AI Notes

Rafael Blecher - z3380867

May 26, 2014

Contents

Part I

Week 1

Chapter 1

Environment Types

An *agent* is a function from *percept sequences* to actions. Ideally, an agent is rational, and picks actions which maximise the performance measure. This performance measure can be evaluated empirically, and sometimes analysed empirically.

1.1 PEAS: Performance, Environment, Actuators, Sensors

1.1.1 Performance

A function that measures the quality of the work done by the agent. Examples include Safe, Fast, Legal, Comfortable trip, Maximize profits

1.1.2 Environment

The environment that the agent operates in is generally described by the following properties:

Fully or partially observable

The environment is fully observable if the agent's sensors give it access to the complete state of the environment at each point in time. Note: the sensors only need to detect all RELEVANT information to the task. Relevance depends on the Performance Measure. Causes for partially observable environments are noise, inaccurate sensors, or missing sensors. Example of partially observable environments: a self-driving car cannot tell what other drivers are thinking.

Deterministic or stochastic or strategic

The environment is deterministic if the next state of the environment is completely determined by the current state and the action executed by the agent. We tend to define whether or not an environment is deterministic by considering the environment from the agent's point of view. Interesting to note is that if an environment is both deterministic and fully observable, the agent does not - in theory, at least - have to be concerned with uncertainty. If the environment

is deterministic except for the strategic actions of other agents, we say that the environment is strategic.

Episodic or sequential

Episodic environments divide the agent's experience into atomic episodes. No episode affects the state of another episode, and so every episode is a self-contained experience of [perceive, act]. An example of an episodic environment is checking for defective parts on as assembly line. The agent only considers the current part, and what the agent perceives, and its subsequent action, have no effect on past or future results.

In a sequential environment, the current action can affect all future decisions. Examples of sequential environments include playing chess, and self-driving cars. They are far more complicated than episodic environments because future moves must be considered before making a decision for the current state.

Discrete or continuous

Discrete/continuous can be understood in relation to state, time, and percepts and actions of the agent. Chess is a discrete-state environment, as there are a discrete number of states of a chess game. Chess also has a discrete set of percepts and actions. Taxi driving is both state- and time-continuous: the location of the taxi and other vehicles flow smoothly over time. Taxi-driving actions are also continuous (think of steering angles).

Sometimes it is difficult to ascertain whether or not an environment is discrete. Input from a digital camera is technically discrete: there are $n * m$ pixels, each of which can take any value in the RGB range 000000 to FFFFFF. On the other hand, the input depicts a real-life image, which forces pixels to be displayed with different intensities and hence feels distinctly continuous.

Single-agent or multi-agent

In a single-agent environment, the one agent does not have to account for the actions of other agents, and can concern itself only with how it interacts with the environment.

An environment is described as competitive if it is multi-agent and each agent is in competition with every other agent. The environment is described as cooperative if the agents are working together to achieve some goal. N.B.: competitive and cooperative are not mutually exclusive.

Static or Dynamic

An environment is described as static if it does not change while the agent is deliberating. Otherwise, the environment is dynamic.

Known or Unknown

The environment is *known* if the rules of the environment are known. Otherwise, the environment is unknown. Examples of rules are game rules, and physics/dynamics of the environment.

Simulated, or Situated

An environment is simulated if a separate program is used to simulate the environment, feed percepts to agents, evaluate performance, etc.
See Wikipedia: Artificial intelligence, situated approach for more.

Situated or Embodied? There is an important distinction to make between situatedness and embodiment.

Situatedness describes an agent being situated in a world - they are only concerned with the "here and now" of the world", rather than abstract descriptions. The state of the environment directly influences the behaviour of the agent system.

Embodiment describes the AI having a body (a robot), which experiences the world directly. The actions are part of a dynamic with the world, and an agent's actions have immediate feedback on the robot's own percepts.

An airline reservation system is situated but not embodied; it deals with requets and its responses change as the database changes.

A spray-painting robot is embodied but not situated; it does not perceive anything about the object presented to it, it just goes through a pre-programmed series of actions. The robot has physical extent and must correct for its interaction with gravity, etc.

The Real World

Using these definitions, we describe our world as partially observable, stochastic, sequential, continuous, multi-agent, dynamic, unknown, and situated.

1.1.3 Actuators

Actuators are the set of devices that the agent can use to perform actions.

1.1.4 Sensors

Sensors allow the agent to collect the percept sequence that will be used to decide on the next action.

1.1.5 Applying PEAS to Wumpus World

Performance Measure

Return with Gold +1000, Death -1000, -1 per step, -10 for using the arrow

Actuators

Left, right, forward, grab, shoot

Sensors

Breeze, Glitter, Stench

1.1.6 Applying PEAS to a self-driving car

Performance measure

safety, reach destination, maximise profits, obey laws, passenger comfort

Environment

city streets, freeways, traffic, pedestrians, weather, customers

Actuators

steer, accelerate, brake, horn, speak/display

Sensors

video, accelerometers, guages, engine sensors, keyboard, GPS

Part II

Week 2

Chapter 2

Agent Types

2.1 Reactive Agent

Reactive agents are quite simple; steps are repeated in a very simple procedure:

1. Perceive environment
2. Act

Reactive agents choose the next action to perform based only on what the currently perceive. The action is based on a simple set of rules (called a *policy*) which are easy to apply.

Reactive agents are sometimes called "simple reflex agents", despite the fact that they can actually perform fairly sophisticated tasks (for example, simulated hockey can be performed by a reactive agent.)

2.1.1 Limitations of Reactive Agents

A reactive agent only knows about what it currently perceives and in many cases, you need to remember previous percepts to make intelligent decisions. Reactive agents may also perform the same action over and over, if what they perceive never changes.

2.2 Model-Based Agent

A model-based agent is slightly more complicated than a reactive agent. A model-based agent includes a world model, which represents the important parts of the environment. The agent adjusts its world model based on its percepts, and the agent's actions are informed by both the world model and its current percept.

2.2.1 Advantages of Model-Based Agents

A model-based agent can remember past percepts.

2.2.2 Limitations of Model-Based Agents

Sometimes we need to plan several steps into the future. Hence, a model-based agent will perform badly in the following cases:

Searching several moves ahead Chess, Rubik's Cube

Complex tasks with many steps Cooking a meal, assembling a watch

Logical reasoning to achieve a goal travel to New York

2.3 Planning Agent

A planning agent adds a planning step to a model-based agent. The world model can be a transition table, a dynamical system, a parametric model, a knowledge base, etc.

The planning step can be a state-based search, a simulation, logical inference, or a set of goals. The action of the planning agent is informed by the planning step.

A planning agent has the capacity to reason about future states. However, sometimes agents appear to be planning but are really just reactive, applying rules which are hard-coded or learned. These agents appear intelligent, but are not flexible in adapting to new situations.

2.4 Learning Agent

A learning agent adds a Learning step to each part of the planning agent. Statistical learning is applied to Perception, reinforcement learinng is applied to the Action, Bayesian learning is applied to the world model, and inference learning is applied to the planning step.

Learning is a set of techniques for improving the existing modules in the planning agent, rather than being a different module in the agent.

Learning is necessary because it may be difficult for a human to design all aspects of a system by hand, and because the agent may have to adapt to new situations without being re-programmed by a human.

Application of learned behaviour is distinct from the learning process. For example, the policy for a simulated hockey player took sevel days of computation to derive, but after that process, can be applied in real-time.

Chapter 3

Solving Problems by Searching

3.1 Motivation

Reactive and model-based agents choose their actions based on what they currently perceive, or what the have perceived in the recent past.

A planning agent can use search techniques to plan several steps ahead in order to achieve its goal(s).

There are two classes of search strategies:

1. Uninformed search strategies can only distinguish goal states from non-goal states
2. Informed search strategies use heuristics to try to get closer to the goal

3.2 Romania Example

Now, let's apply the search strategy to the example covered in lectures; the map of Romania. We are touring Romania, and we are currently in Arad. Our flight leaves tomorrow from Bucharest.

Formulate goal be in Bucharest on time.

Specify task

- states: various cities
- operators or actions (transitions between states): drive between cities

Find solution (action sequences) sequence of cities to drive through

Execute drive through all the cities given by the solution.

3.2.1 Single-State Task Specification

A task is specified by states and actions.

initial state At Arad

state space other cities

actions or operators (or successor function S(x)) Arad $\rightarrow$ Zerind

goal test check if a state is a goal state. Can be explicit ("at Bucharest") or implicit ($NoDirt(x)$)

path cost sum of distances from Arad to Bucharest

total cost = search cost + path cost = offline cost + online cost

A *solution* is a state-action sequence (initial to goal state)

3.2.2 Choosing states and actions

The real world is complex; the state space must be abstracted for problem-solving.

An abstract state is a set of real states.

An abstract action is a complex combination of real actions. For example, "Arad $\rightarrow$ Zerind" describes a complex set of possible routes, detours, rest stops, etc. For guaranteed realisability, any real state must get to some real state (for example, Arad must get to Zerind).

An abstract solution is a set of real paths that are solutions in the real world.

3.2.3 Problem types

Toy problems have a concise, exact description

Real-world problems don't have a single agreed description.

3.2.4 Applying task specifications

The 8-Puzzle

states integer locations of tiles (ignore intermediate positions)

operators move blank left, right, up, down (ignore unjamming, etc.)

goal test = goal state (given)

path cost 1 per move

Rubik's Cube

TODO NOTHING HERE. Any takers?

3.3 Path Search Algorithms

A *search* finds state-action sequences that lead to desirable states. Search is a function with prototype

`solutionType` *search*(*task*)

The basic idea is that a search is an offline, simulated exploration of the state space by generating successors of already-explored states (by "expanding" them).

To generate an action sequence:

- Start with the initial state
- Test if it is a goal state
- Expand one of the states
- If there are multiple possibilities, make a choice
- Procedure: choosing, testing, and expanding until a solution is found or there are no more states to expand

Generating an action sequence can be thought of as building a *search tree*. A search tree is superimposed over the state space. The root of the tree is the initial state, and the leaves of the tree are states that are not expanded, or that generated no new nodes. The state space is not the same as the search tree; in the Romania example there are only 20 states (20 cities) but there are infinitely many paths.

A node data structure is composed of five components:

1. Corresponding state
2. Parent node: the node which generated the current node
3. Operator that was applied to generate the current node
4. Depth: the number of nodes from the root to the current node
5. Path cost

Nodes are distinct from states: a state is [a representation of] a physical configuration. A node is a data structure that makes up part of a search tree. States do not have parents, children, depth, or path cost.
Note: multiple nodes can contain the same state.

3.3.1 Data Structures for Search Trees

The *frontier* is the collection of nodes that have not yet been expanded. It can be implemented as a priority queue with the following operations:

Make-Queue(Items) creates queue with given items

Boolean Empty(Queue) returns TRUE if no items in queue

Remove-Front(Queue) removes the item at the front of the queue and returns it

Queueing-Fn(Items, Queue) inserts new items into the queue.

3.3.2 Search Strategies

A strategy is defined by picking the order of node expansion. Strategies are evaluated along the following dimensions:

completeness does it always find a solution if one exists?

time complexity number of nodes generated/expanded

space complexity maximum number of nodes in memory

optimality does it always find a least-cost solution?

Time and space complexity are measured in terms of

b maximum branching factor of the search tree

d depth of the least-cost solution

m maximum depth of the state space (may be ∞)

3.3.3 Approaches to comparing algorithms

Benchmarking

Run two algorithms on a computer and measure speed. This is not a great way of comparing algorithms, as it depends on implementation, compiler, computer, data, network, ...

Analysis of algorithms

We tend to use Big-O notation to describe algorithms. T(n) $\in$ O($f(n)$) means $\exists k > 0 \exists n_0 : \forall n > n_0 \mathrm{T}(n) \leq kf(n)$ Here, n is the input size and T(n) is the total number of steps in the algorithm. In words, this means that $f(n)$ describes the limiting behaviour of T(n) when n tends towards some value, or infinity. For more, see Wikipedia: Big O notation.

Note: alternatives to T(n) $\in$ O($f(n)$) are T(n) = O($f(n)$) and T(n) is O($f(n)$).

Big O notation:

- is independent of the implementation, compiler, etc.
- provides asymptotic analysis; for large n, O(n) is better than O(n^2)
- abstracts over contant factors. i.e. T($100n+1000$) is better than T(n^2+1) only for $n > 110$.
- is a good compromise between precision and ease of analysis.

3.3.4 Uninformed Search Strategies

Uninformed (or "blind") search strategies use only the information available in the problem definition (can only distinguish a goal from a non-goal state). Examples include

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative Deepening search

Strategies are distinguished by the order in which nodes are expanded.

Breadth-first search

All nodes are expanded at a given depth in the tree before any nodes at the next level are expanded.

Expand root first, then all nodes generated by the root, then all nodes generated by those nodes, ...

Expand shallowest unexpanded node

Implementation: put newly generated nodes at the end of the queue

Finds the shallowest goal first.

Breadth-first search is complete (if b is finite the shallowest goal is at fixed depth d and will be found before any deeper nodes are expanded).

Space complexity: $b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$

Time complexity: $O(b^{d+1})$

Is optimal if all actions have the same cost.

Space is the biggest problem with BFS; it grows exponentially with depth.

Uniform-Cost Search

Expand root node first, then expand least-cost unexpanded node first.

Implementation: enqueue nodes in order of increasing path cost.

Reduces to BFS when all actions have same cost.

Finds the cheapest goal provided the path cost is monotonically increasing along each path (i.e. no negative-cost paths)

Uniform-cost search is complete if b is finite and step cost $\geq \epsilon$ with $\epsilon > 0$

Time complexity: $O(b^{\lceil C*/\epsilon \rceil})$ where $C*$ = cost of optimal solution, and assume that every action costs at least ϵ

Space complexity: $O(b^{\lceil C*/\epsilon \rceil})$ ($b^{\lceil C*/\epsilon \rceil} = b^d$ if all step costs are equal)

Is optimal.

Depth-First Search

Expands one of the nodes at the deepest level of the tree

Implementation: insert newly generated nodes at the front of the queue. Can be alternatively implemented by recursive function calls.

Not complete: fails in infinite-depth spaces, spaces with loops. Can be modified to avoid repeated states along path $\Longrightarrow$ complete in finite spaces.

Time complexity: $O(b^m)$ (terrible if m is much larger than d but if solutions are dense, may be much faster than BFS)

Space complexity: $O(bm)$ (linear space!)

Not optimal, can find suboptimal solutions first.

Depth-Limited Search

Expands nodes like DFS but imposes a cutoff on the maximum depth of path.

Complete? Yes (no infinite loops)

Time complexity: $1 + b^1 + b^2 + \ldots + b^{l-1} + b^l = O(b^l)$ where l is the depth limit

Space complexity: $O(bl)$ linear space similar to DFS

Not optimal, like DFS.

Also comes with the problem of picking a good limit.

Iterative Deepening Search

Tries to combine the memory benefits of DFS and optimality/completeness of BFS by doing a series of depth-limited searches to depth 1, 2, 3, ...

Early states will be expanded multiple times, but that might not matter too much because most of the nodes are near the leaves.

Complete.

Time complexity: nodes at the bottom are expanded once, nodes at the next level are expanded twice, etc. $(d+1)b^0 + db^1 + (d-1)b^2 + ... + 2b^{d-1} + 1b^d = O(b^d)$. For example, given $b = 10$, $d = 5$:

depth-limited: $1 + 10 + 100 + 1000 + 10000 + 100000 = 111111$

iterative-deepening: $6 + 50 + 400 + 3000 + 20000 + 100000 = 123456$

only about 11% more nodes (for $b = 10$).

Space complexity: $O(bd)$

Is optimal if step costs are identical.

Bidirectional Search

The idea of bidirectional search is to search both forward from the initial state and backward from the goal state, and stop when the two searches meet in the middle.

We need an efficient way to check if a new node already exists in the other half of the search. The complexity analysis assumes that this can be done in constant time, using a hash table.

Assuming branch factor b in both directions and that there is a solution at depth d. Then bidirectional search finds a solution in $O(2b^{d/2}) = O(b^{d/2})$ time steps.

However, there are problems with bidirectional search. Searching backwards means generating predecessors from the goal, which may be difficult. There can be several goals (e.g. checkmate positions in Chess). Space complexity is $O(b^{d/2})$ because the nodes of at least one half must be kept in memory.

Complexities of Uninformed Search Strategies

Search	BFS	Uniform-Cost Search	DFS	Depth-Limited Search	Iterative Deepening
Time complexity	$O(b^{d+1})$	$O(b^{\lceil C*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space complexity	$O(b^{d+1})$	$O(b^{\lceil C*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Complete?	Yes[1]	Yes[2]	No	No	Yes[1]
Optimal?	Yes[3]	Yes	No	No	Yes[3]

3.3.5 Informed Search Strategies

Informed (or "heuristic") search strategies use task-specific knowledge. An example of task-specific knowledge: distance between two cities on the map of Romania. Informed search is more efficient than Uninformed search. Uninformed search systematically generates new states and tests them against the goal - informed search attempts to filter input, or order inputs in a way that the most likely candidate is tested before others.

[1] complete if b is finite
[2] complete if b is finite and step costs $\leq \epsilon$ with $\epsilon > 0$
[3] optimal if actions all have the same cost

Part III

Week 3

Chapter 4

Informed Search

4.1 Search Strategies

General search algorithm:

- Add initial state to queue
  - take node from front of queue
  - test if it is a goal state; if so, terminate
  - expand it (generate successor nodes and add them to the queue)

Search strategies are distinguished by the order in which new nodes are added to the queue of nodes awaiting expansion.

BFS and DFS treat all nodes equally; BFS adds new nodes to the end of the queue, DFS adds new nodes to the front of the queue.

Best First Search uses an evaluation function $f()$ to order the nodes in the queue. An example is Uniform Cost Search; $f(n) =$ cost $g(n)$ of path from root to node n.

Informed or heuristic search strategies incorporate into $f(n)$ an estimate of distance to goal. Greedy Search uses $f(n) =$ estimate $h(n)$ of cost from node n to goal. A* search uses $f(n) = g(n) + h(n)$.

4.2 Best-First Search

The Best-First Search family of algorithms have different evaluation functions $f(n)$. A key component of these algorithms is the heuristic function $h(n)$.

Heuristic function h: {Set of nodes} $\rightarrow R$. $h(n) =$ estimated cost of the cheapest path from the current node n to the goal node. Heuristic functions provide an estimate of solution cost.

4.2.1 Greedy Best-First Search

Greedy Best-First Search is a Best-First Search that selects the next node for expansion using the heuristic function as its evaluation function. i.e. $f(n) = h(n)$. $h(n) = 0 \iff n$ is a goal state. Greedy search minimises the estimated

cost to the goal; it expands whichever node n is estimated to be closest to the goal.

Greedy Best-First Search is not complete as it can get stuck in loops. It is complete is finite spaces with repeated-space checking.

Time complexity is $O(b^m)$ where m is the maximum depth in the search space.

Space complexity is $O(b^m)$ as all nodes are kept in memory.

Greedy search is not optimal. Greedy search has the same problems as DFS, but a good heuristic can reduce time and memory costs.

4.2.2 Straight Line Distance as a Heuristic

$h_{SLD}(n)$ = straight-line distance between n and the goal node. If you know map coordinates of n and the goal, you can find the straight line distance using $\sqrt{(n_x - goal_x)^2 + (n_y - goal_y)^2}$

4.2.3 A* Search

A* Search uses $f(n) = g(n) + h(n)$ where $g(n)$ is the cost from the initial node to node n, $h(n)$ is the estimated cost of the cheapest path from n to goal, and $f(n)$ is the estimated total cost of the cheapest solution through node n.

Greedy search minimises $h(n)$ (efficient but not optimal or complete), Uniform Cost Search minimises $g(n)$ (optimal and complete but not efficient). A* minimises $f(n) = g(n) + h(n)$ - the idea is to preserve the efficiency of Greedy without expanding paths that are already expensive. A* is both optimal and complete, provided that $h(n)$ is admissible (never overestimates the cost to reach the goal).

Specifically, a heuristic $h()$ is admissible if $\forall n, h(n) \leq h^*(n)$ where $h^*(n)$ is the true cost from n to the goal. If h is admissible then $f(n)$ never overestimates the actual cost of the best solution through n. $h_{SLD}()$ is admissible because the shortest path between two points is the straight line connecting them.

Proving Optimality of A* Search

Suppose a suboptimal goal node G_2 has been generated and is in the queue. Let n be the last unexpanded node on the shortest path to an optimal goal node G.

$$\begin{aligned} f(G_2) &= g(G_2) \text{ since } h(G_2) = 0 \\ &> g(G) \text{ since } G_2 \text{ is suboptimal} \\ &\geq f(n) \text{ since } h \text{ is admissible} \end{aligned}$$

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion (although it may be generated and put into the queue). In words, A* will keep searching until there is no possibility of finding a shorter solution, even after finding a goal node.

A* is complete unless there are infinitely many nodes with $f \leq$ cost of solution. The time complexity is exponential in *relative error in (h * length of solution)*. All nodes are kept in memory, so space complexity is not great. A* is optimal, given that the heuristic is admissible.

4.2.4 Iterative Deepening A* Search

Iterative Deepening A* is a low-memory variant of A* which performs a series of DFSs, cutting off each search when the sum $f(n) = g(n) + h(n)$ exceeds some pre-defined threshold. The threshold is steadily increased with each successive search. Iterative Deepening A* is asymptotically as efficient as A* for domains where the number of states grows exponentially.

4.3 Heuristics

4.3.1 Dominance of Heuristics

If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 *dominates* h_1 and is better for search. The aim is therefore to make the heuristic $h()$ as large as possible without exceeding $h^*()$.

4.3.2 How to Find Heuristic Functions

Admissible heuristics can often be derived from the exact solution cost of a relaxed (simplified) version of the problem that has some constraints weakened or removed.

One can also use a *composite heuristic*; let $h_1, h_2, \ldots, h_m$ be admissible heuristics. The composite heuristic $h(n) = max(h_1(n), h_2(n), \ldots, h_m(n))$. h is trivially admissible, and more importantly, dominates $h_1, h_2, \ldots, h_m$.

Chapter 5

Reactive Agents

As mentioned in the beginning of chapter 2, reactive agents choose the next action based only on what they currently perceive, using a policy. They have no memory, and are unable to plan or logically reason. However, interesting behaviours can emerge from the simple rules in the policy.

5.1 History of Reactive Agents

5.1.1 Braitenberg Vehicles

Braitenberg showed how simple configurations of sensors and motors can lead to surprisingly sophisticated behaviour. The simplest Braitenberg vehicles have two wheels and two sensors. The sensors respond to light, and the response is inversely proportionate to distance from the light source.

The connections can be straight or crossed, and excitatory (+) or inhibitory. Combinations of these lead to four behaviours: hate, love, fear, and curiosity. Imagine a vehicle with a light source ahead of it to its right.

Hate: Crossed and excitatory

The vehicle will sense the light with its right-hand sensor first, causing its *left* wheel (the wires are crossed) to accelerate (excitatory). When the vehicle has turned so that the light source is directly ahead of it, the left and right sensors will be receiving the same input and the vehicle will travel in a straight line towards the light source.

The vehicle will travel faster and faster before ramming into the light source, and will then attempt to continue driving.

Love: Straight and inhibitory

The vehicle will sense the light with its right-hand sensor first, causing its right wheel to deccelerate. When the vehicle has turned so that the light source is directly ahead of it, the left and right sensors will be receiving the same input and the vehicle will travel in a straight line towards the light source.

The vehicle will slow until it is directly in front of the light source, when it stops.

Fear: Straight and excitatory

The vehicle will sense the light with its right-hand sensor first, causing its right wheel to accelerate. As the vehicle gets closer to the light source it will increase its speed, and will continue turning away from the light source and accelerating until it no longer senses the light.

Curiosity: Crossed and inhibitory

The vehicle will sense the light with its right-hand sensor first, causing its left wheel to deccelerate. The vehicle will approach the light source and slow until a certain point, when the vehicle will turn away from the light, picking up speed in the process.

5.1.2 The Swiss Robots

The Swiss Robots were given a simple set of rule that are used to clean up objects in an area. The rules are as follows:

- Normally, move forward
- If you detect an obstacle to the left or right, turn away from it
- If you detect an obstacle directly in front of you, move forward.

TODO THIS NEEDS COMMENTARY. HALP.

5.2 Behaviour-Based Robotics

Introduced by Rodney Brooks in the late 1980s, behaviour-based robotics was meant as a challenge to GOFAI (Good Old Fashioned AI). It had a few key points:

- robots should be based on insects rather than humans
- tasks like walking around and avoiding obstacles rather than playing Chess
- abandon traditional horizontal decomposition (Sense $\rightarrow$ Plan $\rightarrow$ Act)
- replace vertical decomposition (each layer can connect sensing to an action)

TODO INSERT IMAGES PLZ

5.2.1 Modern Perspective

Each layer in vertical composition is a behaviour. Low-level behaviours such as "avoid hitting things" are reactive, connecting sensors to actuators. Mid-level behaviours like "build maps" make use of a world model. High-level behaviours make use of a world map and planning.

Importantly, higher-level behaviour may take control from lower-level behaviour. e.g. if low-level behaviour has gotten stuck. Lower-level behaviour can also take control from higher-level behaviour, usually in an emergency (i.e. to

avoid getting burned, or falling down a staircase). This is similar to the way that humans' nervous systems have immediate responses to triggers such as heat and pain.

Part IV

Week 4

Chapter 6

Games

6.1 Why Games?

Games have "unpredictable" opponents ⇒ solution is a strategy.

Time limits ⇒ must rely on approximation (tradeoff between speed and accuracy).

Games have been a key driver of new techniques in CompSci and AI.

6.2 Types of Games

6.2.1 Discrete Games

Fully observable, deterministic (chess, checkers, go, othello)

Fully observable, stochastic (backgammon, monopoly)

Partially observable (bridge, poker, scrabble)

6.2.2 Continuous, embodied games

Robocup soccer, pool

6.3 Key Ideas

- Computer considers possible lines of play
- Algorithm for perfect play
- Finite horizon, approximate evaluation
- Machine learning to improve evaluation accuracy
- Pruning to allow deeper search

6.3.1 Minimax

Minimax provides perfect play for deterministic, perfect-information games. The idea is to choose the position with the highest minimax value = best achievable payoff against best play.

Minimax Algorithm

```
function minimax(node, depth)
   if node is a terminal node or depth = 0
      return heuristic value of node
   if we are to play at node
      let α = −∞
      foreach child of node
         let α = max(α, minimax(child, depth − 1))
      return α
   else // opponent is to play at node
      let β = ∞
      foreach child of node
         let β = min(β, minimax(child, depth − 1))
      return β
```

This assumes that all nodes are evaluated with respect to a fixed player (e.g. White in Chess).

Negamax formulation of Minimax Minimax can be simplified to an algorithm called Negamax if we assume that each node is evaluated with respect to the player whose turn it is to move.

```
function negamax(node, depth)
   if node is a terminal node or depth = 0
      return heuristic value of node from perspective of player whose
turn it is
   let α = −∞
   foreach child of node
      let α = max(α, −negamax(child, depth − 1))
   return α
```

Properties of Minimax Complete?
Optimal?
Time complexity?
Space complexity?

6.3.2 Reducing the Search Effort

For chess, $b \approx 35, m \approx 100$ for "reasonable" games $\Rightarrow$ exact solution is completely infeasible.

There are two ways to make the search feasible:

1. don't search to final position; use heuristic evaluation at the leaves
2. $\alpha - \beta$ pruning

Heuristic Evaluation for Chess

Material: Queen = 9, Rook = 5, Knight = Bishop = 3, Pawn=1

Position: some (fractional) score for a particular piece on a particular square

Interaction: some (fractional) score for one piece attacking another piece, etc.

The value of individual features can be determined by reinforcement learning.

Motivation for Pruning

Once we have seen one reply scary enough to convince us the move is really bad, we can abandon this move and search elsewhere.

$\alpha - \beta$ search algorithm

```
function alphabeta(node, depth, α, β, ourTurn)
   if node is a terminal node or depth = 0
      return heuristic value of node
   if ourTurn // we are to play at node
      foreach child of node
         let α = max(α, alphabeta(child, depth − 1, α, β, FALSE))
         if α ≥ β
            return α // β is pruned off
      return α
   else // opponent is to play at node
      foreach child of node
         let β = min(β, alphabeta(child, depth − 1, α, β, TRUE))
         if β ≥ α // α is pruned off
            return β
      return β
alphabeta(origin, depth, −∞, ∞, TRUE)
```

Negamax formulation of $\alpha - \beta$ search

```
function minimax(node, depth)
   return alphabeta(node, depth, −∞, ∞)

function alphabeta(node, depth, α, β)
   if node is terminal or depth = 0
      return heuristic value of node
      // from perspective of player whose turn it is to move
   foreach child of node
      let α = max(α, −alphabeta(child, depth − 1, −β, −α))
      if α ≥ β
         return α
   return α
```

Why is it called $\alpha - \beta$? α is the best value for us found so far, off the current path

β is the best value for opponent found so far, off the current path

If we find a move whose value exceeds α, pass this new value up the tree.

If the current node value exceeds β, it is "too good to be true", so we "prune off" the remaining children.

TODO CAN SOMEONE EXPLAIN THIS? how is it "too good to be true"? what does that mean? lecture slides 6 Games, page 28

Properties of $\alpha - \beta$ $\alpha - \beta$ pruning is guaranteed to give the same result as minimax, but speeds up the computation substantially.

Good move ordering improves effectiveness of pruning.

With perfect ordering, time complexity = $O(b^{m/2})$. To prove that a bad move is bad, we only need to consider one good reply, but to prove that a good move is good, we need to consider all replies.

This means that $\alpha - \beta$ can search twice as deep as plain minimax. An increase in search depth from 6 to 12 could change a very week player into a quite strong one.

6.4 Chess

Deep Blue defeated world champion Gary Kasparov in a six-game match in 1997.

Traditionally, computers played well in the opening (using a database) and in the endgame (by deep search DEFINITION FOR DEEP SEARCH PLZ) but humans could beat them in the middle game by "opening up" the board to increase the branching factor. Kasparov tried this, but Deep Blue was so fast that it was still able to in.

Some experts believe that Kasparov should have been able to defeat Deep Blue, but today chess programs stronger than Deep Blue are running on standard PCs (these programs rely on quiescent search, transposition tables, and pruning heuristics) and could definitely beat the strongest humans.

6.5 Checkers

Chinook failed to defeat human world champion Marion Tinsely prior to his death in 1994, but has beaten all subsequent human champions. Chinook used an endgame database defining perfect play for ass positions involving 8 or fewer pieces on the board - a total of 443,748,401,247 positions. This database has since been expanded to include all positions with 10 or fewer pieces (38 trillion positions).

In 2007, Jonathan Shaeffer released a new version of Chinook and published a roff that it will never lose, His proof method fills out the game tree incrementally, ignoring branches which are likely to be pruned. After many moths of computation, it eventually converges to a skeleton of the real (pruned) tree which is comprehensive enough to complete the proof.

6.6 Go

The branching factor for Go is greater than 300, and static board evaluation is difficult.

Traditional Go programs broke the board into regions and used pattern knowledge to explore each region.

Since 2006, new "Monte Carlo" players have developed using UCB search (TODO WHAT EVEN IS THIS?). A tree is built up stochastically. After a small number of moves, the rest of the game is played out randomly, using fast pattern matching to give prefence to "urgent" moves. Results of random playouts are used to update statistics on early positions.

Computers are now competitive with humans on 9x9 boards, but humans still have the advantage on 19x19 boards.

6.7 Stochastic Games

In stochastic games, chance is introduced by dice, card-shuffling, etc.

Expectimax is an adaptation of Minimax which also handles chance nodes:

if *node* is a chance node

    return average of values of successor nodes

Adaptations of $\alpha-\beta$ pruning are possible, provided the evaluation is bounded.

6.8 Partially Observable Games

Card games are partially observable because some of the opponent's cards are unknown.

This makes the problem very difficult, because some information is known to one player but not the other.

Typically, we can calculate a probability for each possible deal.

The idea is therefore to compute the minimax value of each action in each deal, then choose the action with the highest expected value over all deals.

GIB, the current best bridge program, approximates this idea by

1. generating 100 deals consistent with bidding informatio
2. picking the action that wins most tricks on average

Chapter 7

Motion Planning

7.1 Motion Planning Approaches

7.1.1 Unknown Environments (on-board sensors only)

Occupancy Grid

Divide environment into a Cartesian grid. For each square in the grid, maintain an estimate of the probability of an obstacle in that square.

Potential Field

Treat robot's configuration as a point in a potential field that combines attraction to the goal, and repulsion from objects. Very rapid computation, but can get stuck in local optima, thus failing to find a path.

Vector Field Histogram

Uses a continuously updated Cartesian histogram grid and a Polar histogram based on the current position/orientation of the robot. Candidate valleys are generated (contiguous sectors with low obstacle density). Candidate valleys are then seleced based on proximity to target direction.

7.1.2 Known Environments (overhead cameras)

Delaunay Triangulation

Applicable in situations where the environment is well mapped by overhead cameras (museums, shopping centres, robocup soccer field)

Add line segments between the closest points of obstacles, and sort them according to length in ascending order.

Do not add a segment that crosses an existing segment.

Prune arcs that are too small for the robot to traverse

A* Search can then be applied on the resulting graph.

Parameterized Cubic Splines

Assume that each path segment is of the form
$P(t) = \binom{P_x(t)}{P_y(t)} = \binom{a_x}{a_y}t^3 + \binom{b_x}{b_y}t^2 + \binom{c_x}{c_y}t + \binom{d_x}{d_y}$

Solve these equations for a specified position and velocity at the beginning ($t = 0$) and the end ($t = s$) of the segment. Traditional method was to set $s = 1$. We instead try to minimise s (total time for segment) while satisfying kinematic contraints:

$$|\frac{P\prime\prime(t)}{A} + \frac{P\prime(t)}{V}|^2 \leq 1, \text{ for } 0 \leq t \leq s,$$

where A and V are the maximal acceleration and velocity of the robot.

7.1.3 Minimizing Time instead of Distance

For the problem of a soccer robot getting to the ball, or a wheeled robot navigating a maze, the path with the shortest distance might not be the path that is quickest to traverse. By speeding up and slowing down, the robot could traverse a path with long straight stretches faster than a shorter path with lots of twists and turns.

This requires more work than just path-finding:

1. Delaunay Triangulation
2. A* Search, using paths composed of Parameteric Cubic Splines
3. Smooth entire curve with Waypoint Tuning by Gradient Descent

Part V

Week 5

Chapter 8

Constraint Satisfaction Problems

8.1 Constraint Satisfaction Problems (CSPs)

Constraint Satisfaction Problems are defined by a set of variables X_i, each with a domain D_i of possible values, and a set of contraints C.

The aim is to find an assignment of the variables X_i from the domains D_i in such a way that none of the constraints C are violated.

Path Search Problems and CSPs are significantly different. It is difficult to know the final state for a CSP, but how to get there is easy. Knowing the final state for a Path Search is easy, but it's difficult to get there.

8.1.1 Example: Map Colouring

Variables: WA, NT, Q, NSW, V, SA, T
Domains: $D_i = red, green, blue$
Constraints: adjacent regions must have different colours

The solution is an assignment that satisfies all of the contraints, e.g. WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green
TODO add images

8.1.2 Example: n-Queens Puzzle

Put n queens on an n-by-n chess board so that no two queens are attacking each other.

Describing the puzzle as a CSP:
First, simplify the problem: 4-Queens puzzle.
Assume one queen in each column. Which row does each one go in?
Variables: Q_1, Q_2, Q_3, Q_4
Domains: $D_i = 1, 2, 3, 4$
Constraints:
$Q_i \neq Q_j$ (cannot be in the same row) $|Q_i - Q_j| \neq |i - j|$ (or same diagonal)

8.1.3 Example: Cryptarithmetic

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

Variables: D E M N O R S Y
Domains: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$
Constraints: $M \neq 0, S \neq 0$ (unary contraints)
$Y = D + E$ or $Y = D + E - 10$, etc.
$D \neq E, D \neq M, D \neq N$, etc.

Cryptarithmetic with Hidden Variables

We can add hidden variables to simplify the constraints.

	T	W	O
+	T	W	O
F	O	U	R

Variables: F T U W R O X_1 X_2 X_3
Domains: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Constraints: F, T, U, W, R, O are all different
O + O = R + $10X_i$, etc.

8.1.4 Example: Sudoku

Come on. You know Sudoku.

8.1.5 Real-World CSPs

- Assignment problems (e.g. who teaches which class?)
- Timetabling problems (e.g. which class is offered when and where?)
- Hardware configuration
- Transport scheduling
- Factory scheduling

8.1.6 Types of Constraints for CSPs

Unary constraints involve a single variable, e.g. $M \neq 0$

Binary constraints involve pairs of variables, e.g. $SA \neq WA$

Higher-order constraints involve 3 or more variables, e.g. $Y = D + E$ or $Y = D + E - 10$

Inequality constraints on continuous variables, e.g. $EndJob_1 + 5 \leq StartJob_3$

Soft constraints (preferences) e.g. 11 am lecture better than 8am lecture.

8.1.7 Standard search formulation

Let's start with a simple but slow approach and then see how to improve it. States are defined by the values assigned so far.

Initial state the empty assignment

Successor function assign a value to an unassigned variable that does not conflict with previously assigned variables ⇒ fails if no legal assignments (not fixable)

Goal test the current assignment is complete

This is the same for all CSPs. Every solution appears at depth n with n variables ⇒ use DFS.

8.1.8 Backtracking search

Variable assignments are commutative; [WA = red then NT = green] ≡ [NT = green then WA = red]. Thus, we only need to consider assignments to a single variable at each node. DFS for CSPs with single-variable assignments is called Backtracking search, and is the basic algorithm for all CSPs. It can solve n-Queens for $n \approx 25$.

There are a number of improvements to backtracking search.

General-purpose heuristics can give huge gains in speed:

1. which variable should be assigned next?
2. in what order should its valuen be tried?
3. can we detect inevitable failure early?

Minimum Remaining Values (MRV)

Choose the variable with the fewest legal values.

Degree Heuristic

The degree heuristic acts as a tie-breaker for MRV variables.

The idea is to choose the variable with the most constraints of remaining values.

Least Constraining Value

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

Combining MRV, Degree Heuristic and Least Constraining Value makes 1000-Queens possible.

Forward Checking

The idea is to keep track of remaining legal values for unassigned variables and terminate search when any variable has no legal values.

Constraint propagation Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures. Constraint propagation repeatedly enforces constraints locally. Thus, while Forward Checking will terminate when a variable has no legal values, constraint propagation will terminate one step earlier, when (for example) two variables will be assigned the same value.

Arc Consistency

Simplest form of constraint propagation makes each arc consistent. $X \to Y$ is consistent if for every value x of X there is some allowed value y. If X loses a value, neighbours of X need to be rechecked. For some problems, it can speed up the search by a huge factor. For others, it may slow the search due to computational overheads.

8.1.9 Local Search

Local Search, or Iterative Improvement, is another class of algorithms for solving CSPs.

These algorithms assign all variables randomly in the beginning (thus violating several constraints) and then change one variable at a time, attempting to reduce the number of violations at each step.

Hill-climbing by min-conflicts

Randomly select any variable and select values by the min-conflicts heuristic: choose the value that violates the fewest contraints.

Phase transition in CSPs Given a random initial state, hill climbing by min-conflicts can solve n-Queens in almost constant time for arbitrary n (e.g. $n = 10,000,000$) with high probability.

In general, randomly-generated CSPs tend to be easy if there are very few or very many constraints. They become extra hard in a narrow range of the ratio

$$\mathrm{R} = \frac{numberofconstraints}{numberofvariables}$$

TODO insert image plz.

Flat regions and local optima

TODO insert image plz

Sometimes we have to go sideways or backwards in order to make progress towards the actual solution.

8.1.10 Simulated Annealing

Simulated annealing is stochastic hill-climbing based on the difference between evaluation of previous state h_0 and new state h_1. If $h_1 < h_0$ then definitely make the change, otherwise make the change with probability $e^{-(h_1-h_0)/T}$ where T is a "temperature" parameter.

Simulated annealing reduces to ordinary hill climbing when $T = 0$, and becomes random when $T \to \infty$. Sometimes, we gradually decrease the temperature during the search (TODO WHY?).

Chapter 9

Evolutionary Computation

We use principles of natural selection to evolve a computational mechanism which performs well at a specified task. Start with a randomly initialised population, and then repeat a cycle of

1. evaluation
2. selection
3. reproduction + mutation

We can use any computational paradigm, with appropriately defined reproduction and mutation operators.

9.1 Evolutionary Computation Paradigms

TODO this section is very empty - anything cool to add?

9.1.1 Bit String Operators (Genetic Algorithm)

We can use bitmask/bitstrings to represent gene values of an organism. The correlation goes something like this (source: http://www.doc.ic.ac.uk/~sgc/teaching/pre2012/v231/lecture16.html)

Genetics Terminology	Genetic Algorithm Terminology	Species Terminology
	Population	Population
Chromosome	Bit string	Organism
Gene	Bit (0 or 1)	
	Selection	Survival of the fittest
Crossover	Recombination	Inheritance
Mutation	Mutation	

Crossovers

A crossover, representing inheritance of genes, can be done in different ways.

You can use a one-point crossover, where additions to the population are modified by taking parts of two 'parents'. We choose some arbitrary point in the string and the 'children' become products of the parent strings, like so:

11101001000 00001010101 become 11101010101 and 00001001000
You can also use a two-point crossover, like so:
1101001000 0000101010101 become 11001011000 and 00101000101

Point Mutation

Point mutation represents the mutation of a gene.
11101001000 becomes 11101011000

9.1.2 S-expression trees (Genetic Programming)

S-expression trees can be used similarly. Recombination becomes swapping subtrees of the S-expressions, to give different results upon evaluation.

9.1.3 Continuous Parameters (Evolutionary Strategy)

For Continuous Parameters, reproduction is just copying, and mutation is adding random noise to weights (or parameters) from a Gaussian distribution with a specified standard deviation. Sometimes, the standard deviation can evolve as well.

9.1.4 Lindenmayer System

TODO what is this? :P