CSC384 Notes

---

1. Uninformed and Heuristic Search

Search Algorithms

Breadth First Search - BFS

Algorithm

Expand all nodes reachable from start in $i, i+1 \ldots$ steps

Properties

- b : maximum number of successors of any node
- d: depth of the shortest solution

Time Complexity: $\mathcal{O}(b^{d+1})$

Space Complexity: $\mathcal{O}(b^{d+1})$

Optimality: Yes, shortest solution

Completeness: Yes

✓ Solution is complete and optimal

× Typically run out of space

Depth-First Search - DFS

Algorithm

Place new paths that extend the current path at the front

Properties

Optimality: No

Completeness: Yes if $\exists$ finite and acyclic paths

Time Complexity: $\mathcal{O}(b^m)$, where m is the length longest path

Space Complexity: $\mathcal{O}(bm)$

✓ Only need to store current path + backtracking nodes

× May not be optimal/complete

Depth Limited Search

Algorithm

Perform DFS but only to a depth limit d

Properties

Optimality: No

Completeness: No, only if solution has depth $\leq d$

Time Complexity: $\mathcal{O}(b^m)$, where m is the length longest path

Space Complexity: $\mathcal{O}(bm)$

✓ No infinite length paths now

$\times$ Not optimal/complete

Iterative Deepening Search

Algorithm

Iteratively increase depth limit and perform depth limited search

Properties

Optimality: Yes

Completeness: Yes, if solution exist with depth $= d$

Time Complexity: $\mathcal{O}(b^d)$

Space Complexity: $\mathcal{O}(bd)$

✓ Will find shortest length solution with small space requirement

Cost Sensitive Search

Uniform-Cost Search

Frontier is a priority queue ordered by min cost, always expand the least cost path. Terminate **only** when goal is removed from Frontier.

Path Checking

Avoid running into cycles, will need to check if child about to expand is already a node in the current path

Cycle Checking

Check if SAG was in the frontier then will not add SCG, need to keep track of all possible nodes and its current min cost to reach.

- The first time uniform-cost expands a state, it has found the min cost path to it

Space Complexity: $\approx$ space of BFS

Properties

Optimality: Yes

Completeness: Yes, assuming cost > 0

Time Complexity: $\mathcal{O}(b^{C^*/\epsilon+1})$, where C^* is the cost of optimal solution, and $\epsilon + 1$ is the min cost

Space Complexity: $\mathcal{O}(b^{C^*/\epsilon+1})$

✓ Optimal and complete

$\times$ No information about goal state, need to explore w/o direction

Heuristic Search

Best First Greedy Search

Greedily explore the min cost path using heuristic value.

A* Search

Take into account the cost of getting to the node + estimate of the cost of getting to the goal from the node

$f(n) = g(n) + h(n)$

Property

- Only stop when we dequeue the goal, otherwise may not be best path

Complexity:

- When $h(n) = 0$, then same with uniform-cost search
- When $h(n) > 0$ and admissible, then expand $\leq$ nodes than uniform-cost search

Weighted A* Search

A* but with a bias toward states near the goal

$f(n) = g(n) + \epsilon * h(n)$

Property

- Paths may be sub-optimal, but may implement anytime algorithm to optimize

Iterative Deepening A^* - IDA*

Want to reduce memory of A^*

Iteratively search for solution with cost $< f = g + h$ cutoff, only explore the most promising path, space << A^*.

Time Complexity: $\mathcal{O}(b^m)$, same with A^*

Space Complexity: $\mathcal{O}(bd)$

Optimality: Yes

Completeness: Yes

Admissibility and Monotonicity

Consistency $\Rightarrow$ admissible

Admissible

Let $h^*(n)$ be the true cost of an optimal path from n to a goal node.

Then an admissible heuristic satisfies $h(n) \leq h^*(n)$, so that we never ignores a promising path while searching.

Monotonic (consistent)

A monotone heuristic satisfies the condition

$$h(n_1) \leq c(n_1, a, n_2) + h(n_2)$$

This condition must hold for all transition from $n_1 \rightarrow n_2$

Properties of Monotonicity

1. f-values of states in a path are non-decreasing
2. If n_2 is expanded after n_1, then $f(n_1) \leq f(n_2)$
3. If node n has been expanded, every path with a lower f-value than n has already been expanded
4. The first time A^* expands a node, it has found the minimum cost path to that node

If A^* is consistent, then the first discovered path is the optimal one.

2. Constraint Satisfaction Problems (CSP)

Want to find a set of values for the variables so that the variables satisfy the constraints.

scope: scope of $C(v_1, v_5)$ is $\{v_1, v_5\}$

Backtracking Search

1. Get an unassigned variable
2. Assign it a value in the domain
3. Check if it violates any constraint, repeat (1) or (2)

Constraint Propagation

Forward Checking

1. If constraint C only has X unassigned, check for each value v in $Dom[X]$ if v violates contraint C, remove v from $Dom[X]$ if True.

2. If $Dom[X] = \{\}$, return Domain Wiped Out
3. Else, return ok

Intuition: If assigning a value to Y leaves no legal assignment for X where X shares some constraint with Y, then we do not assign this value to Y.

If we need to backtrack, we should restore all pruned values.

Minimum Remaining Value Heuristics (MRV)

Always branch on the most constrained variable, or the variable with the fewest legal left values in its domain.

Degree Heuristic

Select the variable that is involved in the largest number of constraints with other unassigned variables.

Generalized Arc Consistency (GAC)

- A constraint $C(v_1, v_2, \ldots, v_n)$ is GAC w.r.t v_i if and only if $\forall x \in Dom[v_i]$, there exist a domain value for $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n$ that satisfy the constraint.

- $C(v_1, v_2, \ldots, v_n)$ is GAC if and only if it is GAC w.r.t every variable in the scope.
- A CSP is GAC if and only if all constraints are GAC

Algorithm

1. Get a unassigned variable X
2. For each v in $Dom[X]$, assign a value to X and put all constraints C with X in its scope on a queue
3. For each C on the queue, check if assigning X this value will DMO any variable Y sharing constraint with X
4. If so, then restore all pruned values and check the next value in $Dom[X]$

Time Complexity: $\mathcal{O}(cd^3)$ where c is the number of binary constraints, and d is the max size of variable domain.

Can improve efficiency of GAC by keeping track of support, or the assignments to other variables that satisfies the constraints for a given value to X