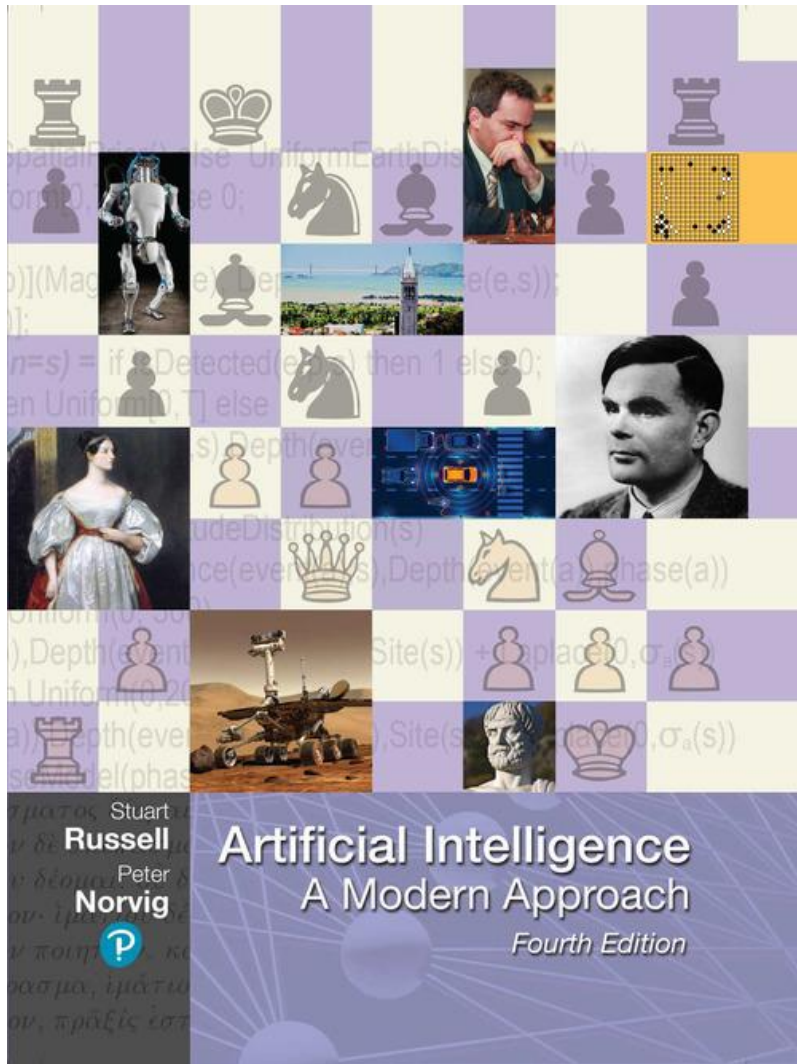


# Artificial Intelligence Fundamentals

2024-2025



*“Plans are of little importance, but planning is essential.”*

— Winston Churchill, former British Prime Minister

## AIMA Chapter 11

### Automated Planning

# Outline

- ◆ Definition of Classical Planning
- ◆ Algorithms for Classical Planning
- ◆ Heuristics for Planning
- ◆ Hierarchical Planning
- ◆ Planning and Acting in Nondeterministic Domains
- ◆ Time, Schedules, and Resources
- ◆ Analysis of Planning Approaches

# Definition of Classical Planning

**Classical planning** is defined as the task of finding a **sequence of actions** to accomplish a goal in a discrete, deterministic, static, fully observable environment.

We have seen **two main methods** to plan ahead already:

- Problem-solving agent (with search algorithms)
- Propositional / FOL Logical agent (with logic inference)

Main limits:

- They both require **ad-hoc knowledge** for each new domain: a heuristic evaluation function for search, and hand-written code for the logic (e.g., definition of the Wumpus world facts and rules).
- They both need to explicitly represent an **exponentially large state space**.

# Planning Domain Definition Language

**Planning Domain Definition Language** (Ghallab et al., 1998). PDDL is a factored representation.

- Basic PDDL can handle classical planning domains, and extensions can handle non-classical domains that are continuous, partially observable, concurrent, and multi-agent.
- Allows us to express all  $4T * n^2$  ( $T$ =time steps,  $n$  locations) actions with a single *action schema*, and does not need domain-specific knowledge (for the “*move forward*” action in the Wumpus world)
- **State**: represented as a conjunction of *ground atomic fluents* (e.g., a single predicate with no variables)
- Uses **database semantics**: the *closed-world assumption* means that any fluents that are not mentioned are false + *unique names assumption* (*entities with different names are different*)

## Definition of Classical Planning

- An **action schema** represents a family of ground actions
- The schema consists of the action **name**, a list of all the **variables** used in the schema, a **precondition** and an **effect**.

*Action(Fly(p, from, to),*

*PRECOND:  $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$*

*EFFECT:  $\neg At(p, from) \wedge At(p, to)$*

- A set of action schemas serves as a definition of a **planning domain**. A specific problem within the domain is defined with the addition of an **initial state** and a **goal state**
- The **initial state** is a conjunction of ground fluents
- The **goal** is just like a precondition: a conjunction of literals (positive or negative) that may contain variables

## Definition of Classical Planning

$Init(At(C_1, SFO) \wedge At(C_2, JFK) \wedge At(P_1, SFO) \wedge At(P_2, JFK)$   
 $\wedge Cargo(C_1) \wedge Cargo(C_2) \wedge Plane(P_1) \wedge Plane(P_2)$   
 $\wedge Airport(JFK) \wedge Airport(SFO))$   
 $Goal(At(C_1, JFK) \wedge At(C_2, SFO))$   
 $Action(Load(c, p, a),$   
    PRECOND:  $At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$   
    EFFECT:  $\neg At(c, a) \wedge In(c, p)$ )  
 $Action(Unload(c, p, a),$   
    PRECOND:  $In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a)$   
    EFFECT:  $At(c, a) \wedge \neg In(c, p)$ )  
 $Action(Fly(p, from, to),$   
    PRECOND:  $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$   
    EFFECT:  $\neg At(p, from) \wedge At(p, to)$ )

**Figure 11.1** A PDDL description of an air cargo transportation planning problem.

[Load(C1,P1,SFO), Fly(P1,SFO,JFK), Unload(C1,P1,JFK),  
Load(C2,P2,JFK),Fly(P2,JFK,SFO), Unload(C2,P2,SFO)] .

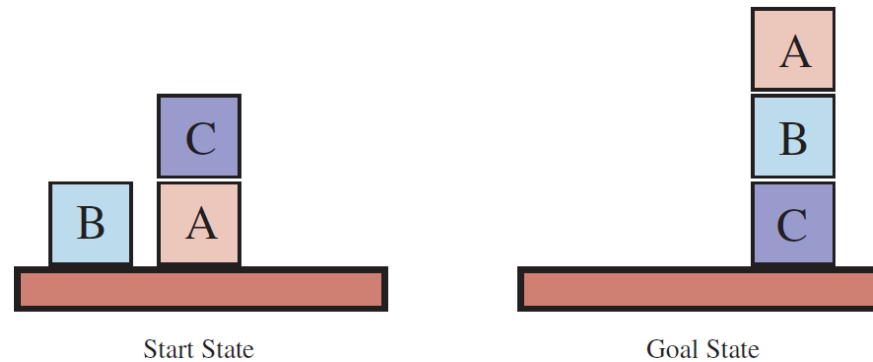
## Definition of Classical Planning

*Init*(*Tire*(*Flat*)  $\wedge$  *Tire*(*Spare*)  $\wedge$  *At*(*Flat*,*Axle*)  $\wedge$  *At*(*Spare*,*Trunk*))  
*Goal*(*At*(*Spare*,*Axle*))  
*Action*(*Remove*(*obj*,*loc*),  
    PRECOND: *At*(*obj*,*loc*)  
    EFFECT:  $\neg$  *At*(*obj*,*loc*)  $\wedge$  *At*(*obj*,*Ground*))  
*Action*(*PutOn*(*t*, *Axle*),  
    PRECOND: *Tire*(*t*)  $\wedge$  *At*(*t*,*Ground*)  $\wedge$   $\neg$  *At*(*Flat*,*Axle*)  $\wedge$   $\neg$  *At*(*Spare*,*Axle*)  
    EFFECT:  $\neg$  *At*(*t*,*Ground*)  $\wedge$  *At*(*t*,*Axle*))  
*Action*(*LeaveOvernight*,  
    PRECOND:  
    EFFECT:  $\neg$  *At*(*Spare*,*Ground*)  $\wedge$   $\neg$  *At*(*Spare*,*Axle*)  $\wedge$   $\neg$  *At*(*Spare*,*Trunk*)  
             $\wedge$   $\neg$  *At*(*Flat*,*Ground*)  $\wedge$   $\neg$  *At*(*Flat*,*Axle*)  $\wedge$   $\neg$  *At*(*Flat*, *Trunk*))

**Figure 11.2** The simple spare tire problem.

[*Remove*(*Flat*,*Axle*),*Remove*(*Spare*,*Trunk*),*PutOn*(*Spare*,*Axle*)]

# Definition of Classical Planning



**Figure 11.3** Diagram of the blocks-world problem in Figure 11.4.

$Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A)$   
 $\wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C) \wedge Clear(Table))$   
 $Goal(On(A, B) \wedge On(B, C))$   
 $Action(Move(b, x, y),$   
 $\quad PRECOND: On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge$   
 $\quad (b \neq x) \wedge (b \neq y) \wedge (x \neq y),$   
 $\quad EFFECT: On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y))$   
 $Action(MoveToTable(b, x),$   
 $\quad PRECOND: On(b, x) \wedge Clear(b) \wedge Block(b) \wedge Block(x),$   
 $\quad EFFECT: On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))$

**Figure 11.4** A planning problem in the blocks world: building a three-block tower. One solution is the sequence  $[MoveToTable(C, A), Move(B, Table, C), Move(A, Table, B)]$ .



# Algorithms for Classical Planning

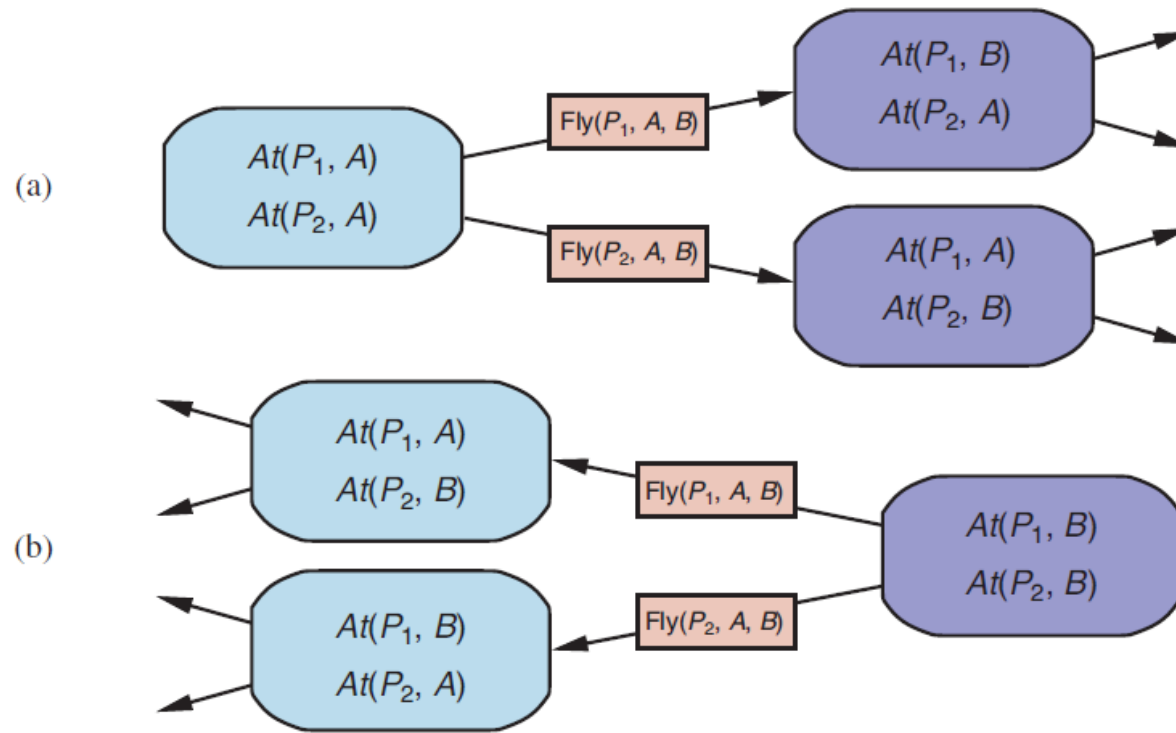
## Forward state-space search for planning

- Start at **initial state**
- To determine the applicable actions we **unify the current state** against the **preconditions** of each action schema
- For each unification that successfully results in a substitution, we apply **the substitution to the action schema** to yield a **ground action** with no variables.

## Backward search for planning

- **Start at the goal** and apply the actions backward until we find a sequence of steps that reaches the initial state
- **Consider relevant actions at each step.**
- **This reduces branching factor**
- A relevant action is one with an effect that unifies with one of the goal literals, but with no effect that negates any part of the goal.

# Algorithms for Classical Planning



**Figure 11.5** Two approaches to searching for a plan. (a) Forward (progression) search through the space of ground states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through state descriptions, starting at the goal and using the inverse of the actions to search backward for the initial state.

# Algorithms for Classical Planning

## Cargo Example

- Consider an air cargo problem with 10 airports, 5 planes and 20 pieces of cargo. The **goal** is to move all the cargo at airport A to airport B. There is a 41-step solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the 20 pieces.
- On average, let's say there are about **2000 possible actions** per state, so the search graph up to the depth of the **41-step solution** has about  **$2000^{41}$  nodes**
- Clearly, even this relatively small problem instance is hopeless without an accurate heuristic search
  - strong **domain-independent heuristics** with PDDL can be derived automatically; that is what makes forward search feasible

# Algorithms for Classical Planning

## Other classical planning approaches

- An approach called **Graph plan** uses a specialized data structure, a **planning graph**
- **Situation calculus** is a method of describing planning problems in first-order logic -> *no big impact in practical applications*, perhaps since FOL provers are not as well developed (as propositional satisfiability programs).
- An alternative called **Partial-Order Planning (POP)** represents a plan as a graph rather than a **linear sequence**:
  - each **action** is a **node** in the graph,
  - for each **precondition** of the action there is an **edge** from another action (or from the initial state) that indicates that the predecessor action establishes the **precondition**.
  - We search in the **space of plans** rather than world-states, inserting actions to satisfy conditions

# Heuristics for Planning

- **Ignore preconditions heuristic:** drops all preconditions from actions
  - Every action becomes applicable
- **Ignore-delete-lists heuristic:** removing the *delete lists* from all actions (i.e., removing all negative literals from effects).

## Domain-independent pruning

- **symmetry reduction:** prune out all symmetric branches of the search tree except for one (if actions are “equivalent” consider only 1)
- **preferred action:** a step in the relaxed plan (solution to a relaxed problem), or it achieves some precondition of the relaxed plan
- **forward pruning:** might prune away an optimal solution

# Heuristics for Planning

## State abstraction in planning

- **state abstraction:** a **many-to-one** mapping from states in the ground representation of the problem to the abstract representation
- relaxations that decrease the **number of states** (not just considered actions): e.g. ignoring some *fluents*
- **decomposition:** dividing a problem into parts, solving each part independently, and then combining the parts
- **Subgoal independence assumption:** the cost of solving a conjunction of subgoals is approximated by the sum of the costs of solving each subgoal independently
- The subgoal independence assumption can be optimistic or pessimistic
  - **optimistic:** negative interactions between the subplans for each subgoal
  - **pessimistic:** inadmissible, when subplans contain redundant actions

# Hierarchical Planning

**Idea:** Higher levels of abstraction with hierarchical decomposition to manage complexity.

**Hierarchical structure** reduces computational task to a small number of activities at the next lower level.

**Computational overhead** to find correct way to arrange those activities for the current problem is small.

## High Level Actions (HLA):

- Hierarchical Task Networks (HTN) or HTN planning
- **Assume** full observability & determinism & **primitive actions** standard precondition-effect schemas
- HLA offers one or more possible *refinements* into a sequence of actions.
- **Implementation of HLA:** An HLA refinement that contains only primitive actions.

## High level Plan (HLP)

- is the concatenation of implementations of each **HLA in the sequence**.
- a high-level plan achieves the goal from a given state if *at least one of its implementations* achieves the goal from that state

# Hierarchical Planning

## Example of Refinement

*Refinement*(*Go*(*Home*, *SFO*),  
  STEPS: [*Drive*(*Home*, *SFO* *LongTermParking*),  
          *Shuttle*(*SFO* *LongTermParking*, *SFO*)] )

*Refinement*(*Go*(*Home*, *SFO*),  
  STEPS: [*Taxi*(*Home*, *SFO*)] )

*Refinement*(*Navigate*( $[a, b]$ ,  $[x, y]$ ),  
  PRECOND:  $a = x \wedge b = y$   
  STEPS: [] )

*Refinement*(*Navigate*( $[a, b]$ ,  $[x, y]$ ),  
  PRECOND: *Connected*( $[a, b]$ ,  $[a - 1, b]$ )  
  STEPS: [*Left*, *Navigate*( $[a - 1, b]$ ,  $[x, y]$ )] )

*Refinement*(*Navigate*( $[a, b]$ ,  $[x, y]$ ),  
  PRECOND: *Connected*( $[a, b]$ ,  $[a + 1, b]$ )  
  STEPS: [*Right*, *Navigate*( $[a + 1, b]$ ,  $[x, y]$ )] )

...

**Figure 11.7** Definitions of possible refinements for two high-level actions: going to San Francisco airport and navigating in the vacuum world. In the latter case, note the recursive nature of the refinements and the use of preconditions.



# Hierarchical Forward Planning

---

```
function HIERARCHICAL-SEARCH(problem, hierarchy) returns a solution or failure
  frontier ← a FIFO queue with [Act] as the only element
  while true do
    if IS-EMPTY(frontier) then return failure
    plan ← POP(frontier)      // chooses the shallowest plan in frontier
    hla ← the first HLA in plan, or null if none
    prefix, suffix ← the action subsequences before and after hla in plan
    outcome ← RESULT(problem.INITIAL, prefix)
    if hla is null then      // so plan is primitive and outcome is its result
      if problem.IS-GOAL(outcome) then return plan
    else for each sequence in REFINEMENTS(hla, outcome, hierarchy) do
      add APPEND(prefix, sequence, suffix) to frontier
```

**Figure 11.8** A breadth-first implementation of hierarchical forward planning search. The initial plan supplied to the algorithm is [*Act*]. The REFINEMENTS function returns a set of action sequences, one for each refinement of the HLA whose preconditions are satisfied by the specified state, *outcome*.

---

# Planning and Acting in Nondeterministic Domains

## Sensorless planning

- **Conditional effect:** an actions effect is dependant on a state (eg: robot's location)
- “**when** condition: effect,” where condition is a logical formula to be compared against the current state, and effect is a formula describing the resulting state.

*Action(Suck,*  
EFFECT:**when** *AtL: CleanL*  $\wedge$  **when** *AtR: CleanR*).

- In general, conditional effects can induce arbitrary dependencies among the fluents in a belief state, leading to belief states **of exponential size in the worst case**
- **All conditional effects** whose conditions are satisfied have their effects applied to generate the resulting belief state; if none are satisfied, then the resulting state is unchanged.

# Planning and Acting in Nondeterministic Domains

## Contingent planning (with observations)

- The generation of **plans with conditional branching** based on **percepts**—is appropriate for environments with **partial observability, nondeterminism, or both**
- When executing this plan, a contingent-planning agent can maintain its **belief state as a logical formula**
- **Evaluate** each branch condition by determining if the belief state **entails the condition formula or its negation**

# Planning and Acting in Nondeterministic Domains

## Online planning

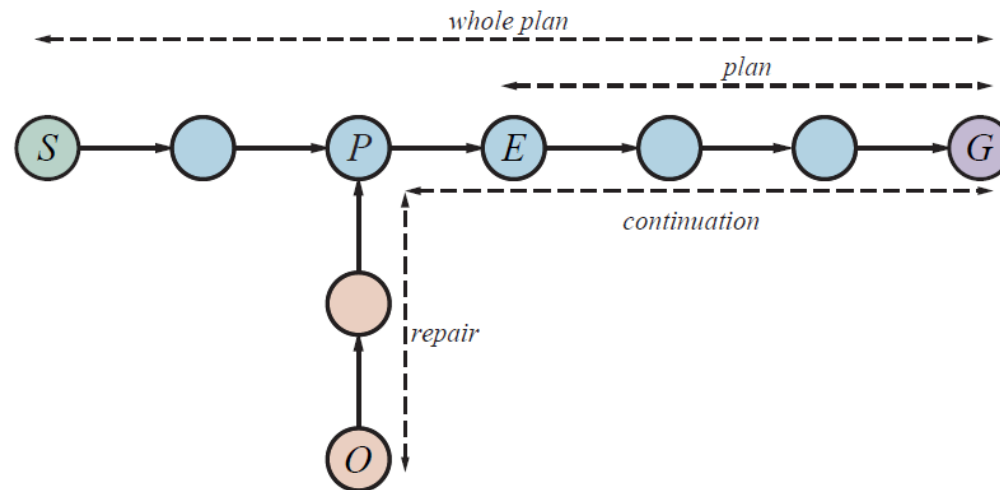
- The need for a new plan: **execution monitoring**
- When there are **too many contingencies** to prepare for
- When a contingency is not prepared, **replanning** is required
- Replanning is needed if the agent's model of the world is **incorrect** (missing precondition, effect or fluent)

For *online planning* an agent monitor based on three approaches:

- **Action monitoring:** before executing an action, the agent verifies that all the preconditions still hold.
- **Plan monitoring:** before executing an action, the agent verifies that the remaining plan will still succeed.
- **Goal monitoring:** before executing an action, the agent checks to see if there is a better set of goals it could be trying to achieve.

# Planning and Acting in Nondeterministic Domains

## Online planning (example)



**Figure 11.12** At first, the sequence “whole plan” is expected to get the agent from *S* to *G*. The agent executes steps of the plan until it expects to be in state *E*, but observes that it is actually in *O*. The agent then replans for the minimal *repair* plus *continuation* to reach *G*.

# Time, Schedules, and Resources

- Classical planning talks about what to do, in what order, but does not talk about **time**: *how long an action takes* and *when it occurs*
- However, the real world has resource constraints

## Representing temporal and resource constraints

- Actions can have a **duration** and **constraints**
- **Constraints:**
  - Type of resource
  - Number of resources
  - Resource is consumable or reusable
- **Aggregation:** representation of resources as numerical quantities
- The representation of resources as numerical quantities, such as *Inspectors(2)*, rather than as named entities, such as *Inspector(I<sub>1</sub>)* and *Inspector(I<sub>2</sub>)*,
- This reduces complexity if multiple quantities are used

# Time, Schedules, and Resources

The approach we take is “**plan first, schedule later**”: divide the overall problem into:

1. **Planning phase:** in which actions are selected, with some **ordering constraints**, to meet the goals of the problem;
2. **Scheduling phase:** in which **temporal information is added to the plan** to ensure that it meets resource and deadline constraints.

This approach is common in **real-world manufacturing and logistical settings**, where the planning phase is sometimes automated, and sometimes performed by human experts.

# Assembling Cars Example

*Jobs*( $\{AddEngine1 \prec AddWheels1 \prec Inspect1\},$   
 $\{AddEngine2 \prec AddWheels2 \prec Inspect2\}$ )

*Resources*(*EngineHoists*(1), *WheelStations*(1), *Inspectors*(2), *LugNuts*(500))

*Action*(*AddEngine1*, DURATION:30,  
USE:*EngineHoists*(1))

*Action*(*AddEngine2*, DURATION:60,  
USE:*EngineHoists*(1))

*Action*(*AddWheels1*, DURATION:30,  
CONSUME:*LugNuts*(20), USE:*WheelStations*(1))

*Action*(*AddWheels2*, DURATION:15,  
CONSUME:*LugNuts*(20), USE:*WheelStations*(1))

*Action*(*Inspect<sub>i</sub>*, DURATION:10,  
USE:*Inspectors*(1))

**Figure 11.13** A job-shop scheduling problem for assembling two cars, with resource constraints. The notation  $A \prec B$  means that action  $A$  must precede action  $B$ .

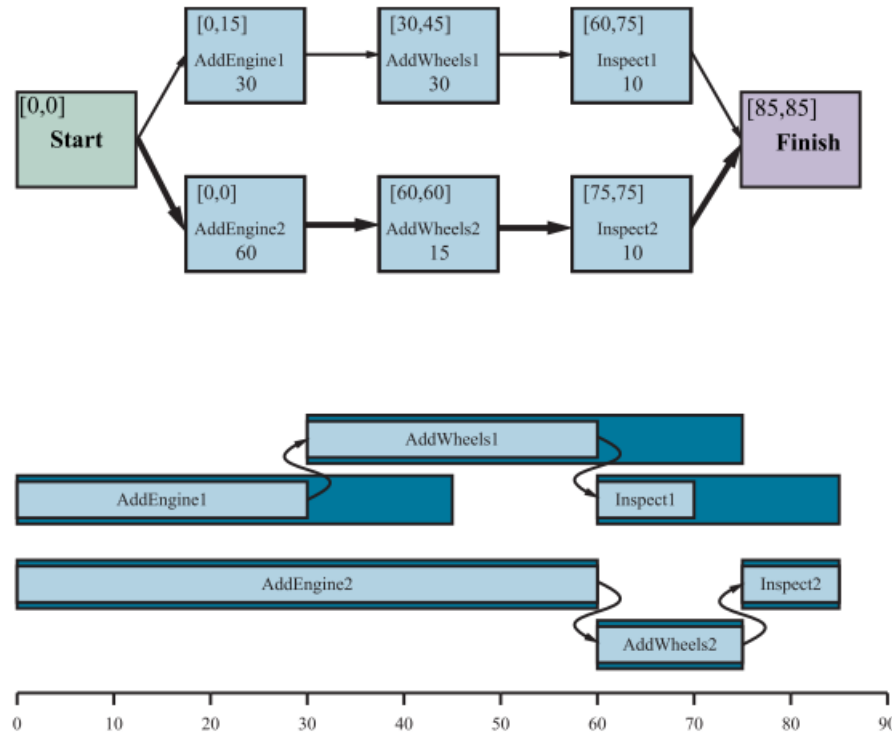


# Time, Schedules, and Resources

## Solving scheduling problems

- We need to model it as a **directed graph** (multiple actions in parallel)
- **Uses critical path method (CPM)** to determine the possible **start** and **end** times of each action
- A **path** through a graph representing a partial-order plan is a **linearly ordered sequence of actions beginning with “Start” and ending with “Finish”**.
- **The critical path** is that path whose total duration is longest; the path is “critical” because it determines the duration of the entire plan
- Actions that are **off the critical path** have a **window of time** in which they can be executed.
- This window of time is known as **Slack**. The window has a earliest possible start time *ES* and latest possible start time *LS*
- Together the *ES* and *LS* times for all the actions constitute a **schedule** for the problem.

# Scheduling in Car Manufacturing



**Figure 11.14** Top: a representation of the temporal constraints for the job-shop scheduling problem of Figure 11.13. The duration of each action is given at the bottom of each rectangle. In solving the problem, we compute the earliest and latest start times as the pair  $[ES, LS]$ , displayed in the upper left. The difference between these two numbers is the *slack* of an action; actions with zero slack are on the critical path, shown with bold arrows. Bottom: the same solution shown as a timeline. Blue rectangles represent time intervals during which an action may be executed, provided that the ordering constraints are respected. The unoccupied portion of a blue rectangle indicates the slack.

# Time, Schedules, and Resources

Many approaches have been tried for **optimal scheduling** including branch-and-bound, simulated annealing, tabu search, and constraint satisfaction / optimization.


Eg: **minimum slack heuristic**:

- on each iteration, schedule for the earliest possible start whichever unscheduled action has all its predecessors scheduled and has the least slack; then update the *ES* and *LS* times for each affected action and repeat.
- It is a **greedy heuristic**
- Resembles minimum-remaining-values (MRV) heuristic in constraint satisfaction.

# Analysis of Planning Approaches

- **Planning** combines the **two major areas of AI** we have covered so far: **search and logic**.
- Combination allows planners to **scale up** from toy problems where the number of actions and states is limited to around a dozen, to real-world industrial applications with **millions of states and thousands of actions**.
- Planning helps in **controlling combinatorial explosion** through the identification of independent subproblems.
- Unfortunately, we do **not yet have a clear understanding** of which techniques work best on which kinds of problems.
- Newer techniques will emerge that provide highly expressive first-order and hierarchical representations
  - Eg: **portfolio planning systems**, where a collection of algorithms are available to apply to any given problem

# Planimation

 **Planimation**


**Planimation**  
User Information  
    Home  
    User Guide  
Developer Information  
Component Information  
Animated Domains  
GitHub Code  
Contributors

## Planimation

Planimation is a modular and extensible open source framework to visualise sequential solutions of planning problems specified in PDDL. We introduce a preliminary declarative PDDL-like animation profile specification, expressive enough to synthesise animations of arbitrary initial states and goals of a benchmark with just a single profile.

### 1 Overview

Planimation was developed as a final year project by students at the Univeristy of Melbourne. Here's a promotional video explaining the motivation behind planimation.



<https://planimation.github.io/documentation/>

# AIMA-python “Planning”

2402 lines (2402 sloc) | 181 KB

<> [icon] Raw Blame [icon] [icon] [icon]

## Classical Planning

---

## Classical Planning Approaches

### Introduction

**Planning** combines the two major areas of AI: *search* and *logic*. A planner can be seen either as a program that searches for a solution or as one that constructively proves the existence of a solution.

Currently, the most popular and effective approaches to fully automated planning are:

- searching using a *planning graph*;
- *state-space search* with heuristics;
- translating to a *constraint satisfaction (CSP) problem*;
- translating to a *boolean satisfiability (SAT) problem*.

```
In [1]: from planning import *
```

### Planning as Planning Graph Search

[aima-python/classical\\_planning\\_approaches.ipynb at master · aimacode/aima-python \(github.com\)](#)

## Summary

- **PDDL**, the Planning Domain Definition Language, describes the initial and goal states as conjunctions of literals, and actions in terms of their preconditions and effects
- **Hierarchical task network (HTN)** planning allows the agent to take advice from the domain designer in the form of high-level actions (HLAs) that can be implemented in various ways by lower-level action sequences.
- **Contingent plans** allow the agent to sense the world during execution to decide what branch of the plan to follow.
- **An online planning agent** uses **execution monitoring** and splices in repairs as needed to recover from unexpected situations, which can be due to nondeterministic actions, exogenous events, or incorrect models of the environment
- Many actions consume **resources**, such as money, gas, or raw materials -> *planning + scheduling*

## In the next lecture... (after the hands-on session)

- ◆ Acting Under Uncertainty
- ◆ Basic Probability Notation
- ◆ Inference Using Full Joint Distributions
- ◆ Independence
- ◆ Bayes' Rule and Its Use
- ◆ Naive Bayes Models
- ◆ The Wumpus World Revisited