Optimizing a Search

Find minimum distance path between 2 points on a rectangular grid.

Represent all possible paths with a tree (shown to just length 3).

Find the shortest path from A to I.

Order Matters

Replace last node in agenda by its children:

Remove first node from agenda. Add its children to end of agenda.

Nano-Quiz Makeup

Wednesday, May 4, 6-11pm, 34-501.
- everyone can makeup/retake NQ 1
- everyone can makeup/retake two additional NQs
- you can makeup/retake other NQs excused by S^3

If you makeup/retake a NQ, the new score will replace the old score, even if the new score is lower!
**Order Matters**

Replace last node by its children (depth-first search):
- implement with **stack** (last-in, first-out).

Remove first node from agenda. Add its children to the end of the agenda (breadth-first search):
- implement with **queue** (first-in, first-out).

**Action Costs**

Some actions can be more costly than others.

Compare navigating from A to I on two grids.

Modify search algorithms to account for action costs
→ **Uniform Cost Search**

**Breadth-First with Dynamic Programming**

Notice that we expand nodes in order of increasing path length.

This algorithm **fails** if path costs are **not equal**.

Visit: A B D C E G F H I

Agenda: **A B C D E G F H I**

---

**Breadth-First with Dynamic Programming**

First consider actions with equal costs.

Visit: A B D C E G F H I

Agenda: **A B C E G F H I**

---

**Today**

Generalize search framework → **uniform cost search**.

Improve search efficiency → **heuristics**.

Visit: A B D C E G F H I

Agenda: **A B C E G F H I**
**Uniform Cost Search**

Associate action costs with actions. Enumerate paths in order of their total path cost. Find the path with the smallest path cost = sum of action costs along the path. → implement agenda with priority queue.

**Priority Queue**

Same basic operations as stacks and queues, with two differences:
- items are pushed with numeric score: the cost.
- popping returns the item with the smallest cost.

```python
pq = PQ()
pq.push('a', 3)
pq.push('b', 6)
pq.push('c', 1)
print(pq.pop())  # 'c'
pq.push('c', 1)
pq.pop()  # 'a'
```

```python
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

    def path(self):
        if self.parent is None:
            return [(self.action, self.state)]
        else:
            return self.parent.path() + [(self.action, self.state)]

    def inPath(self, s):
        if s == self.state:
            return True
        elif self.parent is None:
            return False
        else:
            return self.parent.inPath(s)

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.empty():
        parent = agenda.pop()
        if goalTest(parent.state):
            return parent.path()
        for a in actions:
            (newS, cost) = successor(parent.state, a)
            if not parent.inPath(newS):
                newNode = SearchNode(a, newS, parent, cost)
                agenda.push(newNode, newNode.cost)
    return None
```

**Priority Queue**

Push with cost, pop smallest cost first.

```
>>> pq = PQ()
>>> pq.push('a', 3)
>>> pq.push('b', 6)
>>> pq.push('c', 1)
>>> pq.pop()
'c'
>>> pq.pop()
'a'
```

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

[There are better algorithms!]

```python
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]

    def empty(self):
        return len(self.data) == 0
```

**Uniform Cost Search**

`goalTest` was previously performed when children pushed on agenda. Here, we must defer `goalTest` until all children are pushed (since a later child might have a smaller cost).

The `goalTest` is implemented during subsequent pop.
Dynamic Programming Principle

The shortest path from $X$ to $Z$ that goes through $Y$ is made up of
- the shortest path from $X$ to $Y$ and
- the shortest path from $Y$ to $Z$.

We only need to remember the shortest path from the start state to each other state!

Want to remember shortest path to $Y$. Therefore, defer remembering $Y$ until all of its siblings are considered (similar to issue with goalTest) — i.e., remember expansions instead of visits.

ucSearch with Dynamic Programming

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.empty():
        parent = agenda.pop()
        if not expanded.has_key(parent.state):
            expanded[parent.state] = True
            if goalTest(parent.state):
                return parent.path()
        for a in actions:
            (newS, cost) = successor(parent.state, a)
            if not expanded.has_key(newS):
                newN = SearchNode(a, newS, parent, cost)
                agenda.push(newN, newN.cost)
    return None

Conclusion

Searching spaces with unequal action costs is similar to searching spaces with equal action costs.
Just substitute priority queue for queue.

Stumbling upon the Goal

Our searches so far have radiated outward from the starting point.
We only notice the goal when we stumble upon it.
Example: Start at $E$, go to $I$.

Heuristics

Our searches so far have radiated outward from the starting point.
We only notice the goal when we stumble upon it.
This results because our costs are computed for just the first part of the path: from start to state under consideration.
We can add heuristics to make the search process consider not just the starting point but also the goal.

Heuristic: estimate the cost of the path from the state under consideration to the goal.
Heuristics

Add Manhattan distance to complete the path to the goal.

Developed a new class of search algorithms: uniform cost.

Allows solution of problems with different action costs.

Developed a new class of optimizations: heuristics.

Focuses search toward the goal.


- everyone can makeup/retake NQ 1
- everyone can makeup/retake two additional NQs
- you can makeup/retake other NQs excused by S^-3

If you makeup/retake a NQ, the new score will replace the old score, even if the new score is lower.