COMP219: Artificial Intelligence

Lecture 24: Scheduling in Real World Planning

Overview

- Last time
  - Classical planning; PDDL; planning as a SAT problem
- Today
  - Planning in the real world
    - Time and resource constraints
- Learning outcomes covered today:

  Identify or describe approaches used to solve planning problems in AI and apply these to simple examples

Real World Planning

- Classical planning decides what to do and in what order
- Planners used in the real world for planning and scheduling operations for spacecraft, factories and military campaigns need to talk about time (scheduling):
  - how long an action takes
  - when an action occurs
  - e.g. an airline schedule assigning planes to flights needs to know departure and arrival times
- The real world also imposes many resource constraints
  - e.g. there is a limit on the number of pilots employed, and a pilot can only fly one plane at any one time

Time

- In classical planning we assumed that:
  - actions are instantaneous
  - preconditions must hold before an action is taken
  - the effects of an action persist
- Real world planning domains are more complex:
  - actions take time to execute; how long an action takes to execute may depend on the preconditions
  - preconditions may need to hold during an action's execution as well as before it starts
  - effects may not be true immediately or may persist for only a limited time
  - an action may have multiple effects on a fluent at different times
- In scheduling we usually require a goal to be true *at a given time* or *over a given time interval*

Planning with Time

- Examples:
  - If I hire a carpet cleaning machine to clean my carpets, I need to continue to have the machine while I am cleaning my carpets
  - If I push a lift button, the lift may take time to arrive and the doors will only open for a limited time
  - If I share a printer, my print job will have to wait until the printer is available if someone else is currently printing
- Some actions may have to be taken concurrently:
  - If a fuse blows, I have to strike a match and walk to the fusebox while the match is burning

Resources

- A **resource** is a set of objects whose value or availability determines whether an action can be taken
  - e.g. money, drivers, trucks, surgeons, power
  - time is a resource which PDDL treats as a special case
- Resources can be consumable (e.g. fuel) or reusable (e.g. a plane)
- Resources can be produced by actions (e.g. hire a car, refuel a plane, grow a potato)


Planning with Resources

- A solution is a plan that achieves the goals while allocating resources to actions such that all resource constraints are satisfied
- A **satisficing plan** achieves the goals without violating any temporal and resource constraints
  - e.g. deliver all packages by 09.00
- An **optimal plan** achieves the goals while minimising (or maximising) a cost function, often defined in terms of resource usage
  - e.g. deliver all packages by 09.00, minimising the number of planes and fuel required

Scheduling Approach

- One approach is to **plan first and schedule later**
- Divide the overall problem into
  - Planning phase: select actions (with some ordering constraints) to meet the goals: partially ordered plan
  - Scheduling phase: add temporal information to ensure it meets resource and deadline constraints
- This approach is common in real-world manufacturing and logistical domains, where the planning phase is often done by human experts

Example: Assembly of Cars

```
Jobs({AddEngine1 ≺ AddWheels1 ≺ Inspect1 },
        {AddEngine2 ≺ AddWheels2 ≺ 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(30),
           USE: WheelStations(1))
Action(AddWheels2, DURATION: 15, CONSUME: LugNuts(20),
           USE: WheelStations(1))
Action(Inspect_i, DURATION: 10, USE: Inspectors(1))
```

- Each job has a set of actions with ordering constraints
- A ≺ B means that action A must precede action B
- Each action has a duration and a set of resource constraints
- Each constraint specifies type, number and consumable/reusable

Example from Chapter 11 of AIAMA

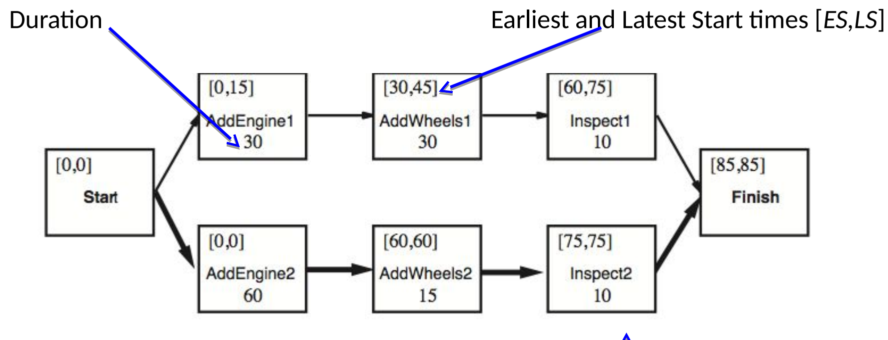
Aggregation

- If all objects are indistinguishable w.r.t. the purpose of the plan, complexity can be reduced by grouping individual objects into quantities – called aggregation
  - e.g. `Inspectors(2)` instead of `Inspector(Bob), Inspector(Jane)` because it does not matter *which* inspector inspects the car in our problem, so we don't need to make the distinction
- Consider a schedule proposing 10 concurrent inspections when there are only 9 available inspectors:
  - Inspectors represented as quantities – failure detected immediately, backtrack and try another schedule
  - Inspectors as individuals – algorithm backtracks to try all 10! ways of assigning inspectors to actions

Time Constraints: Critical Path Method

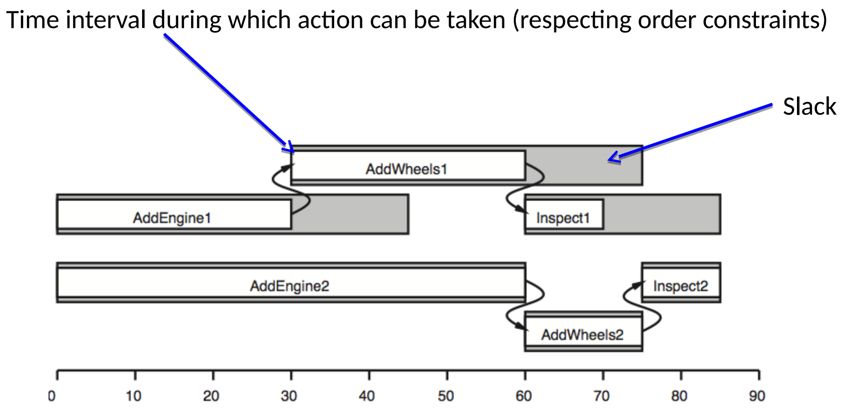
- **minimise the plan duration:** find the *earliest start times* for all actions
  - consistent with the **ordering constraints**
- **Critical path method** can find the possible start and end times for each action
- A **path:** sequence of actions beginning with *Start* and ending with *Finish*
- The **critical path**: path with the longest total duration; 'critical' because it determines the duration of the entire plan:
  - delaying the start of *any action* on the critical path slows down the entire plan
  - shortening other paths does not shorten the whole plan
- Actions *not* on the critical path have a window of time in which they can be executed: **LS – ES** is known as the *slack* for the action (ES earliest possible start time, LS latest possible start time)
- A **schedule** is the ES and LS times for all the actions

Example: Assembly of Cars



- Representation of temporal constraints
- Slack = LS - ES
- Actions with zero slack are on critical path

Example: Assembly of Cars

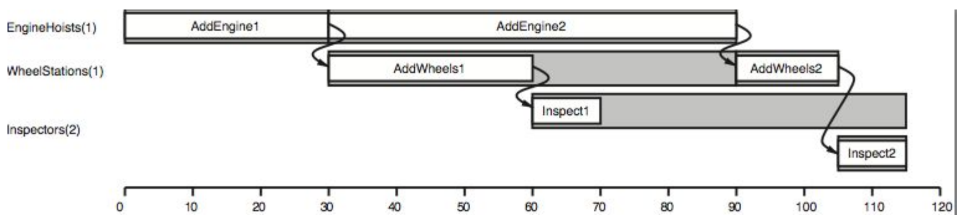


- Solution as a timeline

Resource Constraints

- Finding a minimum-duration schedule given a partial ordering on actions and no resource constraints is easy:
  – Any action can be executed in parallel with any other unless this is prohibited by the partial order specified in the plan
- Resource constraints impose additional restrictions on the ordering of actions – actions which require the same resources can't be executed at the same time
  – e.g. two `AddEngine` actions begin at the same time but both require the same `EngineHoist` and so a constraint "cannot overlap" must be added
- Scheduling with resource constraints is complex

Example: Assembly of Cars with Resource Constraints



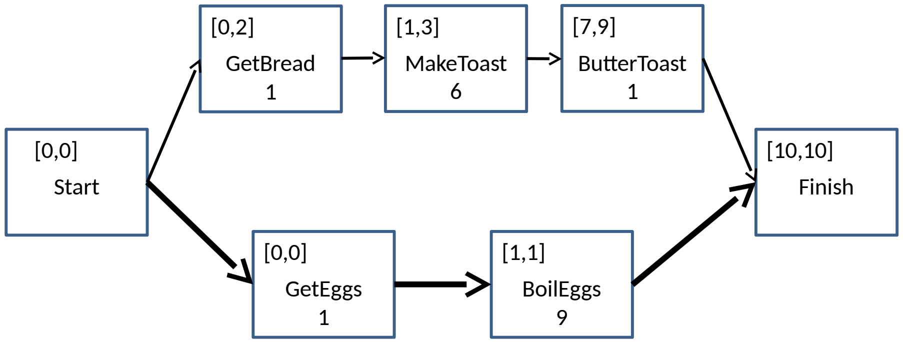
- Solution incorporates "cannot overlap" constraint
- Fastest solution takes 115 mins (30 mins longer)
- No time when both inspectors needed, so only need one for this solution

Exercise

- Draw a diagram to represent the temporal constraints of the following scheduling problem (assume start time [0,0]) and indicate the critical path:

```
Jobs({GetBread < MakeToast < ButterToast}, {GetEggs < BoilEggs})

Resources(Butter(1), Bread(2), Eggs(2), Water(500), Toaster(1),
    Knife(1), Pan(1))

Action(GetBread, DURATION: 1, USE: Bread(2))

Action(MakeToast, DURATION: 6, USE: Toaster(1), Bread(2))

Action(ButterToast , DURATION: 1, CONSUME: Butter(1), USE: Knife(1))

Action(GetEggs, DURATION: 1, USE: Eggs(2))

Action(BoilEggs, DURATION: 9, USE: Pan(1), Eggs(2), Water(500))
```

Solution



Reducing Complexity

- Complexity of scheduling with resource constraints is often seen in practice
  - e.g. challenge posed in 1963 to find the optimal schedule for a problem involving 10 machines and 10 jobs of 100 actions went unsolved for 23 years (Lawler *et al.* 1993)
- Minimum slack algorithm heuristic

```
REPEAT
IF (unscheduled(A) AND all_predec_scheduled(A)
   AND least_slack(A))
THEN schedule A for earliest possible start;
UPDATE ES and LS for all affected actions;
UNTIL solution produced
```

  - But for car assembly problem, solution longer (130 mins)
- Integrating planning and scheduling is active area of research

Managing Complexity: Hierarchical Decomposition

- State-of-the art planning algorithms can generate plans with thousands of actions
- However some planning tasks involve millions of actions, e.g.
  - Planning military operations
  - Plans executed by the human brain: to move about, if this is planned at the level of muscle activations (about 10^3 muscles, activation can be modulated 10 times per second, so planning for just one hour may involve more than 3 million actions)
- Solution: plan at a *higher level of abstraction*, e.g. instead of muscle activations, just an action 'walk to the shop', then refine if necessary

Example: Holiday

- A reasonable plan might be

  [Go to Manchester Airport; Take Emirates Air flight 778 to Dubai; Do holiday stuff for 2 weeks; Go to Dubai Airport; Take Emirates Air flight 779 to Manchester; Go home]
- Each action in the plan is a planning task in itself
  - e.g. 'Go to Manchester Airport' may have a solution [Drive to the airport car-park; park; take the shuttle bus to the terminal]
- Each of these actions may then be *decomposed* further until we reach the right level of actions
- Hierarchical decomposition
- Recall discussion about 'right' level of abstraction w.r.t. search

Hierarchical Decomposition

- *Software*: Hierarchy of subroutines or object classes
- *Armies*: hierarchy of units
- *Government and corporations*: hierarchy of departments, subsidiaries, branch offices
- Key benefit: at each level of the hierarchy a computational task, military mission or administrative function is reduced to a smaller number of activities at the next lower level
  – Computational cost of solving a planning problem is small

Hierarchical Task Networks

- HTN similar to classical planning:
  – States are sets of fluents (ground atomic formulae)
  – Actions correspond to deterministic state transitions
- Planning domain description extended: methods for decomposing tasks into subtasks
- Primitive actions: set of possible actions
- High-level actions: higher level abstraction of actions

High-Level Actions

- Each HLA has one or more possible refinements into a sequence of actions
- Each refinement may include HLAs or primitive actions
- Primitive actions by definition have no refinements
- Refinements may be recursive
- An HLA refinement that contains only primitive actions is called an implementation of the HLA

Example Refinement: Holiday

- The action 'Go to Manchester Airport' represented as *Go(Home,MAN)* might have two possible refinements:

```
Refinement(Go(Home,MAN),
STEPS: [Drive(Home,MANLongStayParking),
        Shuttle(MANLongStayParking,MAN)])
Refinement(Go(Home,MAN),
STEPS: [Taxi(Home,MAN)])
```

Example Refinement: Vacuum World

```
Refinement(Navigate([a,b],[x,y]),
    PRECOND: a=x  ∧ 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])
```

- Recursive refinement: to get to a destination, take a step, and then go to the destination
- `[Right,Right,Down]` and `[Down,Right,Right]` are both implementations of the HLA `Navigate([1,3],[3,2])`

High-Level Plan

- A high-level plan is a sequence of HLAs
- An implementation of a high-level plan 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
  - Not all implementations need to achieve the goal
- If a HLA has exactly one implementation, can compute preconditions and effects as if it were a primitive action

Summary

- Planning in the real world
  - Time constraints, critical path method, minimum slack
  - Resource constraints, abstraction, Hierarchical Task Networks
- This concludes our consideration of the topic Planning

- Next time
  - Machine learning