SEARCH TREE

Node: State in state tree

Root node: Top of state tree

Children: Nodes that can be reached from a given node in 1 step (1 operator)

Expanding: Generating the children of a node

Open: Node not yet expanded

Closed: Node after expansion

Queue: Ordered list of open nodes

SEARCH

BLIND SEARCH: *Systematic Search*

Depth–1st: Continue along current path looking for goal

Breadth–1st: Expand all nodes at current level before progressing to next level

Depth–limited Search: Depth-1st + depth-limit

Iterative Deepening Search: limit=0,limit=1, ...

USING COST: $g(n)$=*cost from start to n*

Uniform-Cost Search (= Branch-and-bound): Select node n with best $g(n)$.

USING HEURISTIC: $h(n)$=*Estimate cost to a goal*

Greedy Search: Select node n with best $h(n)$

A*: Select node n with best $f(n) = g(n) + h(n)$

IDA*: A* + f-cost limit.

Hill-Climbing: Depth-1st exploring best $h(n)$ first

Simulated Annealing: Hill-Climbing + RandomWalk

Beam Search: Breadth-1st keeping only m nodes with best $h(n)'s$ per level

DEPTH–1st SEARCH

1. Put start state onto queue
2. If queue is empty then fail
3. If head of queue is goal then succeed
4. Else remove head of queue, expand it, place children in front of queue
5. Recurse to 2

DEPTH–1st (cont.)

When to use

- Depth limited or known beforehand
- All solutions at same depth
- Any solution will do
- Possibly fast

When to avoid

- Large or infinite subtrees
- Prefer shallow solution

BREADTH–1st SEARCH

1. Put start state onto queue
2. If queue is empty then fail
3. If head of queue is goal then succeed
4. Else remove head of queue, expand it, place children at end of queue
5. Recurse to 2

BREADTH–1st (Cont.)

When to use

- Large or infinite search tree
- Solution depth unknown
- Prefer shallow solution

When to avoid

- Very wide trees
- Generally slow
- May need a lot of space

MODIFICATIONS TO DEPTH/BREADTH 1ST

Depth–limited Search:
Limit the total depth of the depth 1st search.

Iterative Deepening Search:
Repeat depth–limited search with limit 0, 1, 2, 3, . . . until a solution is found.

Bidirectional Search:
Simultaneously search forward from initial state and backward from goal state until both paths meet.

UNIFORM–COST SEARCH (= BRANCH–AND–BOUND)

1. Put start state onto queue
2. If queue is empty then fail
3. If head of queue is goal then succeed
4. Else
   - remove head of queue,
   - expand it,
   - place in queue, and
   - **sort entire queue** with **least cost-so-far** nodes in front
5. Recurse to 2

UNIFORM–COST SEARCH SUMMARY

Advantages

- Optimal (when costs are non–negative)
- Complete

Disadvantages

- Can be inefficient

When to use

- Desire best solution
- Keep track of cost so far

When to avoid

- May not work with negative costs
- May be overly conservative
- Any solution will do

Potential improvement

- Dynamic Programming

UNIFORM–COST SEARCH + DYNAMIC PROG.

1. Put start state onto queue
2. If queue is empty then fail
3. If head of queue is goal then succeed
4. Else
   - remove head of queue,
   - expand it,
   - place in queue,
   - ⋆ remove redundant paths:
     Paths that reach the same node as other paths but are more expensive, and
   - **sort entire queue** with **least cost-so-far** nodes in front
5. Recurse to 2

GREEDY SEARCH (= called BEST–1st SEARCH in other textbooks)

1. Put start state onto queue
2. If queue is empty then fail
3. If head of queue is goal then succeed
4. Else
   - remove head of queue,
   - expand it,
   - place in queue, and
   - **sort entire queue** with **least estimated-cost-to-goal** nodes in front
5. Recurse to 2

GREEDY SEARCH SUMMARY

Advantages

- Can be very efficient
- Paths found are likely to be short

Disadvantages

- Neither optimal nor complete

When to use

- Desire “short” solution

When to avoid

- When an optimal solution is required

A*

1. Put start state onto queue
2. If queue is empty then fail
3. If head of queue is goal then succeed
4. Else remove head of queue, expand it, place in queue, and **sort entire queue** with **least cost-so-far** + **estimated-cost-remaining** nodes in front
5. If multiple paths reach a common goal, keep only lowest cost-so-far path
6. Recurse to 2

- $f(\text{node}) = g(\text{node}) + h(\text{node})$, where
  - $f(\text{node})$ = estimated total cost
  - $g(\text{node})$ = cost-so-far to node
  - $h(\text{node})$ = estimated-cost-remaining (*heuristic*).
- Properties of h:
  - Lower bound ($\leq$ actual cost)
  - Nonnegative

A* SUMMARY

Advantages

- Complete
- Optimal, when h is an underestimate
- Optimally efficient among all optimal search algorithms

Disadvantages

- Very high space complexity

When to use

- Desire best solution
- Keep track of cost so far
- Heuristic information available

When to avoid

- No good heuristics available

HILL CLIMBING SEARCH version 1: with backtracking

1. Put start state onto queue
2. If queue is empty then fail
3. If head of queue is goal then succeed
4. Else remove head of queue, expand it, place **children sorted by** $h(n)$ in front of queue
5. Recurse to 2

HILL CLIMBING SEARCH
version 2: without backtracking
arguably this is the most common version of hill climbing

1. Put start state onto queue
2. If queue is empty then fail
3. If head of queue is goal then succeed
4. Else remove head of queue, expand it, sort the children by $h(n)$, and place only the child with the best $h(n)$ in (front of) queue
5. Recurse to 2

HILL CLIMBING SUMMARY

Advantages

- Complete if backtracking is allowed (like in Winston's book) and the graph is finite

Disadvantages

- Not optimal
- Not complete if backtracking is not allowed

When to use

- Depth limited or known beforehand
- All solutions at same depth
- Desire good solution
- Reliable estimate of remaining distance to goal
- Fast if good estimate

When to avoid

- If optimal solution is required
- Large or infinite subtrees
- No good estimate
- Difficult terrain

BEAM SEARCH

1. Put start state onto queue
2. If queue is empty then fail
3. If head of queue is goal then succeed
4. Else remove head of queue, expand it, place children at end of queue
5. If finishing a level, keep only w best nodes in queue
6. Recurse to 2

BEAM SEARCH SUMMARY

Advantages

- Saves space

Disadvantages

- Neither optimal nor complete

When to use

- Large or infinite search tree
- Solution depth unknown
- Prefer shallow solution
- Possibly fast
- No more than wb nodes stored

When to avoid

- Can't tell which solutions to prune
- Prefer conservative

SEARCH STRATEGIES

Completeness; Optimality; and Time and Space Complexity

Search	Complete?	Optimal?	Time	Space
Depth-1st	N	N	b^d	bd
Breadth-1st	Y	Y*	b^s	b^s
Depth-limited	N	N	b^l	bl
Iter. deepening	Y	Y*	b^s	bs
Uniform-cost	Y	Y	b^s	b^s
Greedy	N	N	b^d	b^d
A*	Y	Y	exp	exp
Hill-climbing	N	N	dep	dep
Beam	N	N	ms	$2m$

(adapted from Russell & Norvig's book)

- Y*: Yes, IF cost of a path is equal to its length. Otherwise No.
- b: branching factor
- s: depth of the solution
- d: maximum depth of the search tree
- l: depth limit
- m: beam size
- exp: exponential depending on heuristic h
- dep: depends on heuristic h

SEARCH STRATEGIES
Summary

Depth 1st: Continue along current path looking for goal

Breadth 1st: Expand all nodes at current level before progressing to next level

Hill Climbing: Like depth 1st, but explore most promising (according to heuristic) children first (if allowing backtracking) or just the most promising child only (if not allowing backtracking)

Beam: Like breadth 1st, but prune unpromising (according to heuristic) children

Greedy: Expand best (according to heuristic) open node regardless of its depth

Uniform-Cost: Expand the least-cost-so-far node until goal reached

A*: Like uniform search, but with
least-cost-so-far node + heuristic value