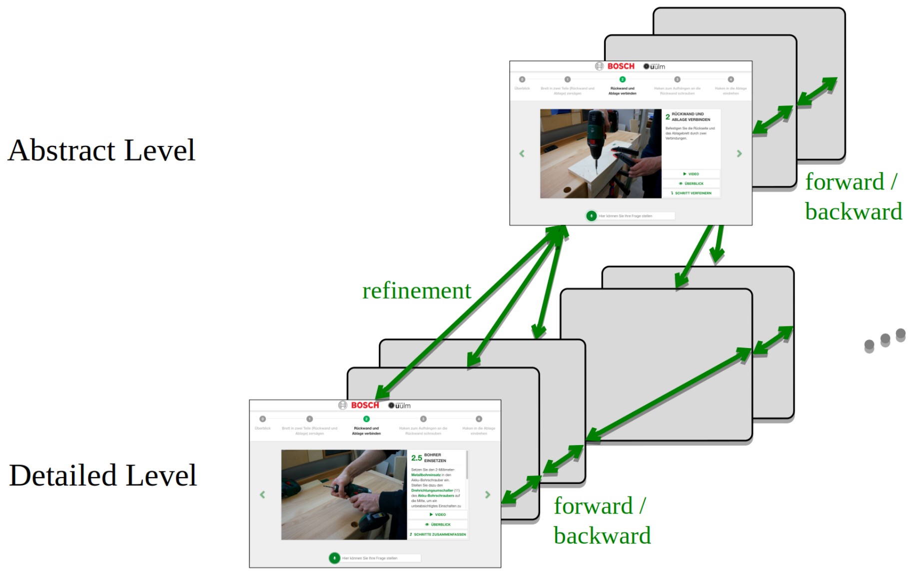
The material:

- Boards (need to be cut first)
- Electrical devices like drills and saws
- Attachments like drill bits and materials like nails

DIY Assistant: User Interface



DIY Assistant: Task Hierarchy



So, why HTN planning?

So, why HTN planning?

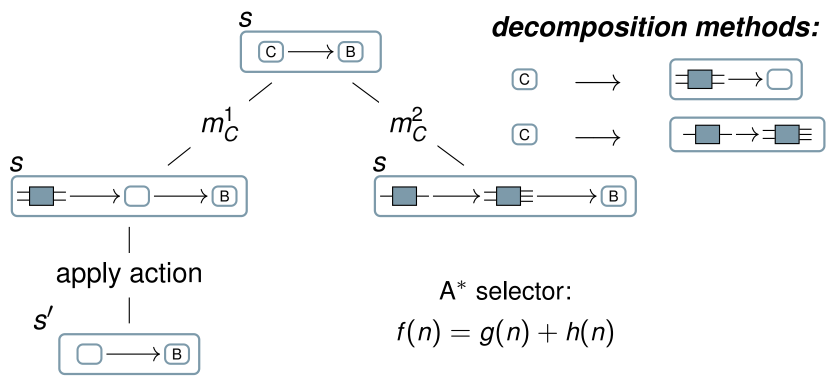
There are three main motivations:

- Modeling should be made easy, and experts often have hierarchical, "rule-based" knowledge.
- We get more control over solutions. We can *exclude* more. More technically:
  - TO HTN planning: exactly context-free, e.g., $L = \{a^n b^n \mid n \in \mathbb{N}\}$.
  - PO HTN planning: strictly above context-free, strictly within context-sensitive.
- Present action plans on multiple levels of abstraction.

Progression Search to Solve HTN Problems

HTN Progression Search via A^*

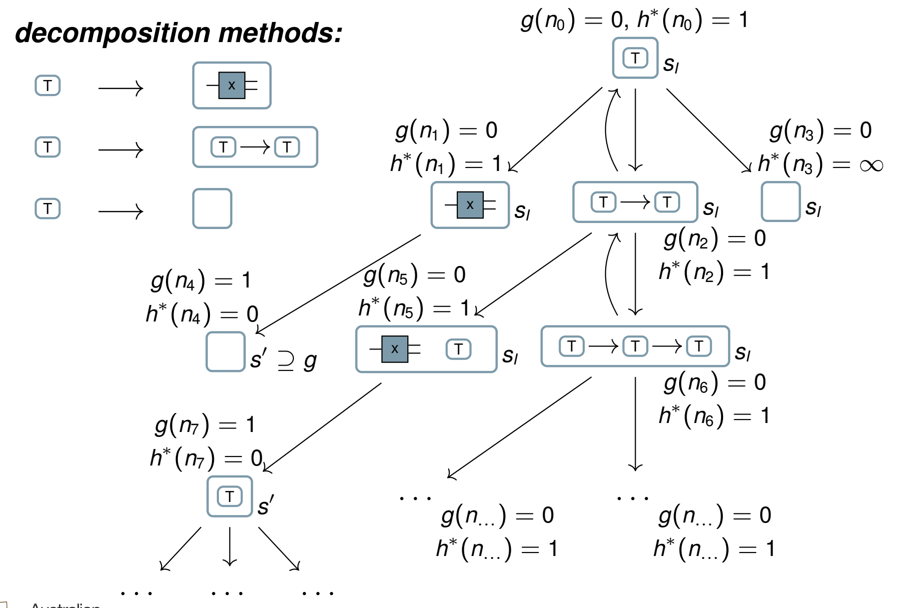
- **node selection:** Select a node with minimal f value.
  - g: cost incurred so far (number of progressed/applied actions)
  - h: estimate of number of actions to still be applied
- **node expansion:**
  - primitive? Progress it! (Update the state.)
  - compound? Apply all its decomposition methods!
- **solution:** The action sequences encoded in the search path.



A* selector:

$f(n) = g(n) + h(n)$

A^* is Incomplete: We have infinitely many nodes with perfect f-value!



Key Messages

So, what about progression search?

- Progression search is a state-of-the-art approach.
- (There are several heuristics and pruning techniques.)
- However, even with *perfect heuristic* and total-order HTN problems, search might get stuck in an infinite loop. *Depending on properties of the hierarchy.* (Keyword: grow-and-shrink cycle!)
- The issue can be solved, but it's not implemented yet.

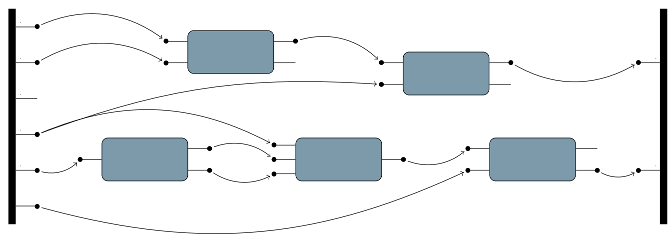
What about other algorithms?

- There's planning as SAT (translation into a series of SAT formulae).
- Search in the space of POCL plans (see later!). But: usually that's done for planning with time only. More crucially: The results from earlier apply to that paradigm as well!

So, currently, only planning as SAT guarantees termination in all cases.

Non-sequential Plans

Motivation

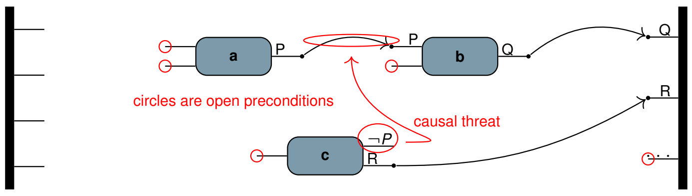


In such plans, every linearization is a classical solution!

Such partially ordered plans are used:

- In many HTN planning formalizations and planners.
- When we want to execute actions in parallel: minimize time (But be cautious, see next slide.)
- It's used for many plan optimizations techniques (see my IJCAI'24 survey on plan optimization)

PO and POCL Plans



When is a Partial-Order Causal Link (POCL) plan a solution?

- When all preconditions are supported by a causal link, and
- there are no causal threats. Threats can be resolved by adding ordering constraints moving *c* appropriately.

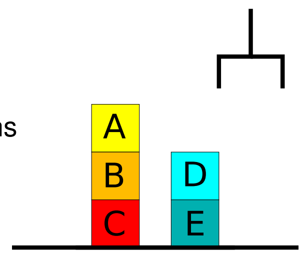
There are also other kinds of non-parallel plans:

- PO plans (same as POCL, but without links)
- parallel plans: a sequence of sets of actions, such that:
  - in each layer, every linearization is executable
  - in each layer, every linearization results into the same (unique) state

Modeling Support

Modeling Blocksworld

Modeling Blocksworld: Blocksworld Revisited



We want automated support in creating actions (and their interactions) like the one below:

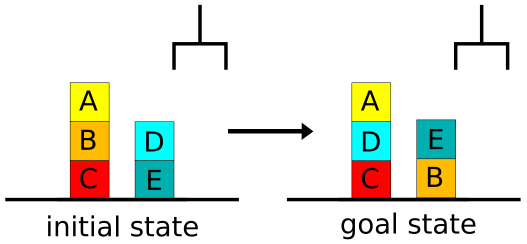
Is the action correctly modeled?

```
(:action unstack
   :parameters (?b1 ?b2 - block)
   :precondition (and (gripperFree)
                      (on ?b1 ?b2) (clear ?b1))
   :effect (and (not (gripperFree)) (holding ?b1)
                (not (on ?b1 ?b2)) (not (clear ?b1))
                (clear ?b2)))
```

Yes! We had that action in slide 1! :)

Modeling Blocksworld: Patient Zero

How about this situation? Is it modeled correctly?



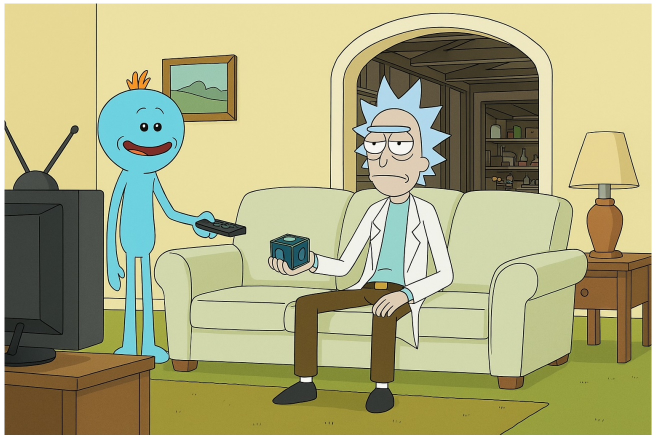

```
(define (problem blocksworld-prob)
  (:domain blocksworld)
  (:init (clear A) (on A B) (on B C) (onTable C)
         (clear D) (on D E) (onTable E))
  (:goal (and (clear A) (on A D) (on D C) (onTable C)
              (clear E) (on E B) (onTable B))))
```

No! The gripper being initially empty is missing!

A more Complex Example

A more Complex Example: Example (Teaser)

We'll provide an example in the following domain:



- We have a remote (in the garage) and Meeseeks box (in the den)
- Rick wants the TV being turned on with the remote.

A more Complex Example: Problem Definition

A lifted classical planning problem $\langle \mathcal{T}, \mathcal{P}, \mathcal{O}, \mathcal{A}, s_I, g \rangle$ consists of:

- $\mathcal{T}$ is a finite set of hierarchical *types*. *Example(s):* character – object
- $\mathcal{P}$ is a finite set of *predicate symbols*, each with fixed arity, i.e., it takes a sequence of typed variables. *Examples:* At(?room – room,?object – object)
- $\mathcal{O}$ is a finite set of (typed) *objects* used to ground action schemas (and predicates). *Examples:* Box, Remote – object; Rick, Meeseeks – character
- $\mathcal{A}$ is a finite set of *action schemas* of the form:

$$(\mathtt{name}(\vec{x}), \mathtt{pre}(\vec{x}), \mathtt{add}(\vec{x}), \mathtt{del}(\vec{x}))$$

  where $\vec{x}$ is a list of (typed) variables. *Examples:* next slide!
- s_I is the initial state, given as a finite set of ground atoms.
- g is the goal description, a finite set of ground atoms.

A more Complex Example: Example: *Lifted* Classical Planning Problem

Types: room, object; character – object (i.e., character is-a object)
Objects: Remote, Box – object; R, M – character; Den, Garage – room

$s_I = \{$At(Den,R), At(Garage,Remote), At(Den,Box), TV-Off()$\}$ $\quad g = \{$TV-On()$\}$

Available action schemata:

PushBox(?room,?character**)**:
({At(?room,Box), At(?room,?character)},{At(?room,M)},$\emptyset$)

GoTo(?room-f,?room-t,?character**)**:
({At(?room-f,?character)},{At(?room-t,?character)},{At(?room-f,?character)})

PickUp(?object,?room,?character**)**:
({At(?room,?character), At(?room,?object)},
{Has(?object,?character)},{At(?room,?object)})

Give(?object,?room,?character-f,?character-t**)**:
({Has(?object,?character-f), At(?room,?character-f), At(?room,?character-t)},
{Has(?object,?character-t)}, {Has(?object,?character-f)})

TurnTVOn(?character**)**:
({Has(Remote,?character), At(Den,?character), TV-Off()},{TV-On()},{TV-Off()})

A more Complex Example: Example Problem, Solutions

Recap: $s_I = \{$At(Den,Box), At(Den,R), At(Garage,Remote), TV-Off()$\}$.

Solution 1 (Rick does it himself):

GoTo(Den,Garage,R)	{At(Den,Box), At(Garage,R), At(Garage,Remote), TV-Off()}
PickUp(Remote,Garage,R)	{At(Den,Box), At(Garage,R), Has(Remote,R), TV-Off()}
GoTo(Garage,Den,R)	{At(Den,Box), At(Den,R), Has(Remote,R), TV-Off()}
TurnTVOn(R)	{At(Den,Box), At(Den,R), Has(Remote,R), TV-On()}

Solution 2 (Rick uses a Meeseeks):

PushBox(Den,R)	{At(Den,Box), At(Den,R), At(Den,M), At(Garage,Remote), TV-Off()}
GoTo(Den,Garage,M)	{At(Den,Box), At(Den,R), At(Garage,M), At(Garage,Remote), TV-Off()}
PickUp(Remote,Garage,M)	{At(Den,Box), At(Den,R), At(Garage,M), Has(Remote,M), TV-Off()}
GoTo(Garage,Den,M)	{At(Den,Box), At(Den,R), At(Den,M), Has(Remote,M), TV-Off()}
Give(Remote,Den,M,R)	{At(Den,Box), At(Den,R), At(Den,M), Has(Remote,R), TV-Off()}
TurnTVOn(R)	{At(Den,Box), At(Den,R), At(Den,M), Has(Remote,R), TV-On()}

Recap: $g = \{$TV-On()$\}$.

A more Complex Example: Modeling is hard... Example 2

Recall the action **PushBox(**?room,?character**)**:
({At(?room,Box), At(?room,?character)}, {At(?room,M)}, $\emptyset$)

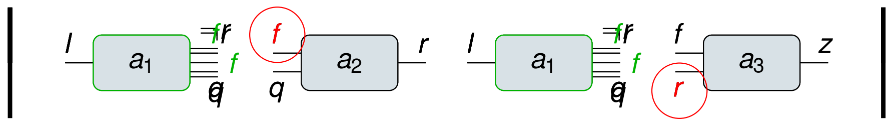
Q: How many Meeseeks can we have at any time?

- **A:** One:
  - If a Meeseeks presses the box, it has to be in said room already! Then, adding At(?room,M) doesn't change anything.
  - If Rick presses the box multiple times, adding At(?room,M) doesn't matter: states are *sets*.
- **Better A:** No! It's one per room!
  - Rick could use the Meeseeks box in any room, or press it again once the Meeseeks left.
  - Maybe unanticipated side effects:
    - Meeseeks get "fused" when walking into a room with a Meeseeks.
    - All Meeseeks share one inventory. (Only one Has(?object,M)!)

Thus, modeling is hard... The model might not do what we think/want...

A more Complex Example: How to provide Support?

We use the test-and-verify approach, based on hitting sets:



$s_I = \{l, r\}$ $\qquad g = \{z\}$

I.e., we provide a set of of plans:

- Some are supposed to be solutions (but are not), *white list plans*
- others should not be solutions (but are). *black list plans*

We aim at a *cardinality-minimal* number of repairs that satisfy these constraints. (For "better" answers we will need LLMs.)

We do this via an NP-complete *Hitting Set* approach.

Summary

We learned about...

- HTN planning = classical planning + grammar as filter to exclude some solutions.
- Partial Order HTN planning can express more than Total-Order HTN planning.
- Even with perfect heuristic, the most famous approach for solving HTN problems, progression search, can get stuck in an infinite loop (even for TO HTN problems).
  - This happens only for "certain kind of hierarchies" – and only for A^*, i.e., for optimal solutions.
  - This can also be fixed by reformulation (not yet implemented)
  - Other optimal approaches exist as well, such as compilation as SAT
- Modeling domains is complex and error-prone, but
- there is support technology, such as providing solution and non-solution plans (to fix the model automatically).

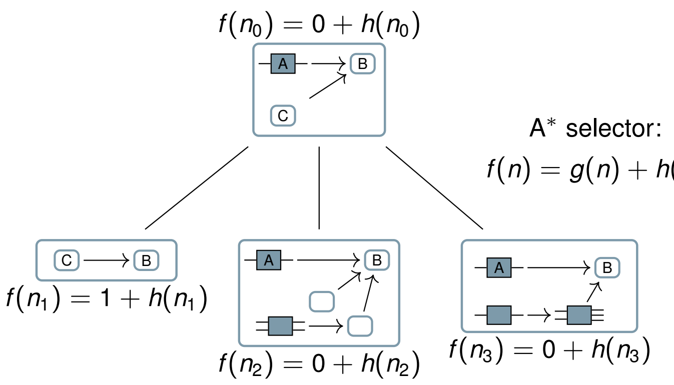
Thank you! :)

Progression Search

A^* for Partial-Order HTN Planning

A^* for Partial-Order HTN Planning: Overview

- Heuristics receive as input a search node (task network *tn*)
- and as output estimate:
  - the cost of a (cheapest) solution reachable from *tn* or
  - the number of progression steps to reach a (cheapest) solution.

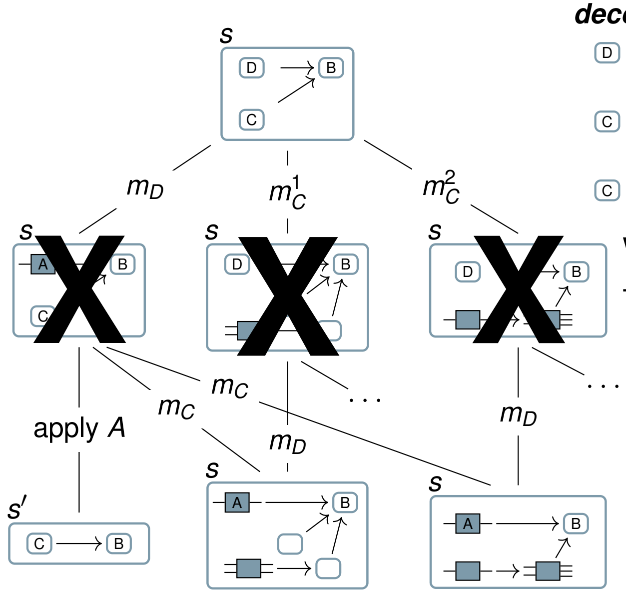


A* selector:

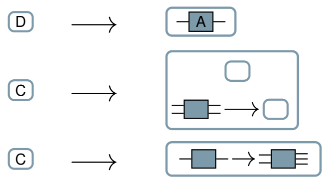
$f(n) = g(n) + h(n)$

Unsystematic Progression Search

Unsystematic Progression Search: We can do better! (Make search systematic)



decomposition methods:



We have redundancy! :(

To fix:

- *Pick* a compound task!
- Don't apply actions unless first compound tasks are gone
- Only use TO HTNs! But:
- A^* even with perfect heuristic is incomplete!

Application Examples

Assembly Assistant

Assembly Assistant: Example: Home Theater Assembly Assistant

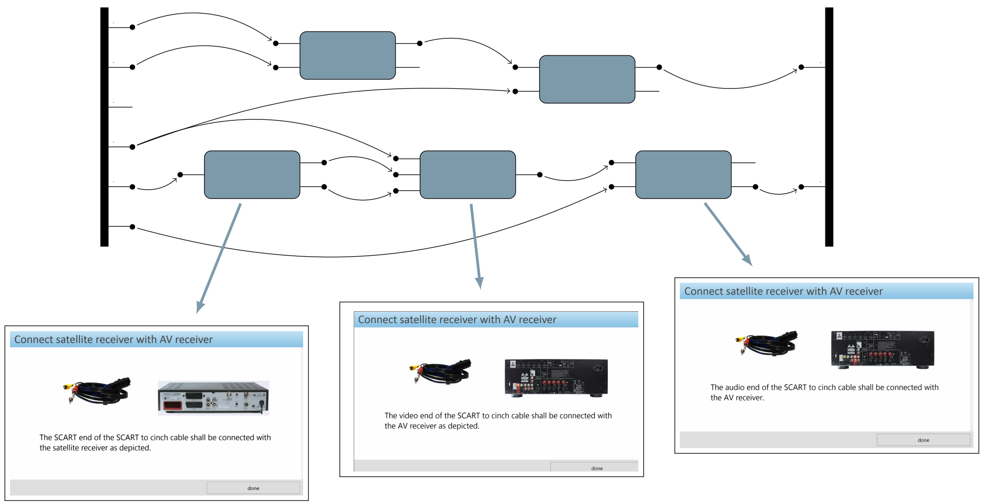
Sink devices:

- Television (requires video)
- Amplifier (requires audio)

Source devices:

- Blu-ray player
- Satellite receiver
  (both produce audio & video)

Assembly Assistant: Solution Plan



Assembly Assistant: Definitions, Examples

- Planning problems are usually defined in terms of a description language based on a first-order predicate logic.
  - Predicates, like $HasPort(?device, ?port)$, express relationships between variables representing objects.
  - Constants, like $AMPLIFIER$ and $CABLE\_HDMI$, represent objects.
- States are sets of (ground) propositions, e.g.,
  $s \supseteq \{HasPort(AMPLIFIER, HDMI),$
  $HasPort(AMPLIFIER, CINCH),$
  $HasPort(CABLE\_HDMI, HDMI),$
  $IsConnected(AMPLIFIER, CABLE\_HDMI, HDMI)\}$


(connected to each other)

- Actions are defined by preconditions and effects, e.g.,

In$(?cable, ?device, ?port)$

precondition: $HasPort(?device, ?port) \wedge$

Assembly Assistant: Planning Problem Definition in the Home Theater Domain

Initial state:

- HasPort(. . . , . . .) // which device has which ports?
- IsConnected(. . . , . . . , . . .) // how are the connections initially?
- HasSignal(. . . , . . . , . . .) // which device has which signals?

Action portfolio:

- ***plugIn(****?cable*, ?*device*, ?*port****)*** // plugging in a cable
- ***plugOut(****?cable*, ?*device*, ?*port****)*** // in case plugging out is allowed

Goal description:

- HasSignal(. . . , . . . , . . .) // e.g., HasSignal(TV, VIDEO, BR) denoting
- . . . that the TV has the video signal of the blu-ray player

Solitaire

Solitaire: Games, e.g., Solitaire

Rush Hour

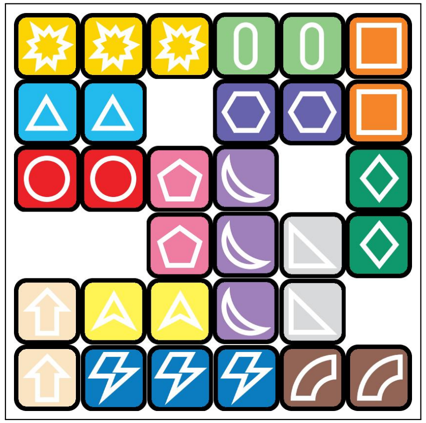
Rush Hour: Games, e.g., ..?

Photo made by Bercher (Dec. 2020) at the ANU.

Rush Hour: Games, e.g., Rush Hour

- Start: any configuration of cars with an exit on one specific side.
- Goal: Get the red car out to the right.

Rush Hour: Games, e.g., Rush Hour



- Start: any configuration of cars with an exit on one specific side.
- Goal: Get the red car out to the right.

Modeling this was a research project; reach out if interested in more!