Artificial Intelligence Foundation – JC3001

Lecture 5: Search I: Uninformed Search II

Prof. Aladdin Ayesh (aladdin.ayesh@abdn.ac.uk)

Dr. Binod Bhattarai (binod.bhattarai@abdn.ac.uk)

Dr. Gideon Ogunniye, (g.ogunniye@abdn.ac.uk)

September 2025

Material adapted from:
Russell and Norvig (AIMA Book): Chapter 3 (3.1–3.5)
Malte Helmert (University of Basel)

UNIVERSITY OF ABERDEEN

Course Progression

- Part 1: Introduction
  1. Introduction to AI ✓
  2. Agents ✓
- Part 2: Problem-solving
  1. **Search 1: Uninformed Search**
  2. Search 2: Heuristic Search
  3. Search 3: Local Search
  4. Search 4: Adversarial Search
- Part 3: Reasoning and Uncertainty
  1. Reasoning 1: Constraint Satisfaction
  2. Reasoning 2: Logic and Inference
  3. Probabilistic Reasoning 1: BNs
  4. Probabilistic Reasoning 2: HMMs
- Part 4: Planning
  1. Planning 1: Intro and Formalism
  2. Planning 2: Algos and Heuristics
  3. Planning 3: Hierarchical Planning
  4. Planning 4: Stochastic Planning
- Part 5: Learning
  1. Learning 1: Intro to ML
  2. Learning 2: Regression
  3. Learning 3: Neural Networks
  4. Learning 4: Reinforcement Learning
- Part 6: Conclusion
  1. Ethical Issues in AI
  2. Conclusions and Discussion

Objectives

- Search Algorithms (Tree/Graph)
- Uninformed Search Strategies

Outline

We can evaluate search strategies in several ways

- **Completeness:** Does it always find a solution if one exists?
- **Optimality:** Is the solution optimal (i.e., lowest cost)?
- **Time Complexity:** How long does it take to find a solution?
- **Space Complexity:** How much memory does it need to perform the search?

We can evaluate search strategies in several ways

- **Completeness:** Does it always find a solution if one exists?
- **Optimality:** Is the solution optimal (i.e., lowest cost)?
- **Time Complexity:** How long does it take to find a solution?
- **Space Complexity:** How much memory does it need to perform the search?

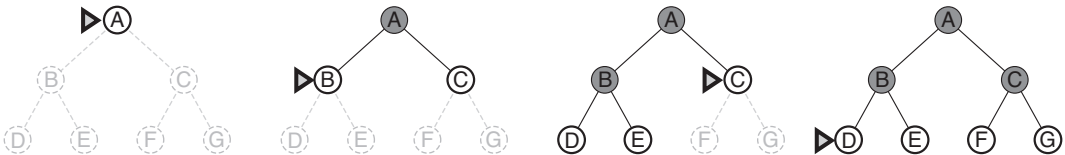
Time and space complexity are measured in terms of

- **b** – maximum branching factor of the search tree
- **d** – depth of the least-cost solution
- **m** – maximum depth of the state space (may be ∞)

Uninformed strategies use only the information available in the problem definition

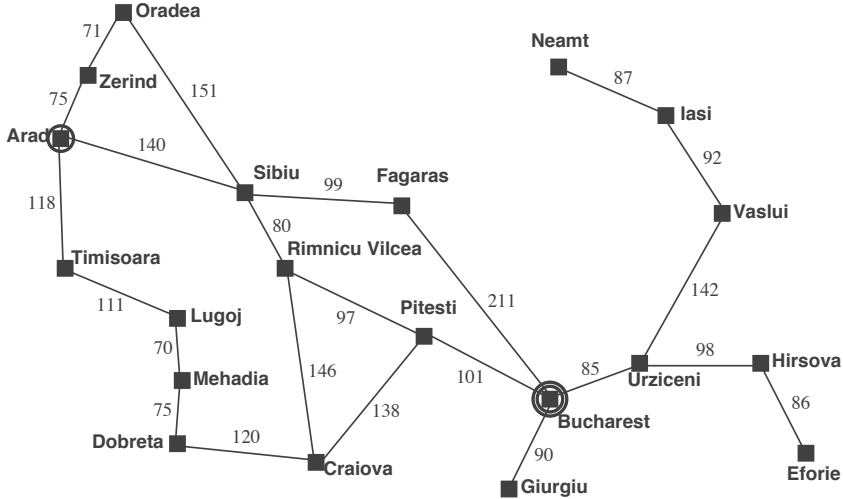
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

- Root node expanded first
- All successors to root node then expanded, then their successors, etc.
- All nodes at a given depth are expanded before the next level is expanded.



Implementation: *fringe* is a **FIFO** (queue)

Breadth-first search

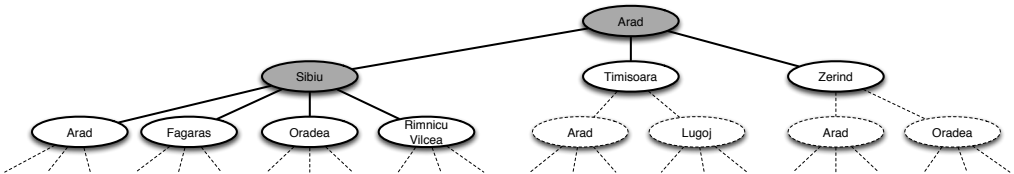


- **Complete?**
- **Time?**
- **Space?**
- **Optimal?**

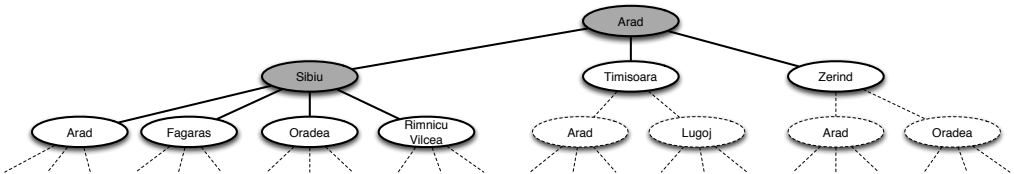
- **Complete** – Yes
- **Time** – $1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$
- **Space** – $O(b^d)$ nodes generated
- **Optimal** only if the path cost is a non-decreasing function of the depth of the node e.g.,?

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Outline

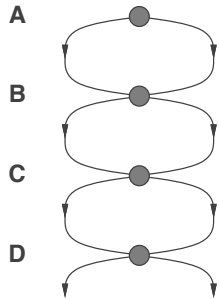


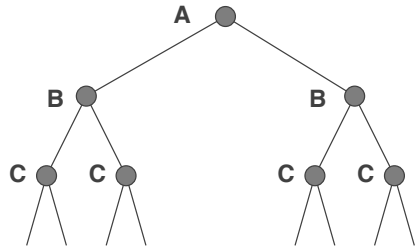
- Recall the Romania problem?
- Is this problem finite?



- Recall the Romania problem?
- Is this problem finite?
  Repeated states in the tree make it infinite

Failure to detect repeated states can turn a linear problem into an exponential one!





- We have been searching through trees so far.
- In graphs, we can expand states that have already been encountered and expanded before.
- Without detecting repeated states, we can loop, making otherwise solvable problems unsolvable.
- By remembering every expanded state and discarding any expansion that revisits this state we can prevent this problem from occurring.
- We store these expanded stats in a closed list

```
function graphSearch(problem p, strategy s)
    closed ← {}                                   ▷ We now keep a list of explored states
    frontier.add(newNode(p.initial))
    loop
        if frontier is empty then
            return fail
        n ← s.removeChoice(frontier)
        closed.add(n.state)                       ▷ Where we keep states we already visited
        if p.goalTest(n.state) then
            return getPath(n)
        for all a ∈ p.actions(n) do
            n′ ← a.result(n.state)
            if n′.state ∉ closed then
                frontier.add(n′)                  ▷ And only explore states we haven't visited
```

function graphSearch(problem p, strategy s)

$closed \leftarrow \{\}$ ▷ *We now keep a list of explored states*
$frontier$.add($newNode(p.initial)$)
loop
  if $frontier$ is empty **then**
    return $\boldsymbol{fail}$
  $n \leftarrow s$.removeChoice($frontier$)

  $closed$.add($n.state$) ▷ *Where we keep states we already visited*
  if p.goalTest($n.state$) **then**
    return getPath(n)
  for all $a \in p$.actions(n) **do**
    $n' \leftarrow a$.result($n.state$)
    if $n'.state \notin closed$ **then**

      $frontier$.add(n') ▷ *And only explore states we haven't visited*

- There are many data structures we can use to make things more efficient. E.g.
  - hash table for the closed set,
  - priority queue to determine what node to expand in uniform cost Search
  - keeping track of shortest path to each node
  - etc…

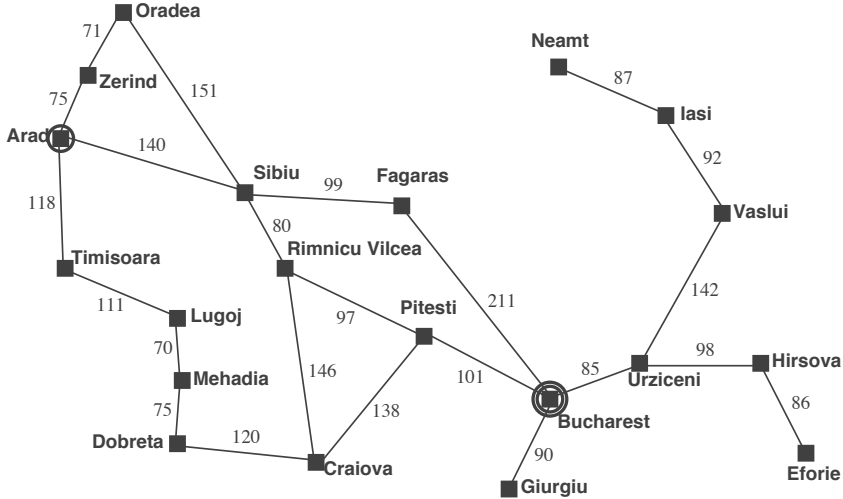
Outline

- Breadth-first search is optimal when all step costs are equal as it always expands the shallowest unexpanded nodes.
  - What if we have different **step-cost** functions?

- Breadth-first search is optimal when all step costs are equal as it always expands the shallowest unexpanded nodes.
  - What if we have different **step-cost** functions?
- We expand the node with the lowest path cost.
- For completeness, we need to always have a path cost (else we can loop forever).
- Implementation: *fringe* is a **FIFO** ordered by cost, lowest cost first (priority queue)

Uniform Cost Search

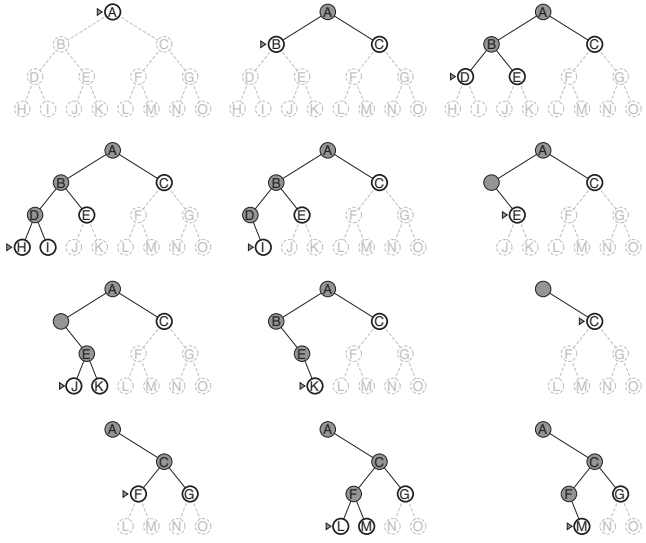
3 Search Strategies Continued



- This type of search expands the deepest node in the current frontier.
- When the end is reached, the search backs up to the next shallowest node that still has unexplored successors.
- Implementation: *fringe* is a **LIFO** (stack)
- What do we need to store in memory compared to BFS and UCS?

Depth-first Search

3 Search Strategies Continued



- **Complete?**
- **Time?**
- **Space?**
- **Optimal?**

- **Complete** – No: fails in infinite-depth spaces, spaces with loops
  Modify to avoid repeated states along path
  $\Rightarrow$ complete in finite spaces
- **Time** – $O(b^m)$ terrible if **m** is much larger than **d**
  but if solutions are dense, may be much faster than breadth-first
- **Space** – $O(bm)$, i.e., linear space!
  10 exab to 156kb
- **Optimal** – No

Outline

1: **function** depthLimitedSearch(problem p, limit l)
2: recursiveDLS($newNode(p.initial), p, l$)
3: **function** recursiveDLS(node n, problem p, limit l)

Depth-limited search


```
1:  function recursiveDLS(node n, problem p, limit l)
2:      cutoffOccurred ← false
3:      if p.goalTest(n.state) then
4:          return getPath(n)
5:      else if node.depth = l then
6:          return cutoff
7:      else
8:          for all successor in p.expand(n) do
9:              result ← recursiveDLS(sucessor, p, l)
10:             if result = cutoff then
11:                 cutoffOccurred ← true
12:             else if result ≠ failure then
13:                 return result
14:         if cutoffOcurred then
15:             return cutoff
16:         else
17:             return failure
```

DFS with depth limit **L**
i.e. nodes at depth **L** have no successors

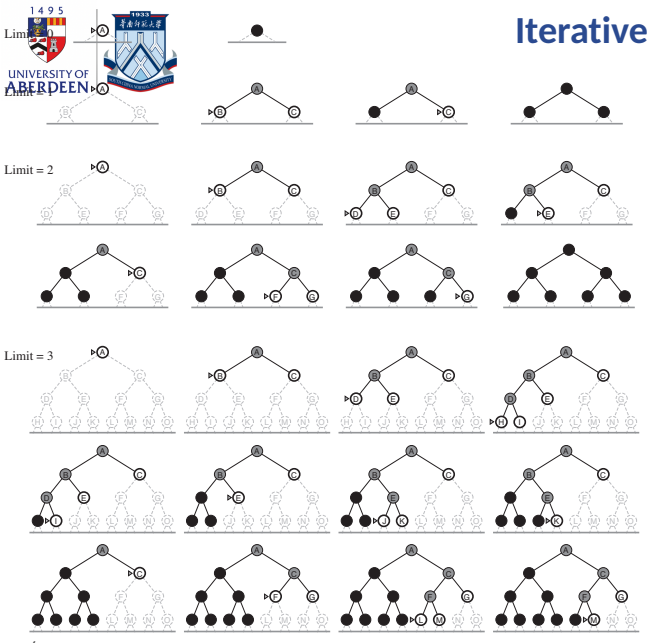
- This search solves the infinite path problem of DFS
- But incompleteness is introduced if the shallowest goal is beyond the depth limit.
- In routing example, if we know there are 20 cities, we know that max depth is 19.
  But examining the problem, we see that any city can be reached in depth 9.
  This is the **diameter** of the state space.

1: **function** IterativeDeepeningSearch(problem p)
2: **for all** $depth \leftarrow 0$ to ∞ **do**
3: $result \leftarrow$ DepthLimitedSearch$(problem, depth)$
4: **if** $result \neq cutoff$ **then**
5: **return** $result$

- This acts as a DFS with a gradually increasing limit until a goal is found.
- Combines benefits of DFS and BFS.
- Note that states are generated multiple times. But this is not costly as most nodes are at the bottom level.

Iterative deepening Depth-first search

4 Search Strategies Combined



Properties of iterative deepening search

- **Complete?**
- **Time?**
- **Space?**
- **Optimal?**

Properties of iterative deepening search

- **Complete** – Yes
- **Time** – $(d)b^0 + (d-1)b^1 + (d-2)b^2 + (d-3)b^3 + \cdots + (!)b^d = O(b^d)$
- **Space** – $O(bd)$
- **Optimal** – Yes, if step cost = 1
  Can be modified to explore uniform-cost tree

Numerical comparison for b = 10 and d = 5, solution at far right leaf:

$$N(IDS) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(BFS) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,100$$

Uninformed Search Summary

4 Search Strategies Combined

- Problem-solving agents
  - Problem types
  - Problem formulation
- Uninformed Search
  - Tree and Graph Search
  - Uninformed Search Strategies

Any Questions.