Artificial Intelligence Foundation – JC3001

Lecture 31: Hierarchical Planning - I

Prof. Aladdin Ayesh (aladdin.ayesh@abdn.ac.uk)

Dr. Binod Bhattarai (binod.bhattarai@abdn.ac.uk)

Dr. Gideon Ogunniye, (g.ogunniye@abdn.ac.uk)

October 2025

Material adapted from:

Russell and Norvig (AIMA Book): Chapter 11 (11.4)

Dana Nau (University of Maryland)

Course Progression

- Part 1: Introduction
  1. Introduction to AI ✓
  2. Agents ✓
- Part 2: Problem-solving
  1. Search 1: Uninformed Search ✓
  2. Search 2: Heuristic Search ✓
  3. Search 3: Local Search ✓
  4. Search 4: Adversarial Search ✓
- Part 3: Reasoning and Uncertainty
  1. Reasoning 1: Constraint Satisfaction ✓
  2. Reasoning 2: Logic and Inference ✓
  3. Probabilistic Reasoning 1: BNs ✓
  4. Probabilistic Reasoning 2: HMMs ✓
- Part 4: Planning
  1. Planning 1: Intro and Formalism ✓
  2. Planning 2: Algorithms & Heuristics ✓
  3. **Planning 3: Hierarchical Planning**
  4. Planning 4: Stochastic Planning
- Part 5: Learning
  1. Learning 1: Intro to ML
  2. Learning 2: Regression
  3. Learning 3: Neural Networks
  4. Learning 4: Reinforcement Learning
- Part 6: Conclusion
  1. Ethical Issues in AI
  2. Conclusions and Discussion

Objectives

- Control Knowledge in Planning
- Hierarchical Planning

Outline

- Domain Knowledge in Planning
  - Heuristics and Control Strategies
  - Domain Knowledge
- Hierarchical Task Network Planning

Motivation

- Domain-independent planners suffer from combinatorial complexity
  - Planning is in the worst case intractable
  - Need ways to control the search

Abstract Search Procedure

- Here is a general framework for describing classical and neoclassical planners
- The planning algorithms we have discussed all fit into the framework, if we vary the details (e.g. the steps do not have to be in this order)

function AbstractSearch(u)
    if Terminal(u) **then return** u
    $u \leftarrow$ Refine(u) ▷ *refinement step*
    $B \leftarrow$ Branch(u) ▷ *branching step*
    $B' \leftarrow$ Prune(B) ▷ *pruning step*
    if $B' = \emptyset$ **then return** failure
    nondeterministically choose $\nu \in B'$
    return AbstractSearch(ν)

- Wrap iterative forward search
- Refinement: compute heuristic information for node u
- Branching: {sets of states from actions applicable to u}
- Pruning: prune actions/nodes such that $h(u') = \infty$

function AbstractSearch(u)
    if Terminal(u) **then return** u
    $u \leftarrow$ Refine(u) ▷ *refinement step*
    $B \leftarrow$ Branch(u) ▷ *branching step*
    $B' \leftarrow$ Prune(u) ▷ *pruning step*
    if $B' = \emptyset$ **then return** failure
    nondeterministically choose $\nu \in B'$
    return AbstractSearch(ν)

Domain Knowledge

- Often, planning can be done much more efficiently if we have domain-specific information
  - classical planning is EXPSPACE-complete
  - block-stacking can be done in time $O(n^3)$
- But we don't want to have to write a new domain-specific planning system for each problem!
- **Domain-configurable** planning algorithm
  - Domain-independent search engine (usually a forward state-space search)
  - Input includes domain-specific information that allows us to avoid a brute-force search
    - Prevent the planner from visiting unpromising states

Domain Knowledge

- If we're at some state s in a state space, sometimes a domain-specific test can tell us that
  - s doesn't lead to a solution, or
  - for any solution below s, there's a better solution along some other path
- In such cases we can prune s immediately
- Rather than writing the domain-dependent test as low-level computer code, we would prefer to talk directly about the planning domain

function AbstractSearch(u)
  if Terminal(u) **then return** u
  $u \leftarrow$ Refine(u) ▷ *refinement step*
  $B \leftarrow$ Branch(u) ▷ *branching step*
  $B' \leftarrow$ Prune(u) ▷ *pruning step*
  if $B' = \emptyset$ **then return** failure
  nondeterministically choose $\nu \in B'$
  return AbstractSearch(ν)

Outline

- Domain Knowledge in Planning
  - Heuristics and Control Strategies
  - Domain Knowledge
- Hierarchical Task Network Planning

Hierarchical domain knowledge

- 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:
    - buy ticket from local airport to remote airport
    - travel to local airport
    - fly to remote airport
    - travel to final destination
- How to enable planning systems to make use of such recipes?

Two Approaches

- Control rules:
  - Write rules to prune every action that **does not** fit the recipe

function AbstractSearch(u)
    if Terminal(u) **then return** u
    $u \leftarrow$ Refine(u) $\triangleright$ *refinement step*
    $B \leftarrow$ Branch(u) $\triangleright$ *branching step*
    $B' \leftarrow$ Prune(u) $\triangleright$ *pruning step*
    if $B' = \emptyset$ **then return** failure
    nondeterministically choose $\nu \in B'$
    return AbstractSearch(ν)

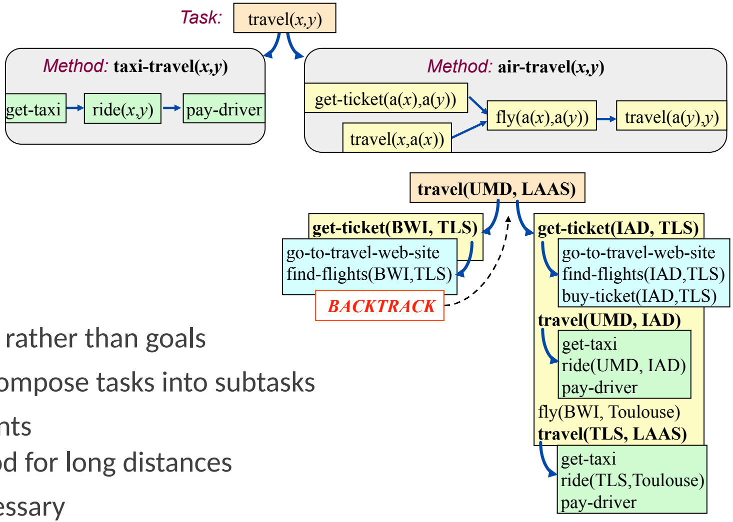
- Control rules:
  - Write rules to prune every action that **does not** fit the recipe

1: **function** AbstractSearch(u)
if Terminal(u) **then return** u
$u \leftarrow$ Refine(u) ▷ *refinement step*
$B \leftarrow$ Branch(u) ▷ *branching step*
$B' \leftarrow$ Prune(u) ▷ *pruning step*
if $B' = \emptyset$ **then return** failure
nondeterministically choose $\nu \in B'$
return AbstractSearch(ν)

Two Approaches

- Control rules:
  - Write rules to prune every action that **does not** fit the recipe
- Hierarchical Task Network (HTN) planning:
  - Describe the actions and subtasks that **do** fit the recipe

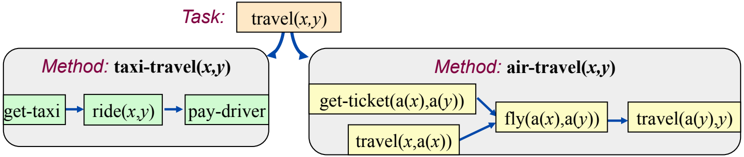
function AbstractSearch(u)
    if Terminal(u) **then return** u
    $u \leftarrow$ Refine(u) ▷ *refinement step*
    $\boxed{B \leftarrow \text{Branch}(u) \triangleright \textit{branching step}}$
    $B' \leftarrow$ Prune(u) ▷ *pruning step*
    if $B' = \emptyset$ **then return** failure
    nondeterministically choose $\nu \in B'$
    return AbstractSearch(ν)



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

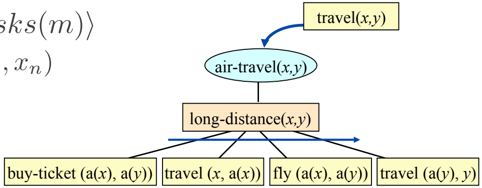
- HTN planners may be domain-specific
- Or they may be domain-configurable
  - Domain-independent planning engine
  - Domain description that defines not only the operators, but also the methods
  - Problem description
    domain description, initial state, initial task network



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)

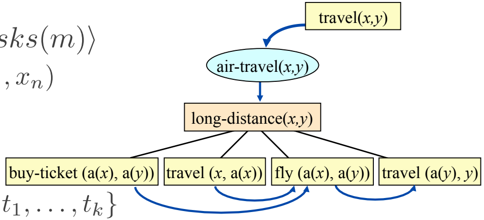
- Totally ordered method: a 4-tuple
  $m = \langle name(m), task(m), precond(m), subtasks(m) \rangle$
  - $name(m)$: an expression of the form $n(x_1, \ldots, x_n)$
    $x_1, \ldots, x_n$ are parameters - variable symbols
  - $task(m)$: a nonprimitive task
  - $precond(m)$: preconditions (literals)
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



- *air-travel*(x, y)

task: *travel*(x, y)
precond: *long-distance*(x, y)
subtasks: $\langle$*buy-ticket*$(a(x), a(y))$, *travel*$(x, a(x))$, *fly*$(a(x), a(y))$, *travel*$(a(y), y)\rangle$

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

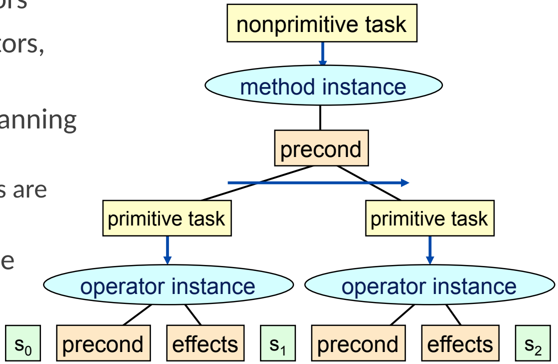


- *air-travel*(x, y)

task: *travel*(x, y)
precond: *long-distance*(x, y)
subtasks: $u_1 =$ *buy-ticket*$(a(x), a(y)), u_2 =$ *travel*$(x, a(x)), u_3 =$ *fly*$(a(x), a(y)), u_4 =$ *travel*$(a(y), y), \{\langle u_1, u_3 \rangle, \langle u_2, u_3 \rangle, \langle u_3, u_4 \rangle\}$

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 nonprimitive tasks
  - operators to primitive tasks



Example (1)

- Two objects:
  banjo, kiwi
- Two operators:
  $\textit{pickup}(A) : pre : \top; \mathit{eff} : \textit{have}(A)$
  $\textit{drop}(A) : pre : \textit{have}(A); \mathit{eff} : \neg\textit{have}(A)$

- Two methods:
  - $swap1$
    $task : swap(X, Y)$
    $pre : \textit{have}(X), \neg\textit{have}(Y)$
    $tn : \langle\textit{drop}(X), \textit{pickup}(Y)\rangle$
  - $swap2$
    $task : swap(X, Y)$
    $pre : \textit{have}(Y), \neg\textit{have}(X)$
    $tn : \langle\textit{drop}(Y), \textit{pickup}(X)\rangle$

Example (2)

- Problem:
  $init$: *have*$(kiwi)$
  tn : *swap*$(kiwi, banjo)$

Example (2)

- Problem:
  $init$: $\mathsf{have}(kiwi)$
  tn : $\mathsf{swap}(kiwi, banjo)$

Applying method $swap1$
$task$: $swap(kiwi, banjo)$
pre : $\mathsf{have}(kiwi), \neg\mathsf{have}(banjo)$ ✓
tn : $\langle \mathsf{drop}(kiwi), \mathsf{pickup}(banjo) \rangle$

- Problem:
  $init$: $\textit{have}(kiwi)$
  tn : $\textit{swap}(kiwi, banjo)$

Applying method $swap1$
$task$: $swap(kiwi, banjo)$
pre : $\textit{have}(kiwi), \neg\textit{have}(banjo)$ ✓
tn : $\langle\textit{drop}(kiwi), \textit{pickup}(banjo)\rangle$

Resulting tn:
$\textit{drop}(kiwi), \textit{pickup}(banjo)$

Example (2)

- Problem:
  $init$: $\textsf{have}(kiwi)$
  tn : $\textsf{swap}(kiwi, banjo)$

Applying method $swap1$
$task$: $swap(kiwi, banjo)$
pre : $\textsf{have}(kiwi), \neg\textsf{have}(banjo)$ ✓
tn : $\langle\textsf{drop}(kiwi), \textsf{pickup}(banjo)\rangle$

Resulting tn:
$\textsf{drop}(kiwi), \textsf{pickup}(banjo)$

After executing $\textsf{drop}(kiwi)$
$state$: $\{\}$

- Problem:
  $init$: $\textsf{have}(kiwi)$
  tn : $\textsf{swap}(kiwi, banjo)$

Applying method $swap1$
$task$: $swap(kiwi, banjo)$
pre : $\textsf{have}(kiwi), \neg\textsf{have}(banjo)$ ✓
tn : $\langle\textsf{drop}(kiwi), \textsf{pickup}(banjo)\rangle$

Resulting tn:
$\textsf{drop}(kiwi), \textsf{pickup}(banjo)$

After executing $\textsf{pickup}(banjo)$
$state$: $\{\textsf{have}(banjo)\}$

To continue in the next session.