Agents (context)

Agents

- An action is rational iff it maximises the expected value of the performance measure given the agent's prior knowledge (the knowledge it is "born" with) and its experiences up to the point of taking the action
    - Rational actions may prove to have been unwise with the benefit of hindsight or decisions that seemed irrational may have good outcomes through chance or unobserved factors
- Belief state of an agent = the information about its past it has access to (its memory)
- A simple reflex agent is an if else if … block from current precept to a command
- A reflex agent with state is an if else if … block from current belief state to a command and new belief state
- A goal based agent has an understanding of how its actions affect the state of the world given its current state and what state of the world constitutes its end goal — using this they can construct a sequence of actions that move the world through a sequence of states that results in the end goal
- A utility based agent is a goal based agent that is able to weigh up the merits of different paths towards the goal
- A learning agent will sometimes take actions with the primary knowledge of gaining knowledge (exploration) rather than moving directly towards its goal (exploitation)

Least cost path

Search

- **Frontier = set of nodes we have discovered but not explored**
  - We can add to this fairly blindly and decide at the point a node is chosen from it whether we actually want to explore or whether we will pick again straight away
  - Most search algorithms only vary on the ordering of the frontier
- Stereotypical search algorithms are called **tree search algorithms — produce a tree of the connected component they start in**
  - Some tree-search algorithms may not terminate if the input graph contains (directed) cycles, breadth-first tree-search will always terminate
- **Having the frontier store the path to the node to be visited instead of just the node converts a tree search algorithm into a graph search algorithm** and remove the risk of non-termination
- **An algorithm is admissible iff it gives a least-cost path to a goal whenever a path to a goal exists**

A breadth-first tree-search algorithm

```
Add start to tree and frontier
while True
    curr = frontier.dequeue()
    if is_goal(curr)
        return tree
    for neigh in curr.neigbours()
        Add neigh to tree below curr
        frontier.enqueue(neigh)
```

A depth-first tree-search algorithm

```
Add start to tree and frontier
while True
    curr = frontier.pop()
    if is_goal(curr)
        return tree
    for neigh in curr.neigbours()
        Add neigh to tree below curr
        frontier.push(neigh)
```

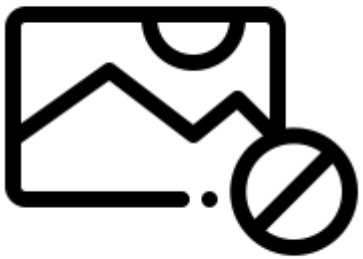
A depth-first graph-search algorithm

```
frontier.enqueue((start,))
while True
    curr = frontier.pop()
    if is_goal(curr.last)
        return curr
    for neigh in curr.last.neighbours()
        Add neigh to tree below curr
        frontier.push(curr.append(neigh))
```

A breadth-first graph-search algorithm

```
frontier.enqueue((start,))
while True
    curr = frontier.dequeue()
    if is_goal(curr.last)
        return curr
    for neigh in curr.last.neighbours()
        Add neigh to tree below curr
        frontier.enqueue(curr.append(neigh))
```

(Breadth-first) Tree-search example



- Note nodes are added to the tree everytime they are discovered

Uninformed search

- Uninformed search explores the entire graph aimlessly until it stumbles across the goal
- Breadth-first search and depth-first search have similar time complexities but breadth-first search has much worse space complexity
- Breadth-first search is admissible if edge weight is a non-decreasing function of depth (e.g. all edge weights are the same)
- **Lowest-cost first search** (Dijkstra's algorithm stopping once a goal is reached)**: Frontier is a priority queue and priorities are current cost of the path**
    - Use subscripts on tuples to denote priorities and superscripts to show the calculation used to obtain the subscript
    - As it uses a priority queue, it has similar characteristics to breadth-first search
    - **Admissible if all edges weights are positive and there are finitely many edges**
- **Dynamic programming (Dijkstra but working backwards from goal(s) to allow us to use any start point): dist(n) = distance of the shortest path from n to a goal = 0 if n is a goal, min {<n, m> (arc cost of nm) + dist(m) for all neighbours m of n} otherwise** — work backwards from goals to compute dist for every node
    - Same admissibility as lowest-cost first (all edges weights are positive and there are finitely many edges)
    - If we often need to find the shortest path from a wide range of start nodes to static goal nodes in a static graph, we can compute the table once then store it and use it quickly every time after

Informed search

- Informed search uses a (goal-based) heuristic (a cheap to compute function from a node to an estimate of the cost of reaching the goal from that node) to guide the search in a sensible direction
- **A heuristic is admissible iff it is non-negative and consistent (never greater than the cost of the lowest cost path from the node to the goal)**
  - **A heuristic is consistent iff for every edge (n, n') with cost c(n, n)', $h(n) \leq c(n, n') + h(n')$ and $h(n) = 0$ if n is a goal**
- Heuristic depth-first search: Uses the heuristic to decide which order to push the neighbours onto the stack in
  - Often gives a "good-enough" solution but still has no optimality guarantees
- **(Greedy) best first search: Frontier is a priority queue with priorities of heuristic value**
  - Often gives a "good-enough" solution but has no optimality guarantees
  - Similar space and time to breadth-first search
- **A* search: Frontier is a priority queue with priorities sum of current cost of path (lowest-cost first) and heuristic value (best-first)**
  - **Frontier is a priority queue with priorities of heuristic value**
  - **Admissible if all costs are positive, there are finitely many edges, and heuristic is admissible**

Pruning

- **Cycle pruning: If we discover a path $(n_0, \ldots, n_k)$ but $n_k \in (n_0, \ldots, n_{k-1})$ we can skip adding this path to the frontier for any search algorithm without affecting admissibility** (given the reasonable assumption we don't need to worry about negative cycles)
- Multiple-path pruning: We only want to ever store the best path to each node we have found so far but for some algorithms by the time we find a better path the worse path may have been built upon
    - Any multiple-path pruning algorithm is a cycle-pruning algorithm
- **The closed list = a hash set of nodes that lay at the end of each path that has been popped from the frontier**
- **Multiple-path pruning that preserves admissibility if paths will never need to be replaced (e.g. lowest-cost first): When a path is selected from the frontier, check if the end node is in the closed list, if it is skip over the path, otherwise proceed as normal (and so add it to the closed list)**
- Multiple-path pruning that preserves admissibility of any algorithm: If end node of path is in closed list, compare costs, if we would be better off keeping the new path and getting rid of the old path loop through frontier and update the path segment of all paths that used the old path to use the new one

Search tricks: Bounding

- **Bounded depth-first search does not add paths to the frontier if their cost exceeds the bound**
- **Iterative-deepening search:**

```
bound = 0
do
    sol = bounded_dfs(G, bound)
    bound++
while (sol is null)
return sol
```

  - The repeated computation does not asymptotically affect the time (is the same (exponential) as depth/breadth-first search)
  - Admissible if breadth-first search is and costs are integer
    - Uses the (linear) space of depth-first search instead of the (exponential) space of breadth-first search

Search tricks: Bounding: Branch and bound

- **Branch and bound:**

```
Let heurisitic_bounded_dfs be bounded_dfs with cut-off
condition cost + heuristic >= bound instead of cost >
bound
bound = +inf
sol = null
do
    prev_sol = sol
    sol = heurisitic_bounded_dfs(G, bound)
    bound = sol.cost
while (sol is not null)
return prev_sol
```

  - The (good) admissibility of A* with the (linear) space of dfs instead of the (exponential) space of A*

Search tricks: Directionality

- Backwards search: Let G be the input graph, s be the start node, and T be the set of goal nodes.
  Generate the inverse graph $G' = (V \cup \{t'\}, \{(n_2, n_1): (n_1, n_2) \in E\} \cup \{(t', n): n \in T\})$.
  Search in G' from t' to s
  - Useful if the branching factor in G' is lower than in G
- Bidirectional search: Search forwards and backwards at the same time

CSPs

Elementary algorithms

- **Scope of a constraint = the set of variables involved in the constraint**
- Generate and test: Naive enumeration through assignment space stopping if we find an assignment that satisfies all constraints
- **Backtracking: Set variables one at a time, after each variable is set, check all the constraints for which their scope is a subset of the set of variables that have been set, if a constraint (whose scope is fully initialised) is violated then undo the most recent assignment, if all constraints are (initialised and) satisfied then return the assignment**
    - Exploits the fact that if a constraint is violated by a partial assignment then it will be violated by all supersets of that assignment
- *SAT can easily be reduced to CSP (and vice versa if the CSP is over finite domains), so CSP is NP-complete — not withstanding P=NP, we can't do much better than this exponential time backtracking algorithm but we can use heuristics for the arbitrary choice of variable and ordering of values of variables to improve the real world performance*
- **Minimum remaining values (MRV) heuristic: Pick the variable with the fewest remaining legal values**
    - Intuition: Try to violate a constraint as quickly as possible so we can rule out this class quickly if we are going to do so
    - **Degree heuristic: If we have a tie in MRV, chose the variable that is involved in the most constraints**
        - Intuition: Narrow down search space as much as possible for future iterations
- **Least constraining value heuristic: Explore the value that ruled out the fewest values of other variables**
    - Intuition: Go down the path with the best chance of success

CSP solving as graph search

- Nodes are assignments of some values to some variables
- Start node is the empty assignment
- is_goal is all constraints are (initialised and) satisfied
- There is an edge (u, v) iff |ass(v)\ass(u)| = 1 and neither u nor v violated any constraint (out of the ones whose scope is fully initialised)
- Backtracking corresponds to depth-first search

Arc consistency

- We can represents the constraints as a graph (the **constraint network**):
  - **Circular node for each variable**
  - **Rectangular node for each constraint**
  - **Edges from each variable to each constraint that involves it**
  - **Binary constraints can be abbreviated by annotating the edge between the variables with the constraint(s) instead of subdividing it with constraint nodes**
- **Let a constraint c have scope (uses variables) {X, Y_1, …, Y_k}. The arc from X to c is arc consistent iff for every legal valuation of X there exists valuations for each Y_i s.t. the constraint is satisfied**
  - **If an arc is not arc consistent, it can be made to be by pruning (removing) from the domain of X the values of X for which there exists no valuations of the other variables that satisfy the constraint — this (obviously) preserves the solutions to the CSP**
    - **Have to repeat this process for the constraints between X and that Y_i if some Y_i has its domain pruned subsequently**
- **If a constraint c has scope {X}, then the arc from X to c is domain consistent iff every value of X satisfies c**
- **A network is arc consistent if all its arcs are (domain and) arc consistent**
- **An arc consistent network is useful for CSP solving:**
  - **If a domain is empty, there is no solution**
  - **If each domain has exactly one value, there is a unique solution and this is it**
  - **If each domain has at least one value but some have more than one value, we still do not know whether there is a solution but we have simplified the problem for further search**

Solving from an ambiguous arc consistent network

- **Domain-splitting: Pick a variable** (it is generally sensible to pick the one with the smallest non-singleton domain) **and split its domain** (e.g. into halves) **to create sub-problems and run arc consistency on each sub-problem (can run domain splitting if still ambiguous) — a solution to any subproblem is a solution to the original problem**
- **Variable elimination:**

```
Let each constraint be represented as the set of all tuples that obey it
Let consWith(X) returns all constraints that contain the variable X
VarEl(vars, cons)
    if |vars| = 1
        return intersection over consWith(vars[0])
    Pick arbitrary X ∈ vars
    R1 = natural join over consWith(X)
    R2 = R1 with X projected out
    R3 = VarEl(vars\{X}, (cons\consWith(X)) ∪ R2)
    return R1 natural join R3
```

  - Performance depends a lot on choice of order in which to eliminate variables
    - *Finding an optimal order would be solving CSP which is NP-hard but there are heuristics*

Local search

Local search

- Sometimes instead of finding a path that minimizes edge weights, we want to find an end state that maximizes (or equivalently minimizes) some objective function e.g. weights for a neural network, a solution to a (potentially soft) CSP etc.
- Local search decides which neighbour of the current state to move to based (more or less) only on the value of the current state

Hill climbing

- Iterative improvement: Repeatedly move to some neighbour of the current state (i.e. a similar state that also satisfies any hard constraints) that improves the value
- **Hill climbing (greedy local search)** is a natural implementation of iterative improvement where we **move to the neighbour with maximal value stopping once all neighbours have strictly lower value than the current state**
- Local maximum = a plateaux for which in all directions the value is decreasing at the point it stops being flat
- Shoulder = a plateaux in which there is a direction in which the value eventually switches from flat to increasing
- In a plateaux moves randomly with a limit on number of steps to try to escape — getting stuck at local maxima is the cost of greediness but the random movement is helpful at shoulders
- May oscillate at ridges — significantly slows the process
    - Ridge = small first derivative, large second derivative
- If our variables are continuous we can use gradient descent: Change value of variable by some small constant (negative in the case of decent) multiple of the partial derivative of the objective function with respect to the variable at its current value
    - Even worse with ridges than hill climbing
    - Still gets stuck at local optima

Randomized methods

- **As they don't store paths, local searches may cycle, to mitigate against this store a tabu list of the last t assignments, use probability 0 if it's in the tabu list**
- **Stochastic local search: Greedy local search with random restart (with low probability: store best solution found so far, reinitialize every parameter (move to a completely random point), then continue as normal updating the best solution if we become the best) and random steps (with low probability: move to a random neighbour instead of the one given by greediness then continue as normal) to try to let it to escape local minima**
- **Simulated annealing: For a current position n, pick a random neighbour n'. Let the objective function values be h(n) and h(n'), we move to n' with probability 1 if h(n) ≥ h(n') and probability $e^{(h(n)-h(n'))/T}$ otherwise (h(n) < h(n')) where T is the temperature** — for maximization problems, negate h
  - *Annealing = the process of melting a metal and allowing it to cool as slowly as practicable to allow it to reach as low an internal energy as practicable*
    - *Solid metal has a crystal structure, molten metal does not, the slower the temperature decrease is the more uniform the final crystal structure will be and so the stronger the metal will be*
  - **Temperature is decreased over time** in accordance with some annealing schedule

Parallel searches

- Parallel search: Run k searches from different starting points, take the best result
  - Very similar effect to random restarts but runs quicker if we have a multicore computer
- **Beam search:** Same idea as parallel but **start off with one search and each step expand the k most promising of the neighbours of all the currently selected nodes (thats k overall, not k each)**
  - Survival of the fittest type idea but no randomness, only ever keeps the current fittest
  - K = 1 is plain greedy descent
  - K = +inf is breadth first search
- **Stochastic beam search — probabilistically chose the k individuals**, order them randomly and cycle through each node n in turn choosing it with **probability $e^{-h(n)/T}$** where T is a temperature until we have picked k nodes
  - Randomness gives diversity and so is better at avoiding getting stuck in local maxima than ordinary beam search
  - Analogous to asexual reproduction

Genetic algorithms

- **Each successor state is derived from two parents** — analogous to sexual reproduction
- **Start with a population of k randomly generated states (individuals)**
- Represent each individual's parameters as a fixed length string over a finite fixed alphabet (a chromosome)
- **Use fitness function (i.e. objective function) to decide which k (not necessarily all distinct) individuals will reproduce** — they pair up in the order they are chosen in
  - **Can cull individuals with fitness below a threshold**
  - **Probabilistic: As in stochastic beam search, choose with probability $e^{-f(n)/T}$ where T is a temperature and f is the fitness function**
    - Changes will naturally become smaller over time even if temperature is not turned down, as advantageous genes will become widespread
  - **Tournaments: Split into a fixed number of random groups and use the fittest from each group**
- **Reproduction process:**
  1. **Pick a random** (1-indexed) **crossover point**
  2. **Concatenate chromosome of one parent up to and including crossover point and chromosome of the other from crossover point and vice versa to produce chromosomes of two children**
  3. **Randomly mutate each character in chromosome of each child with small probability**

Adversarial search

Minimax

- We say that a game tree tree for m players with mn layers is mn-ply but only n moves deep (we define a move as every player getting one turn)
- As well as min and max nodes where the outcome is controlled by the respective player, we can have chance nodes where the outcome is random
- **minimaxValue(n) = utility for max of the state n assuming both players play optimally = Utility(n) if n is a terminal node, max {minimaxValue(s): s ∈ children(n)} if n is a max node, min {minimaxValue(s): s ∈ children(n)} in n is a min node, $\sum_{s \in children(n)} P(s) \times minimaxValue(s)$ if n is a chance node**
- Working from the leaves up to the root we can compute utilities for all the nodes using this then always chose the move that moves us to the child (AWLOG we are playing as max) with the highest minimaxValue
- Minimax is optimal even if the opponent is not optimal (not also using minimax), provided the full tree is explored (no alpha-beta pruning or cut-offs)
- *Can generalise to more than two players using vectors of utilities where player i maximises the ith element of the vector*

Minimax: Alpha-beta pruning

- **Alpha(n) = max {Beta(c): c is a child of n} if n is a non-terminal Max node, Utility(n) if n is a terminal max node**
- **Beta(n) =min {Alpha(c): c is a child of n} if n is a Min node, Utility(n) if n is a terminal min node**
- Each node's alpha/beta is monotone
- **At each expansion step for a node check alpha and beta of it and its parent, if alpha > beta, can prune at it (stop expanding it)**

Monte carlo tree search

- Minimax computes the exact value of states but evaluates the whole tree (or a substantial portion thereof even if alpha-beta is used), monte carlo tree search estimates the value of states instead
- Playout = a simulation of a games starting from a certain state
- Selection policy decides which moves we already have data on to play — balance exploration against exploitation
- Playout policy decides which moves are made during the simulations

1. **Tree starts as a single node (the current game state)**
2. **Chose moves using selection policy until we reach a leaf (i.e. immediately to start with)**
3. **Attach a child to the leaf and perform a playout on this child using the playout policy** — note we don't add new nodes to the tree for the states reached in the playout
4. **For each node on the path we took down the tree, increment number of games played and if we won the playout increment the number of games won**
5. **Go back to root and repeat from step 2**
6. **Once we have done enough iterations, play the move that results in the child of the root <u>with the most number of games played (not most games won!)</u>**
    - Number of games won can be used by policies but is not used here

Knowledge-based systems

Knowledge base (KB)

- Coercion = taking an action that doesn't make much progress towards the goal but does narrow down the possible states and so reduces uncertainty for actions in the future
- **KB ⊨ α (knowledge base entails α) iff for every state s.t. KB is true, α is true**
  - Assuming knowledge is correct, α being entailed means α is definitely true but α not being entailed does not necessarily mean α is false
  - *Implication is a part of some logics whereas entailment is used to reason about logics but the difference is largely academic as they are equivalent under the normal axioms of logic*
- **KB $\vdash_i$ α iff α can be derived from the knowledge base using inference procedure i**
- **i is called sound iff KB $\vdash_i$ α ⇒ KB ⊨ α**
  - If i is not sound, it may make inferences that are not actually true
- **i is called complete KB $\vdash_i$ α ⇐ KB ⊨ α**
  - If i is not complete, there may be inferences that are correct but i is unable to make
- The simplest inference procedure for propositional logic is **enumeration: iterate through all combinations of the valuations of the atomic propositions, for each one evaluate KB and α and check that whenever KB=T we have that α=T, if our check never failed KB ⊢ α**

Knowledge representation

- Knowledge is represented as vectors (o, a, v) where o is drawn from the set of objects O, a is drawn from the set of attributes of objects A, and v is drawn from the set of values of attributes V
  - O, A, and V are said to together make up the vocabulary
  - (o_1, a_1, v_1) and (o_1, a_2, v_2) and … and (o_1, a_n, v_n) abbreviates to **$(o_1\ (a_1\ v_1)\ (a_2\ v_2)$ … $(a_n\ v_n)$ which reads in FOL as an existential claim — there exists an x such that x is an o_1 and x's a_1 is v_1 and x's a_2 is v_2 …**
- The agent's current store of facts is called its working memory
  - Contains the current state of the knowledge base (facts)
  - Contains rules that define how the working memory evolves
  - Each fact (symbol) is called a working memory element
- **A production rule is of the form if P_1 and P_2 and … and P_m, then perform actions Q_1 and Q_2 and … and Q_n**
  - **<u>Note we only have conjunction — for disjunction we must create multiple rules</u>**
  - The conditions P_1 … are object-attribute-specification vectors (same structure as object-attribute-value vectors for facts)
  - **If for each $P_i = (o_i\ (a_1\ s_1)$ … $(a_n\ s_n))$ there exists a working memory element x with type o_i and a_1 of x_1 is s_1 and … and a_n of x_n is s_n, then the actions will be performed**
  - **Actions can and do add/delete/modify facts in working memory**

Inference procedures

- It is not practicable to store every true proposition in the working memory — we need an inference procedure to allow us to determine whether a particular fact (a goal) can be derived from the facts and rules in the knowledge base
- **Forward chaining (bottom-up ground proof procedure): Pick a rule that matches the working memory (using a conflict resolution mechanism if there are multiple options). Fire the rule (update the knowledge base in accordance with the rules actions). Stop and check for the goal only once no rules match or a rule is ran with actions halt**
    - Sound, complete, and easy to implement but may infer lots of unnecessary things
    - The series of rules that fire is called an inference chain
- **Backward chaining (top-down definite clause proof procedure): Pick an element of the goal and find a rule that would result in that element being added to KB, update the goal to replace that element with the antecedent of the rule. Repeat until either the goal is made only of elements that are in the KB or we cannot find a suitable rule to continue**
    - Almost sound and complete but can get stuck in a cycle if there are bidirectional implications in the knowledge base
- **Conflict resolution mechanisms:**
    - **Specificity — fire the rule with the most Ps**
        - Allows exceptions to rules to be added without having to update the original rule to not fire when the exception does
    - **Recency — fire the rule whose antecedent uses the most recent WME**
    - **Refractoriness — Allow each rule to only run once on each WME — typically implicit**

Planning

Planning

- Each step of search produces an ordered sequence of actions whereas each step of planning produces a set of constraints on actions — planning is more flexible and so can allow quicker computation (especially as many real world systems have large branching factors and no good heuristics)
- Simple planning agent:

```
while True
    Tell KB current precept
    If we don't currently have a plan
        Ask KB for a goal
        Use a planner to create a plan for the goal
Pop next action from current plan
Carry out the action and tell the KB we have taken it
```

Situation calculus

- Situation calculus is first order logic (predicate logic in cs130 terms) extended to include fluents (functions or predicates that vary with time and so take the world state as an argument) — ordinary functions and predicates are called eternals
- **Each action has a possibility axiom** *(aka action precondition axiom)***: Poss(a, s) ⇔ Precondition(..., s) where s is a snapshot of the state of the world**
  - Does not necessarily mean it would be sensible to take a, only that it is possible to!
- **Actions can have effect axioms which describe how the world will change if we take an action a**
  - **Poss(a, s) ⇒ ...:** If we take the action and it is possible then … will happen
- The frame problem: we need to explicitly describe which properties of the world are certainly not changed by an action as well as which certainly are
- **Each fluent has a successor state axiom which describes every way that is can become true and every way it can become false**
  - These are similar to effect axioms but solve the frame problem
  - F(...) ⇔ …

STRIPS

- Closed world assumption: If the state description does not contain a certain literal, that literal is false
- States = conjunctions of function-free (i.e. possibly negated but otherwise bare) literals
- Each operator has a precondition, an action, and an effect
  - Precondition is a conjunction of positive (function-free and not negated) literals
  - Effect is a conjunction of function-free literals
- Operator schema = an operator with variables — operator schemas are to operators are predicates are to propositions
- An operator is applicable in a state s iff there is an assignment to variables left free under s such that the precondition of the operator is true

POP: Principles

- **Progression = search forward from initial situation to goal**
- **Regression = search backward from goal to initial situation — POP is a regression planner**
- A plan consists of:
  - A set of steps and ordering constraints on them
    - Search produces a totally ordered sequence of steps whereas plans can be only partially ordered
  - A set of variable bindings
  - A set of causal links (which steps achieve which parts of the preconditions of which other steps, give rise to order constraints)
- **Open condition = precondition that has not yet been fulfilled**
- **A plan is complete iff every precondition is achieved iff every precondition is an effect of an earlier step and it is not possible for any steps between to undo this effect**
- **A plan is consistent iff there are no contradictions in the orderings or bindings**
- Partial plan = a plan where some variables are still free
- Fully initiated plan = a plan in which all variables are bound
- A partial plan can be made less incomplete by:
  - Creating a causal link from an existing action to an open condition
  - Adding a step to fulfil an open condition
  - Reordering existing steps to enable a causal link from an existing action to an open condition to be created

POP: Algorithm

- **Create a minimal plan (that is only start and finish with ordering start < finish and whatever variable bindings they make)**
  while plan is not complete
  Pick a step S_{need} with a precondition c that has not achieved [e.g. precondition of finish]
  try
  Pick a step S_{add} that has effect c (or can have effect c through certain bindings of free variables that maintain consistnecy), record this causal link $S_{add} \rightarrow S_{need}$, the ordering constraints start $\leq S_{add} \leq S_{need} \leq$ finish, and any variable bindings that were necessary
  Resolve any broken casual links
  except NotPossible
  backtrack
- Cloberer = an intermediate step that might destroy a causal link
- Demotion: Move the clobberer to before the step that creates the link
- Promotion: Move the clobberer to after the step that uses the link

No plan survives contact with the enemy

- The real world is not fully observable, static, and deterministic — our information about the state of the world or the effects of actions may be incorrect by the time we reach the point of taking a certain action
    - If the state of the world diverges from what we expect due to an outside influence, we can plan around this
    - If our information about the effect of our actions on the world is incorrect, unless we have learning, we may be unable to succeed
- **Conditional planning: include sensing actions in the plan and create a subplan for each possible outcome of a sensing action**
    - Most subplans won't be needed
- Replaning (monitoring): Plan for the most likely states and effects, during execution of the plan if something unexpected happens replan then
    - Dynamic nature makes it more efficient than conditional planning
    - **Plan monitoring: At each step check that causal links in the remaining plan are still valid for the current state**
    - **Action monitoring: At each step check that preconditions of the next action are still met**
    - **To replan efficiently, find a best continuation point in original plan for our current state and calculate a miniplan from current state to the best continuation point**
- **Mitigation techniques:**
    - **Coercion — take actions that reduce number of possible states**
    - **Abstraction — ignore unknown details**
    - **Aggregation — treat a large number of unpredictable objects as a single predictable object** (think thermal physics)

Bayesian reasoning

Probability

- Sample space Ω is the finite set of possible states of the world (outcomes) — must be atomic (mutually exclusive and exhaustive)
- (Ω, P) is a probability space iff all of the following hold for all $\alpha, \beta \subseteq \Omega$:
  - $P(\Omega) = 1$
  - $0 \leq P(\alpha) \leq 1$
  - $P(\alpha \vee \beta) = P(\alpha) + P(\beta) - P(\alpha \wedge \beta)$
  - **$P(\neg\alpha) = 1 - P(\neg\alpha)$**
  - **$P(\alpha) = P(\alpha \wedge \beta) + P(\alpha \wedge \neg\beta)$ for any β**
- An event E is a set of outcomes (a subset of Ω) — $P(E) = \sum_{s \in E} P(s)$
- A probability distribution is a function over a variable (called a random variable) that assigns a probability to each value in the domain of the random variable
- A joint probability distribution is a probability distribution over multiple random variables
  - **$P(X, Y) = P(X \wedge Y)$**
- $E[X] = \sum_{s \in \Omega} P(s)X(s)$ where $X(s)$ is the value of X in state s
- $Var[X] = E[(X - E[X])^2] = E[X^2] - E[X]^2$
- A probability density function is a map from reals to non-negative reals with integral over the entire domain of exactly 1
  - $P(a \leq X \leq b)$ = integral from x=a to x=b of p(x) dx where p is the probability density function

Conditional probability

- $P(\alpha \mid \beta) = P(\alpha \wedge \beta)/P(\beta)$
  - $P(\alpha \mid \beta)$ is called the posterior probability of α given the evidence of β
  - $P(\beta)$ is called the prior probability of β
  - Does not necessarily tell us anything about a causal relationship between β and α
- **X and Y are conditionally independent given Z iff $P(X \wedge Y \mid Z) = P(X \mid Z)P(Y \mid Z)$** or equivalently $P(X \mid Y \wedge Z) = P(X \mid Z)$
- **Total probability theorem: Let A be a set of events that partition Ω, then, $\forall B \subseteq \Omega$. $P(B) = \sum_{E \in A} P(B|E)P(E)$**
- **Chain rule: $P(a_1 \wedge \ldots \wedge a_i) = P(a_1)P(a_1|a_2)P(a_3|a_1 \wedge a_2)\ldots P(a_i|a_1 \wedge \ldots \wedge a_{i-1})$**
- Bayes' Rule: Let C be a partition of Ω s.t. $\forall E \in C$. $P(E) \neq 0$. Then, $P(A|B) = P(B|A)P(A)/(\sum_{E \in C} P(B|E)P(E))$
  - E.g. by taking C = {A, ¬A}, **$P(A|B) = P(B|A)P(A)/(P(B|A)P(A) + P(B|\neg a)P(\neg A))$**

Bayesian belief networks (BBNs)

- Inference by enumeration: Let X be the query variable, E be the evidence variables, e be the observed values of the variables in E, and Y be the unobserved variables. **$P(X|e) = P(X, e)/P(e) = ((\sum_{y \in Y} P(X, e, y))/(\sum_{(x, y) \in X \times Y} P(x, e, y))$**
- **Suppose we have a joint probability distribution P of the random variables in a set V. A directed graph G=(V, E) is a bayesian belief network iff (G, P) satisfies the markov condition iff every X in V is conditionality independent (given its parents) from its non-descendants**
- **In a BBN, $P(X_1, X_2, \ldots, X_i) = \prod_{j \in \{1, \ldots, i\}} P(X_j|Parents(X_j))$** — this makes inference linear instead of the exponential of enumeration
  - If G is a directed <u>acyclic</u> graph and $P(V=\{X_1, X_2, \ldots, X_n\}) = \prod_{j \in \{1, \ldots, n\}} P(X_j|Parents(X_j))$, then (G, P) satisfies the markov condition
- **Pearl's Network construction algorithm (creating a BBN):**

```
V = {}
E = {}
while there is a variable v that has not been added yet
    Add v to V
    Add edges to E giving v the minimum number of parents required for v to
obey the markov condition
```

  - As always a good ordering would speed things up (reduce number of edges), but if we knew a good ordering we'd have solved our problem already

Reasoning using BBNs

- **The lack of an edge in a BBN implies independence between those nodes but the presence of an edge does not imply a causal link between those nodes**
- **Diagnostic reasoning = travelling only in opposite direction to arcs in BBN**
  - "Symptom to cause"
- **Predictive reasoning = travelling only in direction of arcs in BBN**
  - "Cause to symptom"
- **Intercausal reasoning = reasoning about all possible causes of an effect**
  - "What is the probability the symptom be explained by a different cause to the given cause"

Variable elimination

- We can interpret probabilities as functions and a BBN as a factorization of the joint probability distribution
- **Call each conditional probability a factor with scope of the variables it uses** e.g. P(X|Y,Z) has scope {X, Y, Z}
- **Each factor can be specified as a table of its value for all its inputs** (relation),
- **Conditioning factors (eliminating variables based on observations): If we have a table for P(X|Y,Z) and we find out (or wish to investigate what would happen if) Z=T, then we can discard the rows that corresponded to combinations where Z=F**
- **Multiplying factors (joining tables): Given P(X, Y) and P(Y, Z) we can find P(X, Y, Z) by taking natural join and folding each pair of the probabilities into a single probability by multiplication**
- **Summing factors (eliminating variables using total probability): Given P(X, Y = t, Z) and P(X, Y = f, Z) we can find P(X, Z) by projecting out y, taking natural join, and folding each pair of the probabilities into a single probability by addition**
- **Variable elimination algorithm:?**
  Use conditioning on the observed variables
  Multiply to combine observed
  Sum out all remaining non-query variables
  Multiply all unmodified factors that involve query to combine query to obtain P(query, observed) parameterized over query
  Return the value at the value of query divided by the sum over all entries (all values of query)

Bayesian agents

- $U(O_i|A)$ = utility of the outcome O_i of the action A
  - Depends on action as well as outcome as actions have side effects not captured in our outcomes e.g. we assign lower utility to administering a drug than watchful waiting if they have the same outcome
- **Expected utility of taking A = EU(A) = $\sum_i P(O_i|A) \times U(O_i|A)$**
- **A Bayesian decision maker will take the action that maximises their expected utility**
  - **Given evidence E, maximise over EU(A|E) = $\sum_i P(O_i|A, E) \times U(O_i|A)$** as utility is independent of evidence but probability of outcome is not necessarily
- **A decision tree contains of chance nodes (depicted as circles), decision nodes (depicted as squares), and leaves (concrete utilities)**
  - **EU(n) = Utility at n if n is a leaf, EU(A) where A is the most recent action on the path to n if n is as chance node, max {EU(a): a is an action available at n} if n is a decision node**
- An influence diagram is a combination of a bayesian belief network and a decision tree — chances nodes are circles, decision nodes and squares, utility nodes are diamonds
  - Can factor out nodes that occur in multiple paths in the decision tree

Reinforcement learning

Principles

- Reinforcement learning is like bayesian agents but with the added complication that we must learn the utility function
- **Sequence of experiences: state, action, reward, state, action, reward, …**
- One of the reasons RL is difficult is the reward received immediately after taking an action could have much more to do with an action any number of time steps in the past than the action that was just taken
- To perform RL we will:
  - Learn **state transition function P(s'|a, s) and reward function R(s, a, s')**
    - Solve using markov decision process
  - Learn $Q^*(s, a)$ using P and R
    - Q = estimate of total value of action a in state s assuming some reasonable policy is followed afterward
    - **Q* = estimate of total value of taking action a in state s assuming an optimal policy is followed afterwards**
  - P, R, and Q are all learnt at the same time and interact with each other
- **Suppose the agent receives a sequence of rewards r_1 …, r_n. Then, time discounted total reward = $V = r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots$ where $0 \leq \gamma \leq 1$ is the discount factor**
  - $V_t = r_t + \gamma V_{t+1}$

Markov decision process (MDP)

- **Markovian assumption: Given knowledge of the current world state, no knowledge about the past is relevant to the future. That is that $P(S_{t+1}|S_0,A_0,...,S_t,A_t) = P(S_{t+1}|S_t, A_t)$**
  - Same assumption as BBN but phrased this specific context
- A Markov decision process is a markov chain (like a BBN (markov condition etc) but undirected and potentially cyclic) augmented with actions and values
- **Circles depict states**
- **Squares depict actions**
- **Diamonds depict rewards**
- **Set S of states**
- **Set A of actions**
- **Function $P(S_{t+1}|S_t, A_t)$ describes dynamics**
- **Function $R(S_t, A_t, S_{t+1})$ describes rewards**
  - **R(s, a, s') = expected reward when agent moves from state s to s' using action a**
- A stationary policy is a function $\pi: S \mapsto A$ which tells the agent which action to take using only the current state
- A policy is optimal iff it has maximal expected discounted reward
- For a fully observable MDP with stationary dynamics and long horizon, there exists an optimal stationary policy

Value iteration

- $Q^{\pi}(s, a)$ = expected value of taking action a in state s then following policy π forevermore
- $V^{\pi}(s)$ = Expected value of following policy π in state s
- **$V^{\pi}(s) = Q^{\pi}(s, \pi(s))$**
- $Q^{\pi}(s, a) = \sum_{s'} P(s'|a,s)[R(s,a,s') + \gamma V^{\pi}(s')]$
- $Q^{*}(s, a) = \sum_{s'} P(s'|a,s)[R(s,a,s') + \gamma V^{*}(s')]$
- **$V^{*}(s) = \max \{Q^{*}(s, a): a \in A\}$**
- **$\pi^{*}(s) = \text{argmax} \{Q^{*}(s, a): a \in A\}$**
- Let Q and V be estimators of Q^{*} and V^{*}
- Note we actually only need one of V and Q to compute the other (given P, R, and γ)
- If we are storing V: $V[s] = \max \{\sum_{s'} P(s' \mid s, a)[R(s, a, s') + \gamma V[s']]: a \in A\}$
- If we are storing Q: $Q[s, a] = \sum_{s'} P(s' \mid s, a)[R(s, a, s') + \gamma \max \{Q[s', a']]: a' \in A\}$

Q-learning

- **Q-learning: We store a Q-table and when we have an experience <s, a, r, s'> we update it. $Q[s,a] = (1 - \alpha)Q[s,a] + \alpha[r + \gamma \max \{Q[s', a']: a' \in A\}]$**
  - **$0 < \alpha < 1$ and α is learning rate**
- Q table will (eventually) converge to Q^*
- Exploration strategies:
  - **Greedy: With probability ε choose an action at random, otherwise use action believed to be best**
  - **Softmax: Chose actions probabilistically, probability of choosing an action k in state s = $e^{Q[s,k]/T}/[\sum_{a \in A} e^{Q[s,a]/T}]$ where T is the temperature**
- Q-learning is off-policy learning, it will eventually learn an optimal policy but explores the state space however it likes to get there — if mistakes are costly we need on-policy learning
- On-policy learning doesn't just learn the value of the optimal policy but also the value of the policy it is following to learn the optimal policy
- **SARSA is on-policy learning — has experience <s, a, r, s', a'> instead of the <s, a, r, s'> for Q-learning**
  - **We use the given a' instead of maximising over all a'**

Multi-agent systems

Game theory: Forms of games

- The normal form of a game is:
  - A finite set I of agents {1, …, n}
  - A set of actions A_i available to each agent i
  - An action profile $\sigma=<a_1, \ldots, a_n>$ of the actions taken by agents 1, …, n
  - A utility function u(σ, i) which gives the expected utility for agent i of the outcome from the agents taking those actions
    - We can represent the game as a payoff matrix which contains a 2D vector of utilities for the agents in the cell corresponding to each element of the cross product of the action sets
- The extensive of a game is a finite tree whose nodes are states and arcs are actions
  - Each internal node is labeled with which agent (or "nature" for the external environment) controls what happens at the node
    - Nature nodes are stochastic
  - Each arc from a node corresponds to an action available to the agent who controls that node
  - Each leaf is an outcome and is labeled with the utility vector of utilities for all agents
  - Utility for an agent at a leaf is its entry in the vector
  - Utility for an agent at an internal node is (expected value of if stochastic) utility for the agent at the child chosen by the strategy of the agent who controls the node
- In an imperfect-information (partially observable e.g. agents act simultaneously) agents do not know the exact state of the world when choosing their action
  - In extensive form, agents control a set of nodes (that they cannot distinguish which one they are at) with the same available actions and same strategy

Game theory: Strategies

- Strategy = probability distribution over available actions
- Strategy profile = a collection of strategies, one for each agent
- **σ_i = strategy of agent i**
- **σ_{-i} = Set of strategies of all agents other than i**
- strategy profile = $\sigma = \sigma_i\sigma_{-i}$
- **σ_i is a best response to σ_{-i} iff $\forall \sigma_i'$ utility($\sigma_i\sigma_{-i}$, i) ≥ utility($\sigma_i'\sigma_{-i}$, i)**
- **σ is a Nash equilibrium iff $\forall$ i σ_i is a best response to σ_{-i}**
  - **"No agent would benefit from <u>unilaterally</u> deviating** (assuming others don't change their strategy)"
  - Nash's Theorem: Every finite game has at least one Nash equilibrium
- **s_1 strictly dominates s_2 for agent i if iff $\forall \sigma_{-i}$ utility($s_1\sigma_{-i}$, i) > utility($s_2\sigma_{-i}$, i)**
  - "Gives better utility for every combination of strategies of the other agents"
- *s_1 weakly dominates s_2 for agent i if iff $\forall \sigma_{-i}$ utility($s_1\sigma_{-i}$, i) ≥ utility($s_2\sigma_{-i}$, i) and $\exists \sigma_{-i}$: utility($s_1\sigma_{-i}$, i) > utility($s_2\sigma_{-i}$, i)*
- **Dominated strategies (<u>remember a pure strategy may be dominated by a mixed strategy)</u> can be safely discarded from consideration**
- **We call the set of non-dominated strategies the support set, we can calculate a (possibly mixed) strategy over this that gives a nash equilibrium** (if the game is finite)
  - **Form simultaneous equations for "utilities for the agent whose strategy it is are the same for all strategies of other agents" — <u>don't forget to include that probabilities sum to 1</u>**