Tufts

Lecture 03: Analyzing Search Algorithms

Artificial Intelligence (CS 131)

1

A General Search Technique

Image source: Russell & Norvig (2021)

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

- Our basic search framework depends upon two main things:

1. Choosing nodes: we remove nodes from the frontier, according to some choice function
2. Expanding nodes: some successor function tells us what nodes we can reach from the current one, given available actions

2

Different Search Strategies

- In the basic algorithm, there are two key parts:

1. Choice function**:** order in which nodes are removed from frontier:

$$F \subseteq Nodes, \text{ some subset of our possible search-nodes}$$
$$Choice(F) = node \in F, \text{ some particular node chosen}$$

2. Successor function: what happens when we expand a given node; gives us all the possible next states we can get to, and the action-operators that lead to them:

$$S = \{s \mid s \text{ is a problem state}\}$$
$$A = \{a \mid a \text{ is an action-operator}\}$$
$$Successor : S \to \mathcal{P}(S \times A)$$

- Simple (naïve/uninformed) strategies: use only information from nodes themselves, without any special knowledge about whether or not one such node is likely to be *closer* to the goal or not

3

Uninformed Search I: Breadth-First Search

Image source: Russell & Norvig (2021)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

- A very simple form of search—often one of the first a programmer learns

1. Choosing nodes: we explore nodes in order of depth
2. Expanding nodes: whenever we explore a node, we add its immediate children, reachable in a single step

4

Comparing Breadth-First to Best-First Search

Image source: Russell & Norvig (2021)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Evaluate *before insertion*: Best-First does not check until *expanding* the node (*removing* it from the queue)

- Breadth-first search, although simple, has a very nice property: the *first* time a node is ever reached, we have done so on the shortest possible path to that node
- This allows us to simplify the algorithm with an early goal-test, since we know that the first encounter with the goal is on some best path

5

Comparing Breadth-First to Best-First Search

Image source: Russell & Norvig (2021)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Best-First must check if *better* paths are available

```
if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
  reached[s] ← child
  add child to frontier
```

- The optimality of first path found by Breadth-first also means we can simplify condition for adding new search nodes
- We no longer check if we have found a better path to a node, only whether it is one we have already seen (to avoid loops)

6

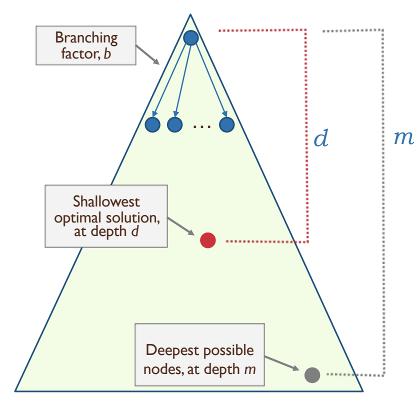
Evaluating Search Strategies

- We can compare search types based upon:
  1. Completeness: Will it always find a solution if it exists?
  2. Cost optimality: Does it always find solution with least cost?
  3. Time complexity: How long does it take to search?
     - Total number of nodes that are generated in searching
  4. Space complexity: How much memory does it need?
     - Maximum number of nodes stored at *any one time*

7

Evaluating Search Strategies

- We can compute time/space complexity in terms of:
  1. Branching factor for search (max), b
  2. Depth of an optimal solution (min), d
  3. Depth of search space (max), $m \leq \infty$



Worst-case time/space to solution for finite domains:

$O(b^d)$

Exponential in solution depth

8

Evaluating Breadth-First Search

- This basic strategy has the following properties
  1. Completeness: Will it always find a solution if it exists?

     Yes, so long as (a) maximum branching-factor b is *finite*, and (b) the problem itself is *finite* and *solvable*.

  2. Optimality: Does it always find solution with least cost?

     Yes: if costs for every action are *identical*.

  3. Time complexity: How long does it take to search?

     Exponential in solution-depth d: $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

  4. Space complexity: How much memory does it take?

     Also exponential, keeping all frontier/reached nodes in memory: $O(b^d)$

9

Optimal Naïve Search

- Breadth-first search is optimal where action costs are uniform (i.e., every action has the same cost for agent)
  - Downside: exponential space and time complexity
- The complexity issues are hard to deal with: some problems simply take a lot of searching to solve naïvely
- Optimality can be extended to problems where some actions have different costs with a simple fix, however: simply order nodes by total path-cost
  - Works so long as all costs are positive

10

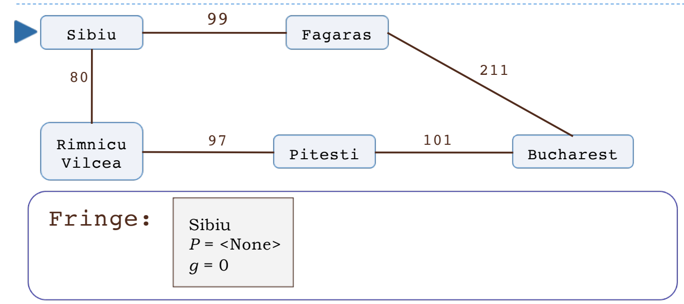
Uniform-Cost Search (Dijkstra's Algorithm)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

1. Best-First routine adds any node n to fringe when a parent is expanded: uses a priority queue for fringe, ordering nodes by total path-cost so far
2. At each step, we expand the first node in the priority queue (the one with least cost so far)
3. If we find a new path to some node on the frontier that is better any found before, we update so that the node now has lower cost and moves closer to front of queue
   - If all costs are positive (greater than 0), then the first time we *expand* a node (*removing* it from queue), we have found the optimal path to it
   - Distinct from Breadth-First, in which we have the optimal path as soon as we expand a node's *parent*, when *adding* nodes to the queue

11

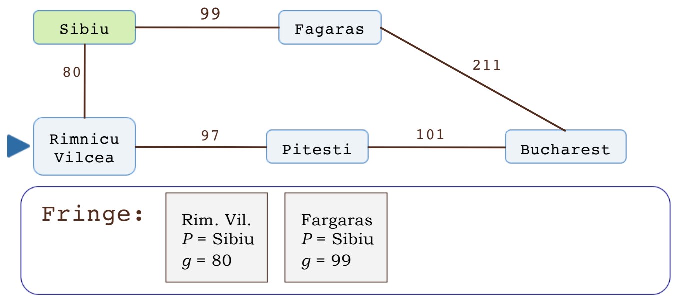
Uniform-Cost Search



- Suppose we are searching for a path from Sibiu to Bucharest
- We start with Sibiu (the initial state) in the fringe priority queue
- It has no parent (P), and the starting path-cost (g) is just 0

12

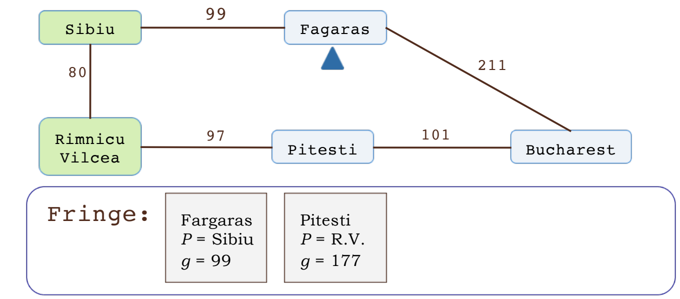
Uniform-Cost Search



- When we expand Sibiu, we add its children to the priority queue
- Each has Sibiu as parent P, and the path-cost g as calculated:
  $$g(n) = g(Parent(n)) + Distance(Parent(n), n)$$
- Since Rimnicu Vilcea has lowest g-value, it is at the front of the queue

13

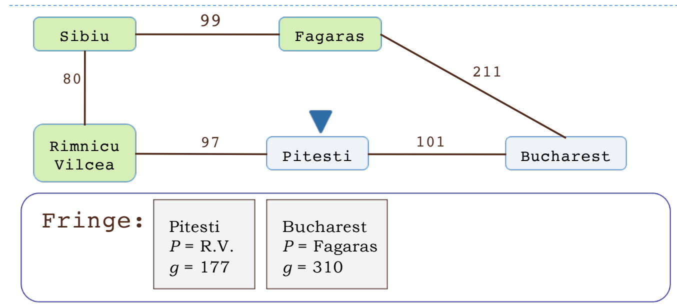
Uniform-Cost Search



- Expanding Rimnicu Vilcea adds Pitesti to the fringe set with path-value:
  $$g(n) = g(Parent(n)) + Distance(Parent(n), n)$$
  $$= 80 + 97 = 177$$
- Now Fagaras has the lowest g-value, and is the next node in the frontier

14

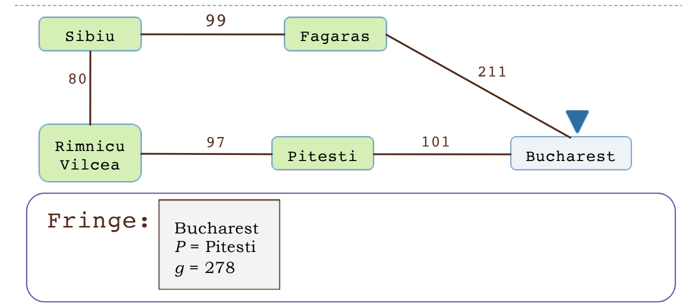
Uniform-Cost Search



- Expanding Fagaras adds Bucharest to the fringe set
- Even though Bucharest is the goal city, we *do not* stop here!
- If we want to guarantee an *optimal* (shortest) path, we must keep looking
- Since Pitesti has lower g-value, it is the next node in the frontier

15

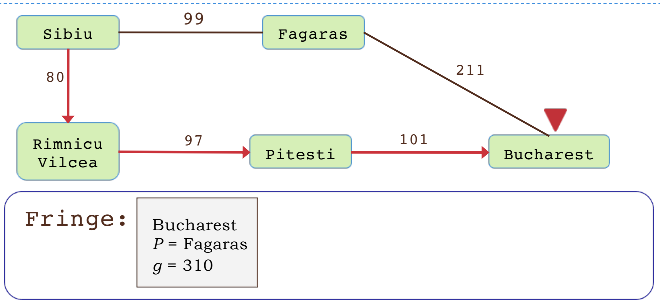
Uniform-Cost Search



- On expanding Pitesti, we see Bucharest ***again***, and calculate a new path-cost
  $$g(n) = g(Parent(n)) + Distance(Parent(n), n)$$
  $$= 177 + 101 = 278$$
- Since the cost we discover is less than the previous path-cost, a new fringe node with different parent and g-value ends up at head of queue

16

Uniform-Cost Search



- Finally, when we expand the goal city of Bucharest itself, we have the optimal path, and can stop searching (even if there is more space left)

17

Evaluating Uniform-Cost Search

- This basic strategy has the following properties
  1. Completeness: Will it always find a solution if it exists?

     Yes, so long as (a) branching-factor b is *finite*, (b) problem itself is *finite* and *solvable*, and (c) all actions have *positive* cost.*

  2. Optimality: Does it always find solution with least cost?

     Yes: under same assumptions about action-costs.

  3. Time complexity: How long does it take to search?

     Expands all nodes with $g(n) \leq C^*$ (the cost of optimal solution).
     If all costs $\geq \varepsilon$ (some positive value) = $O(b^{1+C^*/\varepsilon})$

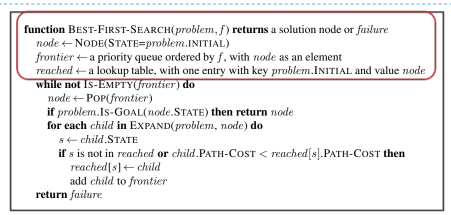
  4. Space complexity: How much memory does it take?

     Same as time = $O(b^{1+C^*/\varepsilon})$
     This can be **far worse** than $O(b^d)$ used by BFS!

18

Image source: Russell & Norvig (2021)

More Informed Search



```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure
```

- Many different uninformed search techniques exist; all will have some exponential worst-case complexity
  - If no solution exists, entire search-space must be seen
- We can still use this general framework in a way that, while *not guaranteed* to do better, often can solve problems much quicker
- We will evaluate nodes in our frontier not just based upon the *current path* found to those nodes, but also upon our *best guess* (heuristic) about which future paths might be best

19