6.01: Introduction to EECS I

Optimizing a Search

May 3, 2011

Nano-Quiz Makeups

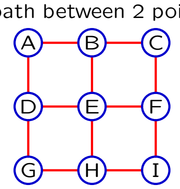
Wednesday, May 4, 6-11pm, 34-501.

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

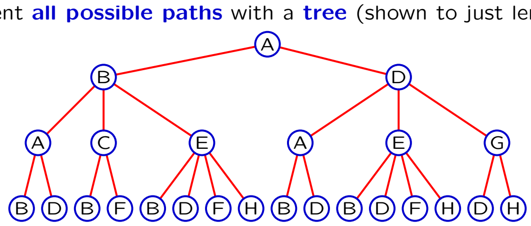
If you makeup/retake a NQ, the new score will replace the old score, even if the new score is lower!

Search Example

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



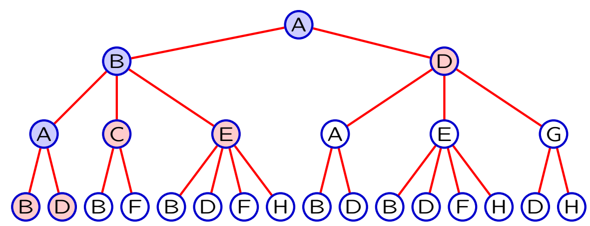
Represent **all possible paths** with a **tree** (shown to just length 3).



Find the shortest **path** from A to I.

Order Matters

Replace first node in agenda by its children:

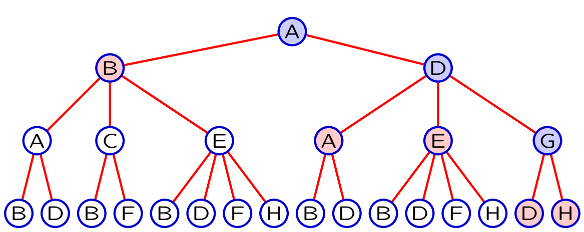


step Agenda
0: A
1: AB AD
2: ABA ABC ABE AD
3: ABAB ABAD ABC ABE AD

Depth First Search

Order Matters

Replace last node in agenda by its children:

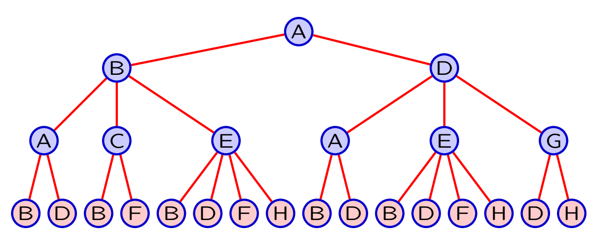


step Agenda
0: A
1: AB AD
2: AB ADA ADE ADG
3: AB ADA ADE ADGD ADGH

also Depth First Search

Order Matters

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



step Agenda
8: ADG ABAB ABAD ABCB ABCF ABEB ABED
ABEF ABEH ADAB ADAD ADEB ADED ADEF ADEH
9: ABAB ABAD ABCB ABCF ABEB ABED ABEF ABEH
ADAB ADAD ADEB ADED ADEF ADEH ADGD ADGH

Breadth First Search

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).

Today

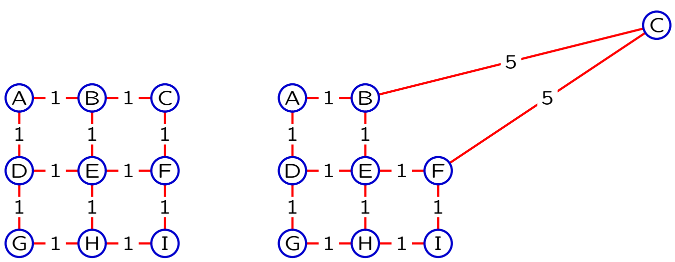
Generalize search framework → uniform cost search.

Improve search efficiency → heuristics.

Action Costs

Some actions can be more costly than others.

Compare navigating from A to I on two grids.

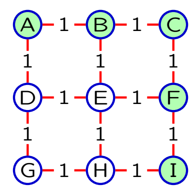


Modify search algorithms to account for action costs

→ **Uniform Cost Search**

Breadth-First with Dynamic Programming

First consider actions with equal costs.

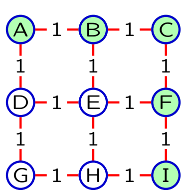


Visited A B D C E G F H I

Agenda: ~~A~~ ~~AB~~ ~~AD~~ ~~ABC~~ ~~ABE~~ ~~ADG~~ ~~ABCF~~ ABEH ABCFI

Breadth-First with Dynamic Programming

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

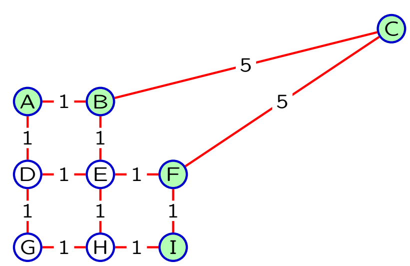


Visited A B D C E G F H I

Agenda:	~~A~~	~~AB~~	~~AD~~	~~ABC~~	~~ABE~~	~~ADG~~	~~ABCF~~	ABEH	ABCFI
	0	1	1	2	2	2	3	3	4

Breadth-First with Dynamic Programming

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



Visited A B D C E G F H I

Agenda:	~~A~~	~~AB~~	~~AD~~	~~ABC~~	~~ABE~~	~~ADG~~	~~ABCF~~	ABEH	ABCFI
	0	1	1	6	2	2	11	3	12

Nodes are **not** expanded in order of increasing path length.

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.

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'
```

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

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

[There are better algorithms!]

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
    def path(self):
        if self.parent == 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 == None:
            return False
        else:
            return self.parent.inPath(s)
```

Uniform Cost Search

```
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):
                newN = SearchNode(a, newS, parent, cost)
                agenda.push(newN, newN.cost)
    return None
```

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.

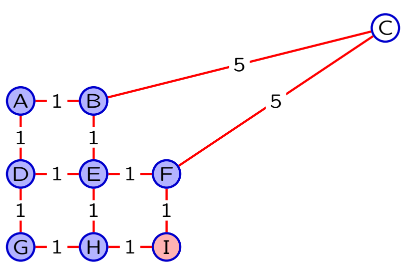
We only need to remember the *shortest* path from the start state to each other state!

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
```

ucSearch with Dynamic Programming



Expanded: A B D E G F H

Agenda:

~~A~~	~~AB~~	~~AD~~	ABC	~~ABE~~	~~ADE~~	~~ADG~~	~~ABEF~~	~~ABEH~~	~~ADGH~~
0	1	1	6	2	2	2	3	3	3

ABEFC	ABEFI	ABEHI
8	4	4

Conclusion

Searching spaces with unequal action costs is similar to searching spaces with equal action costs.

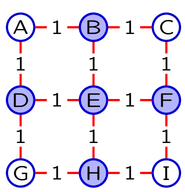
Just substitute priority queue for queue.

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.



Expanded: E B D F H A C G

Agenda:

~~E~~	~~EB~~	~~ED~~	~~EF~~	~~EH~~	~~EBA~~	~~EBC~~	~~EDA~~	~~EDG~~	~~EFC~~	EFI	EHG	EHI
0	1	1	1	1	2	2	2	2	2	2	2	2

Too much time searching paths on **wrong side of starting point!**

Heuristics

Our searches so far have radiated outward from the starting point. We only notice the goal when we stumble upon it.

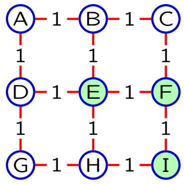
This results because our **costs** are computed for just the first part of the path: from start to state under consideration.

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.



Expanded: E F H

Agenda:	~~E~~	EB	ED	~~EF~~	~~EH~~	EFC	EFI	EHG	EHI
	2	4	4	2	2	4	2	4	2

A* = ucSearch with Heuristics

A **heuristic function** takes input **s** and returns the estimated cost from state **s** to the goal.

```
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.empty():
        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
```

Admissible Heuristics

An **admissible heuristic** always underestimates the actual distance.

If the heuristic is larger than the actual cost from **s** to goal, then the "best" solution may be missed → **not acceptable!**

If the heuristic is smaller than the actual cost, the search space will be larger than necessary → not desireable, but right answer.

The ideal heuristic should be

– as close as possible to actual cost (without exceeding it)

– easy to calculate

A* is guaranteed to find shortest path if heuristic is admissible.

Check Yourself

Consider three heuristic functions for the "eight puzzle":

a. 0

b. number of tiles out of place

c. sum over tiles of Manhattan distances to their goals

Let M_i = # of moves in the best solution using heuristic i

Let E_i = # of states expanded during search with heuristic i

Which of the following statements is/are true?

1. $M_a = M_b = M_c$
2. $E_a = E_b = E_c$
3. $M_a > M_b > M_c$
4. $E_a \geq E_b \geq E_c$
5. the same "best solution" will result for all three heuristics

Summary

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.

Nano-Quiz Makeups: 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!

MIT OpenCourseWare
http://ocw.mit.edu

6.01SC Introduction to Electrical Engineering and Computer Science
Spring 2011

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.