02285 AI and MAS, SP18
Partial-order planning and hierarchical task networks

Curriculum for 4:

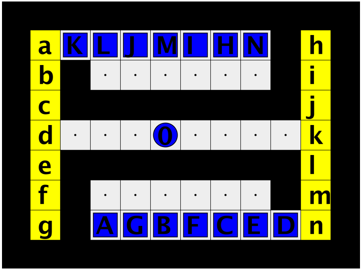
- Section 11.3 *Partial-Order Planning* of **2nd edition** of Russell & Norvig (available on DTU Insider)
- Section 10.5 *Analysis of Planning Approaches* of Russell & Norvig 3ed
- Section 11.2 *Hierarchical Planning* of Russell & Norvig 3ed
- Jeff Orkin: *Three States and a Plan: The A.I. of F.E.A.R.* (available on DTU Inside).

Todays subjects:

- Partial-order planning.
- Hierarchical task networks.

Serialisable sugoals

Planning problem with **serialisable subgoals**: There exists an order of the subgoals, such that they can be achieved in that order without destroying any of the previously achieved subgoals.

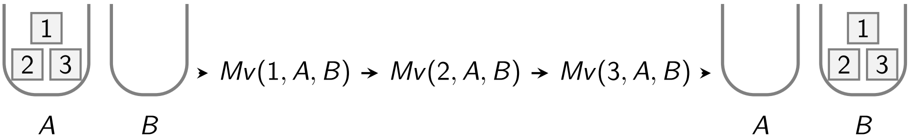


Trying the **Crunch level** from the warmup assignment:

Solver	**Time**	**Plan length**	**Nodes generated**
BFS	297*s*	98	$\approx 9 \cdot 10^6$
BFS utilising ser. subgoals	0.25s	106	27.831
$A^\star$	0.86s	103	24.044
$A^\star$ utilising ser. subgoals	0.12s	106	411
greedy	0.25s	164	2.792
greedy utilising ser. subgoals	0.10s	104	214

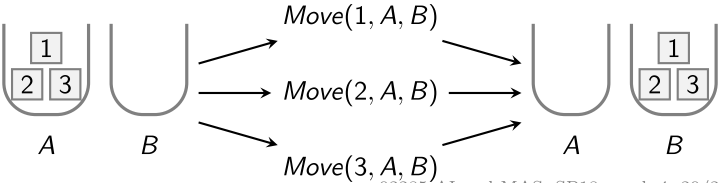
Linear sequences versus partial orders

Progression and regression planning explore **linear sequences of actions**.



This is often not advantageous. E.g. considering in which order to move the boxes will just make the planning process more time-consuming.

We'd like some degree of **partiality** in the ordering of actions, e.g. the order in which to move the boxes or the order in which to buy milk and bananas.



Partial-order planning

A generalised treatment of partiality is found in **partial-order planning (POP)**.

Partial-order planning (POP): A planning technique producing partially ordered plans, ignoring the ordering of independent actions.

Partial-order planning is in particular effective for **partly decomposable** problems like e.g. the gripper problem (or planning a party).

Example: Socks-and-shoes

Putting on your socks and shoes...

Action schemas:

ACTION: *LeftSock*
PRECONDITION:
EFFECT: *LeftSockOn*

ACTION: *LeftShoe*
PRECONDITION: *LeftSockOn*
EFFECT: *LeftShoeOn*

ACTION: *RightSock*
PRECONDITION:
EFFECT: *RightSockOn*

ACTION: *RightShoe*
PRECONDITION: *RightSockOn*
EFFECT: *RightShoeOn*

Initial state: empty.

Goal: $\mathit{LeftShoeOn} \wedge \mathit{RightShoeOn}$.

Note the independence of the actions on the left from the actions on the right. Partial-order planning takes advantage of this.

Partial orders

In partial-order planning a partial order $\prec$ on the required actions is built up in a stepwise fashion.

Example. In the case of the socks-and-shoes problem, the partial order would become:

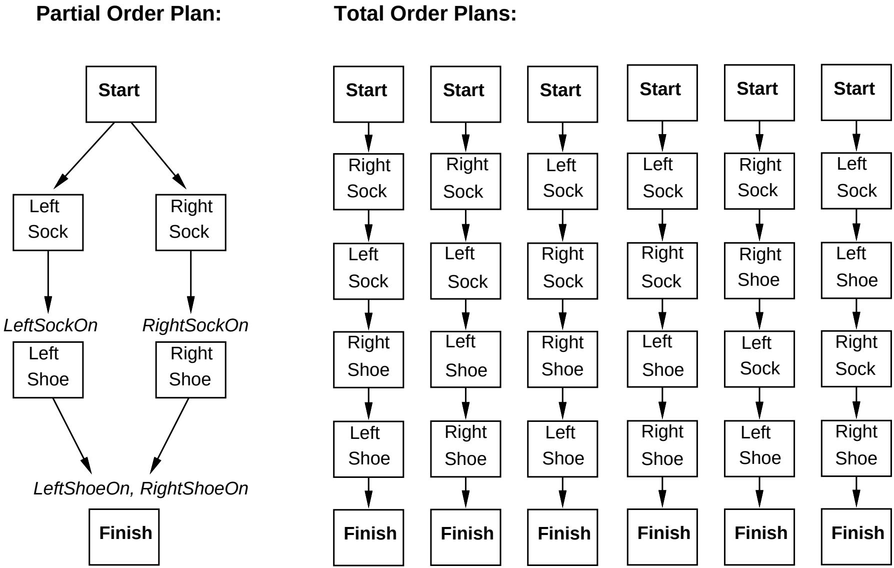
$$\mathit{LeftSock} \prec \mathit{LeftShoe}, \mathit{RightSock} \prec \mathit{RightShoe}.$$

Expresses: put on left sock **before** left shoe, and put on right sock **before** right shoe.

Linearisation of a partial order: Any total order that extends it. Polynomial-time algorithms for linearisation: topological sort.

Solution: A partial order $\prec$ is called a **solution** to a planning problem if any **linearisation** of it constitutes a **plan** (action sequence leading from start to goal).

Partial-order planning for socks-and-shoes



Compare with the simplified Gripper problem.

Ordering constraints

Ordering constraint: An expression of the form $A \prec B$, where A and B are actions. Intended interpretation: Action A has to come before action B.

Labelled ordering constraint: An expression of the form $A \stackrel{c}{\prec} B$, where:

- A and B are actions.
- c is an **effect** of A (one of the literals in the effect of A).
- c is a **precondition** of B (one of the literals in the precondition of B).

$A \stackrel{c}{\prec} B$ is read: "action A **achieves** (pre)condition c for performing action B" or simply "A **achieves** c for B." E.g.

$$RightSock \stackrel{RightSockOn}{\prec} RightShoe.$$

Note: Labelled ordering constraints are called **causal links** in Russell & Norvig (2ed), but I find it simpler to think of everything as ordering constraints.

Partially ordered plans

Partially ordered plan: A directed graph where

- **Nodes** represent **actions**.
- **Edges** represent **(labelled) ordering constraints** between actions.

Ordering constraints:

$$\mathit{RightSock} \overset{\mathit{RightSockOn}}{\prec} \mathit{RightShoe}.$$

become edges:

$$\boxed{\mathit{RightSock}} \xrightarrow{\mathit{RightSockOn}} \boxed{\mathit{RightShoe}}$$

POP algorithm (partial-order planning algorithm): Start with the empty partially ordered plan, then iteratively add more actions (nodes) and ordering constraints (edges).

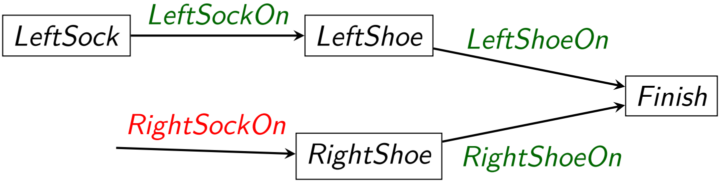
Open preconditions

Recall: $A \overset{c}{\prec} B$ reads "A **achieves** c for B."

Let B denote an action occurring in a partially ordered plan, and let c denote one of its preconditions (precondition literals).

Open precondition: The precondition c is called **open** if it is not achieved by any action in the plan, that is, the plan doesn't contain any ordering constraint of the form $A \overset{c}{\prec} B$.

Example. In the partially ordered plan below, *RightSockOn* is open, but not *LeftSockOn*.



Conflicts

Conflict: An action D is said to **conflict** with an ordering constraint $A \overset{c}{\prec} B$ if D's effect is inconsistent with c (D destroys c). This means that we cannot allow action D to be carried out between carrying out A and carrying out B.

Example. The action $Go(N, H)$ conflicts with the ordering constraint $Go(H, N) \overset{At(N)}{\prec} Buy(M, N)$: we are not allowed to perform the action $Go(N, H)$ between the actions $Go(H, N)$ and $Buy(M, N)$.

Resolving a conflict: If D conflicts with an ordering constraint $A \overset{c}{\prec} B$ we can **resolve** the conflict by adding either the constraint $B \prec D$ or $D \prec A$.

POP algorithm

POP algorithm: Start with the empty plan. Then iteratively do the following:

- Pick an **open precondition** and choose an action that achieves it.
- Resolve any conflicts introduced.

Solution: A partially ordered plan with no open preconditions and no unresolved conflicts will constitute a **solution** to the planning problem (because any linearisation of the ordering $\prec$ of the partially ordered plan will constitute a plan).

Now for the details...

POP algorithm

Initialisation: One starts with the **empty plan** only consisting of the actions *Start* and *Finish*.

Start: An action having the planning problem's start state description as effect.

Finish: An action having the planning problem's goal state description as precondition.

After initialisation the following step is repeated:

Step: Pick an open precondition c on some action B in the current plan and choose an action A that achieves c (either an existing action in the plan or a new one). Then:

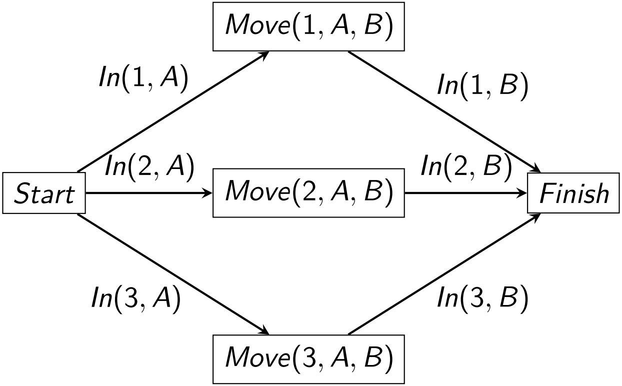
1. Add ordering constraints $A \overset{c}{\prec} B, \mathit{Start} \prec A, A \prec \mathit{Finish}$ (not necessary to draw edges for the two latter).
2. Resolve any conflicts introduced.

Backtrack if:

- An open precondition can't be achieved.
- Some conflict can't be resolved.

POP example: simplified Gripper problem

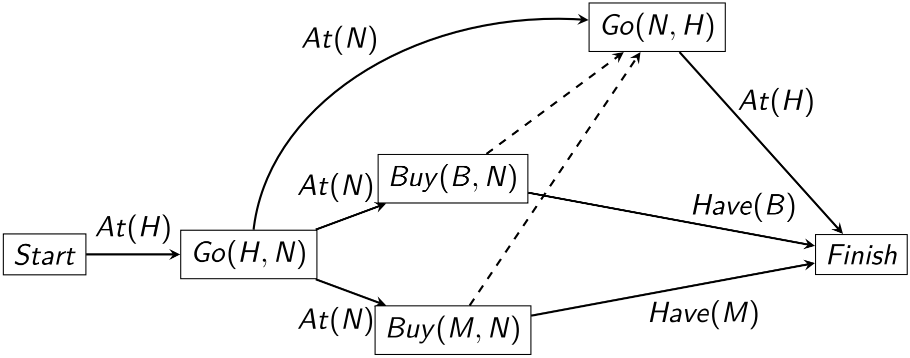
The following is the partially ordered plan resulting from running the POP algorithm on the simplified Gripper problem with $n = 3$:



Note that the graph constructed only has size $O(n)$, even less than the $O(n^2)$ for greedy-best first GRAPH-SEARCH with an optimal heuristics.

POP example: milk-and-bananas

Start is $At(H)$. Goal is $At(H) \wedge Have(M) \wedge Have(B)$. Partially ordered plan produced by POP algorithm (dashed edges are conflict resolutions):



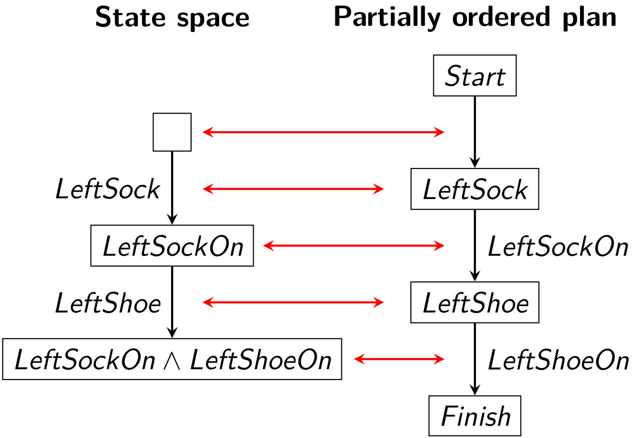
The linearisations are $Go(H, N), Buy(B, N), Buy(M, N), Go(N, H)$ and $Go(H, N), Buy(M, N), Buy(B, N), Go(N, H)$.

Is partial-order planning closer to the way humans make plans?

Duality

Note that partially ordered plans are **dual** to state space graphs! (Nodes in one is edges in the other.)

Example. Initial state is empty, goal is *LeftShoeOn*.



POP Summary

Properties of the **POP algorithm**:

- Works from goal to start state (as regression planning and the additive/max heuristics).
- Builds a graph (partial order plan) dual to a state space graph.
- Any linearisation of the partial order will constitute a solution to the planning problem.
- **Advantages**:
  - Particularly useful for partly decomposable problems. Why?
  - Good for **replanning** in dynamic worlds.
  - Extends naturally to **planning and scheduling problems**.
  - Generates human-friendly plans.
  - Important for applications (e.g. Mars rovers, Maersk container shipping).
- **Disadvantage**: Doesn't currently scale up as well to large problems as state-space planners with good heuristics (hard to come up with good heuristics for POP).

Hierarchical planning

Partial-order planning explores **problem decomposition** to gain efficiency: Independent subgoals are worked on independently.

An alternative problem decomposition is **hierarchical decomposition**: Split **complex tasks** into a small number of **subtasks** that each in turn consists of a small number of subtasks, etc.

Humans have a similar approach to planning.

Example. Assume my goal is to have lunch.

- Top-level plan: *FindFood*, *EatFood*.
- *FindFood* can be split into *ChooseShop*, *BuyFoodAtShop*.
- *BuyFoodAtShop* can e.g. be split into *GoToShop*, *BuyFood*, *GoHome*.

Contrast this with basic planning at the atomic level of, say, individual body movements.

HTN planning

Hierarchical planning can reduce the combinatorial explosion resulting from planning at the atomic level.

Hierarchical Task Network planning (**HTN planning**): A planning method exploring the ideas of hierarchical planning.

The **essential component** of HTN planning is **action decompositions**.

Action decompositions (or **refinements**): The reduction of a **high-level action** (**HLA**) to a sequence of lower-level actions.

HTN planning algorithm: HLA's are recursively refined until only **primitive actions** (non-refinable actions) remain.

Implementation of an HLA: The result of a recursive refinement into primitive actions. Usually not unique.

HTN refinements

HTN **refinements** are represented by **refinement schemas**. A **refinement schema** consists of the following elements:

- **HLA name**. The name of the HLA that the refinement schema defines a decomposition for. E.g. $\mathit{GetFood}$, $\mathit{Buy2Things}(x, y)$ or $\mathit{Navigate}(x, y)$.
- **Precondition**. A conjunction of function-free literals. E.g. $\mathit{At}(\mathit{Shop}) \wedge \mathit{Sells}(\mathit{Shop}, x) \wedge \mathit{Sells}(\mathit{Shop}, y)$. Tells which conditions have to be satisfied in order for the refinement to be applicable.
- **Steps**. A sequence of actions (a totally ordered plan). These actions can either be primitive or high-level. E.g. $[\mathit{Buy}(x, \mathit{Shop}), \mathit{Buy}(y, \mathit{Shop})]$ or $[\mathit{Go}(x, z, r), \mathit{Navigate}(z, y)]$.

Example.
REFINEMENT: $\mathit{Buy2Things}(x, y)$
PRECONDITION: $\mathit{At}(\mathit{Shop}) \wedge \mathit{Sells}(\mathit{Shop}, x) \wedge \mathit{Sells}(\mathit{Shop}, y)$
STEPS: $[\mathit{Buy}(x, \mathit{Shop}), \mathit{Buy}(y, \mathit{Shop})]$

Refinement properties

- **Precondition**. Sometimes the **precondition** of a refinement schema is not explicitly stated, as it can be computed from the **steps** of the schema. How?
- **Multiple refinements**. There can be several refinements schemas for the same high-level action. E.g. a refinement of $Navigate(x, y)$ with steps $[Go(x, z, r), Navigate(z, y)]$ and another with just a single step $[Go(x, y, r)]$. Note the recursion!

High-level plan: A sequence of high-level and possibly primitive actions.

Achieving a goal: A high-level plan **achives a goal** if *at least one* of its implementations (recursive refinements) does.

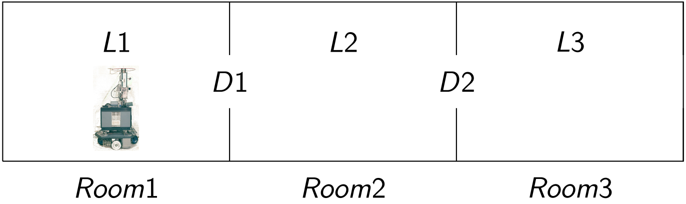
HTN planning example

Example. Refinement schemas:

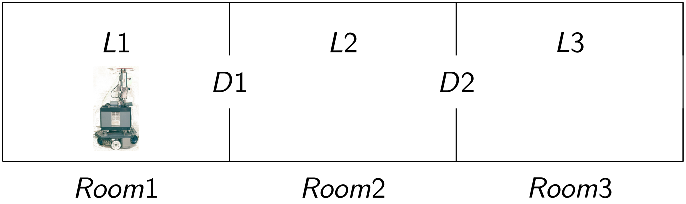
REFINEMENT: $Navigate(x, y)$
PRECONDITION: $At(x) \land In(x, r) \land In(z, r)$
STEPS: $[Go(x, z, r), Navigate(z, y)]$

REFINEMENT: $Navigate(x, y)$
PRECONDITION: $At(x) \land In(x, r) \land In(y, r)$
STEPS: $[Go(x, y, r)]$

Initial state:



Assume Shakey is asked to carry out the HLA $Navigate(L1, L3)$....



Implementation of the HLA $Navigate(L1, L3)$:

1. Refine $Navigate(L1, L3)$ into

$$[Go(L1, D1, Room1), Navigate(D1, L3)].$$

2. Then further into:

$$[Go(L1, D1, Room1), Go(D1, D2, Room2), Navigate(D2, L3).$$

3. And finally into:

$$[Go(L1, D1, Room1), Go(D1, D2, Room2), Go(D2, L3, Room3)].$$

$Navigate(L1, L3)$ **achieves the goal** $At(L3)$.

Representing high-level actions

To allow planning in terms of HLA's before refinement, HLA's are specified using PDDL/STRIPS **action schemas** exactly as primitive actions are.

Example. The $\mathit{Navigate}(x, y)$ HLA can be represented by the following PDDL/STRIPS **action schema**:

ACTION: $\mathit{Navigate}(x, y)$
PRECONDITION: $\mathit{At}(x)$
EFFECT: $\mathit{At}(y) \wedge \neg \mathit{At}(x)$

Downward refinement property of an HLA action schema: At least one implementation of the HLA achieves the stated EFFECT of the schema from the stated PRECONDITION.

Example. The HLA action schema above has the downward refinement property wrt. the refinements defined earlier.

Note: We are here presenting a simplified version of HTN where the effects of a high-level action are unique, not using **angelic semantics** as in Russell & Norvig 3ed.

SELF-STUDY

HTN planning example

Example. Assume now that the robot can only navigate through a room if the light is on in the room:

ACTION: $Go(x, y, r)$
PRECONDITION: $At(x) \land In(x, r) \land In(y, r) \land Light(r)$
EFFECT: $At(y) \land \neg At(x)$

This modification causes a change in the preconditions of the $Navigate(x, y)$-refinements:

REFINEMENT: $Navigate(x, y)$
PRECONDITION: $At(x) \land In(x, r) \land In(z, r) \land Light(r)$
STEPS: $[Go(x, z, r), Navigate(z, y)]$

REFINEMENT: $Navigate(x, y)$
PRECONDITION: $At(x) \land In(x, r) \land In(y, r) \land Light(r)$
STEPS: $[Go(x, y, r)]$

SELF-STUDY

HTN planning example (cont'd)

Recall the action schema of $Navigate(x, y)$:

ACTION: $Navigate(x, y)$
PRECONDITION: $At(x)$
EFFECT: $At(y) \wedge \neg At(x)$

With the refinement modifications carried out above, $Navigate(x, y)$ unfortunately no longer satisfies the downward refinement property. Why?

Naive solution: Add to the precondition of the action schema of $Navigate(x, y)$ that the light should be on in all rooms. This is no good. Why?

Can you think of a better solution?

SELF-STUDY

HTN planning example (cont'd)

Better solution: Replace $Go(x, y, r)$ in the steps of the $Navigate(x, y)$-refinements by the following HLA $Go'(x, y, r)$:

ACTION: $Go'(x, y, r)$
PRECONDITION: $At(x) \wedge In(x, r) \wedge In(y, r)$
EFFECT: $At(y) \wedge \neg At(x) \wedge Light(r)$

REFINEMENT: $Go'(x, y, r)$
PRECONDITION:
$At(x) \wedge In(x, r) \wedge In(y, r) \wedge Light(r)$
STEPS: $[Go(x, y, r)]$

REFINEMENT: $Go'(x, y, r)$
PRECONDITION:
$At(x) \wedge In(x, r) \wedge In(y, r)$
STEPS: $[TurnOn(x, r), Go(x, y, r)]$

Here $TurnOn(x, r)$ is a **primitive action** for turning on the light in the current room:

ACTION: $TurnOn(x, r)$
PRECONDITION: $At(x) \wedge In(x, r)$
EFFECT: $Light(r)$

Downward refinement property re-established: Light doesn't have to be on for $Navigate(x, y)$ to be executable.

HTN planning example (cont'd)

We now have the following 3-level action hierarchy (primitive actions not shown):

REFINEMENT: $Navigate(x, y)$
PRECONDITION: $At(x) \wedge In(x, r) \wedge In(z, r)$
STEPS: $[Go'(x, z, r), Navigate(z, y)]$

REFINEMENT: $Navigate(x, y)$
PRECONDITION: $At(x) \wedge In(x, r) \wedge In(y, r)$
STEPS: $[Go'(x, y, r)]$

REFINEMENT: $Go'(x, y, r)$
PRECONDITION:
$At(x) \wedge In(x, r) \wedge In(y, r) \wedge Light(r)$
STEPS: $[Go(x, y, r)]$

REFINEMENT: $Go'(x, y, r)$
PRECONDITION: $At(x) \wedge In(x, r) \wedge In(y, r)$
STEPS: $[TurnOn(x, r), Go(x, y, r)]$

ACTION: $Navigate(x, y)$
PRECONDITION: $At(x)$
EFFECT: $At(y) \wedge \neg At(x)$

ACTION: $Go'(x, y, r)$
PRECONDITION:
$At(x) \wedge In(x, r) \wedge In(y, r)$
EFFECT:
$At(y) \wedge \neg At(x) \wedge Light(r)$

Note the increase in **planning efficiency**: when planning the route, no branching is produced from considering whether the light is on or not. This problem is dealt with *after* the overall route has been planed.

HTN planning example (cont'd)

We can add further levels on top of our existing plan hierarchy:

ACTION: $MoveBox(b, x, y)$
PRECONDITION: $On(b, x)$
EFFECT: $On(b, y) \wedge \neg On(b, x)$

REFINEMENT: $MoveBox(b, x, y)$
PRECONDITION: $At(l) \wedge On(b, x)$
STEPS: $[Navigate(l, x), NavWithBox(b, x, y), Navigate(y, l)]$

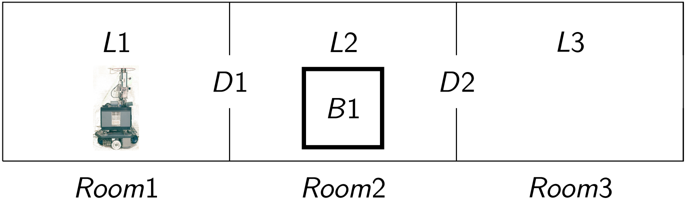
ACTION: $NavWithBox(b, x, y)$
PRECONDITION: $At(x) \wedge On(b, x)$
EFFECT: $At(y) \wedge \neg At(x) \wedge On(b, y) \wedge \neg On(b, x)$

REFINEMENT: $NavWithBox(b, x, y)$
PRECONDITION: $At(x) \wedge On(b, x) \wedge In(x, r) \wedge In(z, r)$
STEPS: $[Push'(b, x, z, r), NavWithBox(b, z, y)]$

REFINEMENT: $NavWithBox(b, x, y)$
PRECONDITION: $At(x) \wedge On(b, x) \wedge In(x, r) \wedge In(y, r)$
STEPS: $[Push'(b, x, y, r)]$

HTN planning example (cont'd)

We now have an HLA $MoveBox(b, x, y)$ for moving arbitrary boxes between arbitrary locations. Consider the following scenario:



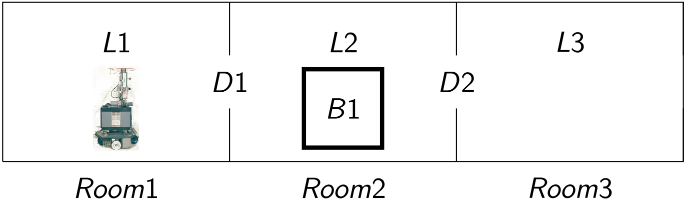
Assume our **goal** is $On(B1, L3)$.

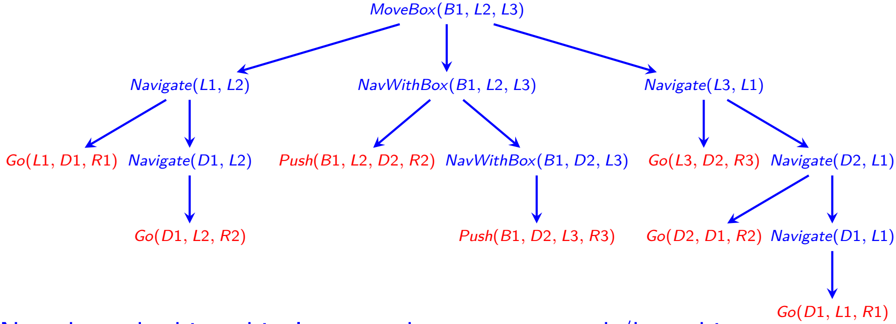
The HLA $MoveBox(B1, L2, L3)$ **achieves** this goal.

The following slide shows a possible implementation.

Remark. For simplicity, we ignore the light aspect (thus using *Go* instead of Go' and *Push* instead of $Push'$).

Implementation example





Note how the hiarachical approach saves on search/branching.

HTN planning advantages

Huge advantage of HTN planning: Can be arbitrarily adjusted along the axis from **purely imperative** (no search) to **purely declarative** (only search):

HTN

Purely imperative.
Single top level action.
Unique implementation.
Linear time.

Purely declarative.
Only primitive actions.
Exponential time.

Allows for combination of subtasks of different types:

- Trivial ones with unique (or few) implementations. (Solution is given).
- Subtasks corresponding to classical planning problems. (Solution completely unknown).

HTN planning advantages

Advantages of HTN planning:

- Allowing **hierarchical decomposition**, an approach widely used in "real life" (consider e.g. writing a large report or planning your vacation).
- "Help" the planner by telling how complex tasks can be decomposed. Can be the crucial difference between **intractability** and **tractability** for **large-scale applications**.

HTN planning has been more popular in **applications** and **industry** than any other planning method:

- Used for production-line scheduling, spacecraft planning and scheduling, equipment configuration, manufacturing process planning, evacuation planning, bridge computers, robotics.
- Used by companies such as Hitachi, Price Waterhouse, and Jaguar Cars.

Final remark

Remember to read the following article (available on CampusNet):

- Jeff Orkin: “Three States and a Plan: The A.I. of F.E.A.R.”. Game Developers Conference (GDC), 2006.

It’s an entertaining and fairly easy read, it presents a nice planning application, and provides a good understanding of some of the strengths and weaknesses of classical planning.