
 This repository Search


Pull requestsIssuesGist

 lucoby / CS6601notes

Unwatch 1Star 0Fork 0

CodeIssues 0Pull requests 0WikiPulseGraphsSettings

Branch: master CS6601notes / 2 - Search.mdFind fileCopy path

 lucoby searchd800213 26 days ago

1 contributor

151 lines (98 sloc) 8.33 KBRawBlameHistory

# Lesson 2 - Search

## Introduction to Search

Regular programming is telling the computer what to do when you know what to do. AI is for telling the computer what to do when you don't know what to do. And search is one of the key areas for planning a sequence of steps when we have no idea what the *first* step should be until we *solve* the search problem.

The complexity of the problem comes from the fact that there are many states. There are many choices to start with and complexity continuing to pick the correct choice. This contrasts with other complex scenarios where all the possible actions or the outcomes of our possible actions aren't immediately obvious.

## Defining the Problem

- Initial state
- Actions (s) -> {a<sub>1</sub>, a<sub>2</sub>, a<sub>3</sub>, ...} Sometimes actions are state dependent, other times actions are state dependent
- Result (s, a) -> s'
- GoalTest(s) -> T | F
- PathCost(s<sub>1</sub> -(a)-> s<sub>2</sub> -(a)-> s<sub>3</sub>) -> n
  - often times it is the sum of StepCost(s, a, s') -> n

## Tree Search

```
function tree_search(problem):
    frontier = {[initial]}
    loop:
        if frontier is empty:
            return fail
        path = remove_choice(frontier)
        s = path.end
        if s is a goal:
            return path
        for a in actions:
            add [path + a -> result(s, a)] to frontier
```

Tree search describes a whole family of algorithms. The flexibility comes in the remove choice function for deciding how to choose the next choice off the frontier.

## Breadth first search

Breadth first selects the shortest path on the frontier to explore next.

## Graph Search

---

Since Breadth first search as a tree search super imposes a search tree on top of the graph that represents the search space. There can be instances where you have a search path that backtracks nodes to a state that you previously searched. In order to deal with that `tree_search` can be modified into `graph_search`

```
function graph_search(problem):
    frontier = {[initial]}
    explored = {}
    loop:
        if frontier is empty:
            return fail
        path = remove_choice(frontier)
        s = path.end
        add to explored
        if s is a goal:
            return path
        for a in actions:
            add [path + a -> result(s, a)] to frontier unless result(s, a) in {frontier, explored}
```

## Breadth First Search

Breadth first search in a graph search algorithm solves the problem of backtracking in a tree search algorithm. One caveat of breadth first search is that general tree search checks and returns the goal state when pulling a path off the frontier rather than when adding it to the frontier. If the goal is simply to find a shortest path in terms of whole steps (and there are only whole steps) breadth first search can be optimized to check for a goal state on adding a result to the frontier and returning a frontier at that point. In that case breadth first is still guaranteed to find the shortest path (in terms of number of steps). But it may not find the shortest path in terms of path cost.

## Uniform Cost Search (Cheapest-First Search)

Rather than popping the shortest path of the frontier you pop the path with the shortest path cost off the frontier.

Based on the generic graph search, the uniform cost search doesn't return a path until it's popped it off the frontier rather than when it's added to the frontier. This ensures that the cheapest path is returned rather the first path that is found.

## Search Comparison

---

Breadth first search is optimal for finding the shortest path.

Cheapest first search is optimal for finding the cheapest cost path

Depth search isn't optimal for finding the shortest path or cheapest path. However, consider the memory requirements for implementing a breadth first or cheapest first search. Breadth first search's frontier is  $2^n$  at a depth of  $n$ . Cheapest first search has an even larger frontier since it can be exploring nodes at varying depth based on which path is cheapest. Depth search only needs a frontier of size  $n$  when searching to depth  $n$ . This memory savings is negated however if the explored set is saved as well.

Another issue is completeness. Both breadth first and cheapest first can find the Goal state if it isn't in the first "depth-branch" of the search in an infinite search space. However, depth first search cannot. It will continue searching down the first branch.

Note: there can be scenarios such as 1) zero cost actions and 2) nodes with infinite successors whose cost form an infinite

series that converge to a cost that less than the goal cost

## More on Uniform Cost Search

Uniform cost search starts search in all directions and isn't directed towards the goal. Depending on where the goal is you can expect to explore a larger space than necessary. However, without more information there isn't much that can be done.

On the other hand if you "know" the straight line distance to the goal you can use this information in a **Greedy Best-First Search**. This is optimal if there are no obstacles in the way. However, if there are obstacles greedy won't necessarily find the optimal path as that could involve back tracking and exploring a path that takes you further away from the goal (temporarily) before leading a much shorter overall path to the goal.

## A\* search

---

$$f = g + h$$

$g(\text{path})$  = path cost

$h(\text{path}) = h(s)$  = estimated distance to the goal

Sometimes it is referred to as the best estimated total path cost first but A\* is traditional

## Optimistic Heuristic

---

A's ability to find the optimal path depends on the  $h$  function. A will only find the best path iff:

- $h(s) < \text{true cost}$ .
- $h$  never underestimates
- $h$  is optimistic
- $h$  is admissible

An intuitive way to think about is that once you've popped the goal state off the frontier you know the actual cost to the frontier and it is less than the actual cost to any of the other frontier nodes + an optimal estimate for the cost to the goal from the frontier. Furthermore it is required that none of the path costs are negative so that an unexplored path may not lead to a lower overall cost path.

## State Spaces

---

Search algorithms are also applicable to searching through state spaces. This is important because even simple systems with a few sub systems that can exist in multiple states can quickly multiply to create a very large state space. Therefore it's good to have algorithms that can efficiently search the state space for optimal paths to take

## Generating optimistic heuristics

---

While search algorithms provide a good way of searching the goal of artificial intelligence is to solve problems with little input knowledge and providing a "good" heuristic function can seem like providing a lot of the intelligence to what ought to be an artificially intelligent agent.

However, with search it is possible to develop heuristics simply given a description of the problem general method. Suppose for a sliding blocks (15) Puzzle. A formal description of the problem is:

A block can move  $A \rightarrow B$  if (A adjacent to B) and (B is blank)

2 optimistic heuristics can be generated from this by simply relaxing the constraints of the problems:

- H1: A block can move  $A \rightarrow B$  if (A adjacent to B) and  $\sim(B \text{ is blank})$

- The minimum number of moves ignoring blank occupied tiles
- H2: A block can move A  $\rightarrow$  B if  $\sim(A \text{ adjacent to B})$  and  $(B \text{ is blank})\sim$ 
  - The number of tiles that are misplaced

## Constraints of search

---

Search works well when:

- The domain is fully observable
- The domain is known: all available actions known
- Discrete: There are a finite number of actions
- Deterministic: We need to know the result of the actions
- Static: There must be nothing else in the world that can change world except for us

## Implementing a Path

---

A path of A  $\rightarrow$  S  $\rightarrow$  F can be represented as a list of nodes. The nodes must contain:

- The state
- The action needed to get from the previous state to the current state
- The cost to get to this state
- A pointer to the parent node

2 data structures deal with nodes the frontier and the explored set.

- Frontier: The two common operations are removing the best item and adding new items which suggest implementing it as a priority queue. But, another common operation is a membership test which suggest representing it as a set built from a hash table or tree. Often good implementations of search utilize both structure simultaneously.
- Explored: All explored needs to do is add new members and check membership which can be done with simply a tree or a hash table

