6.01: Introduction to EECS I

Optimal Search Algorithms

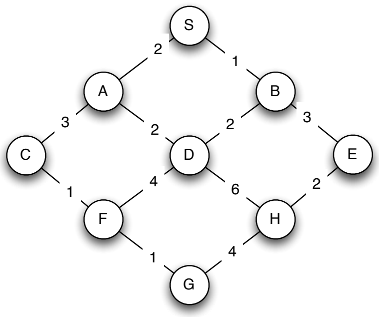
Week 13 *December 1, 2009*

The story so far

- Search domain — characterized by successors function, legal actions, start state, goal function.
- Search tree — an explicit representation for the search space.
- Depth-first search — explore search tree by expanding deepest node.
- Breadth-first search — explore search tree by expanding shallowest node.
- Dynamic programming — do not revisit nodes.

Cost

In many applications, actions have different costs, for example, distance between cities can vary.



Our algorithms thus far ignore this.

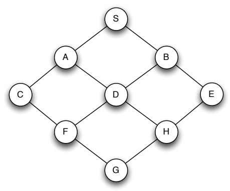
Cost

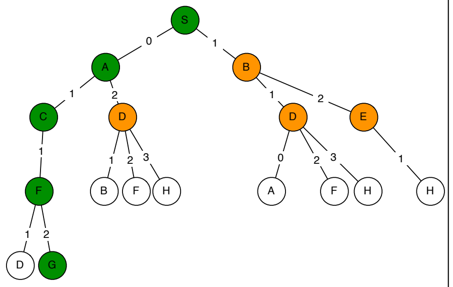
```
map1dist = {'S' : [('A', 2), ('B', 1)],
            'A' : [('S', 2), ('C', 3), ('D', 2)],
            'B' : [('S', 1), ('D', 2), ('E', 3)],
            'C' : [('A', 3), ('F', 1)],
            'D' : [('A', 2), ('B', 2), ('F', 4), ('H', 6)],
            'E' : [('B', 3), ('H', 2)],
            'F' : [('C', 1), ('D', 4), ('G', 1)],
            'H' : [('D', 6), ('E', 2), ('G', 4)],
            'G' : [('F', 1), ('H', 4)]}
```

Path cost is the sum of the action costs along a path.

Breadth-First Search

Enumerates all length 1 paths, then length 2 paths, then length 3 paths, etc.





Uniform-Cost Search

Enumerate paths in order of their total path cost.

Like breadth-first search, but:

- The agenda is a priority queue (returns least cost entry).
- Instead of testing for a goal state when we put an element *into* the agenda, we test for a goal state when we take an element *out of* the agenda.

Guaranteed to find a shortest path.

Priority Queue

A priority queue is a data structure with the same basic operations as stacks and queues, with two differences:

- Items are pushed into a priority queue with a numeric score, called a *cost*.
- When it is time to pop an item, the item in the priority queue with the least *cost* is returned and removed from the priority queue.

Priority Queue

Simple implementation using lists

```
class PQ:
    def __init__(self):
        self.data = []
    def push(self, item, cost):
        self.data.append((cost, item))
    def pop(self):
        (index, cost) = util.argmaxIndex(self.data, lambda (c, x): -c)
        return self.data.pop(index)[1] # just return the data item
    def isEmpty(self):
        return self.data is []
```

The pop operation in this implementation can take time proportional to the number of nodes (in the worst case).

Better algorithms (using trees) reduce run time to be proportional to the log of the number of nodes (in the worst case).

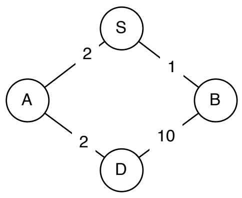
Search Node

```
class SearchNode:
    def __init__(self, action, state, parent, actionCost):
        self.state = state
        self.action = action
        self.parent = parent
        if self.parent:
            self.cost = self.parent.cost + actionCost
        else:
            self.cost = actionCost
```

ucSearch

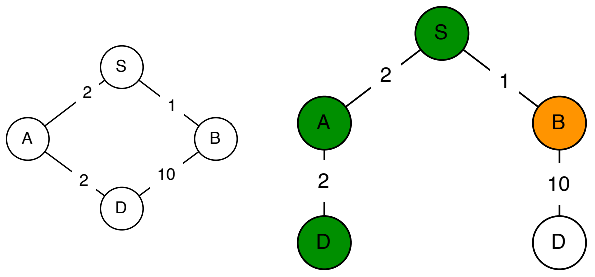
```
def ucSearch(initialState, goalTest, actions, successor):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    while not agenda.isEmpty():
        n = agenda.pop()
        if goalTest(n.state):
            return n.path()
        for a in actions:
            (newS, cost) = successor(n.state, a)
            if not n.inPath(newS):
                newN = SearchNode(a, newS, n, cost)
                agenda.push(newN, newN.cost)
    return None
```

Example



ucSearch: From S to D

Numbers on links are distances not action indices

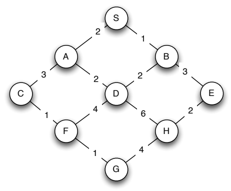


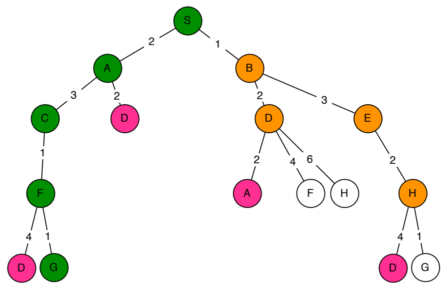
ucSearch with DP

```
def ucSearch(initialState, goalTest, actions, successor):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    expanded = { }
    while not agenda.isEmpty():
        n = agenda.pop()
        if not expanded.has_key(n.state):
            expanded[n.state] = True
            if goalTest(n.state):
                return n.path()
            for a in actions:
                (newS, cost) = successor(n.state, a)
                if not expanded.has_key(newS):
                    newN = SearchNode(a, newS, n, cost)
                    agenda.push(newN, newN.cost)
    return None
```

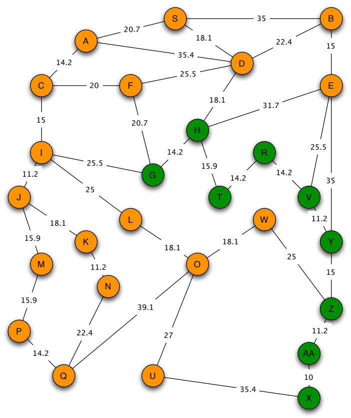
ucSearch with DP: From S to G

Numbers on links are distances not action indices





Search in Big Spaces



Search with heuristics

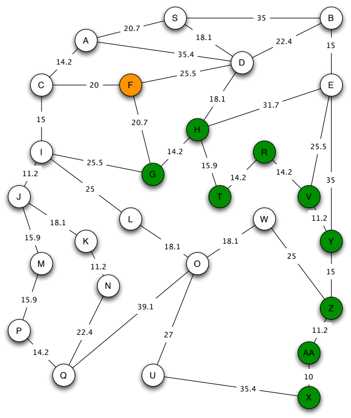
A heuristic function takes a state as an argument and returns a numeric estimate of the total cost that it will take to reach the goal from there.

Used to focus the search in relevant direction.

Actual cost + heuristic is a better estimate of total cost.

For map-like problems, Euclidean distance from node to goal is good heuristic.

Search in Big Spaces



A* = ucSearch with heuristics

```
def ucSearch(initialState, goalTest, actions, successor, heuristic):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    expanded = { }
    while not agenda.isEmpty():
        n = agenda.pop()
        if not expanded.has_key(n.state):
            expanded[n.state] = True
            if goalTest(n.state):
                return n.path()
            for a in actions:
                (newS, cost) = successor(n.state, a)
                if not expanded.has_key(newS):
                    newN = SearchNode(a, newS, n, cost)
                    agenda.push(newN, newN.cost + heuristic(newS))
    return None
```

Good and Bad Heuristics

We want heuristic close to actual distances but cheap to compute.

- The perfect heuristic – solve the problem and use the answer (too expensive).
- Trivial heuristic – 0 for all nodes (cheap but useless).
- Admissible – always an underestimate of the actual distance.

A* is guaranteed to find shortest path with admissible heuristic

Check Yourself

Would the so-called 'Manhattan distance', which is the sum of the absolute differences of the x and y coordinates be an admissible heuristic in the city navigation problem, in general?

Check Yourself

If we were trying to minimize travel time on a road network (and so the estimated time to travel each road segment was the cost), what would be an appropriate heuristic function?

Heuristic search examples

The 8 puzzle

Number test, again