SEARCH ALGORITHMS

Definitions:

Definition of a problem as a search problem:
define the:
States, Initial state, Successor function, Arc cost, Goal test.

Successors function: SUCCESSORS(s) returns all x's successors: An arc exists from a node s to a node s' if s' ∈ SUCCESSORS(s).

Solution: a path connecting the initial to a goal node.

Arc cost: a positive number measuring the "cost" of performing the action corresponding to the arc. For any given problem the cost c of an arc always verifies $c \geq \varepsilon > 0$, where ε is a constant.

Cost of a path: is the sum of the edge costs along this path.

Optimal solution: a solution path of *minimum* cost.

The search tree

Search Nodes ≠ States: If states are allowed to be revisited, the search tree may be infinite even when the state space is finite.

Depth of a node N = length of path from root to N (Depth of the root = 0)

Fringe: the set of all search nodes that haven't been expanded yet. A priority queue.

A *BLIND* Search Algorithm:

INSERT(initial-node,FRINGE)
Repeat:**{**
  If empty(FRINGE) then return failure
  n ← REMOVE(FRINGE)
  s ← STATE(n)
  If GOAL?(s) then return path or goal state
  *For every state s' in SUCCESSORS(s)***{**
    Create a new node n' as a child of n
    INSERT(n',FRINGE)
}}

Completeness: A search algorithm is complete if it finds a solution whenever one exists.

Optimality: A search algorithm is optimal if it returns an optimal solution whenever a solution exists.

Branching factor: Maximum number of successors of any state

Blind strategies:

1. **BFS:** New nodes are inserted at the end of the FRINGE. If a problem has no solution, breadth-first may run for ever. Complete and optimal. Have high space complexity.
2. **DFS:** Complete only for finite search tree. Space efficient, but is neither complete, nor optimal
3. **IDS:** Complete, Optimal if step cost =1. With the same space complexity as DFS and almost the same time complexity as BFS.
4. **Uniform-Cost:** BFS with weighted arcs.

Strategy	Fringe	Time	Space
BFS	at the end of the FRINGE	$O(b^d)$	$O(b^d)$
DFS	at the front of the FRINGE	$O(b^m)$	$O(bm)$ [or $O(m)$]
Iterative Deepening	For k = 0, 1, 2… do: Perform depth-first search with depth cutoff k	$O(b^d)$	$O(bd)$ [or $O(d)$]
Uniform-Cost	sorted in increasing cost of path cost		

b: branching factor

d: depth of shallowest goal node

m: maximal depth of a leaf node

Avoiding Revisited States

1. BFS: Store all states associated with generated nodes in CLOSED
2. DSF: Store all states associated with nodes in current path in CLOSED. ONLY avoid loops. OR Store of all generated states in CLOSED.
3. Uniform-Cost: When a node is expanded, store its state into CLOSED. When a new node N is generated: If STATE(N) is in CLOSED, discard N If there exists a node N' in the fringe such that STATE(N') = STATE(N), discard the node – N or N' – with the highest-cost path

Heuristic (Informed) Search

Evaluation function: $f(N) = g(N) + h(N)$

$g(N)$: Cost of the best path found so far between the initial node and N [Dependent on search tree]

Heuristic function $h(N)$ estimates the distance of STATE(N) to a goal state, [Independent of search tree].

$f(N) = h(N)$ → Greedy BFS

Admissible heuristic:

- Let $h^*(N)$ be the cost of an optimal path from N to a goal node
- The heuristic function $h(N)$ is admissible if: $0 \leq h(N) \leq h^*(N)$
- An admissible heuristic function is always *optimistic*!
- Note: G is a goal node ➔ $h(G) = 0$

A* Search

- $f(N) = g(N) + h(N)$, where:
- $g(N)$ = cost of best path found so far to N.
- $h(N)$ = heuristic function.
- for all arcs: $0 < \varepsilon \leq c(N, N')$

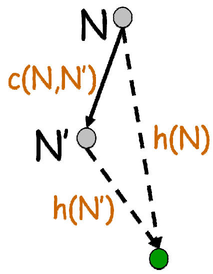
If h is admissible, A is complete and optimal*

! When there is no solution, A runs forever if the state space is infinite or states can be revisited an arbitrary number of time.*

It is not harmful to discard a node revisiting a state if the new path to this state has higher cost than the previous one. A* remains optimal.

A heuristic h is **consistent** or **monotone** if

1) For each node N and each child N' of N: $h(N) \leq c(N, N') + h(N')$.



(triangle inequality)

2) For each goal node G: $h(G) = 0$

A consistent heuristic is also admissible.

RESULT: If h is consistent, then whenever A* expands a node, it has already found an optimal path to this node's state.

Avoiding Revisited States: When a node is expanded, store its state into CLOSED.

When a new node N is generated:
If STATE (N) is in CLOSED, discard N If there exists a node N' in the fringe such that STATE (N') = STATE (N), discard the node – N or N' – with the largest f. CLOSED will be also called VISITED

A* with h≡0 is uniform-cost search

Iterative Deepening A* (IDA*)

1. Initialize cutoff to f(initial-node)
2. Repeat:
   a. Perform depth-first search by expanding all nodes N such that f(N) ≤ cutoff
   b. Reset cutoff to smallest value f of non-expanded (leaf) nodes

Advantages:

1. Still complete and optimal
2. Requires less memory than A*
3. Avoid the overhead to sort the fringe

Drawbacks:

1. Can't avoid revisiting states not on the current path
2. Available memory is poorly used

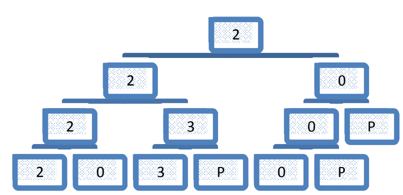
GAME PLAYING

MIN-MAX Algorithm:
Horizon (h): maximal depth of the game tree built each turn.

- Function e: state s → number e(s)
- e(s) > 0 means that s is favorable to MAX (the larger the better)
- e(s) < 0 means that s is favorable to MIN
- e(s) = 0 means that s is neutral

1. Expand the game tree uniformly from the current state (where it is MAX's turn to play) to depth h.
2. Compute the evaluation function e(s) at every leaf of the tree
3. Back-up the values from the leaves to the root of the tree as follows:
   A MAX node gets the maximum of the evaluation of its successors
   A MIN node gets the minimum of the evaluation of its successors
4. Select the move toward a MIN node that has the largest backed-up value

Alpha- Beta pruning:
Update the max(α)/min(β) value of the parent of a node N when the search below N has been completed or discontinued
Discontinue the search below a MAX node N if its max(α) value is ≥ the min(β) value of a MIN ancestor of N.
Discontinue the search below a MIN node N if its min(β) value is ≤ the max(α) value of a MAX ancestor of N.
Pruning example:



Local Search:

Advantages:

1. A Light-memory search method (usually constant)

No search tree; only the current state is represented!

2. OFTEN find reasonable solution in large/infinite state spaces

!!! Only applicable to problems where the path is irrelevant

Hill Climbing Algorithm:

1) *current* ← MakeANode(initialState(problem))
2) Repeat:
   a) *neighbor* ← highest valued successor of *current*
   b) if value(*neighbor*) ≤ value(*current*) then return State(current)
   c) *current* ←neighbor

Possible variations:

- random restart
- try to overcome plateaus
- look k steps ahead
- stochastic hill climbing

Simulated Annealing:

1) S ← initial state
2) Repeat forever:
   a) T = mapping of time
   b) If (T= 0) then return S
   c) S' ← successor of S picked at random
   d) Dh = h(S') - h(S)
   e) if(Dh ≥0) then S ← S'
   f) else
      - S ← S' with probability ~ $e^{(\Delta H/t)}$

Where T is called the "temperature"

Simulated annealing lowers T over the k iterations.

It starts with a large T and slowly decreases T.

"Bad" moves are more likely to be allowed at start.

Genetic Algorithms:

1. Produce a population of solutions (strings)
2. Rank each solutions according to fitness function
3. Repeat:
   a. Select k solutions for breading
   b. Perform crossover to generate offspring
   c. Perform a mutation on offspring
   d. Calculate fitness of offspring
   e. Replace least qualified solutions in population with new offspring

LEARNING ALGORITHMS

Decision tree

Problem: find a hypothesis *h* such that *h* ≈ *f*

h is consistent if it agrees with *f* on all examples.

Aim: find a small tree consistent with the training examples

- (Recursively) choose "most significant" attribute as root of (sub) tree.
- If remaining examples are all positive (or negative) answer yes/no.
- If there are no examples left return a default value (majority of the node parent).
- If there are no attributes left but both positive and negative examples – problem (non consistent examples).
- Most significant = Choose the attribute with the smallest remainder *(A).*

$$remainder(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p + n} I\left(\frac{p_i}{p+n}, \frac{n_i}{p+n}\right)$$

$$I\left(\frac{p}{n+p} + \frac{n}{n+p}\right) = -\frac{p}{n+p} Log\left[\frac{p}{n+p}\right] - \frac{n}{n+p} Log\left[\frac{n}{n+p}\right]$$

$$Gain(A) = I\left(\frac{p}{p+n}, \frac{n}{p+n}\right) - remainder(A)$$

CONSTRAINT SATISFACTION

CSP-BACKTRACKING (A)

1. If assignment A is complete then return A
2. X ← select a variable not in A
3. D ← select an ordering on the domain of X
4. For each value v in D do
   a. Add (X←v) to A
   b. If A is valid then
      i. result ← CSP-BACKTRACKING(A)
      ii. If result ≠ failure then return result
   c. Remove (X←v) from A
   5. Return failure.

! This performs simple DFS on the state tree

CSP-BACKTRACKING (A, var-domains)

1. If assignment A is complete then return A
2. X ← **select** a variable not in A
3. D ← **select** an ordering on the domain of X
4. For each value v in D do
   a. Add (X←v) to A
   b. var-domains ← forward checking(var-domains,X,v,A)
   c. If a variable has an empty domain then return failure
   d. result ← CSP-BACKTRACKING(A, var-domains)
   e. If result ≠ failure then return result
   f. Remove (X←v) from A
5. Return failure

How to select in (2)?

- Most constrained: select the variable with the smallest remaining domain, (Rationale: Minimize the branching factor).
- Most constraining: Among the variables with the smallest remaining domains, select the one that appears in the largest number of constraints on variables not in the current assignment, (Rationale: Increase future elimination of values, to reduce future branching factors)

How to select in (3)?

- least constraining: Select the value of X that removes the smallest number of values from the domains of those variables which are not in the current assignment, (Rationale: Since only one value will eventually be assigned to X, pick the least-constraining value first, since it is the most likely not to lead to an invalid assignment).

Constraint Tree (backtrack free)

- Order the variables from the root to the leaves → (X1, X2, …, Xn)
- For j = n, n-1, …, 2 do
  REMOVE-ARC-INCONSISTENCY(Xi, Xp(i))
- For i=1…n do
  assign any legal value to all Xi consistent with Xp(i).

Xp(i) = the parent of variable X.

REMOVE-ARC-INCONSISTENCY (A, B): remove values from domain B which makes A's domain empty.

Propositional Logic

Truth Table:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
F	F	T	F	F	T	T
F	T	T	F	T	T	F
T	F	F	F	T	F	F
T	T	F	T	T	T	T

$P \Rightarrow Q$: "if P is *true* than I'm claiming that Q is *true*. Otherwise I'm making no claim". $(P \Rightarrow Q) \equiv (\neg P \vee Q)$

Inference rules:

$\alpha \vdash \beta \equiv \frac{\alpha}{\beta}$ Means "β can be derived from α by inference."

Modus Ponens/ Implication elimination	$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$
And- Elimination	$\frac{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}{\alpha_i}$
And- Introduction	$\frac{\alpha_1, \alpha_2, \ldots, \alpha_n}{\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_n}$
Or- Introduction	$\frac{\alpha_i}{\alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_n}$
Double Negation Elimination	$\frac{\neg\neg\alpha}{\alpha}$
Unit Resolution	$\frac{\alpha \vee \beta, \neg\beta}{\alpha}$
Resolution	$\frac{\alpha \vee \beta, \neg\beta \vee \gamma}{\alpha \vee \gamma}$ or $\frac{\neg\alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\neg\alpha \Rightarrow \gamma}$