o Time complexity
o Space complexity
• Correctness
o An algorithm is complete if it will find a solution when one exists and correctly report failure (no solution) when it does not
o An algorithm is optimal if it finds a solution with the lowest path cost among all solutions (i.e., path cost optimal)

Actions: si → A
Function that returns the possible actions, A {a1,..., ak}, at a given state si
Initial state

Transition model, T: (si, aj) → si'
Function that returns the intermediate state transitioned to, si', when action aj is applied at state si

Goal test, isGoal: si → {0, 1}
Function that returns 1 if given state si is a goal state, else returns 0

Action costs, cost: {si, aj, si') → v
Function that returns cost, v, of taking the action aj at state si to reach the intermediate state si'
Generally assume costs >= 0

general search algo

```
Function TreeSearch(initial_state, actions, T, isGoal, cost):
   frontier = {Node(initial_state, NULL)}       # ADT: Node(current_state, parent_node)
   while frontier not empty:                      # add path cost, depth, action as required
         current = frontier.pop()
         if isGoal(current.state): return current.getPath()
         for a in actions(current.state):
               successor = Node(T(current.state, a), current)
               frontier.push(successor)
   return failure       # i.e., no path from the initial state to any goal state
```

BFS

Frontier: Queue

Time Complexity: $O(b^d)$

Space Complexity: $O(b^d)$

Complete: Yes[1]

Optimal: No[2]

b: branching factor
d: depth of shallowest goal
m: maximum depth
1: if finite b AND (finite m OR contains solution)
2: optimal if action costs uniform (and some other cases)

Default assumptions on search spaces:
- b finite
- d finite (contains a solution)
- m infinite
- All actions costs ≥ 0

Exercise questions:
- Show that when b is infinite, BFS is incomplete.
- Show that when b is finite, but m is infinite AND there is no solution, BFS is incomplete.
- Under what (other) cases is BFS optimal?
- Why is the complexity of BFS $O(b^d)$? Is this accurate?

Early goal test for BFS:

Performing goal test on pushing to frontier instead of popping from frontier will prevent this with no change in the expectation on the BFS solution

However, it should be noted that a different (valid) path may be output

UCS:

Frontier: Priority Queue[1]

Time Complexity: $O(b^e)$

Space Complexity: $O(b^e)$

Complete: Yes[2]

Optimal: Yes

Updated default assumptions on search spaces:
- All action costs > ε > 0

Exercise questions:
- Can you show that UCS under BFS completeness assumptions is incomplete when action costs are not > ε > 0 (e.g., when ε = 0)?
- Why is the complexity of UCS $O(b^e)$?
- Why is UCS optimal?
- Would UCS still be optimal with an early-goal test?

b: branching factor
e: $1 + \lfloor C^*/\varepsilon \rfloor$, where C^* is the optimal path cost and ε is some small positive constant
1: prioritising lower path cost, g(n), where g(n) = path cost of the path taken to reach n
2: requires BFS completeness criteria and that action costs > ε > 0
Note: determining path cost for each node is O(1) since we store the current path cost in each node

DFS:

Frontier: Stack

Time Complexity: $O(b^m)$

Space Complexity: O(bm)

Complete: No[1]

Optimal: No

Exercise questions:
- Show that DFS under BFS completeness assumptions is incomplete.
- Can space complexity be improved?

The space complexity of DFS may be improved to O(m) by simply backtracking – i.e., tracing back to parent and last action taken (assuming fixed order of actions – recall that we store parent node and action taken at parent)

b: branching factor
m: maximum depth
1: DFS may be incomplete even if a solution exists

Depth-Limited Search (DLS)

DFS with a depth limit, ℓ
- Search only up to depth ℓ
- Assume no actions may be taken from nodes at depth ℓ

Same guarantees as DFS with ℓ in place of m
- Time complexity: $O(b^\ell)$
- Space complexity: $O(b\ell)$
- Complete: No
- Optimal: No

Iterative Deepening Search (IDS)

Example, b = 10 and d = 5
- N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111
- N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456
- Overhead ≈ 11%

IDS properties
- Time: $O(b^d)$
- Space: O(bd)
- Complete: Yes (is b finite and d finite or contains solution) – same as BFS
- Optimal: No (optimal if costs uniform (and some other cases)) –same as BFS

Summary tree search

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes[1]	Yes[1,2]	No[3]	No	Yes[1]
Optimal Cost?	No[4]	Yes	No	No	No[4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	O(bm)	$O(b\ell)$	O(bd)

1. Complete if b finite and either has a solution or m finite
2. Complete if all actions costs are > ε > 0
3. DFS is incomplete unless the search space is finite – i.e., when b finite and m finite
4. Cost optimal if action costs are all identical (and several other cases)

- Recall that an Early Goal Test is assumed for BFS
- UCS must perform a Late Goal Test to be optimal (this also accounts for the +1 in the index of its complexity)
- DFS is incomplete (even under 1 – note the "or"); if a solution exists, it may infinitely traverse a path without a solution
- DFS space complexity may be improved to O(m) with backtracking (similar for DLS and IDS)

Default assumptions on search spaces:
- b finite
- d finite (contains solution)
- m infinite
- All actions costs are > ε > 0

Tree search allows revisits, cyclic graph, incomplete

Summary graph search

Criterion	BFS	UCS	DFS	DLS	IDS
Complete?	Yes[1]	Yes[1,2]	No[3]	No	Yes[1]
Optimal Cost?	No[4]	Yes	No	No	No[4]
Time	O(|V|+|E|)				
Space					

1. Complete if b finite and either has a solution or m finite
2. Complete if all actions costs are > ε > 0
3. DFS is incomplete unless the search space is finite – i.e., when b finite and m finite
4. Cost optimal if action costs are all identical (and several other cases)

- Time and space complexities are now bounded by the size of the search space – i.e., the number of vertices (nodes) and edges, |V| + |E|
- Note that we do not need to check for cheaper paths under graph search for BFS and DFS since costs play no part in those algorithms and they cannot guarantee an optimal solution anyway

Default assumptions on search spaces:
- b finite
- d finite (contains solution)
- m infinite
- All actions costs are > ε > 0

For CS3243, unless otherwise mentioned, assume tree search

For graph search, assume V2 for UCS and V1 for other uninformed search algorithms

graph search V1

```
Function GraphSearchV1(initial_state, actions, T, isGoal, cost):
   initial_node = Node(initial_state, NULL)
   frontier = {initial_node}
   reached = {initial_state: initial_node}
   while frontier not empty:
         current = frontier.pop()
         if isGoal(current.state): return current.getPath()
         for a in actions(current.state):
               successor = Node(T(current.state, a), current)
               if successor.state not in reached:
                     frontier.push(successor)
                     reached.insert(successor.state: successor)
   return failure
```

Exercise:
- Which uninformed algorithms may benefit from this implementation?

Not optimal bc cannot revisit even with more optimal path

graph search V2

```
Function GraphSearchV2(initial_state, actions, T, isGoal, cost):
   initial_node = Node(initial_state, NULL)
   frontier = {initial_node}
   reached = {initial_state: initial_node}
   while frontier not empty:
         current = frontier.pop()
         if isGoal(current.state): return current.getPath()
         for a in actions(current.state):
               successor = Node(T(current.state, a), current)
               if successor.state not in reached or
                  successor.getCost() < reached[successor.state].getCost():
                     frontier.push(successor)
                     reached.insert(successor.state: successor)
   return failure
```

UCS optimal

Assumptions
- Branching factor (b) is finite
- Solution exists (d finite), or max depth (m) is finite
- Each action cost exceeds a small positive constant, ϵ

Assert that UCS is complete!

Proposition
- Suppose UCS finds a suboptimal path T with goal state n_T
- That implies a different optimal path U with goal state n_U (note n_U may be the same state as n_T)
  - This means that node n_T (via T) was popped before some other node n' along $U \Rightarrow g(n_T) < g(n')$
  - Since both U and T start from the same initial state, sub-paths of U and T must be on the frontier
  - The above states that the last node along U that was not popped is n', and n_T was popped instead
- Thus, we have:
  - $g(n') < g(n_U)$ // assert minimum action cost ϵ > 0
  - $g(n_U) < g(n_T)$ // definition of optimal path
  - $g(n') < g(n_T)$

BFS tracing example

Node	Frontier (push/pop from right to left)
S()	A(S), B(S)
B(S)	A(S), C(S,B), F(S,B)
F(S,B)	A(S), C(S,B), D(S,B,F), H(S,B,F)
H(S,B,F)	A(S), C(S,B), D(S,B,F), D(S,B,F,H), K(S,B,F,H)
K(S,B,F,H)	A(S), C(S,B), D(S,B,F), D(S,B,F,H)
D(S,B,F,H)	A(S), C(S,B), D(S,B,F), G(S,B,F,H,D)
G(S,B,F,H,D)	PATH FOUND = S-B-F-H-D-G

- Common mistakes
  - Specifying nodes explored instead of the final path
  - Excluding node H from the path

Tree vs grapph search
- graph-search algorithms will only explore non-redundant paths
- only explores nodes that either have not been visited or have been visited but previously via less optimal paths

- tree search algorithms have no such restrictions; they consider all paths, including redundant ones

Admissible
paths not ending at a goal are never over estimated
- non goal, n, f(n) = g(n) + h(n) <= g(n) + h*(n)
paths ending at a goal are exact
- goal m, f(m) = g(m) + h(m), where h(m) = 0
if admissible, then A* optimal with tree search

Consistency
h(n) is consistent, if ∀n, n': h(n)≤ cost(n,a,n')+ h(n')
if h(n) is consistent, A* with graph search is optimal

If, h1(n)≥ h2(n), then h1 dominates h2
If h1 is also admissible
– h1 must be closer to h* than h2
– h1 is usually more efficient than h2 when used with A*

Graph search V3

```
Function GraphSearchV3(initial_state, actions, T, isGoal, cost):
   frontier = {Node(initial_state, NULL)}
   reached = {}
   while frontier not empty:
         current = frontier.pop()
         reached.insert(current.state: current)
         if isGoal(current.state): return current.getPath()
         for a in actions(current.state):
               successor = Node(T(current.state, a), current)
               if successor.state not in reached:
                     frontier.push(successor)
   return failure
```

b. Show graph search for Greedy Best-First Search is complete (assuming finite search space).
Proof Sketch:
– When a node is expanded, each successor:
 —is added to frontier, or
 —has already been visited (graph search).
– Implies that at least one path to each state in the search space will eventually be added to the frontier (finite search space).
– So, a goal, if it exists must be found, or else the algorithm terminates when the frontier is empty (all states explored).

Let s be the initial state, n be an intermediate state along the optimal path, t be a suboptimal goal state (i.e., a goal state reached via a suboptimal path), and t* be the goal along the optimal path.
An optimal solution implies that n must be expanded before t.
Proof by contradiction:
• Let us assume that a suboptimal solution is found – i.e., that t is expanded before n, which implies that (A): f(t) ≤ f(n).
• In other words, given the above frontier, only when f(t) < f(n) would we expand t before n.
• However, since t is not on the optimal path but t* is, we have:
o f(t) > f(t*)
o f(t) > g(t*) // since h(t*) = 0
o f(t) > g(n) + p(n, t*) // where p(n, t*) is the actual cost from n to t*
o f(t) > g(n) + h(n) // asserting admissibility
o f(t) > f(n) // this contradicts (A)
• Note: we do not consider f(t) = f(n) since that would mean f(t) is equally optimal.

Prove that A* Search is optimal under graph search v3 with consistent heuristic.
o Base case: When expanding the start node, it is obvious that $g\ s0 = g_*\ s0$
o Inductive case: Assume the inductive hypothesis is true up to sk.
So, from $s0$ to sk, the optimal path is found, and each $g\ si$ Since $g_*\ sk{+}1$ is the optimal path cost from $s0$ to $sk{+}1$,
we have: $\mathbf{g}\ (\boldsymbol{sk{+}1}) \geq \mathbf{g_*}\ (\boldsymbol{sk{+}1})$– (A)
By consistency, we know that sk must be popped before $sk{+}1$.
So, after sk is popped, we know that when $sk{+}1$ is popped,
cost to node $sk{+}1$ is the minimum of all the ways that it has been reached so far (asserting priority queue based on f).
$\mathbf{g}\ (\boldsymbol{sk{+}1})$ + h(sk+1) = MIN (g(sk+1) + h(sk+1), g(sk) + c(sk, sk+1) + h(sk+1))
$\mathbf{g}\ (\boldsymbol{sk{+}1})$ = min{g(sk+1), g(sk) + c(sk, sk+1)}
$\leq g\ (sk) + cost\ (sk,\ sk{+}1)$
$= g_*\ (sk) + cost\ (sk,\ sk{+}1)$
$= \mathbf{g_*}\ (\boldsymbol{sk{+}1})$ – (B)

consistent -> admissible

Solution: The proof is by induction of k(n), which denoted the number of actions required to reach the goal from a node n to the goal node t.
Base case (k = 1, i.e., the node n is one step from t): Since the heuristic function h is consistent, h(n) ≤ c(n, t) + h(t). And since h(t) = 0, h(n) ≤ c(n, t) = h*(n). Therefore, h is admissible.
Induction case: Suppose that our assumption holds for every node that is k – 1 actions away from t, and let us observe a node n that is k actions away from t; that is, the least-actions optimal path from n to t has k > 1 steps.
We write the optimal path from n to t as: n → n1 → n2 → … → nk–1 → t.
Since h is consistent, we have h(n) ≤ c(n, n1) + h(n1).
Now, note that since n1 is on a least-cost path to t from n, we must have that the path n1 → n2 → … → nk–1 → t is a minimal-cost path from n1 to t as well. By our induction hypothesis, we have h(n1) ≤ h*(n1).
Consequently, combining the two inequalities above, we have, h(n) ≤ c(n, n1) + h*(n1).
Note that h*(n1) is the cost of the optimal path from n1 to t; by our previous observation (that n1 → n2 → … → nk–1 → t is an optimal cost path from n1 to t), we have that the cost of the optimal path from n to t – i.e., h*(n) – is exactly c(n, n1) + h*(n1), which concludes the proof.

admissibility but inconsistent
state an admissible heuristic that overestimates the path cost from parent to child (happens when child to goal is understated too much, so h(n) > cost(n to n') + h(n') (h(n') understated)

Completeness assumptions:
- b finite AND (m finite OR has a solution)
- All action costs are > ε > 0

- UCS
  - Optimal under
    - Tree search
    - Graph search (version 2 and 3)
  - Popping node n ⇒ optimal path to n found
- Greedy Best-First Search
  - Not optimal
  - Incomplete under tree search

Assume graph search version 1 for Greedy (unless otherwise stated)

- A* Search
  - Assuming admissible h
    - May not have monotonically increasing path cost
    - Optimal under
      - Tree search
      - Graph search (version 2)
  - Assuming consistent h
    - On popping node n, optimal path to n found
    - Optimal under
      - Tree search
      - Graph search (version 2 and 3)

Assume graph search version 2 and late goal testing for UCS and A* (unless otherwise stated)

h_1	h_2	$MAX(h_1, h_2)$
Admissible	Admissible	Admissible
Admissible	Inadmissible	Inadmissible
Inadmissible	Inadmissible	Inadmissible

h_1	h_2	$MIN(h_1, h_2)$
Admissible	Admissible	Admissible
Admissible	Inadmissible	Admissible
Inadmissible	Inadmissible	Uncertain

What about consistent heuristics?
- MAX of 2 consistent heuristics is also consistent.
- MIN of 2 consistent heuristics is also consistent.

Try to determine these as an exercise! Might appear in the midterm!

h_1: Number of pallets left. Admissible.
- Every move made by the Pacman eats 0 or 1 pallets
  ⇒ Moves required to clear all pallets would be greater than or equal to the number of pallets on the board
  - i.e., Pacman requires n moves to clear n pallets iff. Pacman eats 1 pallet per move.

h_2: Number of pallets left + Manhattan distance to the closest pallet. Inadmissible, by counterexample.
- Consider the case where there is 1 pallet remaining, and Pacman is 1 step away from that pallet.
- Here, we have: $h_2 = 2$ (distance 1 and 1 pallet left), whereas $h^* = 1$. Thus, h_2 overestimates.

h_3: Manhattan distance to the furthest pallet. Admissible.
- To consume that furthest pallet, Pacman must first travel to that pallet, which requires at least Manhattan distance number of moves.
- Note that a common mistake is to begin the proof claiming that $h_3 = h^*$ when only 1 pallet remains on the board.
  - This is untrue since there may still be obstacles on the board – i.e., we instead have $h_3 \leq h^*$.

h_4: Average Euclidean distance to all pallets. Admissible.
- Euclidean distance ≤ Manhattan distance AND average distance ≤ maximum distance ⇒ h_3 dominates h_4
- Thus, since h_3 is admissible, then h_4 is also admissible.

Hill-Climbing Algorithm

```
Function HillClimbing(initial_state, eval_fn, highest_valued_successor):
   current = initial_state
   while True:
         neighbour = highest_valued_successor(current)
         if eval_fn(neighbour) ≤ eval_fn(current): return current
         current = neighbour
```

Stochastic hill climbing
- Changes the function `highest_valued_successor(..)`
- Chooses randomly among states with f-values better than `current`
- May take longer to find a solution but sometimes leads to better solutions

First-choice hill climbing
- Changes the function `highest_valued_successor(..)`
- Handles high b by randomly generating successors until one with better f-value than current is found (instead of generating all possible successors)
- May even work with infinite b

Sideways move
- Replaces ≤ with <
- Allows continuation when `eval_fn(neighbour) == eval_fn(current)`
- Can traverse shoulders / plateaus

Random-restart hill climbing
- Different algorithm
- Adds an outer loop which randomly picks a new starting state
- Keeps attempting random restarts until a solution is found

Local Beam Search
Store k states instead of 1
o Hill climbing just stores the current state
o Beam (window) stores k
• General algorithm
o Begins with k random starts
o Each iteration generates successors for each of the k random start states
o Repeat with best k among ALL generated successors unless goal found
• Better than k parallel random restarts
o Since best k among ALL successors taken (not best from each set of successors, k times)
• Stochastic beam search
o Original variant may still get stuck in a local cluster
o Adopt stochastic strategy similar to stochastic hill climbing to increase state diversity

```
Function BeamSearch(random_initial_state, isGoal, actions, T, choose_best_k):
   frontier = {}
   for i in 1 to k:
         s = random_initial_state()
         if isGoal(s): return s
               frontier.push(s)
   while True:
         new_frontier = {}
         while frontier not empty:
               s = frontier.pop()
               for a in actions(s):
                     new_s = T(s, a)
                     if isGoal(new_s): return new_s
                     new_frontier.push(new_s)
         frontier = choose_best_k(new_frontier)
```

As the search may loop infinitely, it may be wise to define and use an iteration threshold

No duplicate states on beam

We may also apply stochastic beam search where instead of choosing the best k, it chooses successors with probability proportional to the successor's value

- Notice that we may allow states with lower values than their parents to be added since `choose_best_k` will simply choose the best k successor states generated
  - We may vary this rule but then may not be able to always ensure we have k new candidates

CSPs

CSP Formulation

- State representation
  - Variables: X = {x_1, ... , x_n}
  - Domains: D = {d_1, ... , d_k}
    - Such that x_i has domain d_i
  - Initial state: all variables unassigned
  - Intermediate state: partial assignment
- Actions, costs, and transition
  - Assignment of values (within domain) to variables
  - Costs are unnecessary
- Goal test
  - Constraints: C = {c_1, ... , c_m}
    - Defined via a constraint langauge (algebra, logic, sets)
    - Each c_i corresponds to a requirement on some subset of X
    - Can be unary (|scope|= 1), binary (|scope|= 2), or global (|scope|> 2)
  - Objective is a complete and consistent assignment
    - Find a legal assignment $(y_i, ..., y_n)$ s.t. $y_i \in D_i\ \forall i \in [1,n)$
    - Complete: all variables assigned values
    - Consistent: all constraints in C satisfied

Algo

```
Function CSPSolver(variables, domains, constraints):
    assignments = initial_state            # no assignments made
    while assignments incomplete:
        if no possible assignments left: return failure
        current = assign a value to non-assigned variable
        if current is consistent then: assignments.store(current)
    return assignments
```

Search tree size

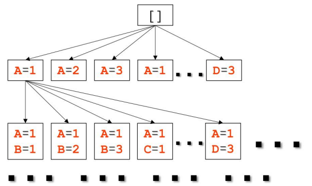
At depth l: $(|X|-l)\cdot|d|$ states

Total number of leaf states:

$$nm \times (n-1)m \times (n-2)m \times \ldots \times 2m \times m = n!m^n$$

where $n = |x|$ and $m = |d|$

Order of variable assignments is not important, just consider assignments to one variable per level (m^n leaves)



Backtracking algorithm

```
def backtrack(csp, assgt) -> bool:
    if assgt is complete:
        return assgt
    var = select_unassigned_variable(csp, assgt)
    for val in order_dom_values(csp, var, assgt):
        if val is consistent with assgt:
            add {var = val} to assgt
            inferences = inference(csp, var, assgt)
            if inferences != failure:
                add inferences to csp
                result = backtrack(csp, assgt)
                if result != failure:
                    return result
                remove inferences from csp
            remove {var = value} from assgt
    return failure
```

Variable-order heuristics

MRV (minimum remaining values)

- Choose variable with fewest legal values
- Most constrained variable
- Sort by domain size, choose variable with smallest domain size
- Places larger subtrees closer to the root so that any invalid state found prunes a larger subtree ⇒ eliminates larger subtrees earlier

Degree Heuristic

- To break ties in MRV heuristics
- Pick unassigned variable with most constraints (highest degree in constraint graph)
- This reduces the branching factor b

Recommended variable selection: MRV, then degree, then random

Value-order Heuristics

LCV (least constraining value)

- Choose the value that rules out the fewest choices from remaining domain values
  - Given assignment of value v to variable x', determine set of unassigned variables U that share a constraint with x' and pick v that maximises sum of consistent domain sizes of variables in U
- Avoids failure by avoiding empty domains

With variables, fail first. With values, fail last. Cuz need to look at all variables but do to look at all values.

BUT if all solutions required, then value-ordering irrelevant

Forward Checking

Track remaining legal values for unassigned variables, terminate search when any variable has no legal values (based on constraints with recently assigned variable)

Constraint propagation

Inference step to ensure local consistency of all variables

- Traverse constraint graph to ensure variable at each node is consistent, eliminate all values in variable's domain that are not consistent with linked constraints

Node-consistent (vertex-consistent)

- Domain of a variable is consistent with its unary constraints
- Achieved through pre-processing step before backtracking

Arc-consistent (edge-consistent)

- Domain of a variable is consistent with its binary constraints
- The variable's domain value must have a partnering domain value in other variable that will satisfy the binary constraint

X_i is arc-consistent w.r.t $X_j \iff \forall x \in D_i\ \exists y \in D_j$ s.t. binary constraint on arc (X_i, X_j) is satisfied

When checking arc (X_a, X_b), remove values from D_a

- Done during pre-processing or after each variable assignment (expensive)

AC-3 Algo

```
function ac3(csp) -> bool:
    queue = queue of all arcs in csp (both dirs)
    while queue:
        (X_i, X_j) = queue.pop()
        if revise(csp, X_i, X_j):
            if len(D_i) == 0:
                return False
            for X_k in X_i.neighbors - {X_j}:
                queue.append((X_k, X_i))
    return True

function revise(csp, X_i, X_j):
    revised = False
    for x in D_i:
        if no value in D_j allows (x, y) to satisfy
        the constraint between X_i and X_j:
            delete x from D_i
            revised = true
    return revised
```

Time complexity of AC-3

With n variables, there are at most $2 \times \binom{n}{2} \in O(n^2)$ directed arcs

Each arc can be reinserted at most d times because X_i has hat most d values to delete where d is domain size

Checking consistency of arc (revise()) takes $O(d^2)$ time

Overall time complexity: $O(n^2 \times d \times d^2) = O(n^2 \times d^3)$

Advesarial search

Assume 2 players, zero-sum game. MAX player wants to maximise value (our agent), MIN player wants to minimise value (opponent)

- If MIN does not play optimally, the actual outcome for MAX can only be better and never worse, but the algo may not select the optimal move.

Formulating games

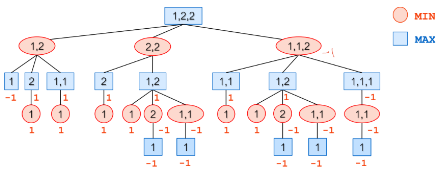
- State representation: as per general search formulation
- TO-MOVE(s): returns p, the player to move in state s
- ACTIONS(s): legal moves in state s
- RESULT(s, a): the transition model, returns resultant immediate state when taking action a at state s
- IS-TERMINAL(s): returns True when game is over, False otherwise
- UTILITY(s, p): defines a numeric value (score) for player p when the game ends at terminal state s

For zero-sum games, at terminal state s,
utility(MAX, s) + utility(MIN, s) = 0.
Just use utility(MAX) = -utility(MIN)

Strategies: optimal decisions via minimax

$$Minimax(s) = \begin{cases} \text{Utility}(s, \text{MAX}) & \text{if Is−Terminal}(s) \\ \max_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if To−Move}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{Minimax}(\text{Result}(s,a)) & \text{if To−Move}(s) = \text{MIN} \end{cases}$$

Remember that Result(s,a) outputs the successor state (given a taken at s) (i.e., MIN wants state with lowest utility, while MAX wants state with highest)



Backwards induction used

Properties

- Complete assuming finite game tree
- Optimal assuming optimal gameplay (opponent plays optimally)
- Time complexity $O(b^m)$
- Space complexity $O(bm)$
- Time polynomial to tree size

Backwards Induction Issue

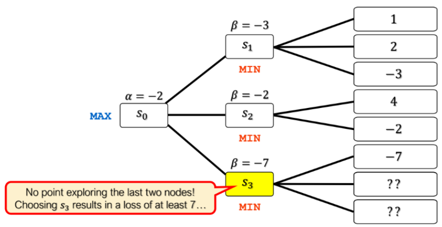
Game trees are massive, impossible to expand entire tree, must find ways to shrink search tree

α - β pruning

General idea: don't explore moves that would never be considered.

α bounds MAX's values, β bounds MIN's values.

Prune subtrees that will never affect Minimax decision



S_3 will be -7 or lower. MAX will not choose it because it already has S_1 or S_2 which are higher than -7.

Minimax algo (a and b are alpha and beta)

```
def minimax(node, depth, isMax, a, b):
  if node is a leaf node:
    return value of node
  if isMax:
    bestVal = -INFINITY
    for each child node:
      value = minimax(node, depth+1, false, a, b)
      bestVal = max(bestVal, value)
      a = max(a, bestVal)
      if b <= a:
        break
    return bestVal
  else:
    bestVal = +INFINITY
    for each child node:
      value = minimax(node, depth+1, true, a, b)
      bestVal = min(bestVal, value)
      b = min(b, bestVal)
      if b <= a:
        break
    return bestVal
minimax(0, 0, true, -INFINITY, +INFINITY)
```

Perfect ordering with alpha beta gives time complexity $O(b^{m/2})$ Random ordering gives time complexity $O(b^{3m/4})$

α - β pruning issues:

- Issue: Depth of the tree could be very large
  - Backwards induction only works with terminal states
  - If depth is very deep, it takes very long before we can reach terminal state
- Solution: Heuristic minimax
  - Cutoff test: have a depth limit as to how deep we search for a terminal node
  - Evaluation function: Estimates the expected utility at that state
  - Run MINIMAX until depth d then start using evaluation function to choose nodes
  - Replaces is_terminal(s) with cutoff_test(s, d).
  - Replaces utility(s, p) with eval(s, p).

Logical agents

Knowledge-based agents

- Represent agent domain knowledge using logical formulas
- Main components of a logical agent are inference engine and knowledge base. Inference engine consists of domain-independent algorithms while knowledge base is domain-specific content

KB agent function

```
def KB_agent(percept) -> action:
    persistent: KB, t (time)

    TELL(KB, MAKE_PERCEPT_SENTENCE(percept, t))
    action = ASK(KB, MAKE_ACTION_QUERY(t))
    TELL(KB, MAKE_ACTION_SENTENCE(action, t))
    t++
    return action
```

Inference via entailment

Entailment

Modelling: v models α (a sentence) if α is true under v

- v corresponds to one set of value assignments
- v corresponds to one instance of the environment (known part of a state)
- Let $M(\alpha)$ be the set of all models for α

Entailment ($\models$): One thing (right) follows from the other (left)

- $\alpha \models \beta \iff M(\alpha) \subseteq M(\beta)$
- E.g. $\alpha = q$ is prime, $\beta = (q \text{ is odd}) \vee q = 2$

Inference

Inference is deriving new knowledge from the KB

KB: environment rules/laws and percepts

$$KB \models \alpha \implies M(KB) \subseteq M(\alpha)$$

$$M(KB) \cup M(\alpha) \equiv M(\alpha)$$

$$M(KB) \cap M(\alpha) \equiv M(KB)$$

$$M(KB) \cap M(\neg\alpha) \equiv \emptyset$$

Given KB, try to infer α, i.e. determine if $KB \models \alpha$.

If $KB \models \alpha$, then α can be added to KB since $M(KB) \cap M(\alpha) \equiv M(KB)$

Soundness & Completeness

- $KB \vdash_A \alpha$
  - Means "sentence α is derived from KB by inference algorithm A"
- Soundness
  - A is sound if $KB \vdash_A \alpha \implies KB \models \alpha$
  - A will not infer nonsense.
- Completeness
  - A is complete if $KB \models \alpha \implies KB \vdash_A \alpha$
  - A can infer any sentence that KB entails

Truth table enumeration

Draw truth table of KB and α, KB entails α if whenever KB true, α true

function TT-ENTAILS?(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, *symbols*, { })

function TT-CHECK-ALL(KB, α, *symbols*, *model*) **returns** *true* or *false*
  if EMPTY?(*symbols*) **then**
    if PL-TRUE?(KB, *model*) **then return** PL-TRUE?(α, *model*)
    else return *true*   // when KB is false, always return true
  else
    P ← FIRST(*symbols*)
    rest ← REST(*symbols*)
    return (TT-CHECK-ALL(KB, α, *rest*, *model* ∪ {P = *true*})
      and
      TT-CHECK-ALL(KB, α, *rest*, *model* ∪ {P = *false* }))

Properties

- $O(2^n)$ time complexity
- $O(n)$ space complexity
- Sound and complete

Theorem proving methods

Validity & Satisfiability

- Validity
  - A sentence α is **valid** if it is true for all possible truth value assignments
  - **tautology**
  - $(KB \models \alpha) \iff (KB \implies \alpha)$ (deduction theorem)
- Satisfiability
  - A sentence is **satisfiable** if it is true for some truth value assignment (i.e. a model exists for that sentence)
  - A sentence is **unsatisfiable** if it is true for no truth value assignments
    - **contradictions**
  - $(KB \models \alpha) \iff (KB \wedge \neg\alpha)$ is unsatisfiable
    - Definition of proof by contradiction

Inference rules

- And-Elimination: $a \wedge b \models a$, $a \wedge b \models b$
- Modus Ponens: $a \wedge (a \implies b) \models b$
- Logical equivalences / Modus Tollens: $(a \vee b) \models \neg(\neg a \wedge \neg b)$
- Contrapositive: $(a \implies b) \models (\neg b \implies \neg a)$
- Syllogism: $(a \implies b) \wedge (b \implies c) \models (a \implies c)$

CNF

CNF = conjunction of disjunctive sentences, e.g. $(x_1 \vee x_2) \wedge (x_2 \vee x_3 \vee x_4)$

Conversion to CNF

- Convert:
  - $\alpha \iff \beta$ to $(\alpha \implies \beta) \wedge (\beta \implies \alpha)$
  - $\alpha \implies \beta$ to $\neg\alpha \vee \beta$
- Expand $\neg$ using De Morgan's and double negation
- Distributive law to convert $(\alpha \vee (\beta \wedge \gamma))$ to $(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$

Cardinality Rules

Suppose n variables, and we want k to be true.

- At least k
  - DNF: Connect each possible k-way conjunction via disjunctions
    - E.g. $n = 3, k = 2 : (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$
  - CNF: Connect each possible i-way disjunction via conjunctions, where $i = n - k + 1$
    - E.g. $n = 3, k = 2, i = n - k + 1 = 2$, so we use pairwise OR: $(A \vee B) \wedge (A \vee C) \wedge (B \vee C)$
- At most k
  - CNF: Negation of $k+1$ ways
    - E.g. $n = 3, k = 1$: 1 + 1 (pairwise) negations of all literals: $(\neg A \vee \neg B) \wedge (\neg A \vee \neg C) \wedge (\neg B \vee \neg C)$
- Exactly k
  - DNF: Choose all combinations of the n variables where k literals are true and $n-k$ are False
    - E.g. $n = 3, k = 2 :$ $(A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge C)$
  - CNF: Conjunction of at least k and at most k
    - E.g. $n = 3, k = 2 :$ $(A \vee B) \wedge (A \vee C) \wedge (B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$

Resolution

If a literal x appears in R_i and its negation $\neg x$ appears in R_j, where $R_i, R_j \in KB$, then it can be removed from both. Then the remaining literals combine to 1 clause

Resolution algo

function PL-RESOLUTION(KB, α) **returns** *true* or *false*
  inputs: KB, the knowledge base, a sentence in propositional logic
    α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new ← { }
  while *true* **do**
    for each pair of clauses C_i, C_j **in** *clauses* **do**
      resolvents ← PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new ← *new* ∪ *resolvents*
    if *new* ⊆ *clauses* **then return** *false*
    clauses ← *clauses* ∪ *new*

- Make a clause list (copy of KB specified in CNF including negation of query $\neg\alpha$)
- Repeatedly resolve two clauses from clause list (add resolvent to clause list)
- Keep doing this till empty clause found or no more resolutions possible
  - If empty clause then can infer α
  - If no more resolutions and not empty clause then cannot infer α

Uncertainties

Conditional probabilities and Bayes RUle

$$Pr[A|B] = \frac{P[A \wedge B]}{P[B]}$$

Bayes rule: $Pr[A|B] = \frac{Pr[B|A]Pr[A]}{Pr[B]}$

Chain rule

$$Pr[R_1 \wedge \ldots \wedge R_k] = Pr[(R_1 \wedge \ldots \wedge R_{k-1}) \wedge R_k]$$
$$= Pr[R_k | R_{k-1} \wedge \ldots \wedge R_1] \cdot Pr[R_{k-1} \wedge \ldots \wedge R_1]$$

$$Pr\left[R_1 \wedge R_2 \wedge \ldots \wedge R_k\right] = \prod_{j=1,\ldots,k} Pr\left[R_j | R_1 \wedge \ldots \wedge R_{j-1}\right]$$

$$Pr[A \wedge B \wedge C \wedge D] = Pr[D|C \wedge B \wedge A] \cdot Pr[C \wedge B \wedge A]$$
$$= Pr[D|C \wedge B \wedge A] \cdot Pr[C|B \wedge A] \cdot Pr[B \wedge A]$$
$$= Pr[D|C \wedge B \wedge A] \cdot Pr[C|B \wedge A] \cdot Pr[B|A] \cdot Pr[A]$$

A and B are independent $\iff Pr[A \wedge B] = Pr[A] \cdot Pr[B]$
$\iff Pr[A|B] = Pr[A]$

Inference via Bayes' Rule

Infer statement of the form "What is the likelihood of an event α given the probabilities of other events"

$$Pr[A \wedge B | C] = \frac{Pr[A \wedge B \wedge C]}{P[C]} = \frac{Pr[A \wedge B \wedge C]}{Pr[B \wedge C]} \times \frac{Pr[B \wedge C]}{Pr[C]}$$
$$= Pr[A|B \wedge C] \times Pr[B|C]$$

Bayesian Networks

- Vertices are random variables
- An edge from X to Y means X directly influences Y
- Each vertex has a conditional distribution given its parents
- In the simplest case, a conditional distribution can be represented as a conditional probability table (CPT)
- CPTs in the BN are the distribution over Z for each combination of parent values

Markov Blanket

- A variable is conditionally independent of all other nodes in the network, given
  - its parents
  - its children
  - its children's parents