Search

Terminology

- **search tree (graph) –** the path the search algorithm follows in exploring the state space via an initial state and a successor function
  - o **search node** – a state from the state space which has a successor function. A node is comprised of the following:
    1. state – the state the node represents
    2. parent – the predecessor of the node.
    3. action – the action applied to the parent to reach the node.
    4. **path-cost *g(n)*** – the cost of the path from the initial state.
    5. depth – the number of search steps along the path.
  - o *expanding a node* – generating a new set of states via the node's successor function. A node is not checked to be terminal until it is expanded.
  - o Note that several nodes in the search tree may contain the same states, generated by different paths. Hence, *the search becomes a graph in state space.*
- **search strategy** – the methodology for choosing the next node to expand.
- **fringe** – the collection of nodes generated but not yet expanded.
  - o this collection typically imposes an ordering on which nodes in the collection will be expanded next based on a preference → **queue**.

Assessing Algorithms

- Performance Measures for our algorithms:
  - o **completeness –** Is algorithm guaranteed to find an existing solution?
  - o **optimality** – Does the algorithm find the optimal solution first?
  - o **time complexity –** How long does it take to find a solution
  - o **space complexity –** How much memory is needed to find a solution?
- Relevant quantities:
  - o **branching factor** *b* – maximum number of successors of a node.
  - o *d* – depth of the shallowest goal node.
  - o *m* – maximum length of any path in the state space.
- **path cost –** a function used to define a numeric cost to each path.
- **search cost –** the cost required to find a particular solution… typically time complexity.
- **total cost –** a combination of search cost and path cost according to some tradeoff between the two.

Uninformed (Blind) search – search solely on the basis of being to expand the successors of a state and being able to distinguish a goal-state.

Criterion	**BFS**	**Uniform**	**DFS**	**DLS**	**Iterative**	**Bidirect.**
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O\left(b^{d+1}\right)$	$O\left(b^{\lceil C^*/\varepsilon \rceil}\right)$	$O\left(b^m\right)$	$O\left(b^l\right)$	$O\left(b^d\right)$	$O\left(b^{d/2}\right)$
Space	$O\left(b^{d+1}\right)$	$O\left(b^{\lceil C^*/\varepsilon \rceil}\right)$	$O(bm)$	$O(bl)$	$O(bd)$	$O\left(b^{d/2}\right)$

1. complete if *b* is finite.
2. complete if step cost is at least $\varepsilon > 0$.
3. optimal if step costs are all identical.
4. if both directions use BFS.

- **Breadth-first Search –** all nodes at a given depth in the search tree are expanded before any of the nodes at larger depths → implemented with FIFO queue
  - *complete*: if the shallowest goal node is at depth *d*, it will be found after searching over all shallower nodes and other nodes at depth *d*.
  - *optimal* if the path cost is a nondecreasing function of depth of a node.
  - *Memory requirements of breadth-first search are crippling.*
    - Every node must remain in memory → $O\left(b^{d+1}\right)$
  - *Uniformed exponential-complexity searches only can be solved for small (trivial) instances.*
- **Uniform-cost Search –** expands the next unexpanded node with the *lowest path cost* → implemented by a priority queue. When costs are equal, becomes BFS.
  - *complete* and *optimal* provided the cost of every step is at least $\varepsilon > 0$.
  - Let C^* be the cost of the optimal solution. Then the space and time-complexity is $O\left(b^{\lceil C^*/\varepsilon \rceil}\right)$.
- **Depth-first Search –** always expands the *deepest* node in the current fringe of the search tree (search backs-up when path unsuccessful) → implemented by Stack.
  - *Incomplete* for non-finite search trees. Always *non-optimal*.
    - In worst case, DFS will end up exploring $O\left(b^m\right)$ nodes where *m* is the maximum depth of any node.
  - Memory requirement only $O(bm)$ where *m* is the maximum depth.
  - **backtracking search –** uses only $O(m)$ memory by only remembering a single successor at each level by having each node "remember" which node to generate as next unexplored successor for backtracking.
    - utilizes idea of generating a successor by modifying current state.

- **Depth-limited Search** – depth-first search with a predetermined depth limit *l*. Becomes DFS when $l = \infty$.
  - *incomplete* if $l < d$. *non-optimal* if $l > d$.
  - time complexity: $O(b^l)$. space complexity: $O(bl)$
  - the **diameter** of the space provides a good clue about the value to choose for *l*, but it is hard to discover the diameter without solving the problem.
- **Iterative Deepening Depth-first Search** – iteratively repeated depth-limited search where *l* is increased by 1 on each iteration from an initial value of 0. This combines the benefits of BFS and DFS.
  - *complete* when branching factor is finite.
  - *optimal* when the cost is a non-decreasing function of depth.
  - space complexity: $O(bd)$
  - *Insight*: most of the nodes are in the bottom-most level so repeating upper levels is not that bad of an idea.
    - time complexity: $O(b^d)$… a factor of *b* better than BFS.
  - *In general, iterative deepening is the preferred method of uninformed search when there is a large search space with unknown solution depth.*
  - **iterative lengthening search –** iterative search on increasing *path-costs* analogous to uniform cost search.
    - incurs substantial overhead compared to uniform-cost search.
- **Bidirectional Search** – simultaneous searches from the initial state forward and from the goal state backwards that stop when the 2 searches meet. Encouraged by the fact that $b^{d/2} + b^{d/2} \ll b^d$
  - *complete* & *optimal* (with uniform step costs) if both algorithms are BFS.
  - Checking a node for membership in the other search tree can be done in constant time via a hash table, but requires that 1 search tree be in memory.
    - Time-complexity: $O(b^{d/2})$ Space-complexity: $O(b^{d/2})$
  - Bidirectional search requires that the **predecessors** of a node be efficiently computable:
    - Easy when actions are *reversible*. Otherwise…
  - To deal with several (explicitly listed) goal states, we make them all have a successor of a single *dummy goal state.*

- **Avoiding Repeated States –** avoiding repeated visits to states that have already been visited could result in substantial savings in space and time. *Algorithms that forget their history are doomed to repeat it.*
    - repeated states are unavoidable is some problems. e.g. reversible actions.
    - Repetition Detection usually requires comparing new node to those that have already been expanded.
        - Once found, one of the paths to a repeated state can be discarded.
    - DFS can only avoid the exponential proliferation of non-looping paths by keeping more nodes in memory.
    - Algorithms can simply remember every state that has been visited.
        - **closed list –** stores all expanded nodes.
        - **open list –** stores all nodes on the fringe.
        - *In worst-case, time/space is proportional to the size of the state space.*
    - Optimality
        - Uniform-cost search and BFS (constant step size) are both optimal graph-search algorithms
        - Iterative-deepening needs to check if new path is better and if so, it must revise costs of all paths going through the altered state.

Searching with Partial Information

- ***Sensorless (Conformant) Problems*** – agent has no sensors.
    - agent must be able to reason about a set of possible states.
    - **belief state –** a set of states representing the agent's belief of what states it might be in. In general, environment of *S* states has 2^S belief states.
    - **coercion –** executing actions that cause the agent's belief state to collapse to a certain set of states.
        - *solution* – coercing the belief state to a set of all goal states.
- ***Contingency Problems* –** environment is partially observable or the outcome of an agent's actions is uncertain.
    - **adversarial –** uncertainty is caused by actions of other agents.
    - **contingency plan** - trees of decisions made based on the current set of percepts made after the last action.
    - *Agent can act before finding a guaranteed plan*
        - idea of acting and seeing what contingences actually arise.
        - *interleaving* of search and execution also useful in exploration.
- ***Exploration Problems*** – when states and actions are unknown, agent must explore.

4: Informed Search and Exploration

- **Informed Search –** uses problem-specific knowledge beyond the problem's definition.
- **Best-First Search** – general Tree (Graph) Search where node's are selected based on an evaluation function *f(n)* – cost of cheapest path to goal through node n.
- **Greedy Best-First Search** – Assumes *f(n)* = *h(n)*; a heuristic function.
    - susceptible to false starts
    - not optimal; incomplete.
    - Worst case time and space: $O(b^m)$.

A* Search – f(n) = g(n) + h(n) where g(n) is the cost to reach the node n and h(n) is a heuristic function estimating the cost to a goal through n ➔ estimated cheapest cost through n.

- A* is *optimal* if h(n) is an *admissible* (and *consistent* for Graph-Search) heuristic.
- A* is *complete*.
- If h(n) is *consistent*, the values of f(n) along any path are nondecreasing!
- A* searches on contours of cost in the state space.
    - C* is the cost of the optimal solution path
        - A* expands all nodes such that $f(n) < C^*$.
        - A* expands some nodes on the goal contour $f(n) = C^*$.
        - A* never explores nodes with $f(n) > C^*$ ➔ **pruned** – elimination of possibilities without considering them.
- A* is **optimally efficient** for any given heuristic since any algorithm that doesn't expand a node n with $f(n) < C^*$ might miss the optimal solution.

Heuristic Functions

- **Heuristic Function** h(n) – estimated cost of cheapest path to goal through node n.
- **Admissible Heuristic –** h(n) never overestimates the cost to reach a goal.
- **Consistent (Monotonic) Heuristic** – h(n) is not more than the cost through n to n' plus h(n'). Thus, a general triangle inequality:

$$h(n) \le c(n, a, n') + h(n')$$

- **Effective Branching Factor (b*) –** the branching factor of a uniform tree of depth d with N+1 nodes would have to have given that A* has generated N nodes with depth d.
- **Dominance –** a heuristic h_1 is said to **dominate** another heuristic h_2 if, for any node n, $h_1(n) \ge h_2(n)$.
    - *A heuristic will never expand more nodes in A* than any other heuristic it dominates.*
    - *Every node surely expanded by search with A* under the dominant heuristic will also surely be expanded by the dominated heuristic.*
- **Relaxed Problem –** a problem with fewer restrictions on the actions allowable in the problem domain.
    - *The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem!*
    - The "relaxed problem" heuristic must obey the triangle inequality, hence it is consistent.
    - The relaxed problem must be easily solved to be used as heuristic.
- <u>MultiHeuristic</u>: if we have a set of heuristics $\{h_i\}$ we can combine them into a single heuristic:

$$\tilde{h}(n) = \max_i \{h_i(n)\}$$

    - if $\{h_i\}$ is admissible, $\tilde{h}$ is admissible.
    - $\tilde{h}$ is also consistent and dominates each of its components.
- **Pattern Databases –** a database of solutions to every subproblem.
    - **Disjoint Pattern Databases –** sum of costs of two subproblems is a lower bound on the cost of solving the entire problem.
    - The solution cost of a subproblem can thus be used to form an admissible heuristic.
- <u>Learning Heuristics</u> – learn a heuristic from experience in solving problem repeatedly.
    - use *inductive learning algorithm* to construct a function h(n) that can predict cost of other states that arise in search.
    - typically use *features* of a state that are relevant to its evaluation. e.g. linear combination of features.

Local Search

- **Local Search Algorithms –** When the path to reach goal is irrelevant, local search are methods for only maintaining *current state* and (generally) only moving to its neighbors. Often used in optimizations where the goal is to minimize a objective function.
- *Advantages:*
    1. small memory requirement
    2. reasonable solutions in large spaces
- **state space landscape** – the space of possible states defined by a "*location*" corresponding to state and a "*elevation*" corresponding to an evaluation, cost, or objective function.
- **complete local search** – always finds a goal (if any exist)
- **optimal local search –** always finds the global min/max.
- **greedy local search –** moves to "good" neighbor without considering future.

- **Hill-Climbing Search** – Always moves in "uphill" direction to maximize objective only searching amongst immediate neighbors of current state and terminating when no improvements can be made ➔ *greedy*.
    - Problems
        - Local Max/Min
        - Ridges
        - Plateaux
    - <u>sideways moves</u> – moves along "flat" objective to get off plateau.
    - **Stochastic Hill Climbing –** random uphill moves where probability of choice depends on steepness of climb.
    - **First-Choice Hill Climbing –** generate successors until one is better than "current" and take it.
    - **Random Restart Hill Climbing –** succession of hill climbs with random initial state. If probability of success is p, expected number of restarts is 1/p.
    - NP-hard problems typically have exponential number of local min/max.

- **Simulated Annealing –** Hill-Climbing with random walk thus giving efficiency and completeness.
    - Candidate move β is randomly selected. If candidate is uphill, it is always accepted. Otherwise, it is accepted with a probability exponentially decreasing with "badness" ΔE and decreasing as temperature T is lowered → *Boltzmann Distrubution*.

      $$P\left(n_{t+1}=\beta\right)=\min\left(1,\exp\left\{\Delta E\left(\beta\right)/T\right\}\right)$$

    - If the *schedule* for T cools "slowly enough", simulated annealing finds global optimum with probability approaching 1.

- **Local Beam Search –** maintains the k "best" successor states; an approach more powerful than k independent searches since information effectively passes between the "search threads."
    - **Stochastic Beam Search –** chooses next k states at random with a probability of a state as an increasing function of the states value. This approach helps alleviate "lack of diversity."

- **Genetic Algorithms** – A variant of stochastic beam search in which successors are generated through combinations of 2 current states.
    - **population –** the k states maintained by the algorithm
    - **individual –** an instance in the state space.
    - **fitness function –** an evaluation function that returns higher values for better states.
    - *Essence of Algorithm*
        - Parents are randomly selected with probabilities related to their fitness.
        - **Crossover** points are selected randomly in accordance with the rules of the state.
        - **Random Mutation** occurs in each successor with some small probability.
    - **Schema –** a substring (representing state) in which some positions of state have been left unspecified. *Instances* are strings that match a schema.
        - If the average fitness of the instances of a schema is above the mean, then the number of instances of that schema within the population will grow over time.
        - GA's work best when schema correspond to meaningful components of the solution.

- *Continuous Spaces*
  - **Gradient Descent (Ascent) –** moves the current solution in the direction of the gradient in the state-space landscape.

    Gradient - $\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$

    Update - $\mathbf{x} \leftarrow \mathbf{x} + \alpha \nabla f(\mathbf{x})$

    - Gradient is local, not global direction.
    - **empirical gradient –** estimation of gradient for non-differentiable objective function by calculating f in a close neighborhood around x.
    - If step size α is too small, too many steps are needed. If it's too large, the steps overshoot the extrema. **Line Search** dynamically chooses α in some scheme.
  - Newton-Raphson Method
    - Newton's Method for iteratively finding roots:

      $$x \leftarrow x - g(x)/g'(x)$$

    - To find min/max, we need to find roots of gradient:

      $$\mathbf{x} \leftarrow \mathbf{x} - \mathbf{H}_f^{-1}(\mathbf{x}) \nabla f(\mathbf{x})$$

      where $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$; the **Hessian**
  - **Constrained Optimization –** an optimization in which solutions must satisfy hard constraints for the values of each variable.
    - **linear programming –** constraints must be linear inequalities forming a convex region and objective function is linear.

Online Agents

- **Online Search** – agent interleaves action and computation by taking actions and observing environment to determine result of the action.
  - Online search necessary in *exploration* where states and actions are unknown.
  - Essentials:
    - Action(s) – returns actions for state s.
    - **step-cost function** c(s,a,s') – determines cost of step s to s' by way of action a.
    - Goal-Test(s) – determines if s is a goal.
  - **competitive ratio** – comparison of cost of path that agent actually travels to the cost of the path of the agent would travel if it knew the search space a priori.
  - *No algorithm can avoid dead ends in all possible search spaces* ➔ *Adversarial Argument.*
  - **safely explorable state space –** space in which every state can reach some goal state eventually.
  - Hill-climbing is already a local search
    - local minimum can be dealt with via *random walks*.
    - estimated cost to reach cost through neighbor s' is the cost to get to s' plus the estimated cost to reach the goal from there (updated):

      $$c(s,a,s')+H(s')$$

    - **optimism under uncertainty** – encourages agent to explore new paths by giving new states least possible cost.