Python Programming

```
from collections import OrderedDict

d = OrderedDict()
d['foo'] = 1
d['bar'] = 2
d['spam'] = 3
d['grok'] = 4

# Outputs "foo 1", "bar 2", "spam 3", "grok 4"
for key in d:
    print(key, d[key])
```

```
>>> import json
>>> json.dumps(d)
'{"foo": 1, "bar": 2, "spam": 3, "grok": 4}'
>>>
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Python Programming

```python
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

```python
min_price = min(zip(prices.values(), prices.keys()))
# min_price is (10.75, 'FB')

max_price = max(zip(prices.values(), prices.keys()))
# max_price is (612.78, 'AAPL')

prices_sorted = sorted(zip(prices.values(), prices.keys()))
# prices_sorted is [(10.75, 'FB'), (37.2, 'HPQ'),
#                   (45.23, 'ACME'), (205.55, 'IBM'),
#                   (612.78, 'AAPL')]
```

```python
prices_and_names = zip(prices.values(), prices.keys())
print(min(prices_and_names))   # OK
print(max(prices_and_names))   # ValueError: max() arg is an empty sequence
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Python Programming

```python
prices = {
    'ACME': 45.23,
    'AAPL': 612.78,
    'IBM': 205.55,
    'HPQ': 37.20,
    'FB': 10.75
}
```

```python
min(prices)   # Returns 'AAPL'
max(prices)   # Returns 'IBM'

min(prices.values())  # Returns 10.75
max(prices.values())  # Returns 612.78

min(prices, key=lambda k: prices[k])  # Returns 'FB'
max(prices, key=lambda k: prices[k])  # Returns 'AAPL'

min_value = prices[min(prices, key=lambda k: prices[k])]
```

```
>>> prices = { 'AAA' : 45.23, 'ZZZ': 45.23 }
>>> min(zip(prices.values(), prices.keys()))
(45.23, 'AAA')
>>> max(zip(prices.values(), prices.keys()))
(45.23, 'ZZZ')
>>>
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Python Programming

```
a = {
    'x' : 1,
    'y' : 2,
    'z' : 3
}
```

```
b = {
    'w' : 10,
    'x' : 11,
    'y' : 2
}
```

```
# Find keys in common
a.keys() & b.keys()   # { 'x', 'y' }

# Find keys in a that are not in b
a.keys() - b.keys()   # { 'z' }

# Find (key,value) pairs in common
a.items() & b.items() # { ('y', 2) }


# Make a new dictionary with certain keys removed
c = {key:a[key] for key in a.keys() - {'z', 'w'}}
# c is {'x': 1, 'y': 2}
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Python Programming

```python
def dedupe(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

```python
def dedupe(items, key=None):
    seen = set()
    for item in items:
        val = item if key is None else key(item)
        if val not in seen:
            yield item
            seen.add(val)
```

```
>>> a = [1, 5, 2, 1, 9, 1, 5, 10]
>>> list(dedupe(a))
[1, 5, 2, 9, 10]
>>>
```

```
>>> a = [ {'x':1, 'y':2}, {'x':1, 'y':3}, {'x':1, 'y':2}, {'x':2, 'y':4}]
>>> list(dedupe(a, key=lambda d: (d['x'],d['y'])))
[{'x': 1, 'y': 2}, {'x': 1, 'y': 3}, {'x': 2, 'y': 4}]
>>> list(dedupe(a, key=lambda d: d['x']))
[{'x': 1, 'y': 2}, {'x': 2, 'y': 4}]
>>>
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Python Programming

```
######    0123456789012345678901234567890123456789012345678901234567890'
record = '....................100          .......513.25     ..........'
cost = int(record[20:32]) * float(record[40:48])



SHARES = slice(20,32)
PRICE  = slice(40,48)

cost = int(record[SHARES]) * float(record[PRICE])
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Python Programming

```
>>> items = [0, 1, 2, 3, 4, 5, 6]
>>> a = slice(2, 4)
>>> items[2:4]
[2, 3]
>>> items[a]
[2, 3]
>>> items[a] = [10,11]
>>> items
[0, 1, 10, 11, 4, 5, 6]
>>> del items[a]
>>> items
[0, 1, 4, 5, 6]
```

```
>>> a = slice(10, 50, 2)
>>> a.start
10
>>> a.stop
50
>>> a.step
2

>>> s = 'HelloWorld'
>>> a.indices(len(s))
(5, 10, 2)
>>> for i in range(*a.indices(len(s))):
...     print(s[i])
...
W
r
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Python Programming

```python
words = [
    'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',
    'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',
    'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',
    'my', 'eyes', "you're", 'under'
]

from collections import Counter
word_counts = Counter(words)
top_three = word_counts.most_common(3)
print(top_three)
# Outputs [('eyes', 8), ('the', 5), ('look', 4)]
```

```
>>> word_counts['not']
1
>>> word_counts['eyes']
8
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Python Programming

```
>>> morewords = ['why','are','you','not','looking','in','my','eyes']
>>> for word in morewords:
...     word_counts[word] += 1
```

```
>>> word_counts['eyes']
9
```

```
>>> word_counts.update(morewords)
```

Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Python Programming

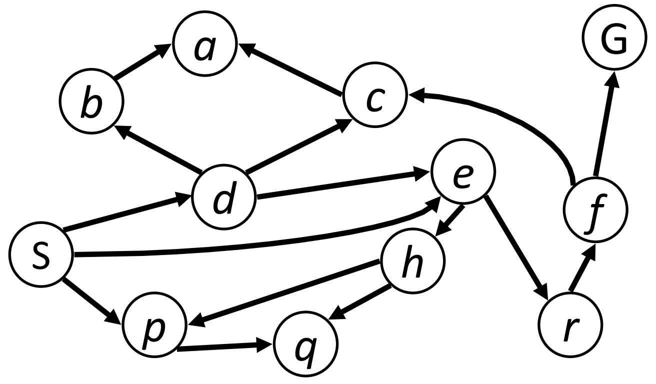
```
>>> a = Counter(words)
>>> b = Counter(morewords)
>>> a
Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2,
         "you're": 1, "don't": 1, 'under': 1, 'not': 1})
>>> b
Counter({'eyes': 1, 'looking': 1, 'are': 1, 'in': 1, 'not': 1, 'you': 1,
         'my': 1, 'why': 1})

>>> # Combine counts
>>> c = a + b
>>> c
Counter({'eyes': 9, 'the': 5, 'look': 4, 'my': 4, 'into': 3, 'not': 2,
         'around': 2, "you're": 1, "don't": 1, 'in': 1, 'why': 1,
         'looking': 1, 'are': 1, 'under': 1, 'you': 1})

>>> # Subtract counts
>>> d = a - b
>>> d
Counter({'eyes': 7, 'the': 5, 'look': 4, 'into': 3, 'my': 2, 'around': 2,
         "you're": 1, "don't": 1, 'under': 1})
```

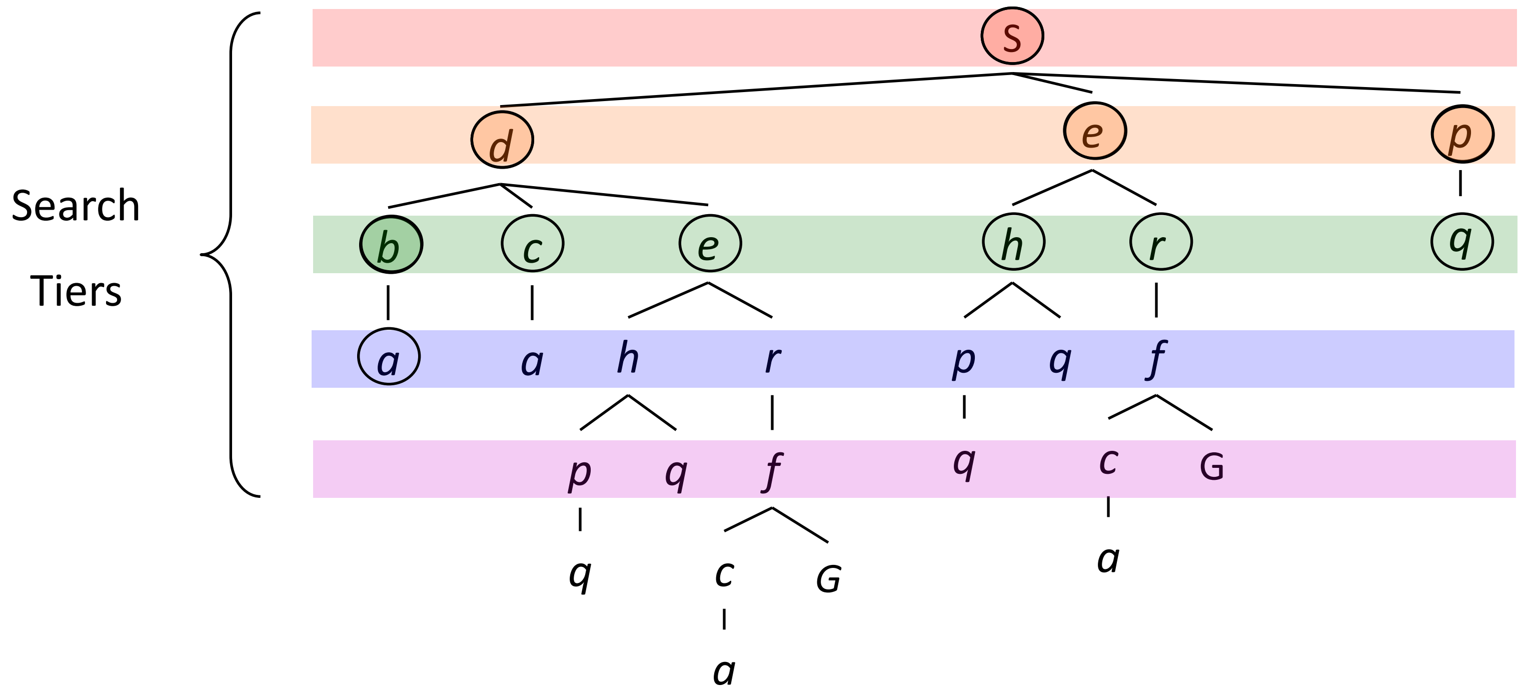
Source: Beazley, David; Jones, Brian K. (2013). *Python Cookbook* (3rd ed.).

Breadth-First Search

Strategy: expand a shallowest node first

Implementation: Fringe is a FIFO queue

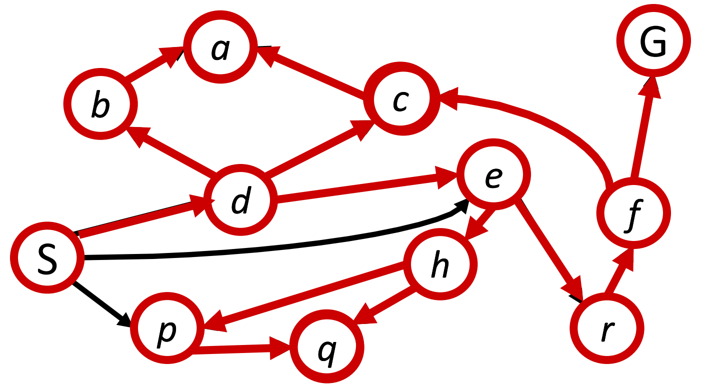


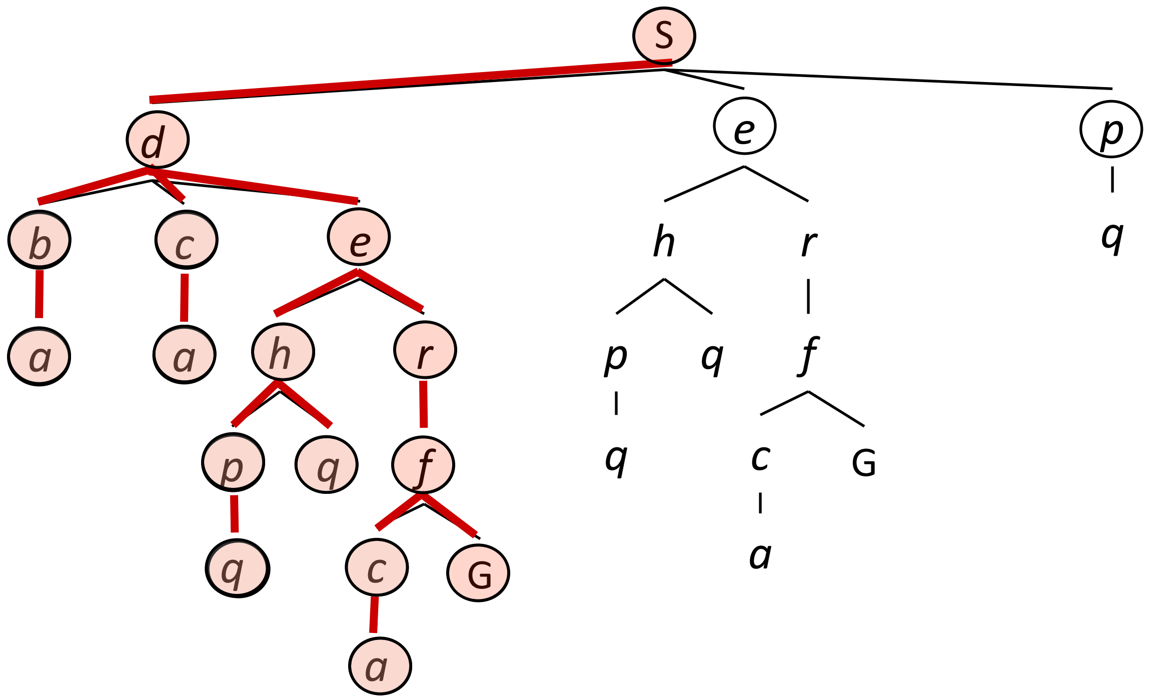


Depth-First Search

Strategy: expand a deepest node first

Implementation: Fringe is a LIFO stack

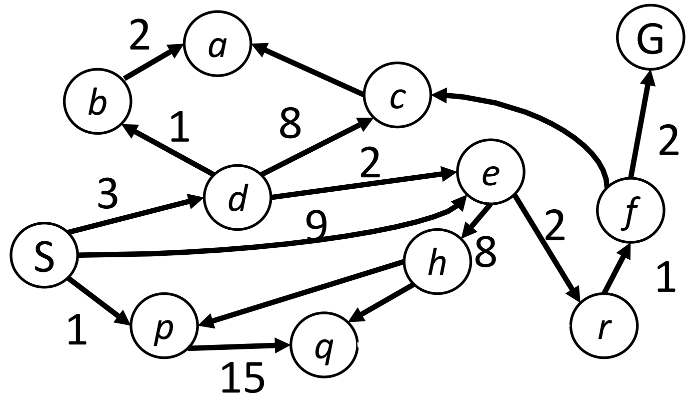


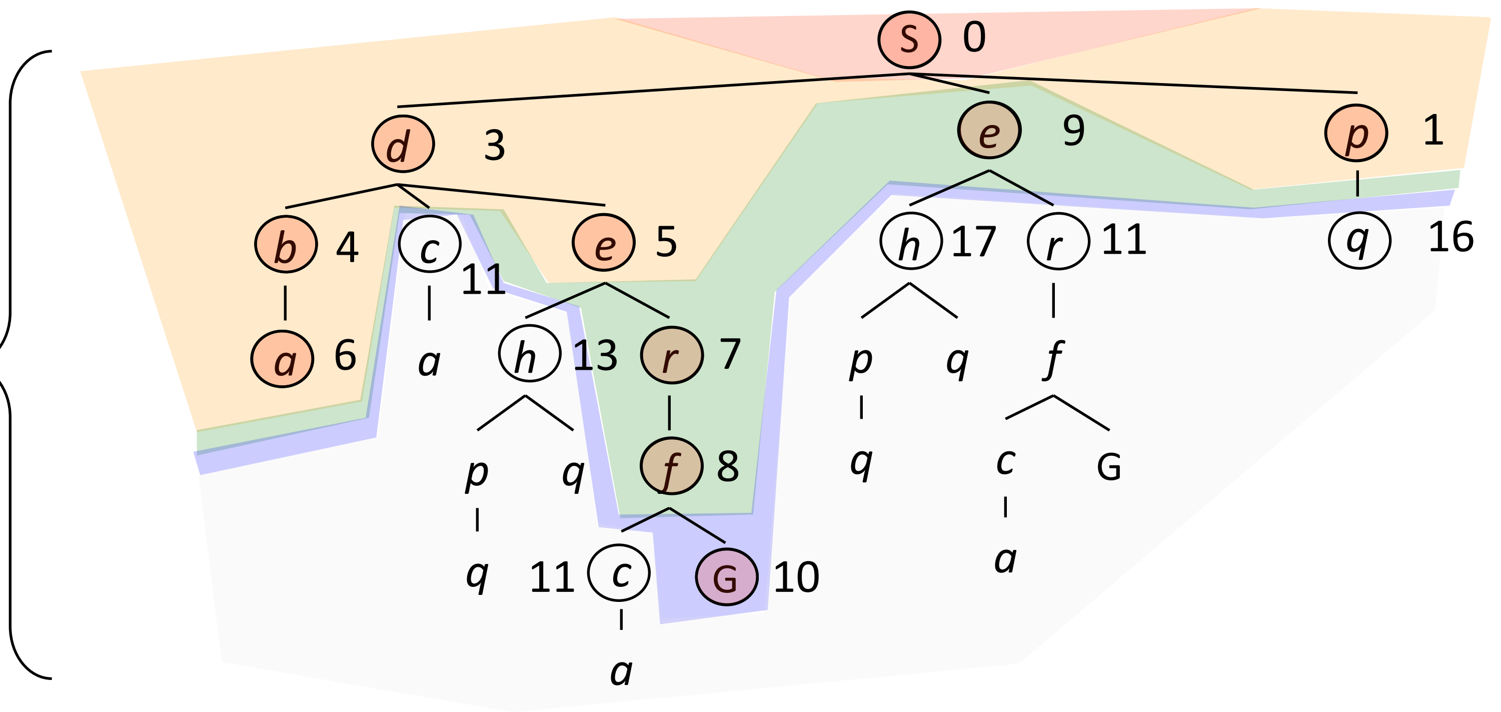


Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)





BFS/DFS/UCS

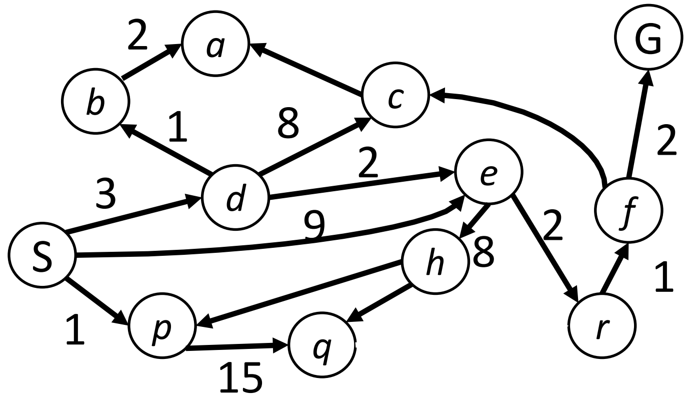
- **Breadth-first search**
  - Good: optimal, works well when many options, but not many actions required
  - Bad: assumes all actions have equal cost
- **Depth-first search**
  - Good: memory-efficient, works well when few options, but lots of actions required
  - Bad: not optimal, can run infinitely, assumes all actions have equal cost
- **Uniform-cost search**
  - Good: optimal, handles variable-cost actions
  - Bad: explores all options, no information about goal location

Basically Dijkstra's Algorithm!

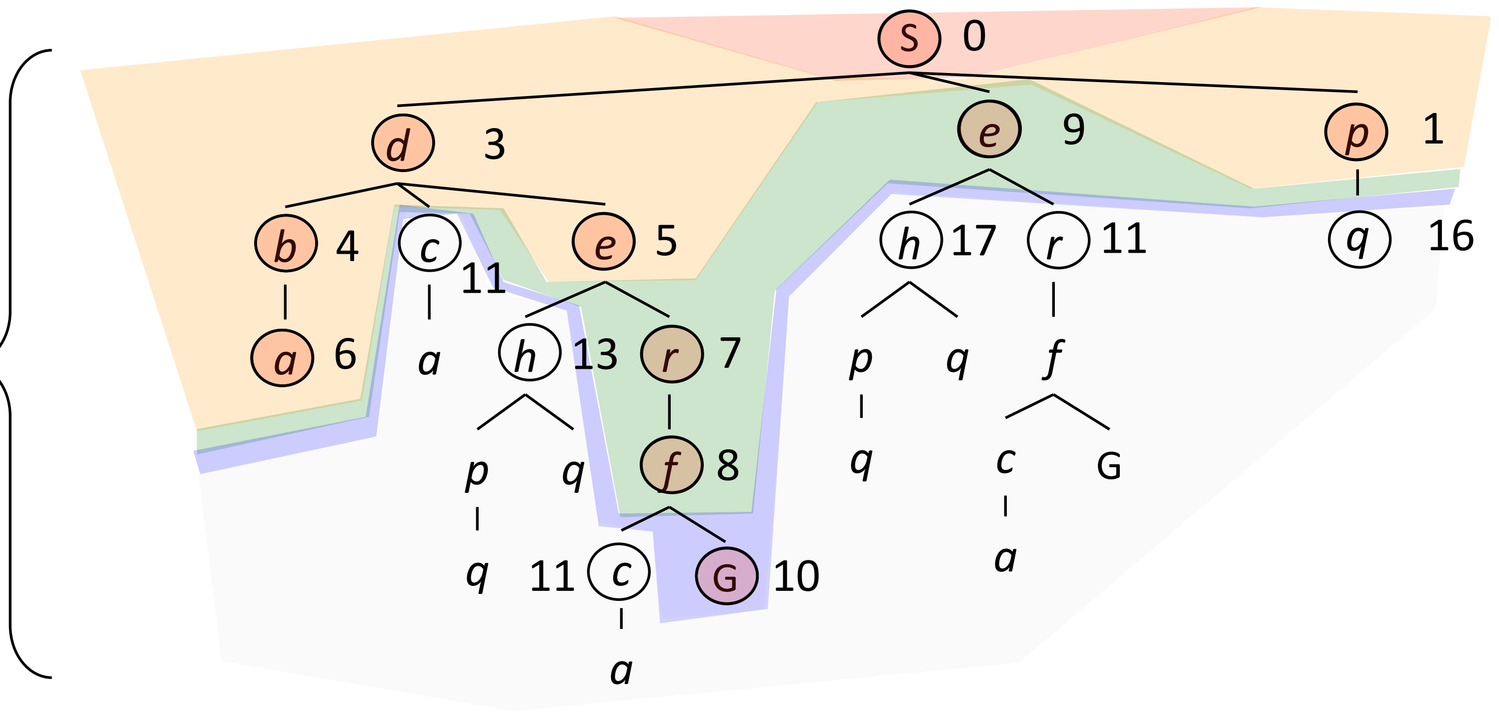
Dijkstra‘s algorithm (Uniform-cost search)

Strategy: expand a cheapest node first:

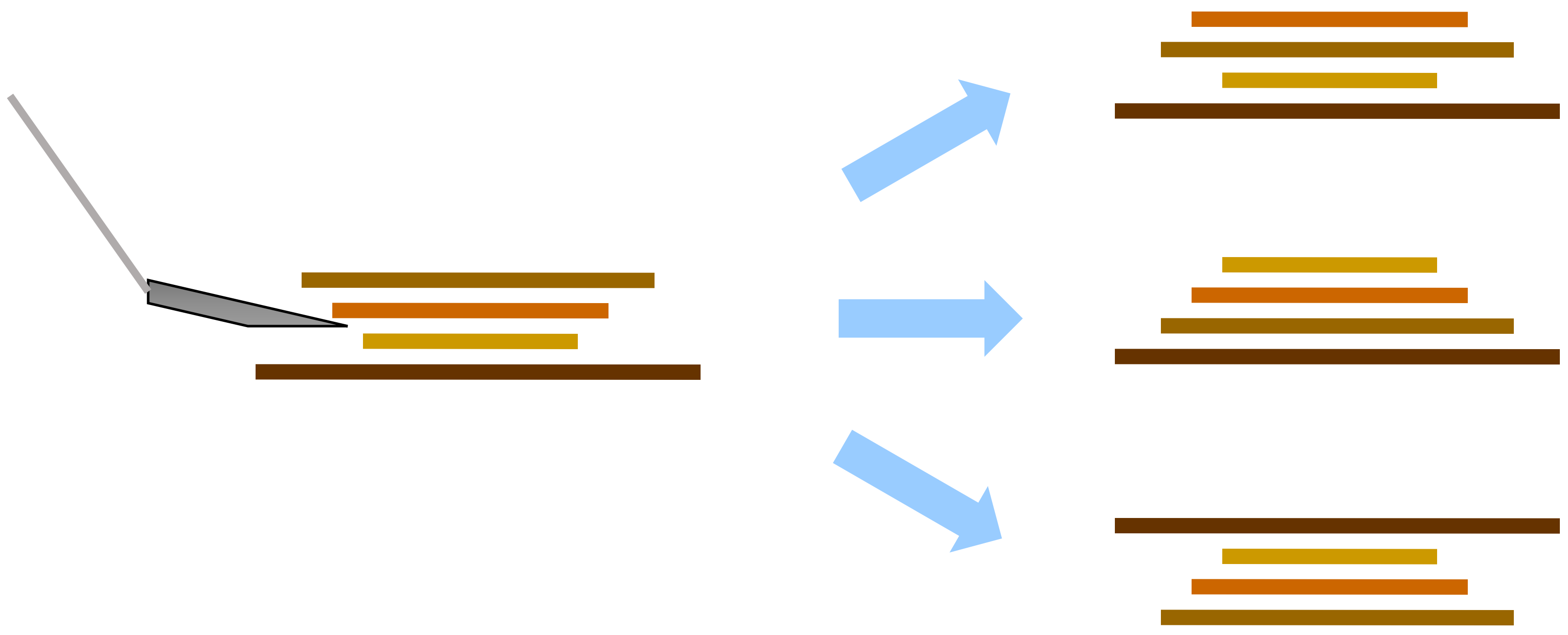
Fringe is a priority queue (priority: cumulative cost)



Cost contours



Search example: Pancake Problem



Rule: a spatula can be inserted at any interval and flip all pancakes above it.

Cost: Number of pancakes flipped.

Pancake BFS

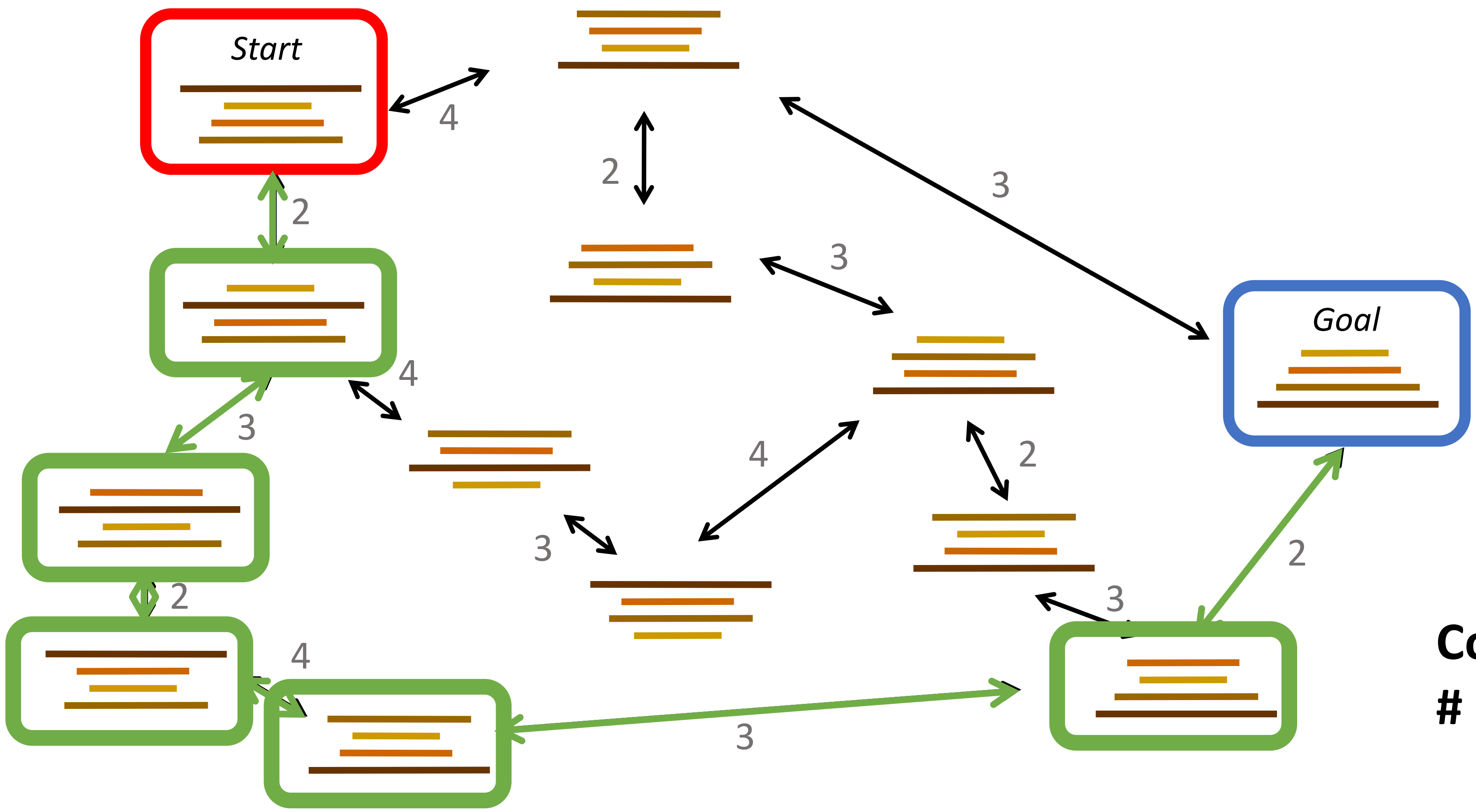
Draw it by yourself!

Pancake UCS

Draw it by yourself!

Pancake DFS

State space graph with costs as weights

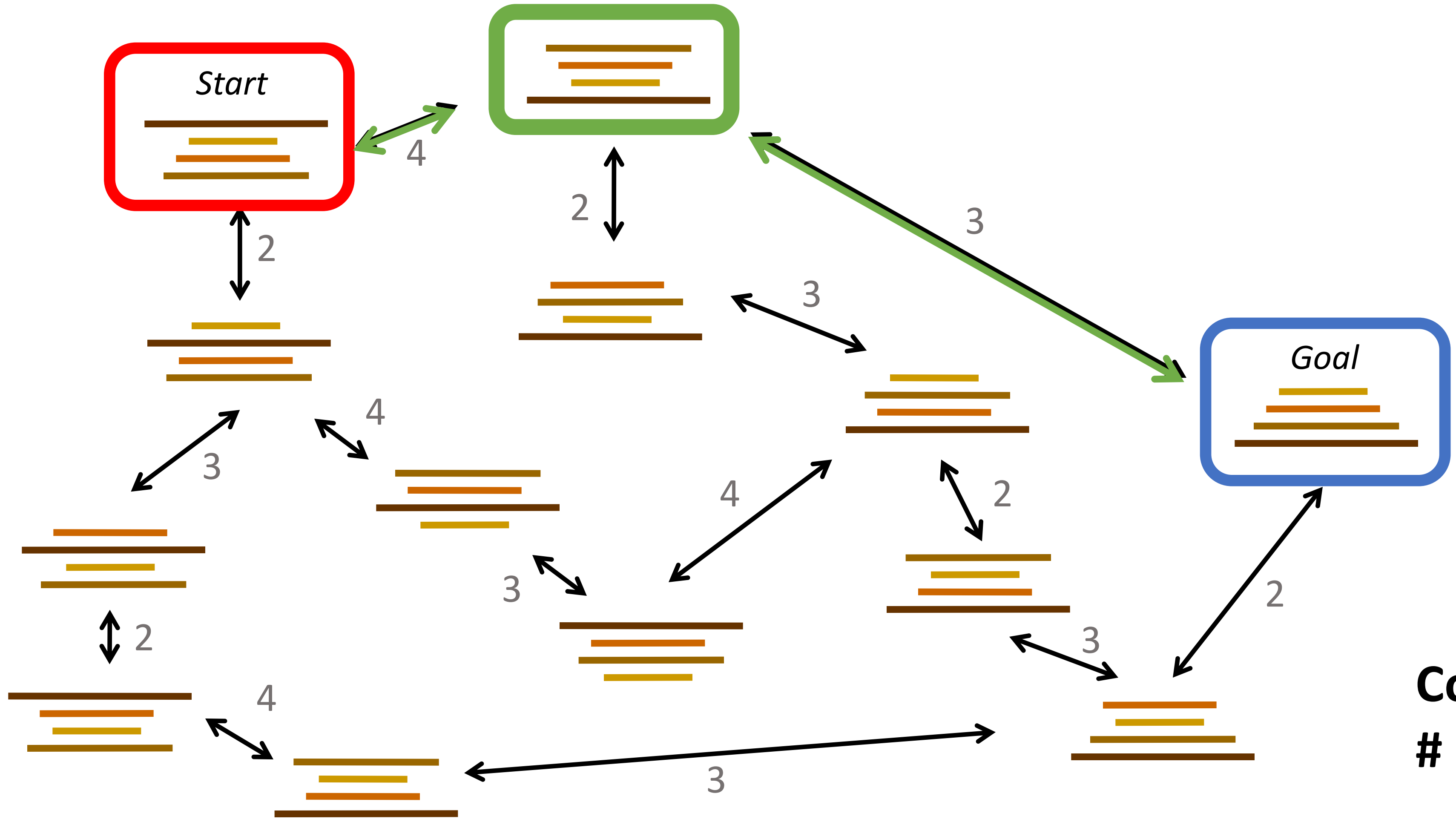


Cost: 16

# Steps: 6

Pancake Optimal

State space graph with costs as weights



Cost: 7
# Steps: 2

Project 1: Search (due 09/13)

Question 1 (3 points): Finding a Fixed Food Dot using Depth First Search

In `searchAgents.py`, you'll find a fully implemented `SearchAgent`, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented -- that's your job.

First, test that the `SearchAgent` is working correctly by running:

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch
```

Question 2 (3 points): Breadth First Search

Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `uninformed_search.py`. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

```
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

Question 3 (4 points): Uniform Cost Search

While BFS will find a fewest-actions path to the goal, we might want to find paths that are "best" in other senses. Consider `mediumDottedMaze`, where food is concentrated in the eastern half of the map, and `mediumScaryMaze`, where that side of the map is full of ghosts.

By changing the cost function, we can encourage Pacman to find different paths through the maze. For example, we can charge more for steps in the eastern half of the map when it's full of dangerous ghosts, and less when it's full of tasty pellets, and a rational Pacman agent should adjust its behavior in response.

Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `uninformed_search.py`.

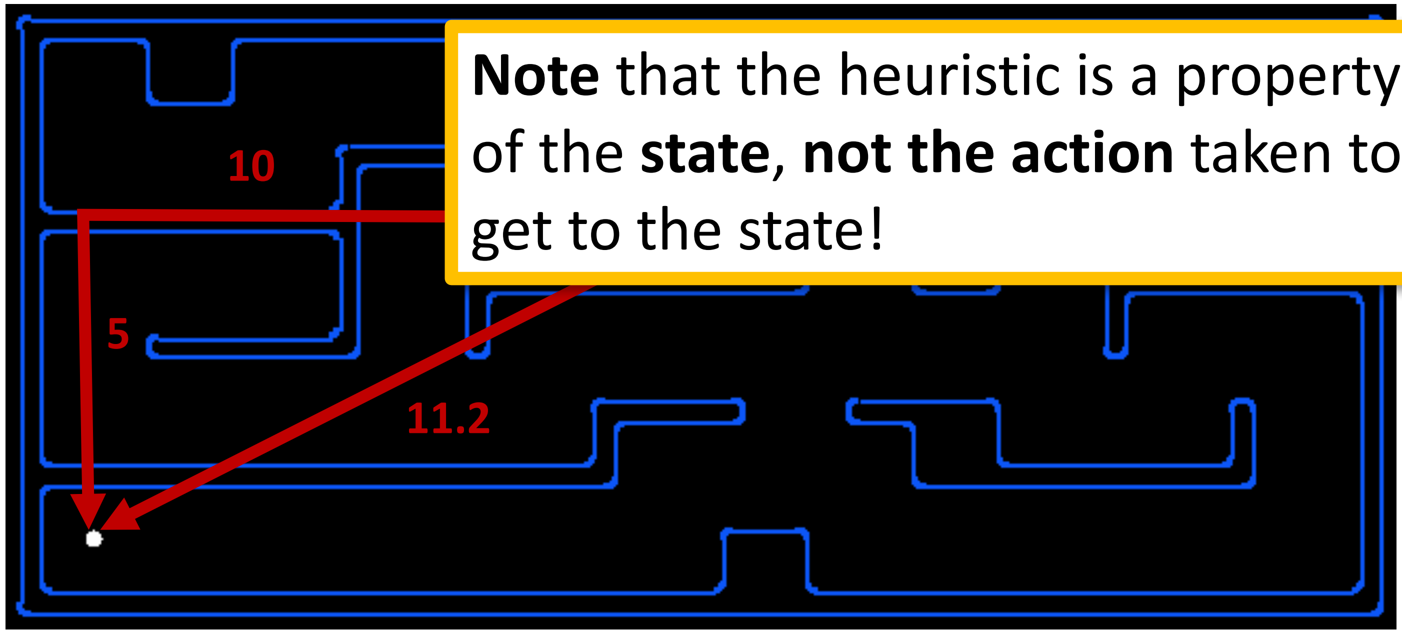
Incorporating goal information

How to efficiently solve search problems with variable-cost actions, using information about the goal state?

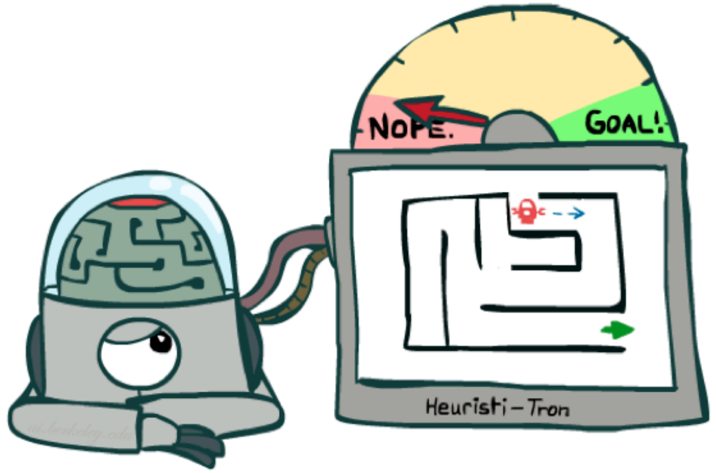
- Heuristics
- Greedy approach
- A* search

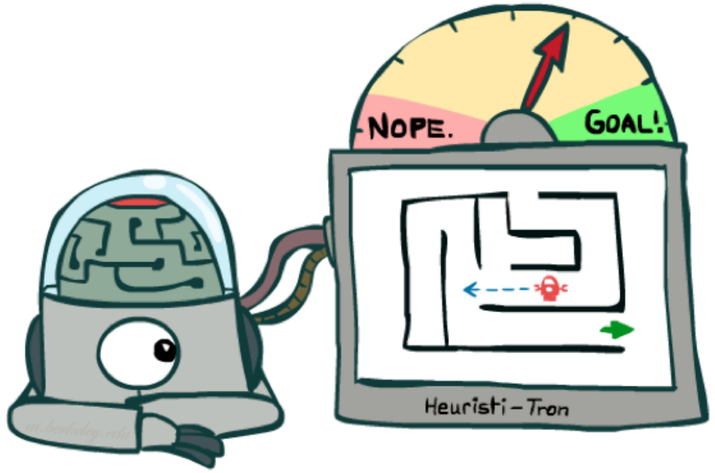
Search Heuristics

- A heuristic is:
  - A function that *estimates* how close a state is to a goal
  - Designed for a particular search problem
  - Examples: Manhattan distance, Euclidean distance for pathing



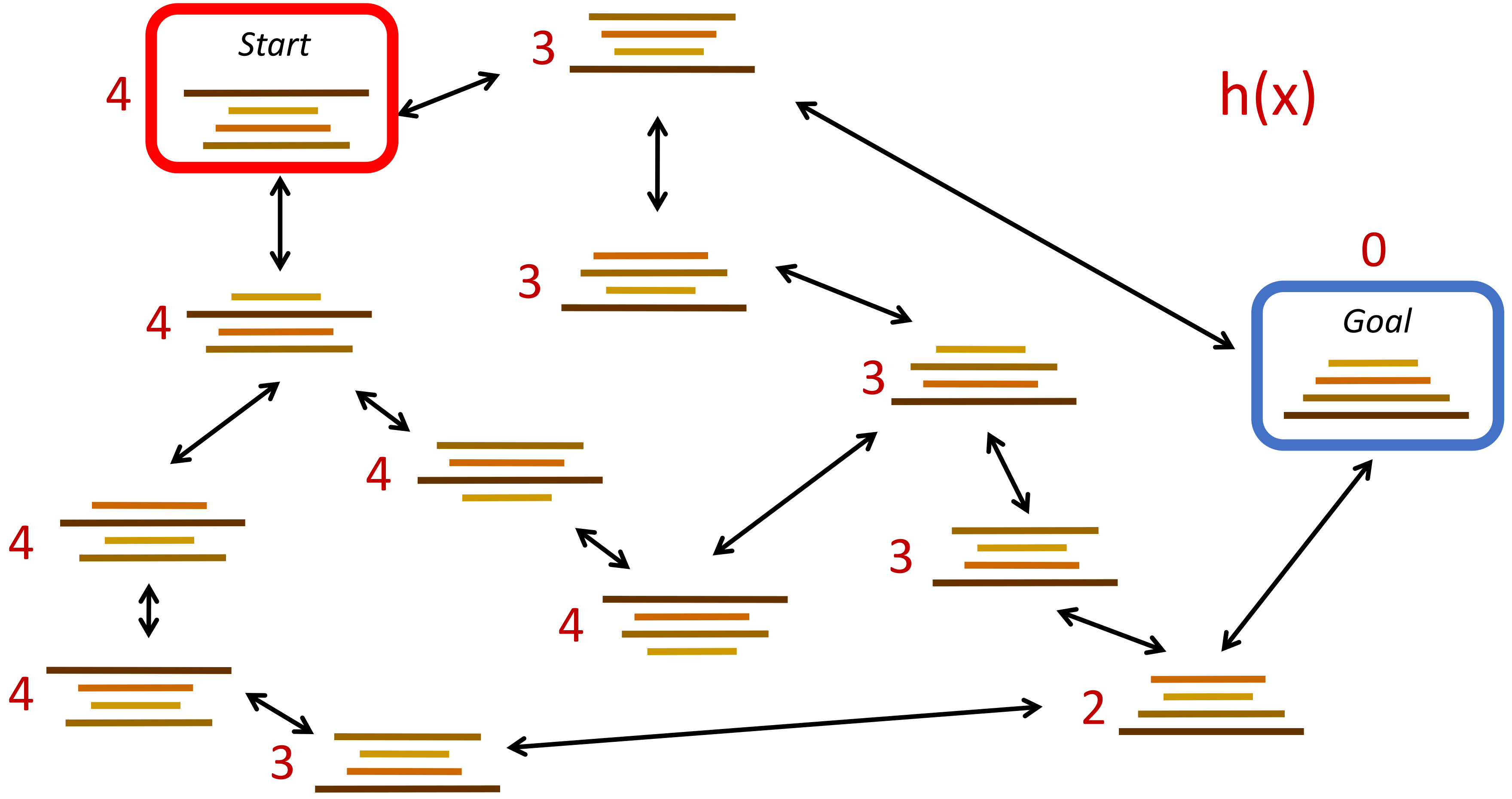
Note that the heuristic is a property of the **state, not the action** taken to get to the state!





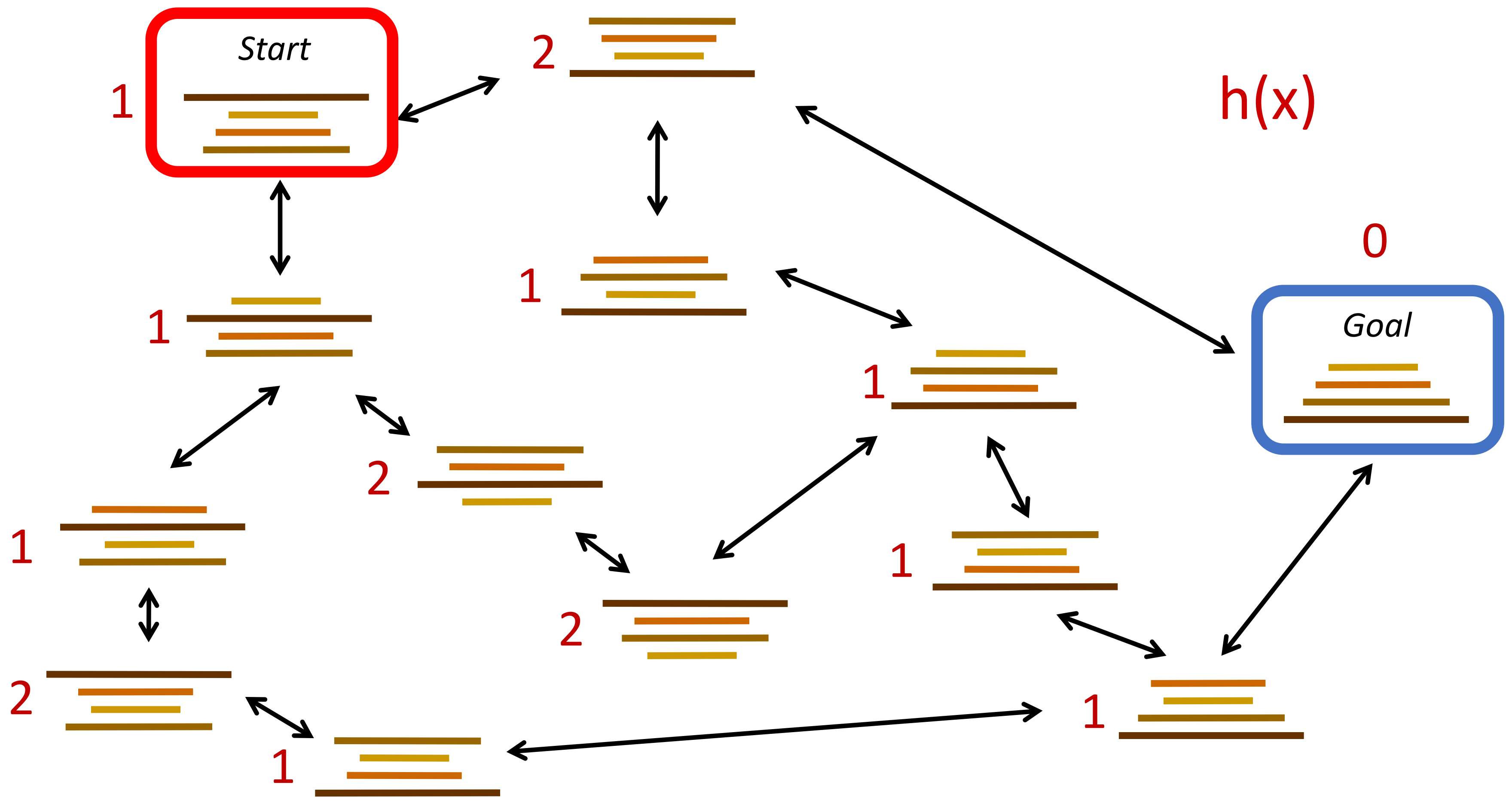
Pancake Heuristics

Heuristic 1: the number of pancakes that are out of place



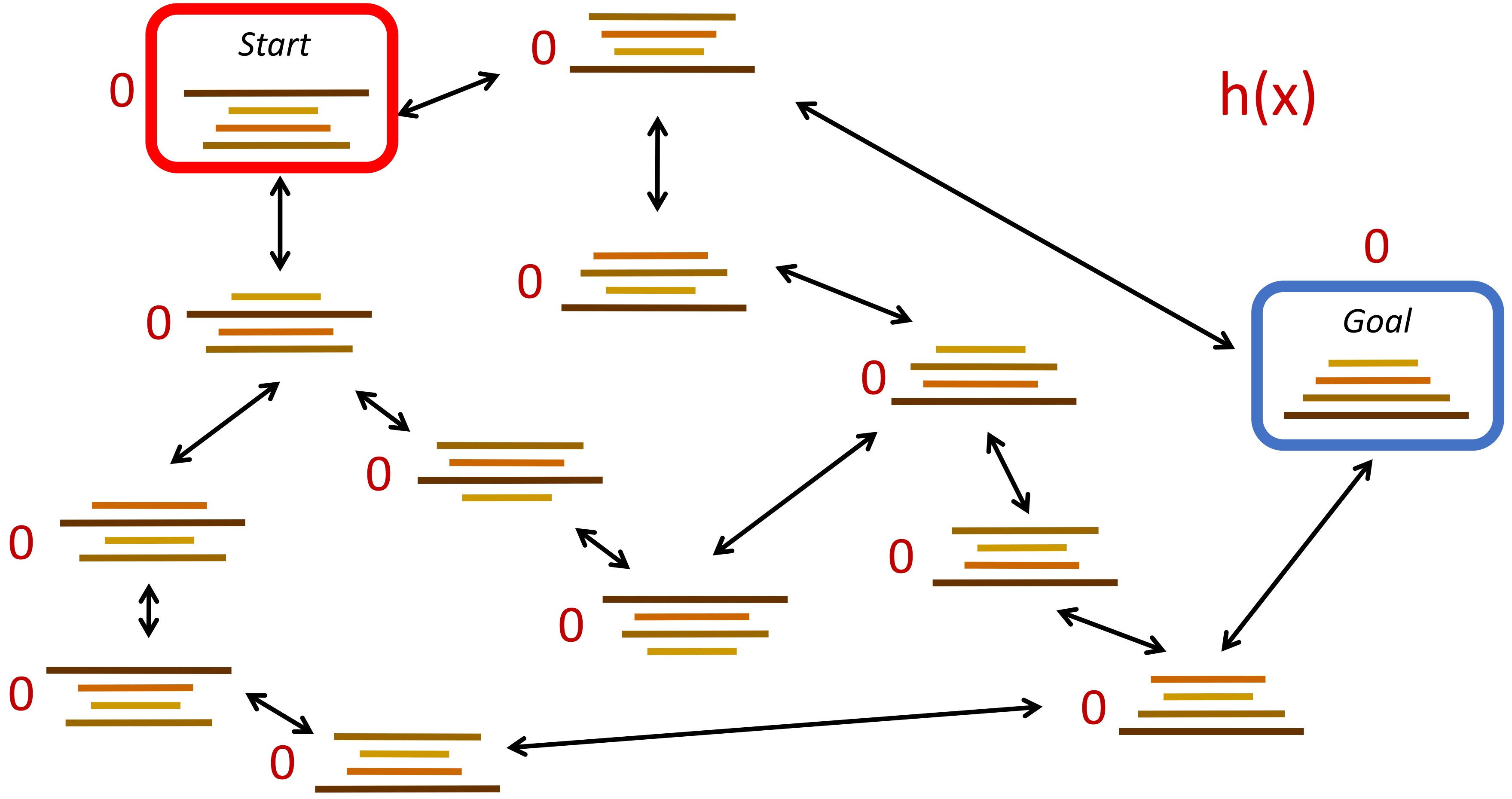
Pancake Heuristics

Heuristic 2: how many pancakes are on top of a smaller pancake?

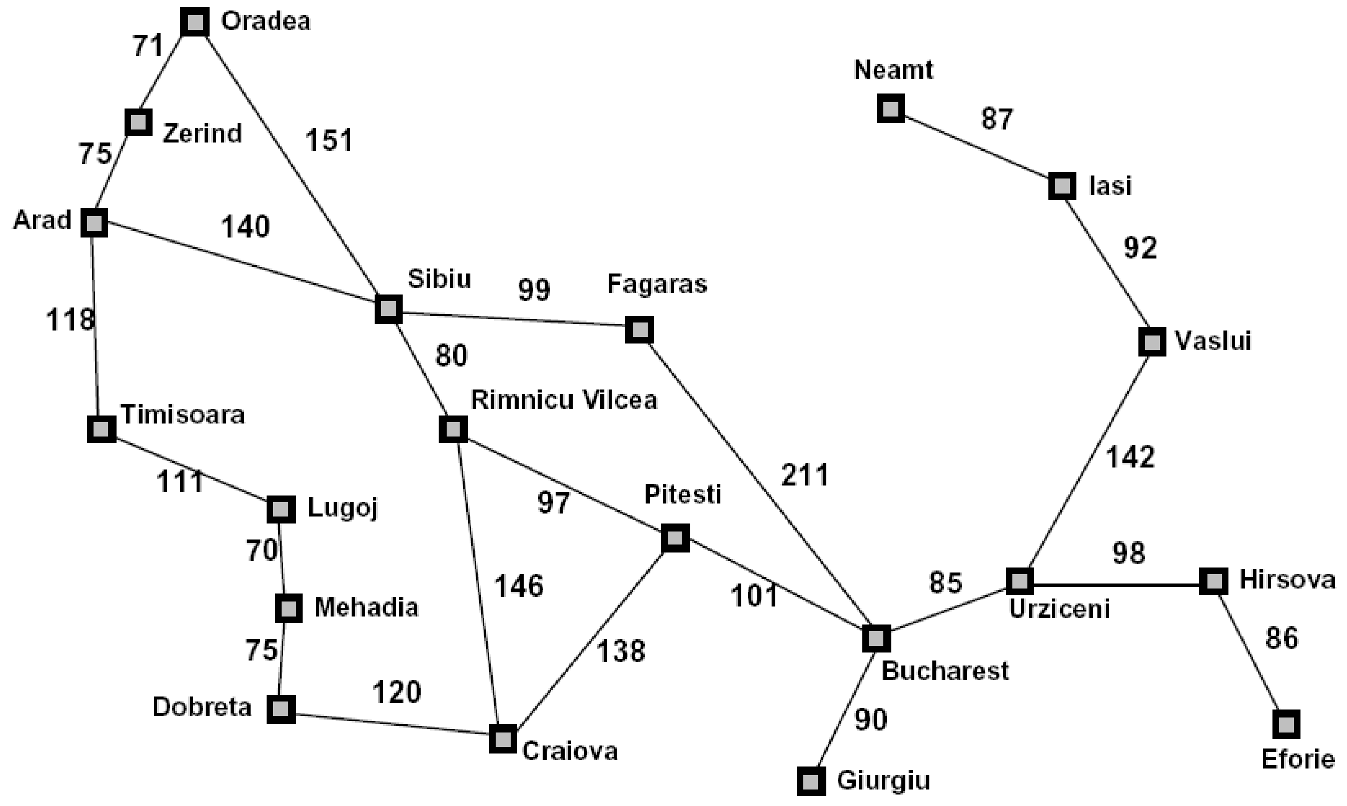


Pancake Heuristics

<u>Heuristic 3</u>: All zeros (aka *null heuristic*, or ”I like waffles better anyway”)



Straight-line Heuristic in Romania



Straight−line distance to Bucharest

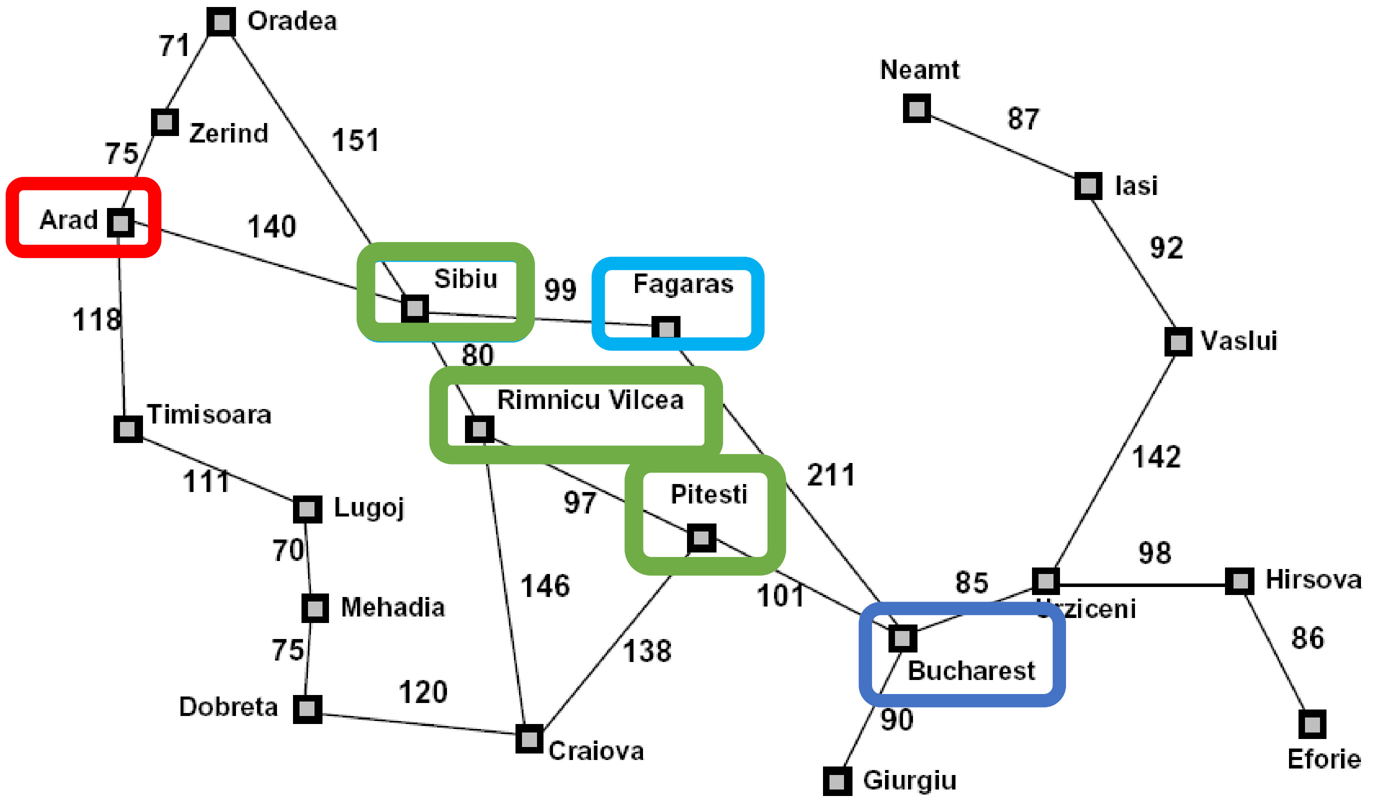
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

Greedy Search

Greedy Straight-Line Search in Romania

- Expand the node that seems closest…



Greedy
- Cost: 450

Optimal
- Cost: 418

Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

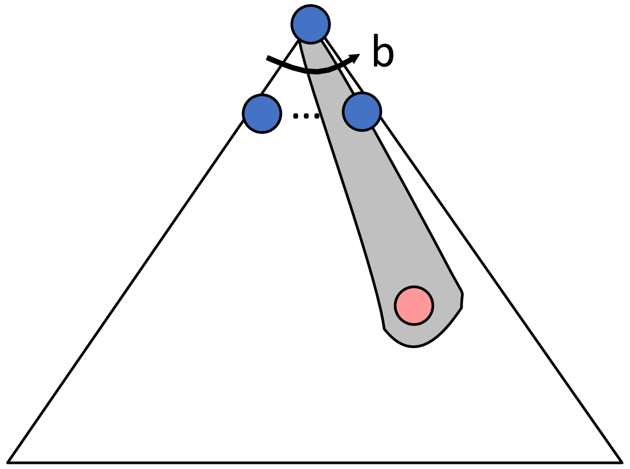
Greedy Search

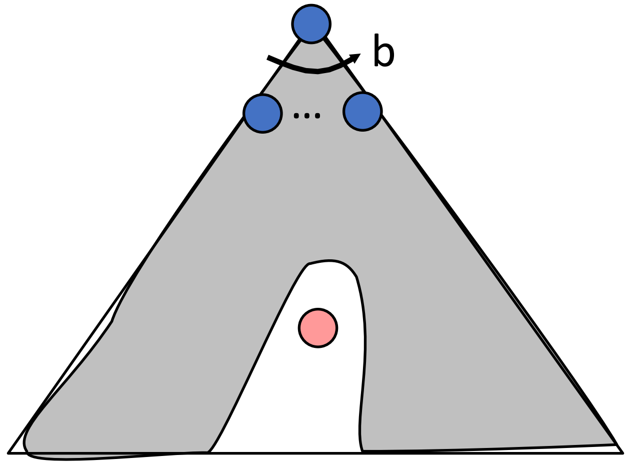
- Strategy: expand a node that you think is closest to a goal state
  - Heuristic: estimate of distance to nearest goal for each state

- A common case:
  - Best-first takes you straight to the (<u>non-optimal</u>) goal

- Worst-case: like a badly-guided DFS

- What goes wrong?
  - Doesn't take <u>real</u> path cost into account





Next class

A* search