Search Review

Classification of Search Problems

Search problems can be divided into three different categories:

1. Classical Search

- observable, deterministic, known environment

2. Optimization

- solution matters, but not how you get it

3. Constraint Satisfaction

- sate is not a black box or indivisible, but it can be represented as a set of variables

Problem Formulation

1. initial state and state space
    - initial: where agent starts from
    - state space: set of all reachable states from
2. action space
    - set of all possible actions that an agent can take
3. transition model
    - returns the next state s' that results from taking action a in current state s
4. goal test
    - returns if a particular state s is the goal state
5. path cost
    - numeric cost for a path (sequence of states connected by a sequence of actions)

Loops When Searching for a Solution

A solution is just a sequence of actions that can be taken. In order to find the appropriate solution we can model all the possible solutions as a graph or as a tree.

Using graph search introduces a set that keeps track of the already visited nodes and removes redundant nodes and cyclic solutions.

Performance

Elements to measure for performance:

1. completeness
2. optimality
3. time complexity
4. space complexity

What does high complexity look like:

- high branching factor
- high depth of goal node
- high max length of any path in the state space

In order to look at performance of the solution and not the algorithm we take a look at the search cost and total cost.

Further Classification

Backtracking (CPS)

Expansion Search

- Uninformed
  - BFS
  - Uniform-cost
  - DFS
    - depth-limit
    - iterative deepening
- Informed
  - Gready BFS
    - Memory Bounded
  - A* Search

Local Search (optimization)

- Hill-climbing and variants
- Simulated Annealing
- Genetic algos

Uninformed Search

These types of searches generate successors and have clear distinguishing factors from a goal state and a non-goal state since it only considers the problem definition itself.

BFS

- expands solution tree by level, one level at a time
- stops whenever a goal node is reached or there are no more nodes
- optimal if path costs increases with depth
- a queue is used to implement it

Uniformed-cost Search

- used when path cost is important as it always expands the node with the lowest path cost
- goal test when a node is selected for expansion
- optimal if step cost is a non-negative number
- can result in worse time results than BFS if all path costs are the same
- an example of this search is Dijkstra's

DFS

- implemented using a stack
- will fully expand each state first and then move to the next. Similarly to backtracking
- sub-optimal
- no clear advantage in term of time complexity
- only need to store a single path form the root to a leaf node and the remaining unexpanded sibling nodes for each node on the path
- can fail embarrassingly in infinite state space

Limited

This solves the case were the state space is infinite and makes an infinite loop with DFS. This only adds a depth limit and does not actually solve the problem with the infinite path case.

Iterative Deepening

- combination of DFS and BFS
- gradually increases the depth limit until a goal is found
- preferred when search space is large and the depth of the solution is unknown

Bidirectional Search

- DFS and BFS can sufferer from exponential time complexity
- run two simultaneous searches one forward from the start state and another backwards from the goal state
- optimal if path cost increases with depth

Informed Search

Greedy BFS (Best-first Search)

This is the same as BFS only difference is that it uses an evaluation function to expand the node that appears to be the closest to the goal based on metrics that you decide.

One way to implement the evaluation function is to use the straight line distance heuristic.

- not optimal and sometimes can get deceived and find really bad solutions since
- relatively fast so it is often used for problems with high complexity

A*

Built to minimize the total estimated solution cost. In order to avoid expanding paths that are already expensive we take a total path cost, f, that adds the cost from the root, g, + the heuristic function, h.

Optimal in trees if it is admissible (underestimate) and optimal in graphs if it is consistent (overestimate).

There can be a path that will "trick" the algo into choosing a less than optimal path, there are some ways to get around it like allowing adding a node that has already been visited into the queue of nodes to explore and update the path cost with new path cost if cheaper.

Local Search

With previous search problems the path taken to get to the goal was the solution but with local search a state is the solution. Ex for the N Queens problem the path to getting to the correct position of the queens on the chess board does not matter, only the final state with all the N queens on the board.

Local search algorithms can be used to:

- keep track of a single current node (rather than multiple paths)
- move only to neighbors of that node
- not keep track of paths

Basic Idea of Local Search

```
# initialize to something, usually a random initial state
# or human-generated initial state
best_found = current_State

for any_amount_of_time:
    if (tried of doing it) then return best_found
```

```
    else
        current_state = MakeNeightbor(current_state)
        if (Cost(current_state) < Cost(best_found)) then
            # keep best result so far
            best_found = current_state
```

Local search in Continuous Space

Gradient descent: - assume we have a continuous cost function - we want to minimize it over continuous variables 1. computer the gradient with respect to every node 2. take a small step downhill in the opposite direction of gradient with step size 3. repeat

Hill Climbing Search

- loop that continuously moves in the direction of increasing value (steepest-ascent version)
  - uses objective function value or heuristic function value
- terminates when it reaches a "peak" where no neighbor has a better value
- no search tree, only record the state and the value of the objective function

Local Search Difficulties

Local Optima:

- local maximum (or minimum) that might be the best option considering a few neighbors but not the entire global scope

Plateau:

- flat area of the state space landscape (a flat local optima)

Ridges:

- a sequence of local optima that algos have to navigate
- neighbors might appear to go the opposite way but have a trend to go upward in the global scope

Escaping Local Optima

Sideways move:

- if no uphill moves, allow sideways moves in hope that the algorithm can escape
- need to place a limit of the possible number of sideways moves to avoid infinite loops

Stochastic (randomized) hill climbing:

- iterate the process of randomly selecting a neighbor for candidate solution
- accept it only if results in an improvement
- can generate a set of neighbors for the current state and pick the best one among the selected neighbors

Random restart:

- if at first you don't succeed try with a different initial state

Simulated Annealing

The idea of this is to escape local optima by allowing some "bad" moves but gradually decrease their frequency. A way of allowing "bad" moves is by instead of picking the best next move, randomly pick a move from a list of successor neighbors.

Local Beam Search

The idea of this is to keep only one node in memory in order to preserve low memory usage in algorithms with extreme memory use.

Steps:

1. create k random initial states
2. generate their children
3. select the k best children
4. repeat indefinitely

Genetic Algorithms

Produce the net generation of states by "simulated" evolution"

- crossover has the effect of jumping to a completely different new part of the search space (non-local)

Constraint Satisfaction Problems

Regular search problems:

- search in a state space
- each state is indivisible (a black box with no internal structure)

Constraint satisfaction problem

- each state has a factored presentation: a set of variables, each has a value
- goal condition consists of a set of sub-conditions: each variable should have a value that satisfies all the constraints on the variable

Consists of 3 components:

- finite set of variables
- finite set of domains
  - one for each variable, finite or infinite domain
- finite set of constraints
  - each constraint involves some subset of the variables
  - <scope, relations>
  - specifies allowable combinations of values for that subset

State space: each stat is defined by an assignment of values to some of all of the variables

Goal: find a complete and consistent assignment

Point of local search for CSP's is to eliminate the violated constraints and to help with choosing the next variable.

Commutativity

- A problem is commutative if the order of any given set of actions has no effect on the outcome
- CSP's are commutative when they can reach the same partial assignment regardless of order
  - Ex. map coloring for Australia (WA=red then NT=green is the same as NT=green then WA=red)
- only consider a single variable at each node in the search tree
- there are leaves
- still need to figure out which variable to choose at each node

Backtracking

Basic `uninformed` search algorithms to solve CSP's similar to DFS, eploring one node at a time then backtrack when there are no more legal moves left to assign while a solution is not found.

Basic example:

```
def backtracking_search(csp):
    return recursive_backtracking({}, csp)

def recursive_backtracking(assignment, csp):
```

```
    if assignment is complete then return assignment

    var = select_unassigned_variable(variables[csp], assignment, csp)

    for each value in order_domain_values(var, assignment, csp):
        if value is consistent with assignment given constraints[csp] then
            assignment.append({var, value})
            result = recursive_backtracking(assignment, csp)
            if result != failure then return result
            assignment.pop()
    return failure
```

Improving Backtracking

Choosing a value Minimum remaining values (MRV):

- way to choose improve on what variable to choose next by going with the var with the fewest legal left values in its domain
- if some variable X has no legal values left select X and failure will be detected immediately avoiding pointless maybe expensive searches through other variables

Degree heuristics:

- select variable that is involved in the largest number of constraints on other unassigned variables
- MRV doesn't help in choosing the first region to color so this helps with that

MRV is usually a more powerful guide, but degree heuristic can be useful as a tie breaker.

Order of trying values Least constraining value:

- Prefer the value that rules out the fewest choices for the neighboring variables in the constraint graph
- leaves the maximum flexibility for subsequent variable assignments
- we use least instead of most because we only care about having one solutions so we look for the most likely values first

Early failure Forward checking:

- keep track of remaining legal values for unassigned variables and cross off bad options
- terminate when any variable has no legal values
- doesn't provide early detection for all failures
-

Constraint propagation:

- using constraints to reduce the domain for a variable, which in turn can reduce the domain for another variable, and so on
- goes further than forward checking because constraints propagation repeatedly enforces constraints locally

Arc Consistency is a systematic procedure for constraint propagation. An arc in consistent if for a value x of X there is some value y of Y consistent with x (without violating the constraint)

Local Search for CSP's

```
def min_conflicts(csp, max_steps):
    current = initial comple assignment for csp

    for _ in range(1, max_steps):
        if current is a solution for csp then return current

        var = randomply chosen conflicted vraible from csp
        value = the value v for var that minimized conflicts(var, v, current, csp)
```

```
    var = value in current
return failure
```