Info

12 October 2020 11:17

2 hour exam on AEP 80%
Coursework is 20%

Book
Moodle

There is a book at the start of each section

Foundations of Computation Agents, Poole and Mackworth

http://artint.info/2e/html/ArtInt2e.html

After reading text book, do exercises at the end of chapters and engage with seminar sheets

Reading

- S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 3rd Edition, Prentice Hall, 2010

Other Reading

The following books give a good discussion of Bayesian AI and of knowledge representation:

- D. Barber, Bayesian Reasoning and Machine Learning, Cambridge University Press, 2012
- R. Brachman and H. Levesque, Knowledge Representation and reasoning, Morgan Kaufmann, 2004
- K. Korb, A. Nicholson, Bayesian Artificial Intelligence, Chapman and Hall, 2004
- Jackson P, Introduction to Expert Systems, Addison Wesley, 1999
- Giarratano J, Riley G, Expert Systems; Principles and Programming (4th ed), 2005.

Other Reading

These books are good background for CS255:

- Callan, Artificial Intelligence, Palgrave
- Ginsberg, Essentials of Artificial Intelligence, Morgan Kaufman
- Nilsson, Artificial Intelligence: A New Synthesis, Morgan Kaufman
- The library has many other AI books containing useful background material (especially for coursework).

Learning Outcomes for CS255

At the end of the module you will:

- develop an appreciation for knowledge based systems, intelligent agents and their architectures,
- understand a wide variety of knowledge representation and artificial intelligence approaches to planning,
- understand various methods for search (uninformed and informed), planning and reinforcement learning, and
- understand various methods for representing and reasoning under uncertainty.

Exam Revision

1. Seminar Sheet 1
2. Seminar Sheet 2
3. Seminar Sheet 3
4. Seminar Sheet 4
5. Seminar Sheet 5
6. Seminar Sheet 6
7. Seminar Sheet 7
8. Seminar Sheet 8

What is AI?

13 October 2020 13:02

1. "The automation of activities that we associate with human thinking, activities such as decision-making problem solving, learning.."

2. "The study of mental faculties through the use of computational models"

3. "The study of how to make computers do things which at the moment people are better"

4. "The branch of computer science concerned with the automation of intelligent behaviour"

- Thinking humanly
- Acting humanly
- Thinking rationally
- Acting rationally

Success measured in terms of human behaviour or against some ideal of rationality

If you can think intelligently, then you will act intelligently

The Turing Test

Acting humanly suggests the main components of AI
Knowledge, reasoning, language, learning

So, is this a good test of intelligence?

Not really – not reproducible and can't be mathematically analysed. Not an intelligence test.
Things that are considered intelligent like object recognition are not tested

Searle's Chinese Room

Thinking Rationally

- Derivation of conclusion from particular premises
- Various forms of logic, notation, and rules of inference
- Task or problem expressed in login -> AI program deduces solution

Problem: Difficult to express tasks in logic
Problem: How to cope with uncertainty
Problem computational expense

Acting Rationally

Acting so as to maximise goal achievement, given the available information and resources
Thought or inference not necessarily involved, e.g. reflex actions
The right thing: acting so as to maximise goal achievement, given the information and resources available

Building Rational Agents

An agent is an entity that perceives and acts
An agent can be viewed as a function from percept histories to actions
F: P -> A

Agents typically required to exhibit autonomy. For any given class of environments and tasks we seek the agents with the best performance
We want to design the best agent we can with our limited resources

Artificial Intelligence

Is the synthesis and analysis of computational agents that act intelligently

An agent acts intelligently if:

- Its actions are appropriate for its goals and circumstances
- It is flexible to changing environments and goals
- It learns from experience
- It makes appropriate choices given perceptual and computational limitations

Rational Agents 1

16 October 2020 13:46

Inputs – in the context of a self-driving car

- Abilities – e.g. steering, braking
- Goals – safety, timeliness
- Prior knowledge – what signs mean
- Stimuli – vision, lasers
- Past experience – effects of steering, friction of surfaces, how people move

Rational Action

- Goals can be defined as a performance measure, defining a numerical value for a given environment history
- Rational action is whichever action maximises the expected value of the performance measure given the percept sequence to date - I.e. doing the right thing
- Previous perceptions are typically important

Rational Agents are not omniscient – they don't know the actual outcome of their actions. They just do the best they can, given the current percepts

Actions that were expected to give a good return but failed to, can still be considered rational

Dimensions of complexity

We can view the design space for AI as being defined by a set of dimensions of complexity. These dimensions define a **design space** for AI; different points in this space are obtained by varying the values on each dimension.

1. Modularity

Flat – one level of abstraction – adequate for simple systems. Continuous or discrete.
Modular – interacting modules that can be understood separately
Hierarchical – agent has modules that are recursively decomposed into modules – complex computers, biological systems etc

2. Planning horizon

Statis – world does not change
Finite – agent reasons about a fixed number of steps into the future
Indefinite – the agent thinks about a finite number of steps but we do not predetermine the number of steps
Infinite – the agent has to keep planning forever – process oriented

3. Representation

- Modern AI is about finding compact representations and exploiting the compactness for computational gain
- Explicitly – e.g. a chess board, one way the world could be
- Features of propositions – states can be described using features 30 binary features can represent 2^30 possible states
- Individuals and relations – there is a feature for each relationship on each tuple of individuals. Often an agent can reason without knowing the individuals or when there are infinitely man individuals
- When describing a complex world, the features can depend on **relations** and **individuals**. What we call an *individual* could also be called a **thing**, an **object** or an **entity**. A relation on a single individual is a **property**. There is a feature for each possible relationship among the individuals.

E.g. With a light switch s_2
Instead of the feature, it could use the relation $position(s2, up)$. *This relation enables the agent to reason about all switches or for an agent to have general knowledge about switches that can be used when the agent encounters a switch.*

By reasoning in terms of relations and individuals, an agent can reason about whole classes of individuals without ever enumerating the features or propositions, let alone the states. An

agent may have to reason about infinite sets of individuals, such as the set of all numbers or the set of all sentences. To reason about an unbounded or infinite number of individuals, an agent cannot reason in terms of states or features; it must reason at the relational level.

4. Computational Limits

Perfect rationality: the agent can determine the best course of action, without taking into account its limited computational resources
Bounded rationality – we have to make good decisions based on limited resources e.g. memory

To take into account bounded rationality, an agent must decide whether it should act or reason for longer. This is challenging because an agent typically does not know how much better off it would be if it only spent a little bit more time reasoning. Moreover, the time spent thinking about whether it should reason may detract from actually reasoning about the domain.

5. Learning from experience

The model is specified a priori

- Knowledge is given
- Knowledge is learned from data or past experience

Usually some mix of prior knowledge is used – nature vs nurture

6. Uncertainty

Sensing and effect

In some cases, an agent can observe the state of the world directly. For example, in some board games or on a factory floor, an agent may know exactly the state of the world. In many other cases, it may only have some noisy perception of the state and the best it can do is to have a probability distribution over the set of possible states based on what it perceives. For example, given a patient's symptoms, a medical doctor may not actually know which disease a patient has and may have only a probability distribution over the diseases the patient may have.

The **sensing uncertainty dimension** concerns whether the agent can determine the state from the stimuli

In each dimension an agent can have

- No uncertainty – e.g. you will run out of power
- Disjunctive uncertainty – there is a set of states that are possible e.g. charge for 30 minutes, or you will run out of power
- Probabilistic uncertainty - Agents need to act even if they are uncertain. We need to predict what might happen in order to decide what to do - e.g. probability you will run out of power is 0.01 if you charge for 30 minutes and 0.8 otherwise

Probabilities can be learned from data and prior knowledge
Acting is gambling – if you don't use probabilities you will lose to agents that do

Sensing uncertainty

- Fully observable – the agent can observe the state of the world
- Partially-observable – there can be a number of states possible given what the agent can perceive

Effect uncertainty

If an agent knew the initial state and its action, could it predict the resulting state?
Deterministic: the resulting state is determined from the action and the state
Stochastic – there is only a probability distribution over the resulting states.

7. Preference

What is the agent trying to achieve?

- Achieve goal is a goal – this can be a complex logical formula
- Complex preference – maybe involve trade-offs between desiderate, perhaps at different times
- Ordinal – the order is the only thing that matters

- Cardinal – counts matter e.g. we want exactly 0 crashes

8. Number of agents

Are there multiple agents?
Single agent – any other agents are part of the environment
Multiple agent reasoning – an agent reasons strategically about the reasoning of other agents

Interaction

When does the agent reason to determine what to do?
Online – while interacting
Offline – before acting

Example: State-space Search

Dimension	Values
Modularity	*flat*, modular, hierarchical
Planning horizon	non-planning, finite stage, *indefinite stage*, infinite stage
Representation	*states*, features, relations
Computational limits	*perfect rationality*, bounded rationality
Learning	*knowledge is given*, knowledge is learned
Sensing uncertainty	*fully observable*, partially observable
Effect uncertainty	*deterministic*, stochastic
Preference	*goals*, complex preferences
Number of agents	*single agent*, multiple agents
Interaction	*offline*, online

The dimensions interact in complex ways

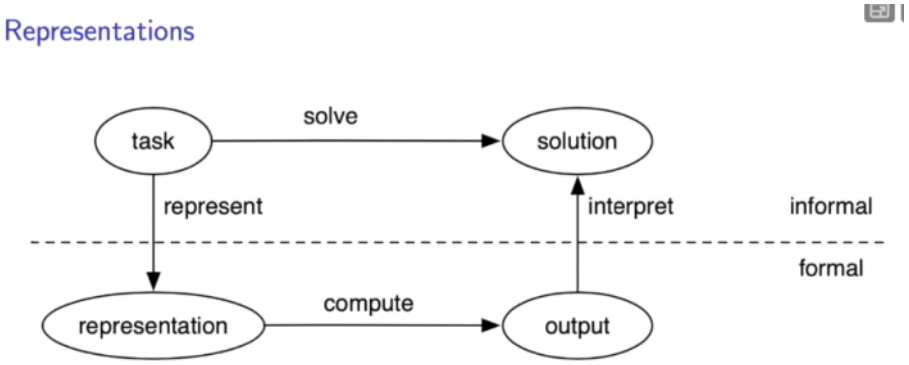
Partial observability makes multi-agent and indefinite horizon reasoning more complex
Modularity interacts with uncertainty and succinctness: some levels may be fully observable, some may be partially
Three values of dimensions promise to make reasoning simpler for the agent

- Hierarchical reasoning
- Individuals and relations
- Bounded rationality

Rational Agents 2

16 October 2020 14:17

Desirable properties for a representation



- determine what constitutes a solution
- represent the task in a way a computer can reason about
- use the computer to compute an output, which is answers presented to a user or actions to be carried out in the environment, and
- interpret the output as a solution to the task.

We have a task, that is represented in the AI system in some way.
The computation is done on this representation of the task, and then the output is relayed back out into the world.

A **representation** of some piece of knowledge is the particular data structures used to encode the knowledge so it can be reasoned with. A **knowledge base** is the representation of all of the knowledge that is stored by an agent.

We want our representation to have a few characteristics:

- Rich in data to express the knowledge needed to solve the problem
- As close to the problem as possible: compact, natural and maintainable
- Amenable to efficient computation – expresses features of the problem that can be exploited for computational gain - Able to trade off accuracy and computation time/space
- Able to be acquired from people, data, and past experiences.

Defining a solution

Given an informal description of a solution, what is a solution?
Typically, much is left unspecified, but the unspecified parts can't be filled in arbitrarily
Muc work in AI is motivated by common-sense reasoning – the computer needs to make common-sense conclusions about unstated assumptions

Quality of solutions

Does it matter if the answer is wrong or answers are missing?

- An optimal solution is a best solution according to some measure of solution quality
- A satisficing solution is one that is good enough according to some description of solutions that are adequate

- An approximately optimal solution is one whose measure of quality is close to the best theoretically possible. E.g. get within 10% of the optimal solution. This is sometimes still just as hard as getting the optimal solution, though.
- A probably solution is one that is likely to be a solution

Decision and Outcomes

- Good and bad decisions can have good and bad outcomes
- Information can be valuable because it leads to better decisions: value of information
- We can often trade off computation time and solution quality – An anytime algorithm can provide a solution in any time, but makes better decisions with more time

Agents are concerned not just about finding the right answer, but finding the information that will allow them to find the right answer

Choosing a representation
We need to represent a problem to solve it on a computer

Physical symbol system hypothesis
A symbol is a physical pattern that can be manipulated
A symbol system allows you to create/modify/delete symbols

The hypothesis states that a physical symbol system has the necessary and sufficient means for general intelligent action

Knowledge & Symbol Levels

Two levels of abstraction seem to be common among entities – biological and computational

- Knowledge level is about the external world – what the agent knows and what its goals are
- Symbol level is about what symbols the agent uses to implement the knowledge level – it is a level of description of an agent in terms of what reasoning it is doing

Mapping from Problem to Representation

- What level of abstraction of a problem to represent?
- What individuals and relations in the world to represent?
- How can an agent represent knowledge to ensure that the representation is natural, modular and maintainable?
- How can an agent acquire the information from data, sensing, experience, or other agents?

Choosing a level of abstraction

- High-level is easier to understand for a human
- a low-level description can be more accurate and more predictive, we lose details when we abstract away details in high level abstractions
- You may not know the information needed for a low-level description

A delivery robot can model the environment at a high level of abstraction in terms of rooms, corridors, doors, and obstacles, ignoring distances, its size, the steering angles needed, the slippage of the wheels, the weight of parcels, the details of obstacles, the political situation in Canada, and virtually everything else. The robot could model the environment at lower levels of abstraction by taking some of these details into account. Some of these details may be irrelevant for the successful implementation of the robot, but some may be crucial for the robot to succeed. For example, in some situations the size of the robot and the steering angles may be crucial for not getting stuck around a particular corner. In other situations, if the robot stays close to the centre of the corridor, it may not need to model its width or the steering angles.

Although no level of description is more important than any other, we conjecture that you do not have to emulate every level of a human to build an AI agent but rather you can emulate the higher levels and build them on the foundation of modern computers. This conjecture is part of what AI studies.

It is sometimes possible to use multiple levels of abstraction

Reasoning and Acting

Reasoning determines what action an agent should do

1. Design time reasoning – done by the designer of the agent
2. Offline computation – done by the agent before it has to act
3. Online computation – computation done by an agent receiving information and acting

Architecture 1

17 October 2020 16:05

Architecture & Hierarchical Control

Agents

By a hierarchic system, or hierarchy, I mean a system that is composed of interrelated subsystems, each of the latter being in turn hierarchic in structure until we reach some lowest level of elementary subsystem.

An **agent** is something that acts in an environment.

Agents interact with the environment with a body. An embodied agent has a physical body. A robot is an artificial purposive embodies agent.

Agents act in the world through their actuators, also called effectors.

Agent Systems

An agent system is made up of an agent and the environment in which it acts.
Agents in an agent system receive stimuli from their environment and carries out actions on the environment.

An agent is made up of a body and a controller. The controller receives percepts from the body and sends commands to the body.

The Agent Function

Agents are situated in time T that can be thought of as discrete, broken into sub-sections. T+1 is one of these intervals after T.

If time is dense, there is always another moment in time between any two other given moments – e.g. time is continuous.

Assume that T has a starting point, which we arbitrarily call 0.

Suppose P is the set of all possible percepts. A **percept trace**, or **percept stream**, is a function from T into P. It specifies what is observed at each time.

Suppose C is the set of all commands. A **command trace** is a function from T into C. It specifies the command for each time point.

A percept trace for an agent is thus the sequence of all past, present, and future precepts received by the controller. A command trace is the sequence of all past, present, and future commands issued by the controller. The commands can be a function of the history of percepts. This gives rise to the concept of a **transduction**, a function from percept traces into command traces.

A transduction is **causal** if, for all times t, the command at time t depends only on percepts up to and including time t. The causality restriction is needed because agents are situated in time; their command at any time cannot depend on future percepts.

A **controller** is an implementation of a causal transduction.

The **history** of an agent at time t is the percept trace of the agent for all times before or at time t and the command trace of the agent before time t.

Thus, a **causal transduction** maps the agent's history at time t into the command at time t. It can be seen as the most general specification of a controller.

Belief State

Although a causal transduction is a function of an agent's history, it cannot be directly implemented due to the fact that an agent does not have access to its entire history. It only has access to its current percepts and those that it has remembered.

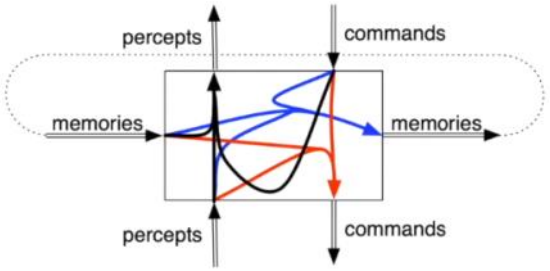
The memory or belief state of an agent at time t is all the information the agent has remembered from the previous times. An agent has access only to the history it has encoded in its belief state. Thus, the belief state encapsulates all the information about an agents history, that the agent can use for current and future commands. At any time, ag agent has access to its belief state and its current percepts.

The belief state can contain any information, subject to the agent's memory and processing limitations. This is a very general notion of belief.

The belief state function determines what we're going to remember - I.e. what will the next belief state look like
The command function uses the current belief state and the current percepts to decide on some action

If there are a finite number of possible belief states, the controller is called a **finite state controller** or a **finite state machine**. A **factored representation** is one in which the belief states, percepts, or commands are defined by features. If there are a finite number of features, and each feature can only have a finite number of possible values, the controller is a **factored finite state machine**. Richer controllers can be built using an unbounded number of values or an unbounded number of features. A controller that has an unbounded but countable number of states can compute anything that is computable by a Turing machine.

Functions Implemented in a Layer



- *memory function:* $remember(memory, percept, command)$
- *command function:* $do(memory, percept, command)$
- *percept function:* $higher\_percept(memory, percept, command)$

Hierarchical Control

There is much evidence that people have multiple qualitatively different levels. Kahneman [2011] presents evidence for two distinct levels: **System 1**, the lower level, is fast, automatic, parallel, intuitive, instinctive, emotional, and not open to introspection, and **System 2**, the higher level, is slow, deliberate, serial, open to introspection, and based on reasoning.

In a hierarchical controller there can be multiple channels – each representing a feature – between layers and between layers at different times.

There are three types of inputs to each layer at each time:

- the features that come from the belief state, which are referred to as the remembered or previous values of these features
- the features representing the precepts from the layer below in the hierarchy
- the features representing the commands from the layer above in the hierarchy.

There are three types of outputs from each layer at each time:

- the higher-level percepts for the layer above
- the lower-level commands for the layer below
- the next values for the belief-state features.

The low-level controllers can run much faster, and react quickly
They also deliver a simple view of the world to the high level controllers.

Architecture 2

17 October 2020 18:05

- Agent Functions
- Types of Agent

We can view agents as being specific by the agent function that maps a percept sequence to a sequence of actions

Ideal rational agents do whatever action is expected to maximise performance measure on basis of percept sequence and built-in knowledge.
So in principle there is an ideal mapping of percept sequences to actions
The simple approach to this is a lookup table, however this is doomed to failure be

The lookup table suggests a notion of an ideal mapping
This would be the rational agent function
We want to implement the rational agent function by implementing a function that approximates the ideal mapping as closely as possible

Agent Types

1. Simple reflex agent – condition action rules
2. Reflex agents with state – retains knowledge about the world
3. Goal-based agents – have a representation of desirable states
4. Utility based agents – ability to discern some useful measure between possible means of achieving state
5. Learning agents – able to modify their behaviour based on their performance, or in light of new information

Goal Based Agents
If we want to solve more sophisticated problems, we might need to look at a GBA
It maintains some representation of the state, which gets updated. It has some knowledge about how the world works, and a set of goals

To achieve goals, you need a sequence of actions. We need to keep track of how close we are to achieving our goal. GBA's are much more flexible.

Utility Based Agents
Similar to GBA, but rather than purely thinking about what action we should do in terms of the goal, we think about the benefit a particular action has.

Sometimes there are many ways of achieving a goal, and many actions you can take to get there.

Utility based agents have a utility function that allows us between which goals we should achieve, and alternative ways of achieving a goal

Learning Agents

Can be built upon any of the previous types – all 4 can fit into the performance element

The performance element takes some input, and chooses some action on the environment.
The learning agent has a critic, a problem generator and a learning element

The learning element uses information about how the agent operates and the particular algorithms, and changes how the performance element operates. It does this by getting information on how it's doing from the critic, and new things to try out from the problem generator.

Useful when we don't know much about the environment and the agent has to be able to learn.

Learning provides an agent with a degree of autonomy

Learning results from interaction between the agent and the world.

The critic uses a performance standard to tell the agent how it is doing. Performance standard should be a fixed measure, external to the agent.

Problem generator – responsible for suggesting actions in pursuit of new and informative experiences

Uninformed Search

18 October 2020 21:18

- Problem solving agents
- Problem types
- Problem formulation
- State space graphs
- Basic tree search and graph search algorithms

An agent could be programmed to act in the world to achieve a fixed goal or set of goals, but then it would not adapt to changing goals, and so would not be intelligent. An intelligent agent needs to reason about its abilities and its goals to determine what to do.

A problem-solving agent is a goal-based agent that will determine a sequence of actions that will achieve some goal state

There are 4 steps a problem solving agent must take

- Goal formulation
- Problem formulation
- Search – find the sequence of actions
- Execution

In the simplest scenario, an agent has no uncertainty, and has a state based model of the world, with a goal to achieve. This is either a flat representation or a single level of a hierarchy. The agent is able to determine how to achieve this goal by searching in its representation of the world state space for a way to get from its current state to a state that satisfies its goal. Given a complete model, it tries to find a sequence of actions that will achieve its goal before it has to act in the world.

This problem is like finding a path from the start node to an end node in a directed graph.

Problem Types

- Deterministic, fully observable – single state problem
- Deterministic, partially observable – multi-state problem
- Stochastic, partially observable – contingency probably
- Unknown state space

Searching

"searching" in this chapter means searching in an internal representation for a path to a goal – from start node to end node.

Search underlies much of artificial intelligence. When an agent is given a problem, it is usually given a description that allows it to recognise a solution, but not an actual algorithm for a solution. The agent has to search for a solution itself.

The difficulty of search and the fact that humans are able to solve some search problems efficiently suggests that computer agents should exploit knowledge about special cases to guide them to a solution. This extra knowledge beyond the search space is called **heuristic knowledge**. This chapter considers one kind of heuristic knowledge in the form of an estimate of the cost from a node to a goal.

State Space
One general formulation of intelligent action is in terms of a state sapce. A state contains all of the information necessary to compute the effects of an action and to determine whether a states satisfies a goal.

When we do a state space search, we assume:
The agent has perfect knowledge of the state space and is planning for the case where it observes what state it is in: there is full observability
The agent has a set of actions that have known deterministic effects
The agent can determine whether a state satisfies a goal

A solution is a sequence of actions that will get the agent from the current state to the state that satisfies the goal.

A state space problem consists of

- A set of states
- A distinguished state called the start state
- For each state, a set of actions available to the agent in that state
- An agent function that, given a state and an action, returns a new state
- A goal specified as a Boolean function that is true when state satisfies the goal, in which case we can say that s is a goal state
- A criterion that specifies the quality of an acceptable solution. For example, any sequence of actions that gets the agent to the goal state may be acceptable, or there may be costs associated with actions and the agent may be required to find a sequence that has minimal total cost. A solution that is best according to some criterion is called an optimal solution. We do not always need an optimal solution, for example, we may be satisfied with any solution that is within 10% of optimal.

Graph Searching

In this chapter, the problem of finding a sequence of actions to achieve a goal is abstracted as searching for paths in directed graphs.

To solve a problem, we first define the underlying search space, then apply a search algorithm to that search space. Many problem solving tasks are transformable into the problem of finding a path in a graph.

A directed graph consists of a set of nodes and a set of directed arcs between nodes. The idea is to find a path along these arcs from the start node to a goal node.

In representing a state-space problem, the states are represented as nodes, and the actions as arcs.

The abstraction is necessary because there may be more than one way to represent a problem as a graph. The examples in this chapter are in terms of state-space searching, where nodes represent states and arcs represent actions.

Formal Graph Searching

A directed graph has a set N of nodes and a set A of arcs, where an arc is an ordered pair of nodes.

Nodes are neighbours if there is an arc between them. Note that the relationship is not symmetrical – it doesn't necessarily go both ways.

A path from node s to node g is a sequence of nodes (n0, n1, … nk) such that s = n0 and g = nk, and there is an arc between each ni and ni+1.

A goal is a Boolean function on nodes. If goal(n) is true, we say that node n satisfies the goal, and n is a goal node.

To encode problems as graphs, one node is identifies as a start node. A solution is a path from the start node to a node that satisfies the goal.

Sometimes there is a cost, a non-negative number associated with the arcs. We write the cost of an arc as cost(ni, nj). The costs of arcs induce a cost of paths. Given a path p, cost of path p is the sum of the costs of all the arcs in the path.

An optimal solution is the solution that has the lowest cost.

That is, an optimal solution is a path p from the start node to a goal node such taht there is no path p' from the start node to a goal node where the cost is lesser.

A cycle is a non-empty path where the end node is the same as the start node. A directed graph without any cycles is called a directed acyclic graph.

A tree is a DAG where there is one node with no incoming arcs and every other node has exactly one incoming arc. The node with no incoming arcs is the root of the tree. A node with no outgoing arcs is a leaf. In a tree, neighbours are called children.

In many problems, the search graph is not given explicitly, but is constructed as needed. For the search algorithms, all that is required is a way to generate the neighbours of a node, and to determine if the node is a goal node.

The forward branching factor of a node is the number of outgoing arcs. The backward branching factor is the number of incoming arcs to the node. These factors provide measures for the complexity of graph algorithms. We assume the branching factors are bounded, when we discuss the space and time complexity.

Tree Search

18 October 2020 23:09

- Basic tree search and graph search algorithms

In the vacuum example, there were 2 squares, and a binary dirt option – dirt or no dirt It was simple to compute the state space diagram for this scenario. In real life, e.g. a self-driving car on a street, there are going to be thousands of variables and thus an intractable number of potential states – how do we deal with this?

Tree Search Algorithms

Offline, simulated exploration of the state space
Starting with a start state, expand one of the explored states by generating its successors, e.g. find neighbours by considering possible actions to build a search tree.

```
function TREE-SEARCH(problem,strategy)
    returns solution, or failure
  initialise search tree using initial state of problem
  loop do
    if no candidates for expansion then return failure
    choose leaf node for expansion according to strategy
    if node contains a goal state
        then return corresponding solution
    else expand node and add resulting nodes to
        search tree
  end
```

Search Strategies
Evaluation
Completeness
Optimality – always find a least-cost solution
Time complexity – maximum branching factor, depth of least cost solution, maximum depth of state space
Space complexity – maximum number of nodes in memory

Graph Search

18 October 2020 23:28

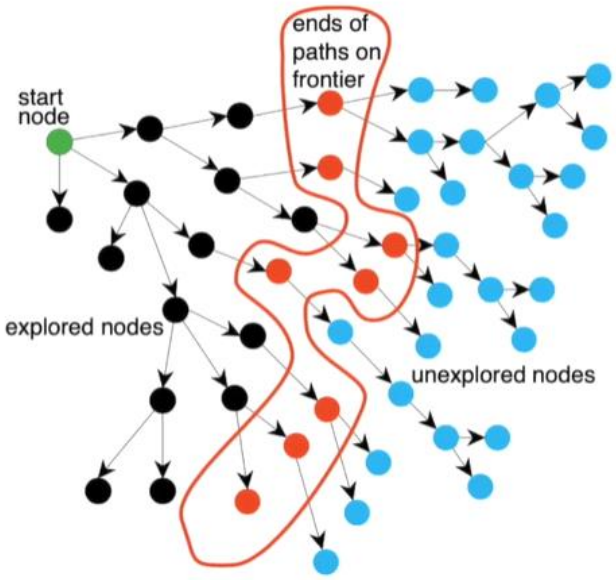
Objectives:

- Graph search algorithms

With a tree search, state spaces with loops give rise to repeated states that cause inefficiencies and infinite loops

Graph search is a practical way of exploring the state space that can account for such repetitions

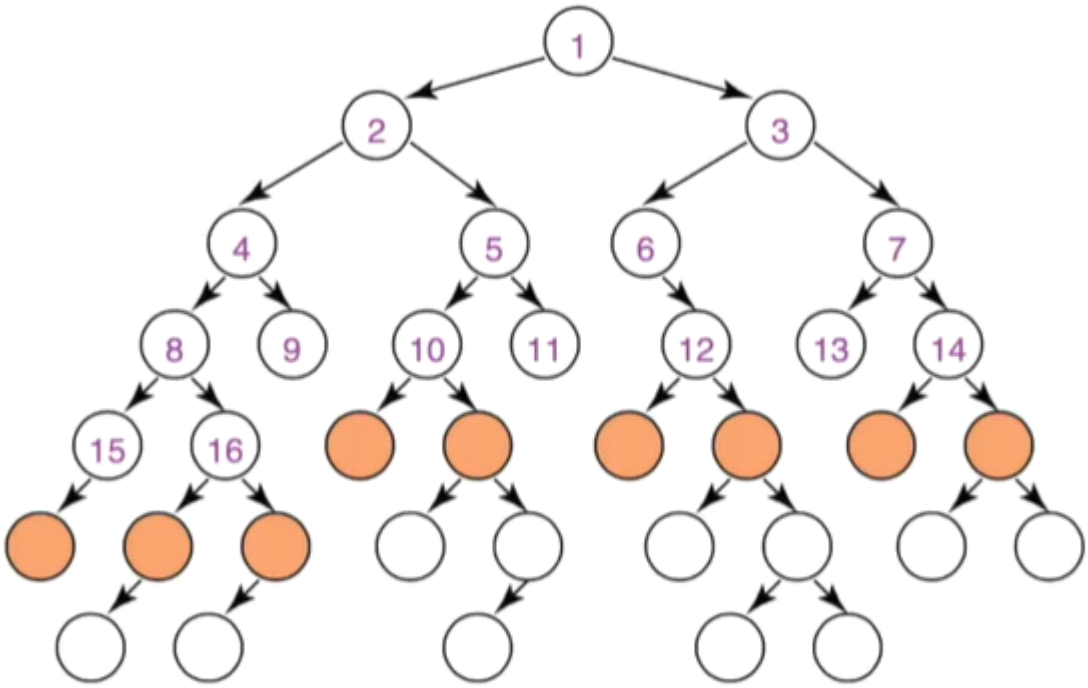
1. Given a graph, incrementally explore paths from the start nodes
2. Maintain a frontier of paths
3. As search proceeds, the frontier expands into the unexplored nodes until a goal node is encountered
4. The way in which the frontier is expanded is defined by the search strategy



Starts with a set of nodes
We check if we're at the goal
We start with a frontier – at the start, we just have the start node
We check the frontier, and look for a goal. If none found, add next nodes to frontier and repeat.

Breadth-First Graph Search

Treat frontier as a queue
Select earliest elements added to frontier



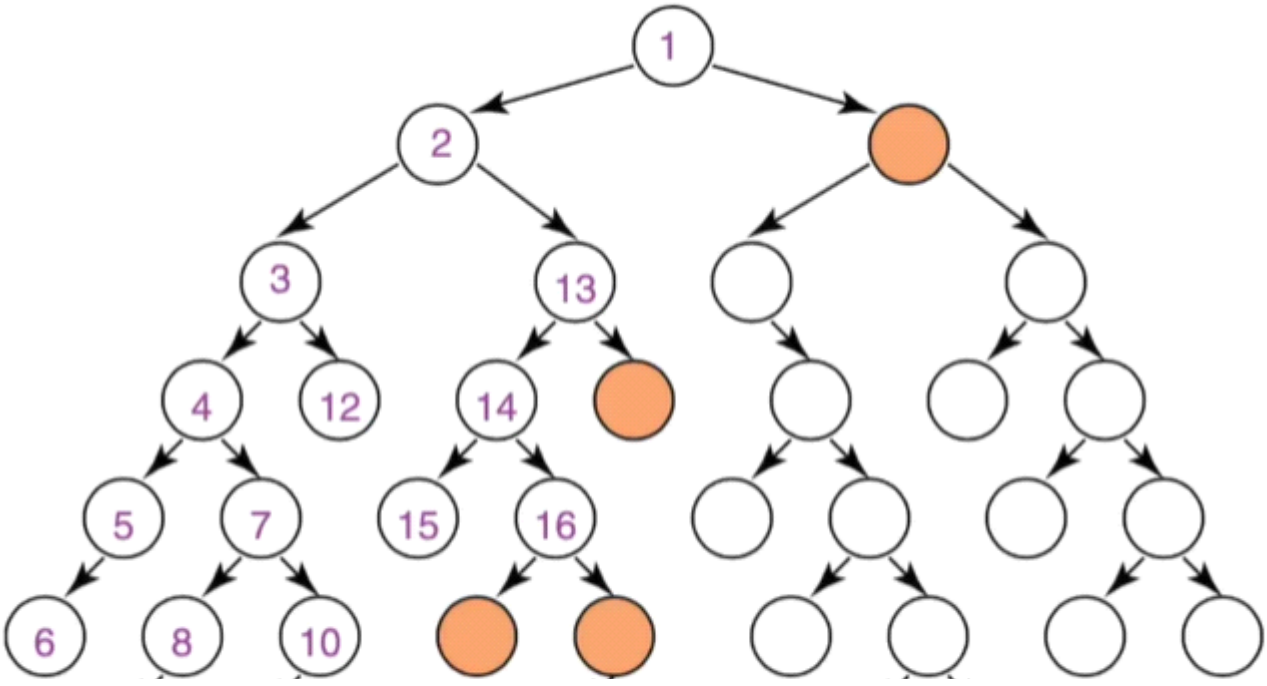
- If branching factor for all nodes is finite, bf graph search will find a solution
- Time complexity is exponential in path length - branching factor the power of n where n is the path length
- The space complexity is exponential in path length b^n
- The search is unconstrained by the goal

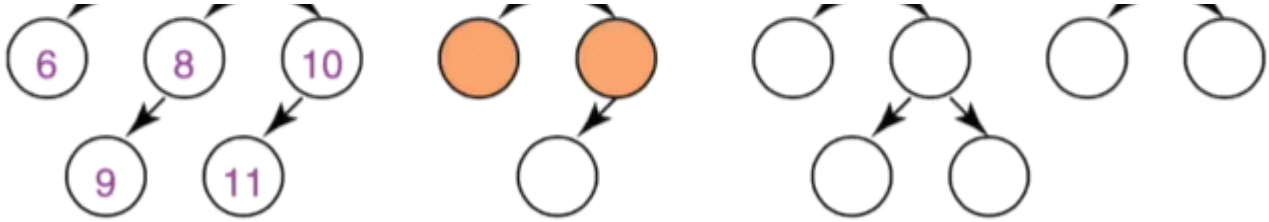
Depth-First Graph Search

Treat the frontier as a stack
Select the last element added to the frontier

If the list of paths on the frontier is p1, p2, p3...

- P1 is selected and the paths that extend p1 are added to the front of the stack in front of p2
- P2 is only selected when all of the paths from p1 have been explored





Complexity of a Depth-First Graph Search

- Not guaranteed to halt if we have cycles of infinite graphs
- The space complexity is linear in the number of arcs from the start of the current node
- If the graph is a finite tree, with a forward branching factor <= b and all paths from the start having at most k arcs, worst case time complexity is O(b^k)
- The search is unconstrained by the goal

Lowest-Cost-First Search

- Sometimes there are costs associated with arcs
- The cost of a path is the sum of the costs of its arcs
- An optimal solution is one with the minimum cost
- At each stage, the lowest-cost-first search selects a path on the frontier with the lowest cost
- The frontier is a priority queue ordered by path cost
- The first path to a goal is a least-cost path to a goal node
- When arcs costs are equal then breadth-first search

Summary of uninformed search strategies

Strategy	Frontier Selection	Complete	Halts	Space
Breadth-first	First node added	Yes	No	Exp
Depth-first	Last node added	No	No	Linear
Lowest-cost-first	Minimal $cost(p)$	Yes	No	Exp

Complete — guaranteed to find a solution if there is one (for graphs with finite number of neighbours, even on infinite graphs)
Halts — on finite graph (perhaps with cycles)
Space — as a function of the length of current path.

Informed Search

23 October 2020 21:44

Objectives:

- Best first search
- A* search
- Pruning search
- Depth bounded search
- Brand and bound search
- Heuristics

AIFCA 3.6-3.8

Uninformed search is generally very inefficient; if we have extra information about the problem we should use it
Improve search using problem specific knowledge
This is still offline problem solving since we have complete knowledge of the problem and solution

Heuristic

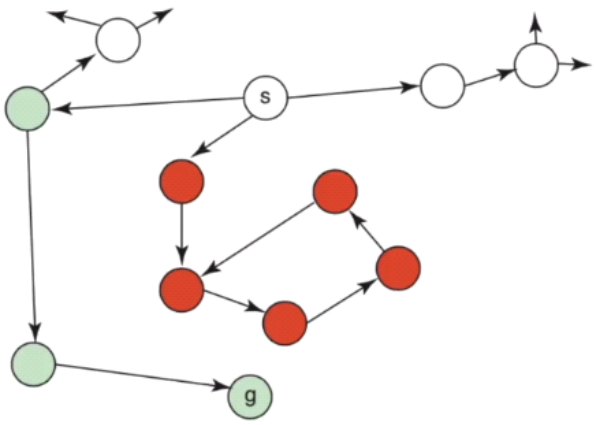
- idea: don't ignore the goal when selecting paths
- Often there is extra knowledge that can be used to guide the search: heuristics
- H(n) is an estimate of the cost of the shortest path from node n to a goal not
- H can be extended to paths
- H(n) is an underestimate if there is no path from n to a goal with cost strictly less than h(n)
- An admissible heuristic is a nonnegative heuristic function that is an underestimate of the actual cost of the path to the goal

Example Heuristic Function

- If the nodes are points on a Euclidean plane and the cost is the distance, h(n) can be the straight-line distance from n to the closest goal
- If the nodes are locations and cost is time, we can use the distance to a goal divided by max speed

Best-First Search Informed Search

- We can use the heuristic function to determine the order of the stack representing the frontier
- Idea: Select the path or node that is closet to a goal according to the heuristic function
- Heuristic depth-first search selects a neighbour so that the neighbour is selected first
- Greedy best fs selects a path on the frontier with the lowest heuristic value
- Best first search treats the frontier as priority
- queue ordered by h



The above is bad for a simple best-first search: heuristic depth first search will select the node below s and never terminate. Greedy best-first search will cycle between the nodes below s never finding an alternative route.

Complexity of Greedy best-first search

- The space complexity is exponential in path length b^n
- Time complexity is exponential in the path length
- Not guaranteed to find a solution, even if one exists
- It does not always find the shortest path

A* Search

It can be seen as an extension of Edsger Dijkstra's 1959 algorithm. A* achieves better performance by using heuristics to guide its search.

- Uses both path cost like in lowest-cost-first, and heuristic values, like in greedy best-first search
- Cost(p) is the cost of path p
- H(p) estimates the cost from the end of p to a goal
- Let(p) = cost(p) + h(p)
- F(p) estimates the total path cost of going from a start node to a goal via p

- A* is a mix of lowest-cost-first and best-first-search
- It treats the frontier as a priority queue ordered by f(p)
- It always selects the node on the frontier with the lowest estimated distance from the start to a goal node constrained to go via that node.

A search algorithm is *admissible* if, whenever a solution exists, it returns an optimal solution

A* is admissible if

1. The arc costs are greater than 0
2. The branching factor is finite
3. The heuristic h is a non-negative and an underestimate

Why is it admissible?

Why is A^* admissible?

- If a path p to a goal is selected from a frontier, can there be a shorter path to a goal?
- Suppose path p' is on the frontier. Because p was chosen before p', and $h(p) = 0$:

$$cost(p) \leq cost(p') + h(p')$$

- Because h is an underestimate:

$$cost(p') + h(p') \leq cost(p'')$$

  for any path p'' to a goal that extends p'
- So $cost(p) \leq cost(p'')$ for any other path p'' to a goal.

- The frontier always contains the initial part of a path to a goal, before that goal is selected
- A* halts, as the costs of the paths on the frontier keeps increasing, and will eventually exceed

any finite number

- Note admissibility does not guarantee that every node selected from the frontier is on an optimal path
- Although it does ensure that the first solution found will be optimal, even in graphs with cycles

Iterative deepening A*

IDA* performs repeated depth-bounded depth-first searches. Instead of the bound being on the number of arcs in the path, it is a bound on the value of f(n). The threshold is initially the vlaue of f(s), where s is the start node. IDA* then carries out a depth-first depth-bounded search but never expands a path with a high f-value than the current bound.

How can a better heuristic function help?

- A* expands all paths from the start in the set
- A* also expands some paths from the set
- Increasing h while keeping it admissible reduces the size of the first of these sets
- If the second set is large there can be significant variability in the space and time of A*

Complexity of A*

Exponential time complexity
Exponential space complexity – you keep all nodes in memory

Strategy	Selection from Frontier	Path found	Space
Breadth-first	First node added	Fewest arcs	Exponential
Depth-first	Last node added	No	Linear
Iterative deepening	—	Fewest arcs	Linear
Greedy best-first	Minimal $h(p)$	No	Exponential
Lowest-cost-first	Minimal cost (p)	Least cost	Exponential
A^*	Minimal cost $(p) + h(p)$	Least cost	Exponential
IDA*	—	Least cost	Linear

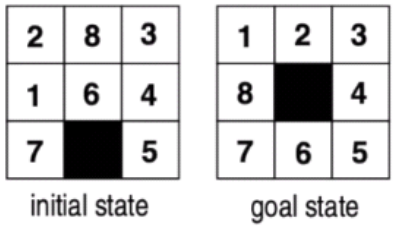
Heuristics

04 November 2020 20:32

A **heuristic function** $h(n)$, takes a node n and returns a non-negative real number that is an estimate of the cost of the least-cost path from node nn to a goal node. The function $h(n)$ is an **admissible heuristic** if $h(n)$ is always less than or equal to the actual cost of a lowest-cost path from node n to a goal.

How do we design our heuristics to make the best search algorithm possible?

Admissible heuristics: Example



- 8-puzzle is just hard enough to be interesting
- Branching factor 3(ish), typical solution around 20 steps
- Exhaustive search: 3^{20} states $(= 3.5 \times 10^7)$
- Eliminating repeating states: $9! = 362880$ states
- Need a decent heuristic.

Possible heuristics

H(n) = number of misplaced tiles
H(n) = Manhattan distance

Characterising Heuristics

- If A* tree-search expands N nodes and solution is depth d, then the effective branching factor * is the branching factor a uniform tree of depth d would have to contain N nodes.
- The closer the effective branching factor is to 1, then the larger the problem that can be solved.
- Can estimate b* experimentally (usually fairly consistent over problem instances)
- Q) is h2 always better than h1? If h(n) is bigger than another h(n) for all n then that heuristic is said to dominate the other heuristic

Deriving Heuristic

Can derive admissible heuristics of the exact solution cost of a relaxed version of the problem
A problem with less restrictions on operators is a relaxed problem
E.g. 8 puzzle, we can relax it and say that a tile can move from A to B if B is blank

If one dominates, use that. Else, calculate the h value for each state and use maximum value. If all options are admissible, then the chosen one will be admissible

Subproblems

Derive admissible heuristics from solution cost of a subproblem of given problem
Cost of subproblem = lower bound on cost of complete problem

We can store exact solution costs in a database which we can use to lookup values

We can combine pattern databases

Disjoint pattern databases
If we can divide up the problem so moves only affect a single subproblem

Statistical Approach
Run search over training problems and gather statistics

Pruning

04 November 2020 20:40

Pruning the Search Space

The preceding algorithms can be improved by taking into account multiple paths to a node. We consider two pruning strategies. The simplest strategy is to prune cycles; if the goal is to find a least cost path, there is no use considering paths with cycles. The other strategy is only ever to consider one path to a node, and to prune other paths to that node.

Cycle Pruning

- The search can prune a path that ends in a node already on the path, without removing an optimal solution
- In depth-first methods, checking for cycles can be done in a constant time in path length
- For other methods checking for cycles can be done in linear time in path length

A simple method of pruning the search while guaranteeing that a solution will be found in a finite graph, is to ensure that the algorithm does not consider neighbours that are already on the path from the start.

Cycle pruning checks whether the last node on the path already appears earlier on the path from the start node to that node. Paths where the end node is already in the path are not added to the frontier, or are discarded when removed from the frontier.

The complexity of cycle pruning depends on which search method is used. For DFS, overhead can be constant if a hash function is used that sets a bit when the node is in the path. Alternatively, for methods that have exponential space, cycle pruning takes time linear in the length of the path being searched. These algorithms cannot do better than simply searching up the initial path being considered, checking to ensure that they do not add a node that already appears in the path.

Multiple path pruning

- prune a path to a node if we've already found a path to that node
- This is done using a closed list of nodes at the end of expanded nodes
- When a path is selected, if the end node is in the closed list then the path is discarded, otherwise the node is added to the closed list and the algorithm proceeds as before..

Multiple-path pruning is implemented by maintaining an explored set (**closed list**) of nodes that are at the end of paths that have been expanded. The explored set is initially empty. When a path is selected, if the node at the end is already in the closed list then we can discard the path. Otherwise, we add the node at the end to the closed list and the algorithm proceeds.

This does not guarantee that we don't discard the least cost path. Something more sophisticated would need to be done in order to ensure the optimal solution is found. To ensure that the search algorithm can still find a lowest-cost path to a goal, we can do one of the following:

1. Make sure that the first path found to any node is a lowest-code path to that node, then prune all subsequent paths found.
2. If the algorithm finds a lower-cost path, remove all paths that used the higher-cost path to the node
3. Whenever the search finds a lower-cost path to a node than a path to that node already found, it could incorporate a new initial section on the paths that have extended the initial path.

Multiple-Path Pruning & A^*

- Suppose path p' to n' was selected, but there is a lower-cost path to n'. Suppose this lower-cost path is via path p on the frontier
- Suppose path p ends at node n
- p' was selected before p (i.e. $f(p') \leq (p)$), so: $cost(p') + h(p') \leq cost(p) + h(p)$
- Suppose $cost(n, n')$ is the actual cost of a path from n to n'. The path to n' via p is lower cost than via p' so: $cost(p) + cost(n, n') < cost(p')$
- From these equations: $cost(n, n') < cost(p') - cost(p) \leq h(p) - h(p') = h(n) - h(n')$
- We can ensure this doesn't occur if $|h(n) - h(n')| \leq cost(n, n')$.

Problem: What if a subsequent path to n is shorter than the first path to n?

1. Ensure this doesn't happen by making sure that the shortest path to a node found first
2. Remove all paths from the frontier that use the longer path

A* does not guarantee that when a path to a node is selected for the first time it is the lowest cost path to that node. Note that the admissibility theorem guarantees this for every path to a goal node but not for every path to any node. Whether it holds for all nodes depends on the properties of the heuristic function.

Consistent Heuristics

A non-negative function h(n) on node n that satisfies the constraint

$h(n) \leq cost(n, n') + h(n')$ for any two nodes n' and n, where $cost(n, n')$ is the cost of the least-cost path from n to n'.

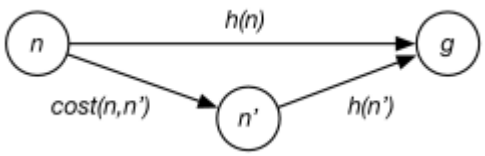
We can guarantee that a heuristic is consistent if it satisfies the monotone restriction

$h(n) \leq cost(n, n') + h(n')$ for any arc $\langle n,n' \rangle$

It is easier to check the monotone restriction as it only depends on the arcs, whereas consistency depends on all pairs of nodes.

Monotone Restriction

- Heuristic function h satisfies the monotone restriction if h(m) - h(n) <= cost(m,n) for every arc <m,n>
- If h satisfies the monotone restriction it is consistent meaning h(n) <= cost(n,n') + h(n') for any two nodes n and n'
- A* with a consistent heuristic and multiple path pruning always finds the shortest path to a goal
- This is a strengthening of the admissibility criterion



Consistency and the monotone restriction can be understood in terms of the triangle inequality which specifies that the length of any side of a triangle cannot be greater than the sum of lengths of the other two sides.

With a consistent heuristic, multiple-path pruning can never prevent A search from finding an optimal solution.*

This can be proved using proof by contradiction (see end of 3.7.1)

Type equation here.

A* **search** in practice includes multiple-path pruning; if A* is used without multiple-path pruning, the lack of pruning should be made explicit. It is up to the designer of a heuristic function to ensure that the heuristic is consistent, and so an optimal path will be found.

Multiple-path pruning is **preferred over cycle pruning for breadth-first methods** where virtually all of the nodes considered have to be stored anyway.

Depth-first search does not have to store all of the nodes at the end of paths already expanded; storing them in order to implement multiple-path pruning makes depth-first search exponential in space. For this reason, **cycle pruning is preferred over multiple-path pruning for depth-first search**.

Sophisticated Search

04 November 2020 21:19

We can refine the aforementioned strategies using some

Depth-First Brand-And-Bound search

- Comines DFS with heuristic information
- Finds optimal solution most useful when there are multiple solutions and we want an optimal one
- Use the space of a DFS

- Suppose bound is the cost of the lowest cost path found to a goal so far
- What if the search encounters a path p such that the cost of p plus the heuristic applied to p is greater than the bound - This means we can prune path p
- If we find a non-pruned path to the goal then we can set the bound equal to the cost of that path and then remember this as the best solution
- We should use DFS for linear space use
- We can guarantee an optimal solution with this algorithm

How do we initialise the bound? We can start it at infinity, or if we have an estimate we can use that

Notes on DFS BNB

- Cycle pruning works well with DFS BNB
- Multiple path pruning is not appropriate as storing explored set defeats space saving of dfs
- Can be combined with iterative deepening to increase the bound until either a solution is found or to show there is no solution

Context: Bounded Depth First Search

A bounded depth-first search takes a bound and does not expand paths that exceed the bound
Explores part of the search graph
Uses space linear in the depth of the search

The bound has to be the same or greater than the cost of getting to an optimal goal.
If we don't know that then we can do an iterative deepening search

Context: Iterative Deepening Search

1. Starts with a bound b=0
2. Do a bounded DFS with bound b
3. If a solution is found return that solution
4. Otherwise increment b and repeat

- This will find the same first solution as BFS
- Since using a depth-first search iterative deepening uses linear space
- Iterative Deepening has an asymptotic overhead of (b/(b-1)) times the cost of expanding the nodes at depth k using a BFS
- When b = 2 there is an overhead factor of 2, when b = 3 there is an overhead of 1.5 and as b get higher the overhead factor reduces

Direction of Search

Bidirectional Search

- Search backward from the goal and forward from the start simultaneously
- This is effective since 2b^(k/2) < b^k and so can result in an exponential time saving
- The main problem is ensuring that the frontiers meet
- This is often used with one breadth-first method that builds a set of locations that can lead to the goal and in the other direction another method can be used to find a path to these interesting locations

Island Driven Search

- Find a set of islands between s and g
- There are m smaller problems rather than 1 big problem
- This can be effective since mb^(k/m) < b^k
- The problem is to identify the islands that the path must pass through it is difficult to guarantee optimality

Dynamic Programming

- For statically stored graphs build a table of dist(n) the actual distance of the shortest path from node n to a goal
- This can be built backwards from the goal
- This can be used locally to determine what to do
- There are two main problems
- It requires enough space to store the graph
- The dist function needs to be recomputed for each goal

Constraint Satisfaction Problems

31 October 2020 15:06

Instead of reasoning explicitly in terms of states, it is typically better to describe states in terms of **features** and to reason in terms of these features. Features are described using **variables**. Often features are not independent and there are **hard constraints** that specify legal combinations of assignments of values to variables

When the goal state is defined in terms of restraints and you're looking for a solution that satisfies the constraints

- Types of CSP's
- Backtracking Search
- Arc-consistency in a constraint graph
- Domain splitting to solve
- Using CSP Problem structure
- Variable elimination for CSP

What is a constraint satisfaction problem?

- A CSP is characterized by a set of variables
- Each variable has an associated domain of possible values
- There are hard constraints on various subsets of the variables which specify legal combinations of values for the variables
- A solution to the CSP is an assignment of a value to each variable that satisfies all the constraints

CSP's as optimisation problems

- For optimisation problems there is a function that gives a cost for each assignment of a value to each variable
- A solution is an assignment of values to the variables that minimizes the cost function

Types of Variables

1. Discrete Variable – domain is finite or countably infinite
2. Binary Variable – a discrete variable with two values in its domain e.g. Boolean variable
3. Continuous Variable – not discrete e.g. a variable that corresponds to a real line

Given these variables, an assignment on the set of variable is a function from the variables into the domains of the variables:

$\{X1, X2, \dots, Xk\}$ as $\{X1 = v1, X2 = v2, \dots, Xk = vk\}$, where vi is in $dom(X_i)$

A possible world is a complete assignment of variables. That is, a function from variables into values that assigns a value to every variable

We use variables because we can reason about many worlds with just a few variables

Types of Constraints

In many domains , not all possible assignments of values to variables are permissible
A constraint specifies legal combinations of assignments of values to some of the variables.

A scope is a set of variables

A relation on a scope is a function that returns true or false to an assignment on a scope
A constraint is a scope and a relation on S. It involves each of the variables in its scope

A model is a possible world that satisfies all of the constraint

Constraints are defined by their **intension** in terms of formulas or by their **extension**, listing all the assignments that are true. Constraints defined extensionally can be seen as relations of legal assignments as in relational databases.

- Unary constraints – involve a single variable e.g. some variable can't be green or > 10
- Binary constraints – involve pairs of variables that can't be equal
- Higher-order constraints – involve 3 or more variables
- Preference or soft constraints – e.g. 1005 is better than 0805

Constraint Satisfaction Problems

- A set of variables
- A domain for each variable
- A set of constraints

A finite CSP has a finite set of variables and a finite domain for each variable

Given a CSP, we can do some useful tasks:

- Determine or not whether there is a model
- Find a model
- Count the number of models
- Enumerate all the models
- Find the best model given a measure of how good models are
- Determine whether some statement holds in all models

Real World CSP's

- Assignment problems
- Timetabling
- Hardware configuration
- Spreadsheets
- Floor planning

Solving CSP's

- A CSP can be solved by graph-searching
- A node is an assignment of values to some of the variables
- The start node is the empty assignment
- A goal node is a total assignment that satisfies the constraints

Generate-and-Test Algorithms

A finite CSP could be solved by an exhaustive algorithm. The assignment space D is the set of total assignments. The generate and test algorithm can be run to just return the first variable found.

Generate the assignment space I.e. the set of total assignments
Test each assignment with constraints

$$\begin{aligned}\mathbf{D} &= \mathbf{D}_A \times \mathbf{D}_B \times \mathbf{D}_C \times \mathbf{D}_D \times \mathbf{D}_E \\ &= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \\ &\quad \times \{1,2,3,4\} \times \{1,2,3,4\} \\ &= \{\langle 1,1,1,1,1\rangle, \langle 1,1,1,1,2\rangle, \ldots, \langle 4,4,4,4,4\rangle\}.\end{aligned}$$

How many assignments need to be tested for n variables, each with a domain of d? D to the power of n.

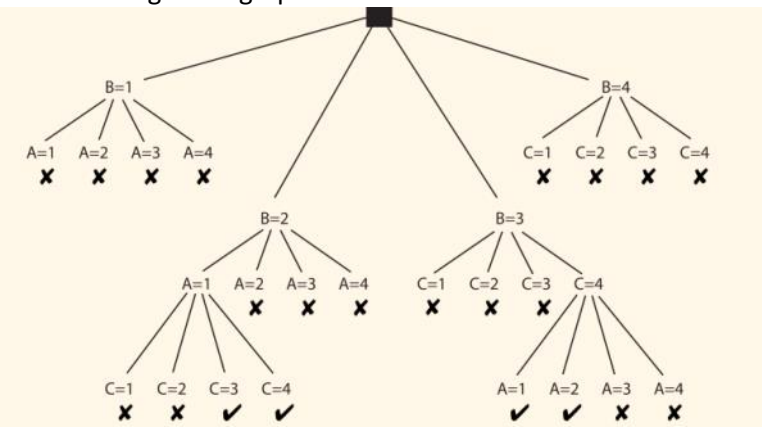
Algorithms for CSP's are trying to cut down this assignment space as it is so large.

Backtracking Algorithms

Generate-and-Test algorithms assign values to all variables before checking constraints. Because individual constraints only involve a subset of the variables, some constraints can be tested before all of the variables have been assigned values. If a partial assignment is inconsistent with the constraint then any total assignment that extends the partial assignment will also be inconsistent.

We can use backtracking instead, where we construct a search space that can be explored by the search algorithms discussed previously.

The following is the graph when A < B and B < C



1. Systematically explore D by instantiating the variables one at a time
2. Evaluate each constraint predicate as soon as all its variables are bound
3. Any partial assignment that does not satisfy the constraint can be pruned.

- Every solution appears at depth n so we can use a DFS
- Path is irrelevant so can use complete state formulation
- Branching factor b = (n-l)d at depth l hence n! x d^n leaves – top level branching factor is nd since any of d values can be assigned to any of n variables' next level branching factor si s(n-1) d and so on
- Variable assignments are commutative e.g. x=5, y=10 === y=10, x=5
- Only need to consider assignments to a single variable at each node
- Backtracking search si the basic uninformed algorithm for CSP's
- Can solve n-queens for n = 25

This is much more efficient that a generate-and-test algorithm because there we only test the constraints at the leaf nodes, whereas in a backtracking algorithm we prune all nodes immediately that violate a constraint, leaving us only with valid avenues to a possible model.

Backtracking Search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING(assignment,csp) returns solution/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment given CONSTRAINTS[csp] then
      add {var = value} to assignment
      result ← RECURSIVE-BACKTRACKING(assignment,csp)
      if result ≠ failure then return result
      remove {var = value} from assignment
  return failure
```

Improving Search Efficiency

1. Which variable should be assigned next? MRV & degree heuristic
2. What order should we try its values? LCV
3. Can we detect inevitable failure early? Consistency Algorithms
4. Can we take advantage of the problem structure? Custset conditioning & variable elimination

Minimum remaining values (MRV)

Choose the variable with the fewest legal values

- Also called fail-first heuristic: will pick variable most likely to cause failure – if exists a variable with 0 possible assignments will pick and fail immediately

Degree Heuristic

Tie breaker among MRV variables

- Choose the variable with the most constraints on remaining variables
- Attempts to reduce branching factor of future choices

Least Constraining Value LCV

- Given a variable, choose the least constraining value: the one that rules out the fewest values in the remaining variables

Combining these we can solve n queens for n = 1000

Consistency Algorithms

Although DFS over the search space of assignments is usually a substantial improvement over generate and test, it still has various inefficiencies that can be overcome.

Idea: prune the domains as much as possible before selecting values from them

- A variable is domain consistent if no value of the domain of the node is ruled impossible by any of the constraints
- Example: if domain is Domain(B) = {1,2,3,4} and we have a rule that B can't equal 3, then D is not domain consistent

Constraint Network

1. There is a circular node for each variable

2. There is a rectangular node for each constraint
3. There is a domain of values associated with each variable node
4. There is an arc from variable X to each constraint that involves X

Arc Consistency

If there is a constraint that acts on some variables X, Y, Z, then the arc <X, c> can be called arc consistent if for each value of x in the domain(X) there are some values y and z in the domain(Y) and domain(Z) such that if we set X = x, Y = y, Z = z, the constraint is satisfied.

A network is arc-consistent if every arc is arc-consistent

What if an arc is not consistent?

If the arc <X, c> is not consistent then there are some values of X for which there are no values for Y, Z for which the constraint holds. In this case, all values of X in the domain(X) for which there are no corresponding values for the other variables can be deleted from the domain(X) to make the arc consistent.

When a value is removed from the domain of a variable it is possible that it will make some other arcs that were previously consistent, no longer consistent.

Arc Consistency Algorithm

See figure 4.3 in textbook

The arcs can be considered in turn making each arc consistent
When an arc has been made arc consistent, does it ever need to be checked again? YES – an arc needs to be revisited if the domain of one of following variables is reduced

Three possible outcomes when all arcs are consistent

- One domain is empty – no solution
- Each domain has a single value – unique solution
- Some domains have more than one value – there may or may not be a solution

Domain Splitting

Another method for simplifying the network is domain splitting, or case analysis
The idea is to split the problem into a number of disjoint cases and solve each case separately. The set of all solutions to the initial problem is the union of the solutions to each case.

In the simplest case, imagine we have a variable X with a domain of {t,f}. All solutions either have X = t, or X = f. One way to find the solutions is to set X = t, find all of the solutions with this assignment, then assign X = f, and find all those solutions.

If we make the problem into a smaller set of subproblems, we can get massive improvements in efficiency:

- Suppose each subproblem has c of n total variables
- There are n/c subproblems each of which takes at most d^c to solve. Worse case solution is therefore n/c x d^c, I.e linear in n

If the variable has > 2 elements, we can split it in a number of ways.

- Split it into a case for each value
- Always split the domain down the middle into 2 disjoint sets

We can be more efficient by interleaving arc consistency with the search
We would solve a CSP by using arc consistency to simplify the network before each step of domain splitting

After domain splitting, we do not need to start arc consistency from scratch. We can simply check the arcs that are possibly no longer arc consistent as a result of the split.

Complexity of Generalized Arc Consistency Algorithm

- If there are c binary constraints, and the domain of each variable is of size d. There are 2c arcs.
- Checking an arc <X, r(X,Y)> involves in the worst case iterating through each value in the domain of Y for each value in the domain of X, which takes d^2 time.
- This arc may need to be checked once for every element in the domain of Y, thus GAC for binary variables can be done in time O(cd^3) which is linear in C – the number of constraints
- The space used is O(nd) where n is the number of variables and d is the domain size.

Hard and Soft Constraints

Given a set of variables, assign a value to each variable that either:

- Satisfies some set of constraints – satisfiability problems – hard constraints
- Minimizes some cost function where each assignment of values to variables has some cost – optimisation problems – soft constraint
- Many problems are a mix of hard and soft constraints

Tree-Structured CSP's

Theorem: If the constraint graph has no loops, the CSP can be solved in O(nd^2)
Compare this to general CSP's, where worst case time is O(d^n)
This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.

Algorithm

Choose variable as root, order variables from root to leaves such that ever nodes parent precedes it in the ordering
For j from n down to 2, remove inconsistent domain elements for the parent.
For j from 1 to n, assign X consistently with Parent

Nearly Tree-Structured CSP's

- Conditioning – instantiate a variable, prune its neighbours domains, I.e. assign variable so remained is a tree.
- Cutset conditioning: instantiate a set of variables such that the remaining constraint graph is a tree
- Cutset size c -> runtime O(d^c x (n-c)d^2) very fast for small c.

Variable Elimination

Arc consistency simplifies the network by removing values of variables. A complementary method is variable elimination which simplifies the network by removing variables.

The idea is to remove the variables one by one. When removing a variable X, VE constructs a new constraint on some of the remaining variables, reflecting the effects of X on all the other variables.

This new constraint replaces all of the constraints that involve X, forming a reduced network that does not involve X.

When we eliminate X, the influence of X on the remaining variables is through the constraint relations that involve X. First, the algorithm collects all of the constraints that involve X.

Variable Elimination Algorithm

If there is only one variable return the intersection of the unary constraints that contain it

- Select a variable x
- Join the constraints in which X appears, forming constraint R1
- Project R1 onto its variables other than X, forming R2
- Replace all of the constraints in which X appears by R2
- Recursively solve the simplified problem, forming R3
- Return R1 joined with R3

- When there is a single variable remaining, if it has no values, the network was inconsistent
- The variables are eliminated according to some elimination ordering
- Different elimination orderings result in different size intermediate constraints

Figure 4.6 gives a recursive algorithm for variable elimination to find all the solutions for a CSP.

The number of variables n the largest relation returned for a particular variable ordering is called the **treewidth of the graph for that variable ordering**. The **treewidth of a graph is the minimum treewidth for an ordering**.

The complexity of VE is exponential in treewidth and linear in the number of variables.

There are some heuristics that exist that can help us get the minimum treewidth:

1. Min-factor – at each stage, select the variable that results in the smallest relation
2. Minimum deficiency or minimum fill – at each stage, select the variable that adds the fewest arcs to the remaining constraint network. The intuition is that it is okay to remove a variable that results in a large relation as long as it does not make the network more complicated.

Example: eliminating C

$r_1 : C \neq E$

C	E
3	2
3	4
4	2
4	3

$r_2 : D < C$

C	D
3	2
4	2
4	3

$r_3 : r_1 \bowtie r2$

C	D	E
3	2	2
3	2	4
4	2	2
4	2	3
4	3	2
4	3	3

$r_4 : \pi_{\{D,E\}} r_3$

D	E
2	2
2	3
2	4
3	2
3	3

↪ new constraint

NB: $r_1 \bowtie r_2$ = join of r_1 and r_2, $\pi_S(r)$ = projection of r onto S.

Local Search

05 November 2020 13:11

1. Hill climbing
2. Greedy descent
3. Randomized algorithms
4. Simulated annealing
5. Genetic algorithms

Iterative Improvement Algorithms

Cases where we don't care about the path – we just want a goal state

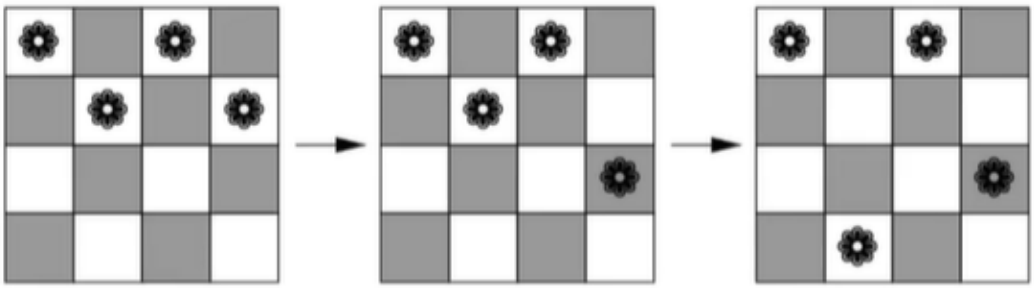
Systematically searching large spaces is not practical in many cases. We can use these iterative improvement algorithms to deal with these large search spaces.

The methods find solutions quickly on average, but do not guarantee that a solution will be found, even if one exists – they are therefore not able to prove a solution exists. They are useful when we already know a solution exists or is very likely to exist.

Local search methods start with a total assignment of a value to each variable and try to improve this assignment iteratively by taking improving steps, by taking random steps, or by restarting with another total assignment.

- Uses a single current state (not multiple paths) and typically moves to neighbours of state
- Not systematic, but low memory usage and can find solutions in continuous spaces
- Useful for optimisation problems including CSP's

The state space is the set of all configurations. The goal is a particular configuration
e.g. the travelling salesman or n-queens

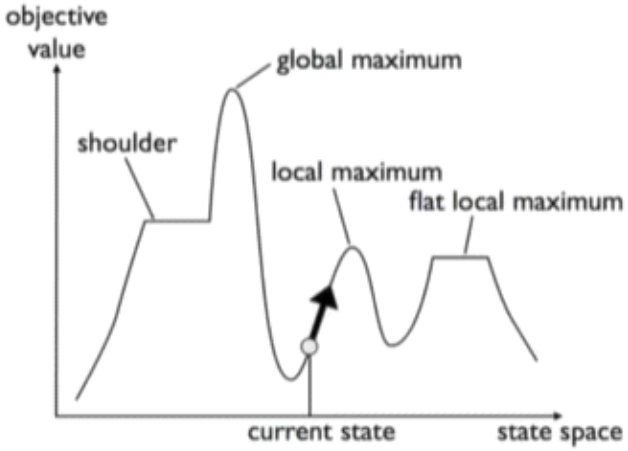


we can use iterative improvement algorithms on these problems, by keeping the current state and trying to improve it.

These algorithms often have constant space

Hill-Climbing

- Greedy local search – always tries to improve the current state, or reduce the cost if evaluation function is cost
- Each iteration moves in the direction of increasing value, or decreasing if evaluation is cost
- No search tree – just keep current state and its cost
- If > 1 alternative with equal cost, choose at random



Problems include local maxima where we have a local peak that is lower than the highest peak, but the algorithm will halt with a suboptimal solution

Also ridges, plateaux – flat area will conduct a random walk

Greedy Descent

Maintain an assignment of values for each variable
Repeatedly select a variable to change and a value for that variable

- Aim is to find an assignment with zero unsatisfied constraints
- Given an assignment of a value to each variable, a conflict is a violated constraint
- The goal is an assignment with zero conflicts
- Heuristic function to be minimized the number of conflicts

To choose a variable to change and add a new value:

- Find a variable-value pair that minimizes the number of conflicts
- Select a variable that participates in the most conflicts
- Select a value that minimizes the number of conflicts
- Select a variable that appeaars in any conflict
- Select a value that minimizes the number of conflicts
- Select a variable at random
- Select a value that minimizes the number of conflicts
- Select a variable and value at random, accept this change if it does not increase the number of conflicts

Complex Domains

- When the domains are small or unordered, the neighbours of an assignment can correspond to choosing another value for one of the variables
- When the domains are large and ordered, the neighbours of an assignment are the adjacent values for one of the variables
- If the domains are continuous, gradient descent changes each variable proportionally to the gradient of the heuristic function in that direction

The value of variable X_i goes from v_i to $v_i - \eta \frac{\partial h}{\partial X_i}$ where η is the step size.

Randomized Greedy Descent

As well as downward steps, we can allow for:

- Random steps: move to a random neighbour
- Random restart: reassign random values to all variables

Stochastic Local Search

Combination of:
Greedy descent – move to a lowest neighbour
Random walk
Random restart

Random Walk

Randomly sometimes choose a random variable value pair
When selecting a variable then a value:

- Sometimes choose any variable that participates in the most conflicts
- Sometimes choose any variable that participates in any conflict
- Sometimes choose any variable

Sometimes choose the best value and sometimes choose a random value

Simulated Annealing

1. Pick a variable at random and new value at random
2. If it is an improvement, adopt it
3. If it is not an improvement, adopt it probabilistically depending on a temperature parameter, T which is reduced over time

e.g. if T = 10 then there is a big chance we'll accept the proposed random change, but if T = 0.1 it is very small

- To prevent cycling we can maintain a tabu list of the last k assignments
- Do not allow an assignment that is already on the tabu list
- If k = 1, we do not allow an assignment of the same value to the variable chosen
- Can we very expensive if k is large

Simulated annealing requires an annealing schedule which specifies how T is reduced as the search progresss. Geometric cooling one of the most widely used schedules.

Parallel Search

A total assignment is called an individual
Idea: maintain a population of individuals instead of one
At every stage, update each individual in the population
Whenever an individual is a solution, it can be reported
Like k restarts but uses k times the minimum number of steps

Beam Search

Like parallel search, with k individuals but choose the k best out of all the neighbours
When k = 1, it is a greedy descent
Whe k = infinity it is a BFS
The value of k lets us limit space and parallelism

This can be made into a stochastic beam search is we probabilistically choose the k individuals at the next generation

The probability that a neighbour is chosen is proportional to its heuristic value
This maintains diversity among individuals

The heuristic value reflects the fitness of the individual

Like **asexual reproduction**: each individual mutates and the fittest ones survive

Genetic Algorithms

- Related to stochastic beam search
- Successor states obtained from two parents
- Start with a population of k individuals
- Each individual represented as a string over a finite alphabet, e.g. string of 0's and 1's. N-

queens would be [1 5 6 3 4 2 3 8] for one potential layout, for example.

Fitness Function

- Each individual in the population is evaluated by a fitness function
- Fitness function should return higher vlaues for better states
- Fitness function determines probability of being chosen for reproduction
- Pairs of individuals chosen according to these probabilities – those below a threshold can be culled

Crossover

For each chosen pair, a random crossover point is chosen from the string representation
Offspring are generated by crossing the parents strings at a chosen point
First child gets first part of string from 1 and second from 2, and second gets the opposite

We then subject the new individuals to mutation

Adversarial Search

05 November 2020 15:22

- Adversarial Search
- Minimax
- Alpha-Beta Pruning
- Imperfect Decisions
- Games with Chances

A competitive multi-agent environment where goals are in conflict – gives rise to games

Other agents – opponents – which introduce uncertainty

- An adversarial search agent must deal with contingency
- High complexity and time sensitive – typically have to make a best guess based on experience and time available
- Chess has a branching factor of 35 and a game is 100 moves – thus we have 35^100 nodes – we have to make the best move given the situation

Uncertainty

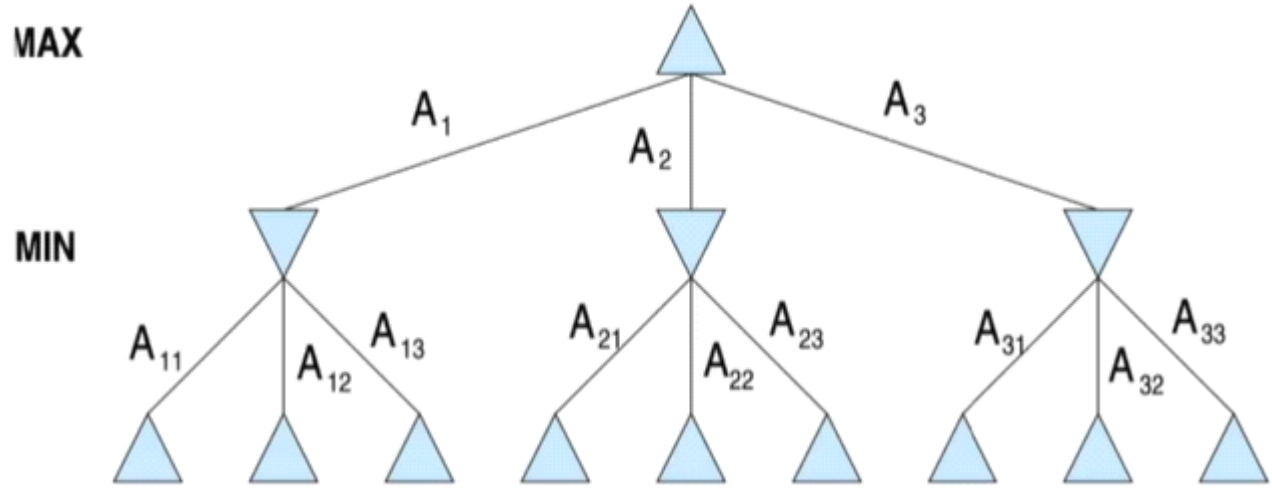
- From the opponent trying to make the best move for themselves
- Randomness – e.g. throwing a dice
- Insufficient time to determine consequences

Formal view of a game
Initial state – the board position and player to move
Set of operators defining legal moves and resulting states
A terminal test to determine a terminal state
A utility function or payoff function – gives a numeric value for terminal states e.g. +1, -1, 0 for chess win lose draw

We can build a game tree based on initial state and operators for each player

Ply

A single move in a 2 player game is takes 2 half-moves, or is "2 ply"



Minimax

In the case where two agents are competing so that a positive reward for one is a negative reward for the other, we have a two-agent zero-sum game. The value of such a game can be characterized by a single number that one agent is trying to maximise and the other agent is trying to minimize. Having a single value fora two-agent zero sum game leads to a minimax strategy. Each node is either a max node, if it is controlled by the agent trying to maximise, or is a min node if it is controlled by the agent trying to minimize.

- Gives and optimal strategy for max
- Choose move with the highest minimax value

The minimax value of a state is the utility (for Max) of being in that state assuming both players play optimally from that state until the end of game

Obtains best achievable payoff against best play

Algorithm

Generate a complete game tree
Use utility function to rate terminal states
Use utility of terminal states to give utility of nodes one level up
Continue backing up tree until reaching the root
Max should choose move that leads to highest utility

This is the minimax decision:

Maximises utility under the assumption that the opponent will play to minimise it

- Complete if tree is finite
- Optimal against an optimal opponent
- Space - O(bd) because DFS
- Time – O(b^d) - a killer for real games
- Minimax requires a complete search tree which is not practical
- Forms basis for realistic algorithms

Multi-player Minimax

Can extend minimax to multiple players by using vectors of utilities

Alpha-Beta Pruning

Complete search tree is impractical – alternative is to prune branches that will not influence decision

- Consider a node n that might be chosen
- If a better choice exists, then n will never be reached so prune it
- As soon as we discover a better choice than n, prune it

Minimax is a DFS, and ABP gets its name from the parameters backed up the path

- Alpha = value of best choice along the path for Max (highest utility)
- Beta = value of best choice for min (lowest utility for Max)

ABP updates alpha and beta as it searches, pruning as soon as value of current node is known to be worse than the current alpha or beta for max or min respectively
Pruning is done by terminating the recursive call

Order of Examining Successors

- The effectiveness of ABP is dependent on order of examining successors
- So, we should try to examine the best successors first
- If we could do this, the ABP looks at O(b^(d/2)) instead of O(b^d) for minimax – roughly twice the lookahead
- For random order successors, APB looks at O(b^(3d/4) nodes
-

In practice, a simple ordering function can give significant advantage
e.g for chess we might look at capture moves first then threats, then forward moves, then backward moves

Imperfect Decisions

APB prunes much of the search tree, while minimax needs the complete tree
But APB still needs to search to the terminal states for some of the tree which is still impractical

Alternative: cut off the tree earlier, using:

- A heuristic evaluation function to get a value for states
- A cut-off test to determine when to stop going down the tree

Evaluation function gives an estimate for expected utility for a given position

Cutting off tree turns nonterminal nodes into terminal leaves.

Eval function should:

- Order terminal states as per utility function
- Approximate actual utility state

Uncertainty is unavoidable since we are not considering a complete tree

Most evaluation functions calculate features of a state e.g. number of each piece
These give equivalence classes for states which will lead to a win, draw or less with some probability
Combine features with a weighted linear function
Assumes features are independent

Simplest approach is to set a fixed depth – cut-off test succeeds at depth d

More robust approach is to use iterative deepening: continue until out of time then return best move found so far

Cutting Off Search

Simplest approach is to set a fixed depth – cut-off test succeeds at depth d
More robust approach is to use an iterative deepening, continue until out of time, then return the best move found so far
Both of these approaches are unreliable

Solution: only apply eval function to quiescent positions – those whose value is unlikely to change significantly in the near future

- Non-quiescent positions expanded until quiescent positions reached
- This extra search is called quiescent search
- Quiescent search restricted to certain types of more to quickly resolve uncertainties in position
- Horizon problem: faced with unavoidable damaging move from opponent, a fixed depth search is fooled into viewing stalling moves as avoidance

Singular extension search as a means of avoiding horizon problem
Singular extension is a move that is clearly better than all others

In chess, can search to see whether opponent can advance pawn to 8th row, turning it into a queen

Forward pruning: immediately prune some moves from a node with no further considerations

- Only safe in special cases like when the two moves are symmetric or equivalent and only consider on of them, or nodes are very deep in search tree

Games with Chance

- Many games contain chance
- Legal moves are dependent on roll of dice, so cannot construct a complete game tree
- Have to include chance nodes to solve this
- These nodes can be labelled with the possibility of the expected value taken over the chance nodes

Knowledge Bases

21 December 2020 14:26

Knowledge Bases

Knowledge Base – a database for the system that contains all the facts and beliefs that the system knows. It is a representation of the systems idea of the world
Inference engine – domain-independent algorithms – a mechanism for reasoning about those beliefs

The knowledge is *domain specific* but the inference engine is *domain independent*

Knowledge Bases = **sentences in a formal knowledge representation language** - but implementation could be anything – linked lists, arrays, databases etc

1. A declarative approach to building an agent – we tell is what it needs to know – it can then ask itself what to do. Answers should follow from the KB through inference
2. **TELL** and **ASK** are standard names for adding sentences and querying KB
3. Result of ASK must follow from previous TELLS as determined by inference mechanism

Each time the agent program is called, it firstly TELLS the knowledge base what it perceives, and then ASKS the knowledge base what action it should perform

We have to build agents that can take a knowledge base and use some inference mechanism to perform actions – **planning is about building that inference engine**

Knowledge Based Agents

- Can reason using inference and their knowledge.
- Can accept new tasks in the form of goals
- Can adapt to environmental change by updating knowledge
- Are able to infer unseen properties of the world from perceptions
- Can often find better solutions than simple search

Moreover, significantly more flexible with respect to **adopting new goals**, **partially observable environments, dynamic environments**

Characterising KBA's

1. **Knowledge Level** – what is known? Allows us to work at an abstract level of ASK and TELL (AKA epistemological level) e.g. a taxi might know that the Golden Gate Bridge links San Francisco to Marin County
2. **Logical Level** – knowledge encoded in formal sentences e.g. links(GGB, SF, M)
3. **Implementation Level** – data structures in KB and algorithms that manipulate them e.g. this could be a 1 in a 2D array of places, where 1 means Links X to Y

Simple KBA

The agent must be able to:

- Represent states and actions
- Incorporate new percept's
- Update internal representations of the world
- Deduce hidden properties of the world

- Deduce appropriate actions

Algorithm:

```
function KB-AGENT(percept) returns an action
  static : KB, a knowledge base
    t, a time counter, initially 0
  TELL(KB,MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB,MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

- Knowledge base may contain initial background knowledge
- Each iteration, TELL KB of perceptions, ASK What actions to perform
- Note: TELL and ASK refer to KB – they are internal
- Representation details hidden by MAKE-PERCEPT-SENTENCE and MAKE-ACTION-QUERY allow us to work at knowledge level
- Inference details hidden in TELL and ASK

Wumpus World

- Managed to get the goal because we can make inferences about knowledge gained from perceptions
- Combining knowledge obtained at different times and in different places allows us to infer more about the world
- Using lack of a particular perception rather than just the existence of a perception allows us to extract more knowledge from the world
- We rely on persistence of knowledge – the world is not fully observable

First-Order Logic

22 December 2020 14:29

A logic that is sufficient for building Knowledge Based Agents

Before, we've used propositional logic as our representation language because it is one of the simplest languages that demonstrates all the important points. Unfortunately, propositional logic has a very limited ontology, making only the commitment that the world consists of facts. This makes it difficult to represent even something simple

FOL or First-Order Predicate Calculus makes a stronger set of ontological commitments. The main one is that the world consists of objects, that is things with individual identities and properties that distinguish them from other objects.

Among these objects, various relations hold. Some of these relations are functions - relations for which there is only one value for a given input. It is easy to start listing examples of objects, properties, relations and functions

Objects - people, houses, numbers, theories, colours, baseball games, wars
Relations - brother of, bigger than, inside of, has colour, occurred after, owns
Properties - red, round, prime
Functions - father of, best friend, third inning of, one more than

FOL makes no commitments to time, categories and events. A logic that tried to, would only have limited appeal as there are so many different ways of interpreting them. Thus, FOL remains neutral and gives us the freedom to describe these things in a way that is appropriate for the domain. Freedom of choice is a general characteristic of FOL

Syntax and Semantics

In propositional logic, every expression is a sentence, which represents a fact. First-Order Logic has sentences, but it also has terms, which represent objects.
Terms are built from constant symbols, variables, and function symbols.

FOL BNF

Sentence -> AtomicSentence | Sentence Connective Sentence | Quantifier Variable, ... Sentence | ! Sentence | (Sentence)
AtomicSentence => Predicate(Term,...) | Term = Term
Term -> Function(term, ...) | Constant | Variable
Connective -> => | /\ | \/ | <=>
Quantifier -> For All | For Some
Constant -> A | John
Predicate -> Before | HasColor | Raining
Function -> Mother | LeftLegOf

Constant

Which object in the world is referred to by each constant symbol? Each constant symbol names exactly one object, nbut not all objects need to have names, and some can have several names. Thus, the symbol john, in one particular interpretation might refer to a specific king, but the symbol king could refer

Predicate Symbols

An interpretation specifies that a predicate symbols refers to a particular relation in the model

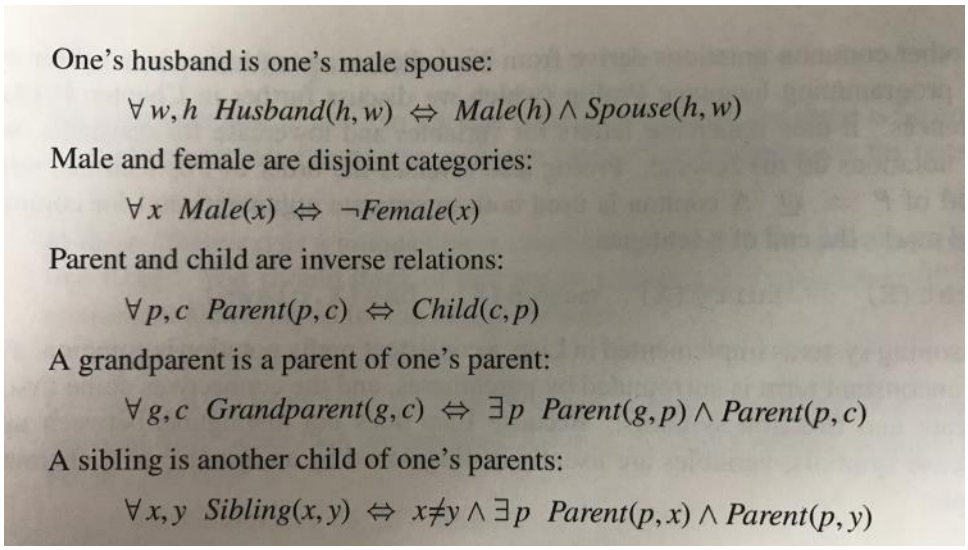
Brother might refer to the relation of brotherhood
A relation is defined by the set of tuples of objects that satisfy it
Thus, a predicate symbol is a symbol that points to this list of satisfying tuples

Function Symbols

- Some relations are functional - that is, any given object is related to exactly one other object by the relation.
- For example, any angle has only one number that is its cosine, and person has only oner person that is his or her father.
- In such cases, it is often more convenient to define a function symbol e.g. cosine that refers to the appropriate relation between angles and numbers.
- In the model, the mapping is just a set of n+1 tuples with a special property, namely that the last element of each tuple is the value of the function for the first n elements.
- A table of cosines is an example of this set of tuples.
- Unlike predicate symbols which hare used to state that relations hold among certain objects, function symbols are used to refer to particular objects without using their names.

Using First-Order Logic

In knowledge representation, a domain is a section of the world about which we wish to express some knowledge


One's husband is one's male spouse:

$$\forall w,h \;\; Husband(h,w) \Leftrightarrow Male(h) \wedge Spouse(h,w)$$

Male and female are disjoint categories:

$$\forall x \;\; Male(x) \Leftrightarrow \neg Female(x)$$

Parent and child are inverse relations:

$$\forall p,c \;\; Parent(p,c) \Leftrightarrow Child(c,p)$$

A grandparent is a parent of one's parent:

$$\forall g,c \;\; Grandparent(g,c) \Leftrightarrow \exists p \;\; Parent(g,p) \wedge Parent(p,c)$$

A sibling is another child of one's parents:

$$\forall x,y \;\; Sibling(x,y) \Leftrightarrow x \neq y \wedge \exists p \;\; Parent(p,x) \wedge Parent(p,y)$$

Representation, Reasoning & Logic

22 December 2020 12:08

The object of knowledge representation is to express knowledge in computer-tractable form, such that it can help the agent to perform well.

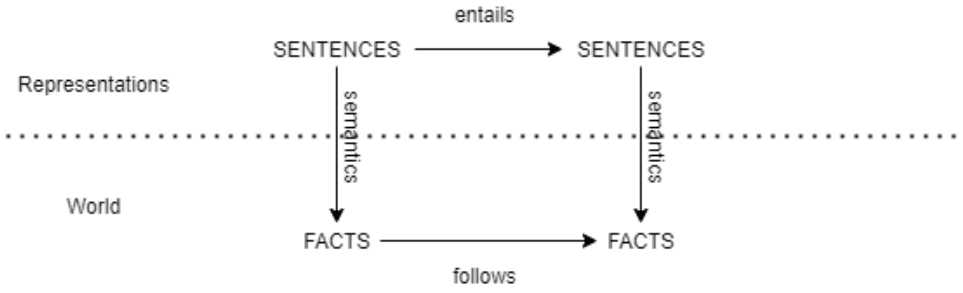
There are two aspects that are key in knowledge representation - syntax and semantics

- The syntax of a language describes the possible configurations that can constitute sentences. E.g. although X, Y and > are valid symbols in a language, X Y > might not be syntactically correct, whereas X > Y might be.
- Semantics determines the facts in the world to which the sentences refer. Without semantics a sentence is just an arrangement of electrons or a collection of marks on a page. With semantics, each statement makes a claim about the world. With semantics, we can say that when a particular configuration exists within an agent, the agent believes the corresponding sentence.

Provided we have a precisely syntax and semantics, we can call the language a logic.

From the syntax and semantics, we can derive an inference mechanism, that uses the logic.

- Facts are parts of the world
- Representations are encoded in some way that can be physically stored in an agent
- Because sentences are physical configurations of parts of the agent, reasoning must be a process of constructing new physical configurations from old ones. Proper reasoning should ensure that the new configurations represent facts that actually follow from the facts that the old configurations represent



An inference procedure can do two things:

1) Given a KB, generate new sentences that are entailed by KB
2) Given a KB and a new sentence alpha, decide whether or not KB entails alpha.

Entailment

We want to generate new sentences that are necessarily true, given that the old sentences are true. This relationship between sentences is called **entailment**, and mirrors the relations of facts following from each other.

KB |= alpha

Entailment is important since it provides a strong way of showing that if certain propositions are true, then some other proposition must be true

- KB entails sentence alpha if and only if alpha is true in all worlds where KB is true
- e.g. if the KB has "James is male" and "James is 34" then the KB entails "James is male or 34"
- Semantics give mapping of sentences to facts
- Logical inference generates sentences that are entailed by existing sentences and should ensure relationship mirrored in real world
- By considering the semantics of a language we can extract the proof theory of the language – what reasoning steps are sound

Inference procedures that generate only entailed sentences are sound or truth preserving

Inference

The term "inference" generally covers any processes by which conclusions can be reached. We are mainly concerned with sound reasoning, which is called "logical inference", or "deduction". Logical inference is a process that implements the entailment relation between sentences.

if i can derive alpha from KB, then we would write:

$$\mathrm{KB} \vdash_i \alpha$$

means that sentence "alpha can be derived from KB by inference procedure i", or "i derives alpha from KB"

- **Soundness**: i is sound if whenever KB |-i alpha, it is also true that KB |= alpha
- **Completeness**: i is complete if whenever KB |= alpha, it is also true that:
  - $\mathrm{KB} \vdash_i \alpha$

We need a logic which is expressive enough to say almost anything of interest and for which there exists a sound and complete inference procedure. That is, the procedure will answer any question whose answer follows from what is known by the KB

Logics

A formal system for describing states of affairs, consisting of

1) the syntax of the language, which describes how to make sentences
2) the semantics of the language, which states the systematic constraints on how sentences relate to states of affairs
3) The proof theory, a set of rules for deducing the entailments of a set of sentences.

there are two main types of logic - propositional and first-order logic

Propositional logic

Symbols represent whole propositions, or facts. We combine propositions with boolean connectives to generate sentences with more complex meanings.

- Negation - if S is a sentence, then not S is a sentence
- Conjunction - if S and B are sentences then S and B is a sentence
- Disjunction - if S and B are sentences then S or B is a sentence
- Implication - if S and B are sentences then S ==> B is a sentence
- Equivalence is S and B are sentences then S <=> B is a sentence

This creates a lot of problems - we need a lot of rules and logic to create a semi competent agent.

E.g. in Wumpus World, rule "Don't go forward if a wumpus is in front of you" requires 64 rules - 16 squares with 4 orientations

First-Order Logic

Represents the world in terms of objects and predicates on objects, e.g. properties of objects or relations between objects, as well as using connectives and quantifiers, which allow sentences to be written about everything in the universe at once.

Summary

We have introduced the idea of a KBA, and showed how we can define a logic with which the agent can reason about the world and be guaranteed to draw correct conclusions, given correct premises. We have also showed how an agent can turn this knowledge into action

- Intelligent agents need knowledge about the world in order to reach good decisions
- Knowledge is contained in agents in the form of sentences in a knowledge representation language, stored in a knowledge base
- A knowledge based agent is composed of a knowledge base and an inference mechanism. It operates by storing sentences about the world in its knowledge base, using the inference mechanism to infer new sentences, and using them to decide what action to take.
- A representation language is defined by its syntax and semantics, which specify the structure of sentences and how they relate to facts in the world
- The interpretation of a sentence is the fact to which it refers/ If it refers to a fact that is part of the actual world then it is true.
- Inference is the process of deriving new sentences from old ones/ We try to design sound inference processes that derive true conclusions given true premises. An inference process is complete if it can derive all true conclusions from a set of premises.
- A sentence that is true in all worlds under all interpretations is called valid. If an implication sentence can be shown to be valid, then we can derive its consequent if we know its premise. The ability to show validity independent of meaning is essential
- Different logics make different commitments about what the world is made of and what kinds of beliefs we can have regarding facts
- Logics are useful for the commitments they do not make, because the lack of commitment gives the knowledge base writer more freedom
- Propositional logic commits only to the existence o facts that may or may not be the case in the world being represented. It has a simple syntax and semantics, but suffices to illustrate the process of inference.
- Propositional logic can accomodate certain inferences needed by a logical agent, but quickly becomes impractical for even very small worlds.

Questions

What if we expand our knowledge base from our initial knowledge using entailment, but then the premises change. Does it invalidate all the entailed knowledge?

E.g. we instantiate our agent with some knowledge about the world in its KB, like "the earth is spherical", "pi = 3.141592", "g = 9.81", and from this knowledge the agent can deduce other things. What happens if we then discover that the earth is flat, or g = 10? This would invalidate all the new knowledge that the agent has.

What does it do? Does it keep a track of where it gained its knowledge from, and delete all knowledge that stems from the incorrect fact? This could get confusing if you deduce facts from your existing knowledge, then use those facts to deduce more information, and so on and so on, until you have a sprawling KB based on a faulty premise.

Planning can help us here.

Planning

18 December 2020 17:12

1. Knowledge bases
2. Reasoning using knowledge and inference
3. Search vs. Planning
4. Partial-order Planning
5. Conditional Planning
6. Monitoring and Replanning

Search Vs. Planning

In most situations, the branch factor of the search problem is so large that search becomes virtually useless.

General search problems..

1. In search, we must specify an initial state, operators, and optionally a heuristic function
2. Branching factor may be huge depending on how we specify these operators
3. Path length may be very long, and thus there are too many states to consider
4. The agent is forced to construct a full sequence of actions and must decide what to do in initial state first.

Difficulties with heuristics...

1. Heuristics can only choose which state is closer to a goal, but cannot eliminate actions from consideration
2. Evaluation function ranks these guesses, but must still consider them all
3. We need to work on the appropriate part of the sequence.

The main problem with basic problem solving agents is that they consider actions in sequence, starting from the initial state. Until the agent has worked out HOW to obtain the items we're looking for, it cannot really decide where to go. The agent therefore needs a more flexible way of structuring its deliberations so that it can in a non-linear fashion.

3 Key Ideas Behind Planning

The **first key idea** in planning is that we 'open up' the representation of states, goals and actions.

1. Planning algorithms use descriptions in some formal language - usually first-order logic
2. States and goals are represented by sets of sentences
3. Actions are represented by logical descriptions of preconditions and effects

This allows the planner to make direct connections between states and actions.
e.g. if the agent knows that the goal is a conjunction that includes have(milk), and it knows that buy(x) achieves have(x) then the agent knows that it is worthwhile to consider a plan that includes buy(milk). It need not consider other irrelevant actions such as buy(orange)

The **second key idea** is that the planner is free to add actions to the plan wherever they are needed, rather than in an incremental sequence starting at the initial state. For example, the agent may decide it needs to buy(milk) even before it has decided how to do such a thing.

There is no necessary *connection between the order of planning and the order of execution*.

The **third key idea** is that most parts of the world are independent of most other parts, and thus it makes it feasible to take a conjunctive goal and solve it using a divide-and-conquer strategy. A subplan involving going to the supermarket can be used to achieve the first two conjucts and another subplan involving going to the hardware store can be used to achieve the third. The

supermarket subplan can be further divided into a milk subplan and a bananas subplan. We can then put all the subplans together to solve the whole problem.

Planning Systems

Open up action, state and goal representations to allow selection - represent in first-order logic

- States and goals = sets of sentences
- actions = description of preconditions and effects

Planning systems allow a planner to make direct connections between states and actions

- We can divide-and-conquer by subgoaling
- Planner can consider several smaller easier problems, and then combine solutions
- Works because little interaction between subplans, otherwise cost of combining solution outweighs the gain e.g. no help for 8 puzzle to consider each tile separately
- Relax requirement for sequential construction of solutions
- Allows planner to add actions where needed, so can make obvious or important decisions first to reduce branching factor

There is no connection between the order of planning and execution. We can do this because of logic - At(Supermarket) represents a class of states, but search requires a complete state description, and so we could not do this.

In the real world, planning tends to do better than search

SPA Algorithm

- Update KB
- if not already executing a plan, generate a goal and construct a plan to achieve it
- Agent must be able to cope if the goal is infeasible or achieved (set action to NoOp)
- Once agent has a plan, it will execute to completion
- Minimal interaction with environment: perceive to determine initial state but then just execute plan - no relevance checks

Simple Planning Agent

```
function SIMPLE-PLANNING-AGENT(percept)
    returns an action
  static : KB, a knowledge base
    p, a plan, initially NoPlan
    t, a time counter, initially 0
  local G, a goal
    current, a current state description
  TELL(KB,MAKE-PERCEPT-SENTENCE(percept,t))
  current ← STATE-DESCRIPTION(KB,t)
  if p = NoPlan then
    G ← ASK(KB,MAKE-GOAL-QUERY(t))
    p ← IDEAL-PLANNER(current,G,KB)
  if p = NoPlan or p empty then action ← NoOp else
      action ← FIRST(p)
      p ← REST(p)
  TELL(KB,MAKE-ACTION-SENTENCE(action,t))
  t ← t + 1
  return action
```

Situation Calculus

- A way of describing change in first-order logic
- World viewed as a sequence of situations, snapshots of the state of the world
- Situations generated from previous situations by actions
- **Fluent**: Functions and predicates that change with time given a situation argument
- Those that do not change are called **eternal** or **atemporal**
- Change represented by function Result(action, situation) which denotes the result of performing action in situation

- *Possibility axioms* — Describes when it is possible to execute an action (Precondition $\implies$ Poss(a,s)) e.g. $At(Agent, x, s) \wedge Adjacent(x, y) \implies Poss(Go(x, y), s)$
- *Effect axioms* — changes due to action (Poss(a,s) $\implies$ changes), e.g. $Poss(Go(x, y), s) \implies At(Agent, y, Result(Go(x, y), s))$

Planning In Situation Calculus

- Planning can be seen as a logical inference problem using situation calculus
- Logical sentences to describe initial state, goal and operators
- Initial state - sentence about the situation - At home, no milk, no bananas etc
- Goal state - logical query for suitable situations - At home and have milk and have bananas etc

- *Given*
  - Initial State: $At(Agent, [1, 1], S_0) \wedge At(G, [1, 2], S_0) \wedge Gold(G)$
  - Possibility Axioms
    - $At(Agent, x, s) \wedge Adjacent(x, y) \implies Poss(Go(x, y), s)$
    - $Gold(g) \wedge At(agent, x, s) \wedge At(g, x, s) \implies Poss(Grab(g), s)$
  - Effect Axioms
    - $Poss(Go(x, y), s) \implies At(Agent, y, Result(Go(x, y), s))$
    - $Poss(Grab(g), s) \implies Holding(g, Result(Grab(g), s))$
- *Goal State*: $\exists seq,\ At(G, [1, 1], Result(seq, S_0))$
- From first Possibility Axiom, $Poss(Go(x, y), s)$
- From first Effect Axiom, $At(Agent, y, Result(Go(x, y), s))$
- So can Agent grab the gold?

Nothing in the KB base says that the location of the gold remains unchanged

Frame Axioms

These tell us how the non-changes due to action.
If some object is at some state, and is not the agent and is not being held by the agent, then the object is still in situation S when the agent moves.

If we have f fluents (things that can change) and a actions, it requires O(AF) frame axioms

The Frame Problem

- representational - proliferation of frame axioms (original frame problem)
- representation problem now largely solved
- inferential - having to carry properties through inference steps, even if remain unchanged
- inferential problem avoided by planning; we do not address it for inference systems.

We solve the representational frame problem with **successor state axioms**

- Each axiom is about a predicate (not an action per se)
- General form: p true afterwards = (an action made P true OR P true already and no action made p false)
- We need a successor state axiom for each predicate that can change over time
- Axiom must list all ways the predicate can become true or false

Practicality of Planning

With first order logic, predicates and situational axioms and calculus, we theoretically have all that is required - but this is unpractical (time, space, semi-decidability) etc.

Thus, we need a restricted language. This reduces the number of possible solutions to search through.

Actions represented in a restricted language, allows creation of efficient planning algorithms
So we need a language and a planning algorithm for that language.

STRIPS is a restricted language that lends itself to efficient planning algorithms, while retaining much of the expressiveness of situation calculus representations

Basic Representations for Planning

Representing States and Goals

States are represented by conjunctions of function-free ground literals - or predicates applied to constant symbols, possible negated.

These state descriptions do not need to be complete. An incomplete description, corresponds to a set of possible complete states for which the agent would like to obtain a successful plan.

Many systems adopt the negation as failure convention - if a state description does not mention a given positive literal,
Remember - a goal given to a planner asks for a sequences of actions that makes the goal true if executed, while a query given to a theorem proves asks whether the query is true given the KB

State example: At(Home AND !Have(milk) AND !Have(bananas)
Goal example - we want to be at a shop that sells milk: At(x) AND Sells(x, Milk)

Representing Actions

STRIPS operators have 3 components

- **action description** - what an agent actually returns to the environment in order to do something
- **precondition** - conjunction of atoms (positive literals) that says what must be true before the operator can be applied
- **effect** of an operator is a conjunction of literals (positive or negative) that describes how the situation changes when the operator is applied

Op(ACTION:Go(There), PRECONDITION:At(here) AND Path(here, there), EFFECT:At(there) AND ! At(here))

An operator with variables is known as an operator schema, because it does not correspond to a single executable action but rather to a family of actions, one for each different instantiation of the variables.

We say that an operator is applicable in a state s if there is some way to instantiate the variables in o so that every one of the preconditions of o is true in s.

For ecample, if the initial state includes the literals At(home), path(home, supermarket) then the action Go(supermarket) is applicable and the resulting situation contains the literals !at(home), At(Supermarket), Path(Home, Supermarket)

Representation for Plans

If we're going to search through plan space, we need to eb able to represnt them. We can settle on a good represntation for plans by considering partial plans for a simple problem: putting on a pair of shoes.

least-commitment - only make choices about things you care about currently, leaving the other choices to be worked out later. This is good for search since you are likely to make the wrong choice and have to backtrack later. A least commitment planner could leave the ordering of the two steps unspecified. This helps us avoid bad plans, by delaying the decision making process until we have more information, allowing the agent to make better choices.

A planner that can represent plans in which some steps are ordered with respect to each other and other steps are unordered is called **partial order planner.**
The alternative planner that plans with a simple list of steps, is a **total-order planner.** A totally ordered plan that is derived from a plan P by adding ordering constraints is called a linearization.

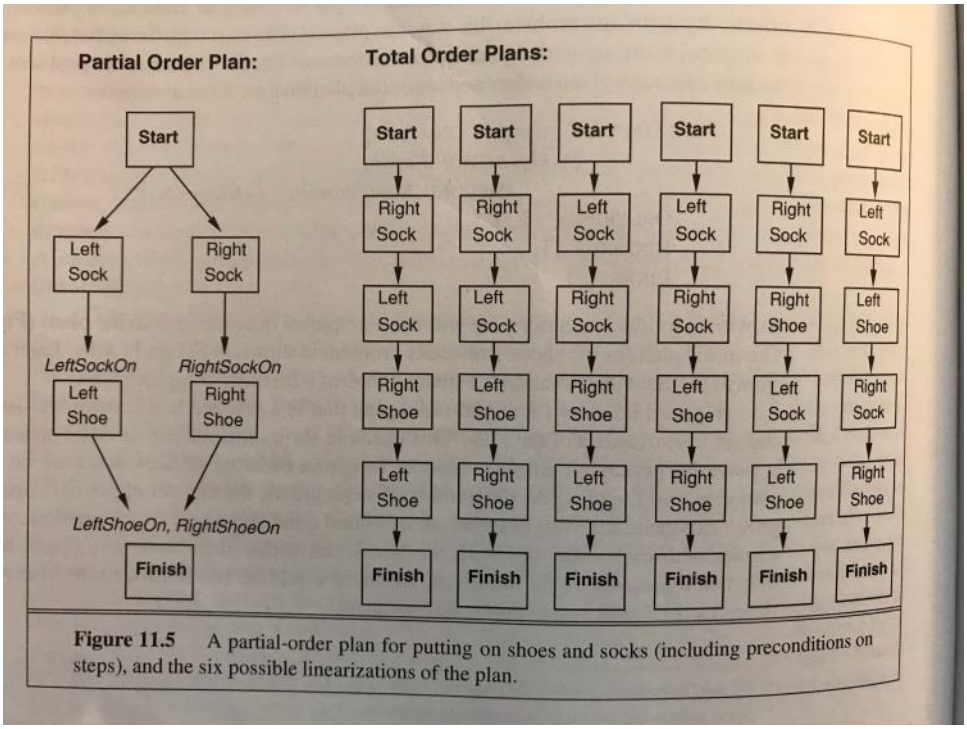


Figure 11.5 A partial-order plan for putting on shoes and socks (including preconditions on steps), and the six possible linearizations of the plan.

Situation Space vs Plan Space

There are a lot of situations that can occur in a world. A path through this space constitutes a plan for a problem. If we wanted, we could take a problem described in the STRIPS languages and solve it by starting at the initial state and applying operators one at a time until we reached a state that includes all the literals in the goal. We could use standard search methods for this

An algorithm that did this would be a problem solver, but it could also be considered a planner. The algorithm would operate in **situation space**. It would be a **progression planner** because it would search forward from the initial situation to the goal situation. Obviously, the branching factor for this method makes it problematic.

One way to cut the branching factor is to search backwards, from a goal state to the initial state. This is a **regression planner**. This approach is possible because the operators contain enough information to regress from a partial description of a result state to a partial description of the state before an operator is applied. We cannot get complete descriptions of states this way, but luckily we don't need to.

This regression approach is desirable because usually, the initial state has many applicable operators that could potentially be used, whereas to move back from the goal state there are typically only a few conjuncts, each of which will only have a few operators.

Searching backwards is hard because we have to achieve a *conjunction of goals*, rather than just one.

Alternatively, we can search through the **space of plans**, rather than the space of plans rather than the space of situations. That is, *we start with a simple, incomplete plan*, which we call a partial plan. Then we *consider ways of expanding the partial plan* until we come up with a complete plan that solves the problem . The operators in this search are operators on plans:

- Adding a step
- Imposing an ordering that puts one step before another
- Instantiating a previously unbound variable.

The solution is the final plan, and the path taken to achieve it is irrelevant.

Refinement operators on plans – take a partial plan and add a constraint. These eliminate plans from the set, but never add new ones
Modification operators – anything that's not a refinement operator is a modification operator. Some planners work by creating an incorrect plan then debugging it with modifications operators.

Plan

- A plan is a data structure consisting of:
- A set of plan steps - each step is one of the operators for the problem
- A set of step ordering constriants e.g. s must occur before x
- A set of variable binding constriants - in the form v = x, where v is a var in some step, and x is either a constant or another variable
- A set of causal links - s_i --c--> s_j, read as "si achieves c for sj". Causal links serve to record the purpose of steps in the plan: here a purpose of s_i is to achieve the precondition of s_j

The initial plan before any refinements take place, simple describes the unsolved problem. It has two steps - Start and Finish and the constraint that start is before finish. There are not links or bindings.

Plan Solution

A solution is a plan that an agent can execute, that guarantees achievement of the goal.
To check a plan is valid, it makes sense to insist on a fully instantiated, totally ordered plan.
However, this is unsatisfactory:

1. Agents can perform tasks in parallel so it makes sense to allow solutions with parallel actions
2. There are many linearisation's of a plan – it is more natural to just return the PO plan than arbitrarily pick a plan
3. If we're creating plans that will be combined with larger plans, it makes sense to maintain flexibility

Therefore, we accept plans that are partially ordered, and

- Complete – every precondition of very step is achieved by some other step – A step achieves a condition if the condition is one of the effects of the step, and if no other step can possible cancel out the condition.
- Consistent – a consistent plan is one in which there are no constrictions in the ordering or binding constraints. A contradiction occurs when both Si must be before Sj and Sj must be before Si – remember plans are transitive.

Partial-Order Planning

31 December 2020 18:53

Quick Recap

You should understand:
1. The idea of an agent that can reason
2. The idea of a KB and an agent performing inference
3. How we adapt that into something that makes plans
4. How plans can do things differently to search
5. Why we might prefer planning over search
6. What the issues are with planning
7. Situation calculus and what it's used for
8. STRIPS and what it can do

Closed-world assumption - most planners assume that if state descriptions do not mention a positive literal, we can assume it to be false - this can be dangerous
Goals - conjunctions of literals, may contain variables
Planner - a system you can ask for a sequence of actions that make the goal true if executed

Operators comprise three components
- **action** e.g. buy(x) - buy an x
- **precondition** e.g. At(p), Sells(p, x) - we're at p and p sells x
- **effect** e.g Have(x) - thus we now have x

The goal is to gradually move from incomplete and vague plan to complete and correct plans

Key Terminology
- Partial Plan – an incomplete plan that we consider ways of expanding, until we come up with a plan that solves the problem
- Operator schema – an operator that takes variables – basically a function.
- Least commitment – one should only make choices about things that you currently care about, leaving other choices to be worked out later
- Partial order – A planner that can represent plans in which some steps are ordered with respect to each other, and some are unordered, is called a partial order planner.
- Total order – a plan that has all the steps in some order – there is no ambiguity about when the steps will take place.
- Linearisation – a totally ordered plan that is derived from a partially ordered plan is called a linearisation of the plan.
- Fully instantiated plans – a plan in which every variable is bound to a constant
- Causal links – a causal link is written as "Si achieves c for Sj". They serve to record the purpose of steps in the plan – here the purpose of Si is to achieve the precondition c of Sj

Plan
A plan is formally a data structure consisting of;
1. A set of plan steps – each step is an operator to the problem
2. A set of step ordering constraints – each ordering constraint is in the form Step I before Step j, meaning that step I has to take place before step j, although not necessarily directly before.
3. A set of variable binding constraints – each variable constraint is of the form v = x where v is a variable in some step, and x is either a constant or another variable
4. A set of causal links – as described above

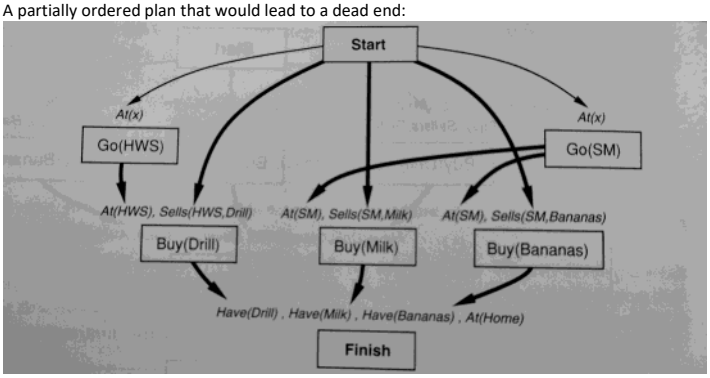
Outline of POP
We sketch an outline for a partial-order regression planner that searches through plan space.

The planner starts with an initial plan representing the start and finish steps, and on each iteration, adds one more step. If this leads to an inconsistent plan, it backtracks and tries another branch of the search space

To keep the search focused, the planner only considers adding steps that serve to achieve a precondition that has not yet been achieved.

If you have two conditions that contradict each other – e.g. go(HardwareStore) and go(superMarket) which both require at(Home), then you reach a dead end. There is no way to go(superMarket) if we're at(hardwareStore) if the precondition for go(superMarket) is at(home). Thus, we have a flawed plan.

A partially ordered plan that would lead to a dead end:



Interestingly, the planner could notice this partial plan is flawed without wasting a lot of time. The key is that the causal links in a partial plan are protected links. A causal link is protected by ensuring that threats (steps that might delete or clobber the protected condition) are ordered to come before or after the new step.

If we place the clobbering step that threatens a condition prior to the step, then we call it a demotion
If we place the step that threatens the condition c after the causal link that protects that condition, we call it a promotion

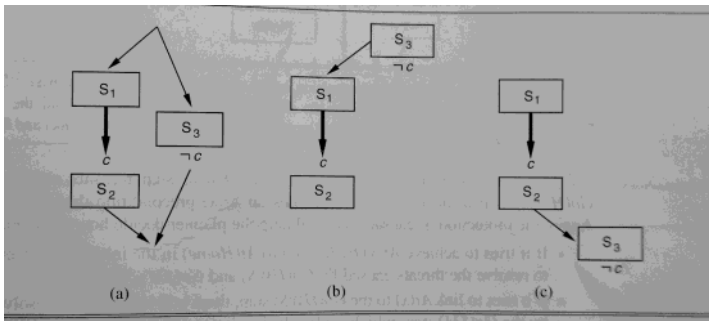


Figure 11.10 Protecting causal links. In (a), the step S_3 threatens a condition c that is established by S_1 and protected by the causal link from S_1 to S_2. In (b), S_3 has been demoted to come before S_1, and in (c) it has been promoted to come after S_2.

Here we have an example of causal link protection in a shopping plan:

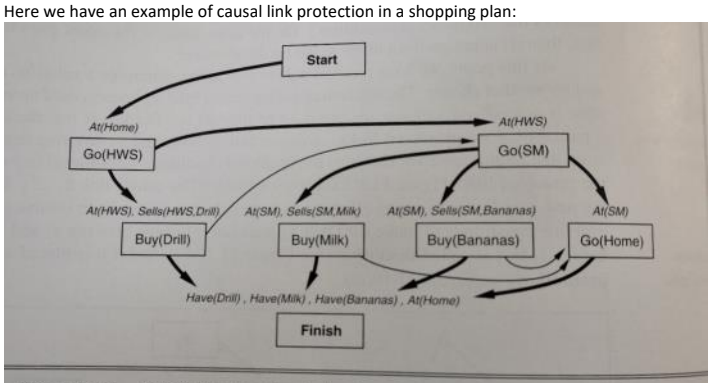
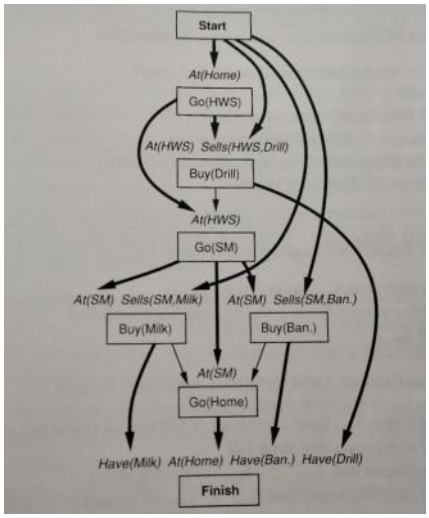


Figure 11.11 Causal link protection in the shopping plan. The $Go(HWS) \xrightarrow{At(HWS)} Buy(Drill)$ causal link is protected ordering the $Go(SM)$ step after $Buy(Drill)$, and the $Go(SM) \xrightarrow{At(SM)} Buy(Milk/Bananas)$ link is protected by ordering $Go(Home)$ after $Buy(Milk)$ and $Buy(Bananas)$.

This is what a finished solution to the shopping problem might look like. Note that the result is almost a totally ordered plan, the only ambiguity being that Buy(Milk) and Buy(bananas) can come in any order.



Our partial order planner can now take a problem that would require thousands of search states for a problem-solving approach, and solve it with only a few search states. Moreover, the least commitment nature of the planner means it only needs to search at all in the places where subplans interact with each other. Finally, the causal links allow the planner to recognise when to abandon a doomed plan without wasting a lot of time expanding irrelevant parts of the plan.

POP Algorithm

POP starts with a minimal partial plan, and at each step extends the plan by achieving a precondition c of a step S_need.

It does this by *choosing some operator*, either from the existing steps of the plan or form the pool of operators, that achieves the precondition. *It records the causal link for the newly achieved precondition, and then resolves any threats to causal links.*

The new step may *threaten an existing causal link* or an existing step may threaten the new causal link. If at any point the algorithm fails to find a relevant operator or resolve a threat, it backtracks to a previous choice point. An important subtlety is that the selection of a step and precondition in select-subgoal is not a candidate for backtracking.

function POP(*initial*,*goal*,*operators*)
  returns a plan
  plan ← MAKE-MINIMAL-PLAN(*initial*,*goal*)
  loop do
    if SOLUTION?(*plan*) **then return** *plan*
    S_{need}, c ← SELECT-SUBGOAL(*plan*)
    CHOOSE-OPERATOR(*plan*,*operators*,S_{need},c)
    RESOLVE-THREATS(*plan*)
  end

function SELECT-SUBGOAL(*plan*)
  returns plan step and precondition (S_{need}, c)
  pick step S_{need} from STEPS(*plan*)
    with precondition c that has not been achieved
  return S_{need}, c

function CHOOSE-OPERATOR(*plan*,*operators*, S_{need},c)
  pick step S_{add} from *operators* or STEPS(*plan*)
      that has effect c
  if no such step **then fail**
  add causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS(*plan*)
  add ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS(*plan*)
  if S_{add} newly added step from *operators* **then**
    add S_{add} to STEPS(*plan*)
    add $Start \prec S_{add} \prec Finish$ to ORDERINGS(*plan*)

procedure RESOLVE-THREATS(*plan*)
  for each S_{threat} that threatens a link
      $S_i \xrightarrow{c} S_j$ **in** LINKS(*plan*) **do**
    choose either
      Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)

The new step may *threaten an existing causal link* or an existing step may threaten the new causal link. If at any point the algorithm fails to find a relevant operator or resolve a threat, it backtracks to a previous choice point. An important subtlety is that the selection of a step and precondition in select-subgoal is not a candidate for backtracking.

The reason is that every precondition needs to be considered eventually, and the handling of preconditions is commutative: Handling c1 and then c2 leads to exactly the same set of possible plans as handling c2 and then c1. So we can just pick a precondition and move ahead without worrying about backtracking. The pick we make effects only the speed, and not the possibility of finding a solution.

Notice that we start at the goal and move backwards until we have a solution. Thus, POP is a sound and complete regression planner. Every plan it returns is a solution.

$S_i \xrightarrow{c} S_j$ **in** LINKS(*plan*) **do**
choose either
Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS(*plan*)
Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS(*plan*)
if not CONSISTENT(*plan*) **then fail**

Problems with POP and STRIPS

STRIPS cannot express:

Hierarchical plans

- we often want to specify plans at different levels of detail.
- We want to have several levels before reaching executable actions.
- This makes computation manageable and resulting plan understandable – analogous to high level and low level machine code.
- This allows human specification of abstract partial plans to guide the planner because we often want to give the planner guidance.

Complex conditions

- Operations have variables, but we have no quantification
- STRIPS use of variables is limited – we cannot express that pick(bag) also causes all objects in the bag to be lifted
- Operators are unconditional - we cannot express actions having different effects according to conditions

Solution – conditional effects – avoids premature commitment – have effect when condition

Time

- In situation calculus, time is discrete, and actions occur instantaneously
- We need to represent that actions taek time, may only be applicable at certain times, and that the goal may have a deadline

Resources

- Real problems have limited resources – financial time, quantity of materials, machinery available
- Action's have a cost that we need a way to represent
- Actions descriptions need to represent the requirements of performing the action
- Planner needs to take cost into consideration

Solution: extend STRIPS and modify POP accordingly

1. Hierarchical decomposition can be added to STRIPS in the form of non-primative operators. We can then modify the planner to repalce non-primatives with decomposition. We do this by adding abstract operators
2. Abstract operators that can be decomposed into steps
3. Decompositions are predetermined and stored in a library of plans – works best when there are several possible decompositions

Broadening Operator Descriptions

We want to make operator descriptions more expressive to widen the applicability of POP

1. Conditional effects – avoids premature commitment - e.g. have effect WHEN
2. Allow negated goals – ability to call CHOOSE-OPERATOR with !p instead of p
3. Disjunctive preconditions – allow SELECT-SUBGOAL to make a nondeterministic choice between disjuncts - use principle of least commitment
4. Disjunctive effects – introduces nondeterminism – may be able to address with coercion
5. Universally quantified preconditions – instead of clear(b) we can do "for all blocks, remove blocks from b
6. Resource constraints – add numeric-value measures such as money(200) or fuel(20) - a measure fluent
7. Temporal constraints – we can treat time as a resource – remember we can do things concurrently so time is the max of the jobs not the sum

With resource constraints, we want to plan for scarce resources first, delaying choice of causal links where possible. We can check for failure then without a finished plan, so we don't have to try all operators.

Conditional Planning

So far we have assumed that the world is fully observable, static and deterministic– we also assumed that action descriptions are correct and complete. Unfortunately, the real world is not like this– typically have to deal with both incomplete and incorrect information.

In the real world we have incomplete information– we don't know the preconditions of an operator, and we don't know the effects of an operator e.g. inflate(tire) might cause inflated(tire), slowhiss(tire), burst(tire), breakPump(tire) etc

We can also have incorrect information. We might think we have a spare tire, then find we don't.

We can never finish listing all the required preconditions and possible conditional outcomes of actions in

the real world

Conditional Planning
- Plan to obtain information
- Subplan for each contingency that could happen
- Expensive because it plans for many unlikely cases

Replanning

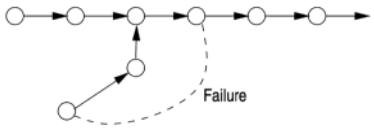
Action Monitoring:
- Preconditions of next action note met

Plan monitoring:
- Effects of an action might not be as predicted
- Failure = preconditions of remaining plan are not met
- Preconditions = causal links at current time
- Check current preconditions with perceived state

In each case, we fail and need to replan.

There are 2 types of plan failure –
1. **bounded indeterminacy**: when unexpected effects of actions can be enumerated – we can use conditional planning to deal with this
2. **Unbounded indeterminacy**: set of possible outcomes is too large to enumerate – we can plan for (at most) a limited number of contingencies and must replace when things go wrong – e.g. driving

Replanning



A naive approach would be to replan from scratch, but this is inefficient and we might get stuck in a cycle of a bad plan

A better approach would be to plan to get back on track by connecting the old plan as show above
Ideally we introduce learning so the agent doesn't loop forever

Alternative solutions to replanning

Coercion – force the world into a particular state to reduce uncertainty
Abstraction – ignor details of a problem that may be unknown
Aggregation – Treat a large number of objects which have individual uncertainty as a single, aggregate precticable object

Reactive Planning
Abandon domain-independent planning and use domain specific knowledge - the agent has procedural knowledge – library of partial plans representing a collection of behaviours

Problem Sheet 5 – Planning

18 November 2020 11:15

seminar_sheet_5_qu...

Q2)

Problem Solving:
- All state representations are complete
- Goal test and heuristic functions are block boxes (hard-coded) representations
- Solution constructed as continuous sequence from initial state to goal (or visa versa if backwards braching factor < forwards branching factor)
- Means that if something changes or goes wrong, have to start over again

Planning:
- Opens up representation of actions, states and goal test
  - Model the consequences of actions and more explicitly aware of our goals
  - State no longer discrete -> modelled by the presence or absence of a variety of conditions
- No requirement for sequential construction of solution -> planner selects appropriate part of solution to work on (uses per and post conditions)
- Can use subgoals to divide and conquer

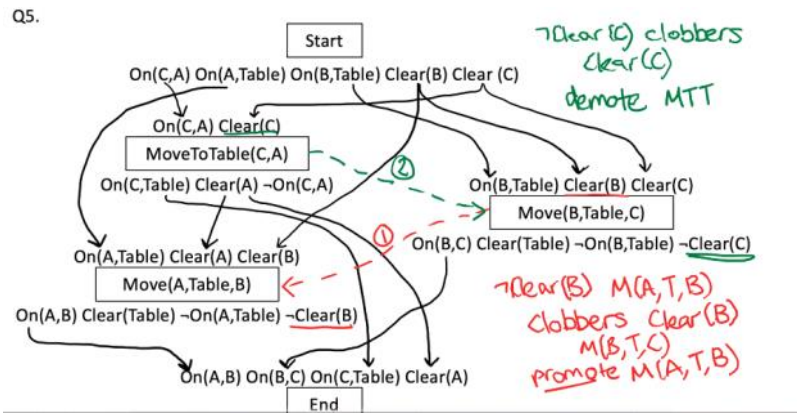
Planning is more flexible, because we can replan with no black boxes
Planning is more complex, but more flexible for the real world

Q3)

Q3.

POP:
Each iteration:
- Select step to achieve a precondition C of a step S_{next} that has an open precondition
- Choose an operator O to achieve C (from existing steps or by adding a new one)
- Add operator O to plan if needed, record causal link from O→ S_{need}
- If O is new, add Start ≺ O and O ≺ End to ordering constraints

Q5)
We have a start state and a goal state.



Q6)

Q6.

Conditional Planning:

Addresses 3 problems:

- Incomplete information:
  - unknown preconditions or effects
- Incorrect information:
  - state information, preconditions, or effects incorrect
  - Plan may not complete as expected
- Qualification problem:
  - If many preconditions then may not be certain of effect and may have too many possible outcomes (i.e. uncertainty)

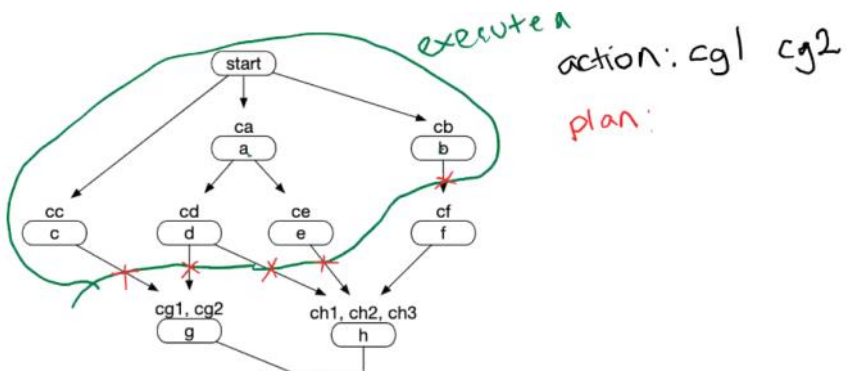
Q7.

Action monitoring:

- Check preconditions of next action

Plan monitoring:

- Check preconditions of entire remaining plan
- If an action has a causal link that says it must happen after another action, and the other action hasn't been executed, then do not need to include its preconditions



If we're doing action monitoring, we need to check