ISBAT UNIVERSITY
BLENDED LEARNING PLATFORM

BAI1103– FUNDAMENTALS OF ARTIFICIAL INTELLIGENCE AND ITS APPLICATIONS

CHAPTER FOUR
UNINFORMED SEARCH

BSc.AI&ML

A **search algorithm** is the step-by-step procedure used to locate specific data among a collection of data.

In Artificial Intelligence, search algorithms can be broadly categorized into two types:
Informed and Uninformed search algorithms.

These algorithms are used to navigate problem spaces, find solutions or optimize outcomes in various AI tasks.

In this chapter, we shall look into Uninformed search strategies in more detail.

Uninformed search

Qn. What is uninformed search?

Uninformed search strategies are those that **have no additional information** about states beyond that provided in the problem definition.

Uninformed search

Qn. What is uninformed search?

An uninformed search algorithm is one that generates the search tree without using any domain specific knowledge.

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Uninformed search

QN. What is uninformed search?

All they can do is generate successors and distinguish a goal state from a non-goal state.

Strategies that **know whether one non-goal state** is "more promising" than another are called **informed search or heuristic search strategies.**

Uninformed search

REMEMBER:

Uninformed search strategies :

- Have no additional information about the states other than what is provided in the problem.

- Operate in a brute force way where they systematically generate and explore all possible states and paths in the search until they find a solution.

- Can distinguish a goal state from a non goal state.

- Can generate successors.

-All non-goal nodes in frontier look equally good

Uninformed search

QN. How are Uninformed Search strategies useful in AI?

Uninformed search plays a fundamental role in AI.

1. Uninformed search algorithms provide a **systematic and exhaustive approach** to solving problems.

2. Uninformed search algorithms are **guaranteed to find a solution if one exists**, provided the **search space is finite**.

This **completeness** is crucial in AI tasks where ensuring that a solution is found is essential, such as route planning, puzzle-solving, and pathfinding.

Uninformed search

QN. How are Uninformed Search strategies useful in AI?

3. Uninformed search serves as a **foundation for more advanced search techniques, such as informed search algorithms**. Informed search combines the systematic exploration of uninformed search with heuristics to make more informed decisions about which paths to explore, leading to more efficient solutions.

Uninformed search

QN. How are Uninformed Search strategies useful in AI?

4. Understanding the **time and space complexities** of uninformed search algorithms is important in AI. This knowledge helps in selecting the appropriate search methods for different problem domains, taking into account computational resources and performance constraints.

5. Uninformed search algorithms can **help agents or robots determine the sequence of actions required to achieve a particular goal**. This is crucial in automated planning, robotics.

Uninformed search

There are 6 different Uninformed search algorithms, namely:

1. Breadth first search
2. Uniform Cost Search
3. Depth first search
4. Depth limited search
5. Iterative deepening depth first search
6. Bidirectional search.

Uninformed search

All search strategies are distinguished by the order in which nodes are expanded.

Strategy = order of tree expansion

In the context of uninformed search, the strategy refers to the **order in which a search algorithm explores nodes** in a search tree or graph while trying to find a solution to a problem.

Implemented by different queue structures (LIFO, FIFO, priority)

Uninformed search

1. LIFO (Last-In-First-Out)

This strategy, often implemented using a stack data structure, explores nodes in a depth-first manner.

The algorithm will continue to explore as deeply as possible along one branch of the search tree before backtracking to explore other branches. LIFO is useful for:

exploring deep paths, but it may not guarantee optimality in finding the shortest path to the goal.

Uninformed search

2. FIFO (First-In-First-Out)

This strategy, implemented using a queue data structure, explores nodes in a breadth-first manner.

It systematically expands all nodes at the current depth level before moving on to nodes at the next depth level.

FIFO is guaranteed to find the shortest path to the goal, but it may require a lot of memory and may not be as efficient as LIFO for deep search spaces.

Uninformed search

Priority Queue

In this strategy, nodes are expanded based on their priority, which is determined by a heuristic or a cost function. Priority queue-based strategies, such as A* search, prioritize nodes that are expected to lead to the goal more quickly. This can be more efficient than LIFO and FIFO for certain search problems, but it requires the design of a good heuristic function.

Search Strategies

Dimensions for evaluation

- Completeness- always find the solution?
- Optimality - finds a least cost solution (lowest path cost) first?
- Time complexity - # of nodes generated (worst case)
- Space complexity - # of nodes in memory (worst case)

- Time/space complexity variables
- b, maximum branching factor of search tree
- d, depth of the shallowest goal node
    - m, maximum length of any path in the state space (potentially □)

Uninformed search

In the context of uninformed search, such as breadth-first search (BFS) and depth-first search (DFS), various dimensions are used to evaluate the performance of search algorithms.

1. Completeness

Completeness refers to whether the search algorithm is **guaranteed to find a solution if one exists** in the search space.

RECAP OF UNINFORMED SEARCH ALGORITHMS

Big O notation, often denoted as O(), is a mathematical notation used to describe the upper bound or worst-case behavior of an algorithm's time complexity.

Big O notation provides an upper bound on the worst-case running time of an algorithm. It indicates that the actual running time of the algorithm will not grow faster than the function "f(n)" for sufficiently large input sizes.

Uninformed search

Time Complexity

Time complexity is measured by the number of nodes generated in the worst case, which depends on the branching factor (b) and the depth of the shallowest goal node (d).

In the worst case, BFS generates O(b^d) nodes, making it exponential in terms of time complexity.

DFS can potentially generate an enormous number of nodes in the worst case, especially in infinite or deep state spaces, making it difficult to provide a general time complexity bound.

Uninformed search

Space Complexity:

Space complexity is measured by the maximum number of nodes that need to be stored in memory at any point during the search, again depending on the branching factor (b) and the depth of the shallowest goal node (d).

DFS typically uses less memory compared to BFS, as it explores one branch deeply before backtracking. However, it may require more memory in the worst case if the search space is very deep, approaching O(bm), where m is the maximum length of any path.

Breadth-first search

Idea:
Expand shallowest unexpanded node

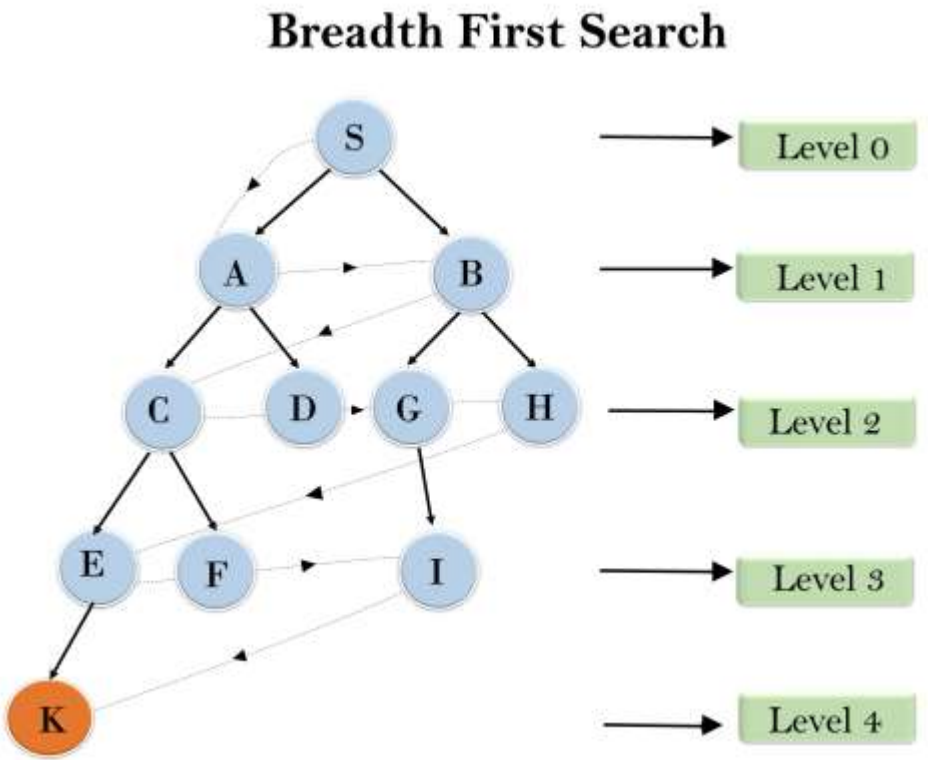
Implementation:
frontier is FIFO (First-In-First-Out)
Queue: Put successors at the end of frontier successor list.

Breadth-first search

In this type of search, the agent **considers every state as a node of a tree data structure**. Breadth-first search is a simple search strategy in which the **root node is expanded first**. All the **successors of the root node** are **expanded next**.

- It first **checks the current node** and then **evaluates all the neighboring nodes**.
- After all the neighboring nodes are checked, it **moves towards the next set of neighboring nodes for any of the neighbor nodes**, and this **process continues** until the search is ended.
- In BFS, the nodes of the tree are **traversed level after level**.

Breadth-first search



Breadth-first search

Breadth-first search is a simple strategy in which **the root node is expanded first**, then **all the successors of the root node are expanded next, then their successors, and so on.**

In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breadth-first search

In BFS, the shallowest unexpanded node is chosen for expansion.

This is achieved very simply by using a FIFO queue for the frontier.

Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.

Breadth-first search always has the shallowest path to every node on the frontier.

Breadth-first search

BFS is **Complete** - If the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes (provided the branching factor b is finite).

BFS can be quite memory-intensive, as it needs to store all nodes at a given level before moving to the next level. In the worst case, it can be O(b^d) in terms of space complexity.

Breadth-first search

BFS is **Optimal** - Breadth-first search is optimal if the path cost is a non decreasing function of the depth of the node.

This means that, in the context of a search problem, as you move deeper into the search tree (increasing the depth), the path cost does not decrease. In other words, the cost of moving from one state to another does not get cheaper as you go deeper into the search.

The most common such scenario is that all actions have the same cost.

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time Complexity? $1+b+b^2+b^3+ \ldots +b^d = O(b^d)$
- Space Complexity? $O(b^d)$ (keeps every node in memory)
- Optimal?

Yes, if cost = 1 per step
(not optimal in general)

b: maximum branching factor of search tree

d: depth of the least cost solution

m: maximum depth of the state space (∞)

Breadth-first search (simplified)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node <- a node with STATE = problem.INITIAL-STATE, PATH-COST=0 if
  problem.GOAL-TEST(node.STATE) then return SOLUTION(node)

  frontier <- a FIFO queue with node as the only element
  loop do
    if EMPTY?(frontier) then return failure
    node <- POP(frontier) // chooses the shallowest node in frontier
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child <- CHILD-NODE(problem, node, action)

      if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
      frontier <- INSERT(child, frontier)
```

Position within queue of new items determines search strategy

Subtle: *Node inserted into queue only after testing to*

From Figure 3.11 Breadth-first search (ignores loops, repeated nodes)

Exponential Space (and time) Not Good...

- Exponential complexity uninformed search problems *cannot* be solved for any but the smallest instances.
- *(Memory* requirements are a bigger problem than *execution* time.)

DEPTH	NODES	TIME	MEMORY
2	**110**	**0.11 milliseconds**	**107 kilobytes**
4	**11110**	**11 milliseconds**	**10.6 megabytes**
6	**10^6**	**1.1 seconds**	**1 gigabytes**
8	**10^8**	**2 minutes**	**103 gigabytes**
10	**10^{10}**	**3 hours**	**10 terabytes**
12	**10^{12}**	**13 days**	**1 petabytes**
14	**10^{14}**	**3.5 years**	**99 petabytles**

Fig 3.13 Assumes b=10, 1M nodes/sec, 1000 bytes/node

Review: Depth-first search

- **Idea:**
  - Expand *deepest* unexpanded node

- **<u>Implementation:</u>**
  - *frontier* is LIFO (Last-In-First-Out) Queue:
    - —Put successors at the *front* of *frontier* successor list.

Properties of depth-first search

- **Complete?** **No: fails in infinite-depth spaces, spaces with loops**
  - Modify to avoid repeated states along path
    → 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

$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$

- **Space?** ***O(b*m),* i.e., linear space!**
- **Optimal?** **No**

b: maximum branching factor of search tree
d: depth of the least cost solution
m: maximum depth of the state space (∞)

Depth-first vs Breadth-first

- **Use depth-first if**
  - *Space is restricted*
  - There are many possible solutions with long paths and wrong paths are usually terminated quickly
  - Search can be fine-tuned quickly

- **Use breadth-first if**
  - *Possible infinite paths*
  - Some solutions have short paths
  - Can quickly discard unlikely paths

UNIFORM COST SEARCH

- **Uninformed** search algorithm
- **Systematically explores** a graph or network
- Considers paths based on their **cumulative cost** from the start node
- Prioritizes nodes with **lower path costs**, ensuring that it finds the optimal solution, typically the shortest path, as long as the edge costs are non-negative.

UNIFORM COST SEARCH

UCS is considered an informed search algorithm because it uses cost information to guide its exploration.

However, in the broader classification scheme used in the field of artificial intelligence and search algorithms, UCS is typically categorized as an **uninformed search algorithm**. This is because:

UNIFORM COST SEARCH

In the context of search algorithms, the term "informed" usually refers to algorithms that use **heuristics—domain-specific estimates of the cost from a state to a goal state**.

Informed search algorithms, like A* or Greedy Best-First Search, **utilize heuristics to prioritize nodes based on the estimated cost to reach the goal.**

UCS, on the other hand, **prioritizes nodes solely based on the actual path cost**, not heuristic estimates.

UNIFORM COST SEARCH

UCS is often classified as an uninformed search algorithm because it doesn't **use heuristic information. It doesn't have access to any knowledge about the domain beyond the actual path costs.**

UCS explores the search space in a straightforward manner, considering nodes in ascending order of their path costs, without favoring any particular direction or path.

UNIFORM COST SEARCH

- BFS and UCS are **similar in several ways**, primarily because Uniform cost search can be seen as an extension or variation of BFS when applied to weighted graphs.

- Both BFS and UCS explore **nodes in a systematic and incremental manner.** They start with the initial node and proceed to expand nodes **level by level.**

UNIFORM COST SEARCH

- Both BFS and UCS use a queue-like data structure to maintain the order in which nodes are explored. BFS uses a **regular queue**, while UCS uses a **priority queue** based on the cost of the path.

- Both BFS and UCS may **consume significant memory**, particularly in deep or complex search spaces, as they need to maintain nodes at various levels in memory.

UNIFORM COST SEARCH

- BFS and UCS rely on **deterministic exploration strategies**. There is no randomness or heuristics involved in their decisions about which nodes to explore next.

- In both BFS and UCS, the search process results in a tree structure that represents the **shortest path from the start node to all explored nodes.**

- Both algorithms are capable of handling **edge costs.** While BFS treats all edges as having equal cost (for unweighted graphs), UCS considers the actual cost of each edge when determining the order of exploration.

UNIFORM COST SEARCH

Some of the differences between BFS and UCS:

BFS is typically applied to unweighted or uniformly weighted graphs, where all edges have the same cost. It doesn't consider the cost associated with traversing edges.

UCS is designed for weighted graphs, where each edge has an associated cost. It explicitly considers these edge costs when determining the order in which nodes are explored.

UNIFORM COST SEARCH

Some of the differences between BFS and UCS:

BFS guarantees finding the optimal solution, which is the shortest path, in unweighted or uniformly weighted graphs. It always finds the shallowest solution.

UCS guarantees finding the optimal solution in weighted graphs as long as edge costs are non-negative. It considers the cumulative path cost to determine the optimal solution.

UNIFORM COST SEARCH

Some of the differences between BFS and UCS:

BFS guarantees finding the optimal solution, which is the shortest path, in unweighted or uniformly weighted graphs. It always finds the shallowest solution.

UCS guarantees finding the optimal solution in weighted graphs as long as edge costs are non-negative. It considers the cumulative path cost to determine the optimal solution.

UNIFORM COST SEARCH

Some of the differences between BFS and UCS:

BFS often uses **a simple queue data structure** to maintain the frontier of nodes to be explored.

UCS uses a **priority queue** to maintain the frontier and ensure that nodes with lower path costs are expanded first.

BFS may not be efficient for finding the shortest path in weighted graphs, as it doesn't consider edge costs.

UCS is effective for finding the shortest path in weighted graphs but may not be efficient for unweighted graphs where BFS is more appropriate.

UNIFORM COST SEARCH

Dimensions of evaluation for UCS

1. **Completeness** – It is a complete algorithm, therefore, it will find a solution if one exists in a finite search space.

2. **Optimality** – It is an optimal algorithm, and guarantees finding the shortest path, assuming all edge costs are equal to or greater than zero, Uniform cost search is also optimal only when the search space is finite (meaning that there are a finite number of nodes and edges to explore).

UNIFORM COST SEARCH

3. **Time complexity** – In the worst case, the time complexity of Uniform cost search is O(b^d) where:

"b" represents the branching factor, which is the maximum number of successor nodes generated from any given node.
"d" represents the depth of the shallowest goal node, which is the depth of the optimal solution.

The higher the branching factor and depth, the longer the time taken to explore the entire search space.

Depth-limited search: A building block

- **Depth-First search *but with depth limit l.***
  - i.e. nodes at depth *l* *have no successors.*
  - No infinite-path problem!

- **If *l* = *d* (by luck!), then optimal**
  - But:
    - **If *l* < *d* then incomplete** ☹
    - **If *l* > *d* then not optimal** ☹

- **Time complexity:** $O(b^l)$
- **Space complexity:** $O(bl)$ ☺

DEPTH LIMITED SEARCH

4. **Space complexity** - In the worst case, the space complexity of Uniform Cost Search is also O(b^d).

Uniform Cost Search needs to keep track of nodes at various levels in the search tree.

Uniform Cost Search may need to store a substantial number of nodes in memory, especially if the branching factor is high or the optimal solution is deep.

DEPTH LIMITED SEARCH

Depth-Limited Search (DLS) is a search algorithm that is a variant of Depth-First Search (DFS), with the added feature of a depth limit. It explores a search space up to a specified depth level and avoids going deeper.

DEPTH LIMITED SEARCH

Depth Limit: DLS is defined by a depth limit, which determines how deep the search goes in the search tree. Any paths that exceed this depth limit are not explored.

Depth-First Nature: Like DFS, DLS explores a branch of the search space as deeply as possible before backtracking to explore other branches. It descends as far as the depth limit allows and then backtracks when necessary.

DEPTH LIMITED SEARCH

Completeness: DLS is not complete by itself. It can miss a solution if the depth limit is set too low and the solution lies beyond that limit. However, it can be made complete by gradually increasing the depth limit in an iterative deepening search.

Memory Efficiency: DLS is memory-efficient compared to BFS because it doesn't need to store nodes at deeper levels in the search tree. It only maintains nodes within the specified depth limit in memory.

DEPTH LIMITED SEARCH

Applications: DLS is commonly used in game-playing algorithms to explore a game tree to a certain depth. It is also used in problems where a solution is not found or required at a deeper level.

Optimality: DLS does not guarantee optimality. It may find a solution within the depth limit, but that solution may not be optimal if a deeper solution exists.

DEPTH LIMITED SEARCH

- **Breadth-first**
  - ☑ Complete,
  - ☑ Optimal
  - ☒ *but* uses $O(b^d)$ space
- **Depth-first**
  - ☒ Not complete *unless m is bounded*
  - ☒ Not optimal
  - ☒ Uses $O(b^m)$ time; terrible if m >> d
  - ☑ *but* only uses ***O(b∗m) space***

How can we get the best of both?

DEPTH LIMITED SEARCH

Multiple Iterations: To ensure completeness and explore deeper levels, an iterative deepening version of DLS can be used. It involves running DLS with increasing depth limits until a solution is found.

Heuristic-Free: DLS, like DFS, doesn't rely on heuristics or cost information. It explores nodes solely based on depth.

DEPTH LIMITED SEARCH

Efficiency Trade-off: The depth limit in DLS represents a trade-off between efficiency and completeness. Lower depth limits make the search faster but less complete, while higher depth limits make it more complete but potentially slower.

Backtracking: DLS backtracks when a path reaches the depth limit, allowing it to explore other paths within the limit.

ITERATIVE DEEPENING DEPTH FIRST SEARCH

DEFINITION: It is a general strategy, often used in combination with Depth first tree search, that finds the best depth limit.

This search gradually increases the limit until a goal is found (occurs when the depth limit reaches d, the depth of the shallowest goal node).

-Combination of both DFS and BFS.

BFS -> All the nodes are expanded at each level, and the algorithm is complete and optimal.

DFS -> Has the lesser memory requirement of DFS, and is also implemented using a stack.

ITERATIVE DEEPENING DEPTH FIRST SEARCH

Advantages:

- No loops
- Fast, less memory
- Incorporates benefits of both DFS and BFS

Disadvantages:

- The process/ work is repetitive.

ITERATIVE DEEPENING DEPTH FIRST SEARCH

DIMENSIONS OF EVALUATION:

Completeness: The algorithm is complete if b (where b is the branching factor) is finite.

Optimal: The algorithm is optimal (if step costs are all identical)

Time complexity: This is denoted by O(b^d)

Space complexity: This is denoted by O(bd)

Iterative deepening depth first search

- **A general strategy to find best depth limit *l*.**
  - **Key idea: use Depth-limited search as subroutine, with increasing *l* .**

```
For d = 0 to ∞ do
        depth-limited-search to level d
        if it succeeds
                then return solution
```

  - ***Complete & optimal*: Goal is always found at depth *d*, the depth of the shallowest goal-node.**

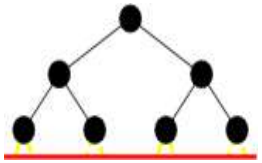
Could this possibly be efficient?

Nodes constructed at each deepening

- **Depth 0: 0 (Given the node, doesn't *construct* it.)**

- **Depth 1: b^1 nodes**

- **Depth 2: b nodes + b^2 nodes**



- **Depth 3: b nodes + b^2 nodes + b^3 nodes**
- **…**

ID search, Evaluation II: Time Complexity

- **More generally, the time complexity is**
  - $N(IDS) = (d)b + (d-1)b^2 + \ldots + (1)b^d = O(b^d)$

- ***As efficient in terms of O(…)*** **as Breadth First Search:**
  - $N(BFS) = b + b^2 + \ldots + b^d = O(b^d)$

Total nodes constructed:

- Depth 0: 0 (Given the node, doesn't *construct* it.)
- Depth 1: $b^1 = b$ nodes
- Depth 2: b nodes + b^2 nodes
- Depth 3: b nodes + b^2 nodes + b^3 nodes
- …

Suppose the first solution is the last node at depth 3:

Total nodes constructed:

3*b nodes + ***2****b^2 nodes + ***1****b^3 nodes

ID search, Evaluation III

- **Complete: YES (no infinite paths)** ☺
- **Time complexity:** $O(b^d)$
- **Space complexity:** $O(bd)$ ☺
- **Optimal: YES if step cost is 1.** ☺

Summary of algorithms

Criterion	Breadth-First	Depth-First	Depth-limited	Iterative deepening
Complete?	**YES**	**NO**	**NO**	**YES**
Time	b^d	b^m	b^l	b^d
Space	b^d	bm	bl	bd
Optimal?	**YES**	**NO**	**NO**	**YES**

BIDIRECTONAL SEARCH

-> Two different searches are run simultaneously. This makes it fast.

-> Single search graph is replaced with 2 small graphs (forward and backward)

-> Any search technique can be used (BFS or DFS).

-> STOP condition for search (the search stops when):-
When graphs interact.

Advantages: - Fast, less memory

Disadvantages:- Implementation is difficult

BIDIRECTONAL SEARCH

DIMENSIONS OF EVALUATION (if applicable):-

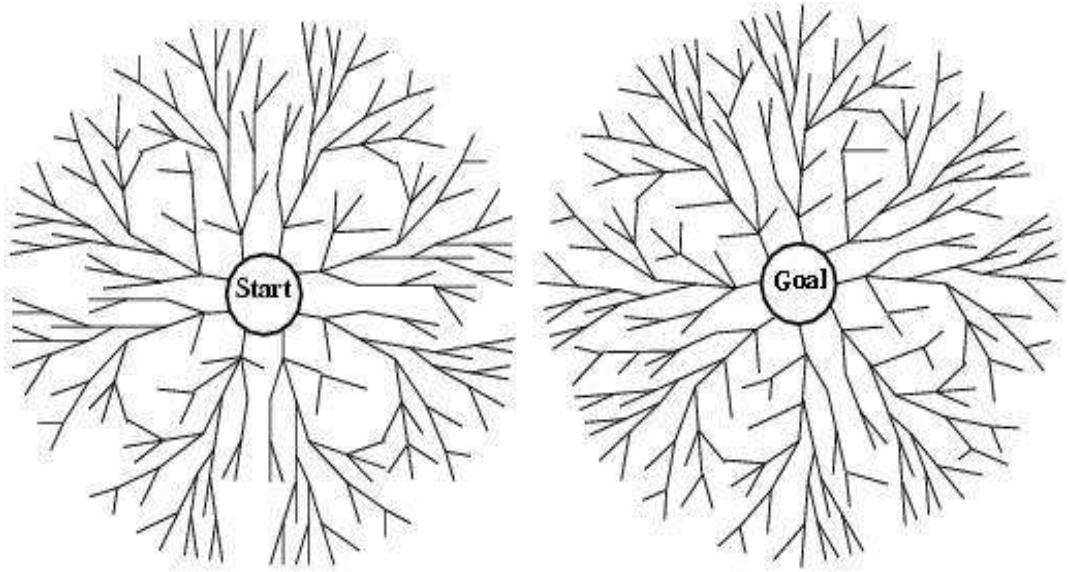
Complete: YES if b is finite, if directions use the same search algorithm

Optimal: if step costs are all identical, if both directions use the same algorithm

Time complexity: O(b^d/2)

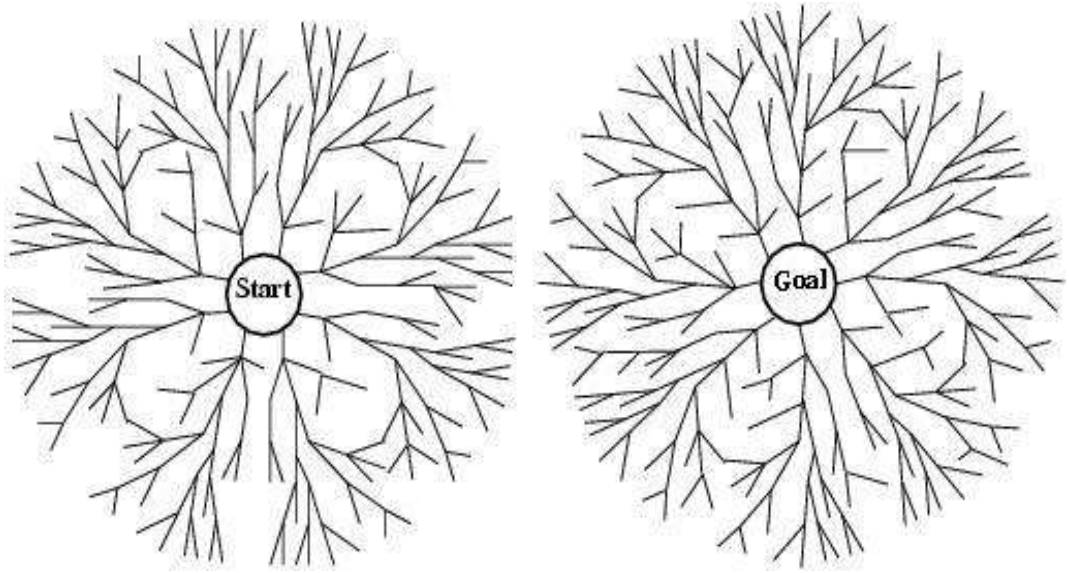
Space complexity: O(b^d/2)

Very briefly: Bidirectional search



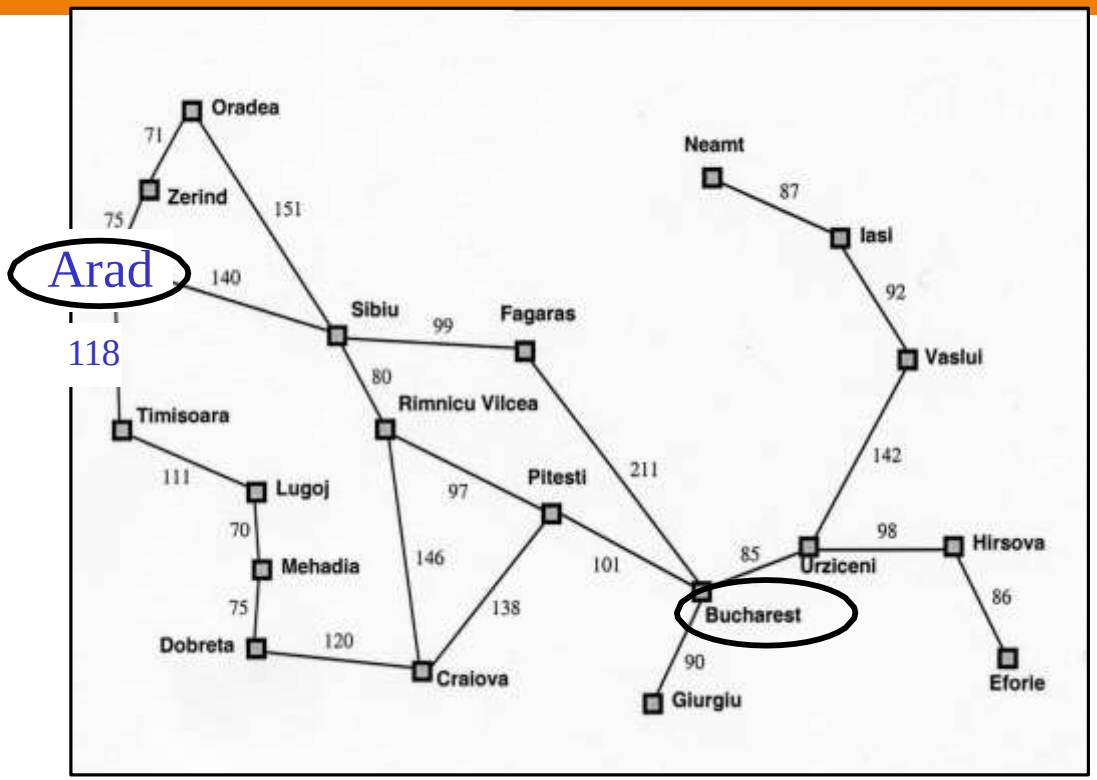
- **Two simultaneous searches from start an goal.**
  - Motivation: $b^{d/2} + b^{d/2} < b^{d}$
- **Check whether the node belongs to the other frontier before expansion.**
- **Space complexity is the most significant weakness.**
- **Complete and optimal if both searches are Breadth-First.**

How to search backwards?



- **The predecessor of each node must be efficiently computable.**
  - Works well when actions are easily reversible.

Motivation: Romanian Map Problem



- **All our search methods so far assume *step-cost = 1***
- ***This is only true for some problems***

g(N): the *path cost* function

- **If all moves equal in cost:**
  - Cost = # of nodes in path-1
  - g(N) = depth(N) in the search tree
  - Equivalent to what we've been assuming so far

- **Assigning a (potentially) unique cost to each step**
  - N_0, N_1, N_2, N_3 = nodes visited on path *p* from N_0 to N_3
  - C(i,j): Cost of going from N_i to N_j
  - If N_0 the root of the searchtree,

$$g(N_3)=C(0,1)+C(1,2)+C(2,3)$$

Uniform-cost search (UCS)

- **Extension of BF-search:**
  - **Expand node with *lowest path cost***
- **Implementation:**
  ***frontier* = priority queue ordered by g(n)**
- **Subtle but significant difference from BFS:**
  - Tests if a node is a goal state when it is selected for expansion, not when it is added to the frontier.
  - Updates a node on the frontier if a better path to the same state is found.
  - So always enqueues a node *before checking whether it is a goal.*

WHY???

Uniform Cost Search

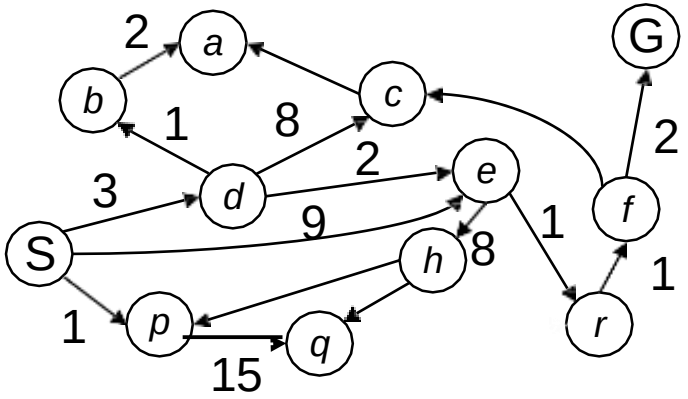
Slide from Stanford CS 221 (from slide by Dan Klein (UCB) and many others)

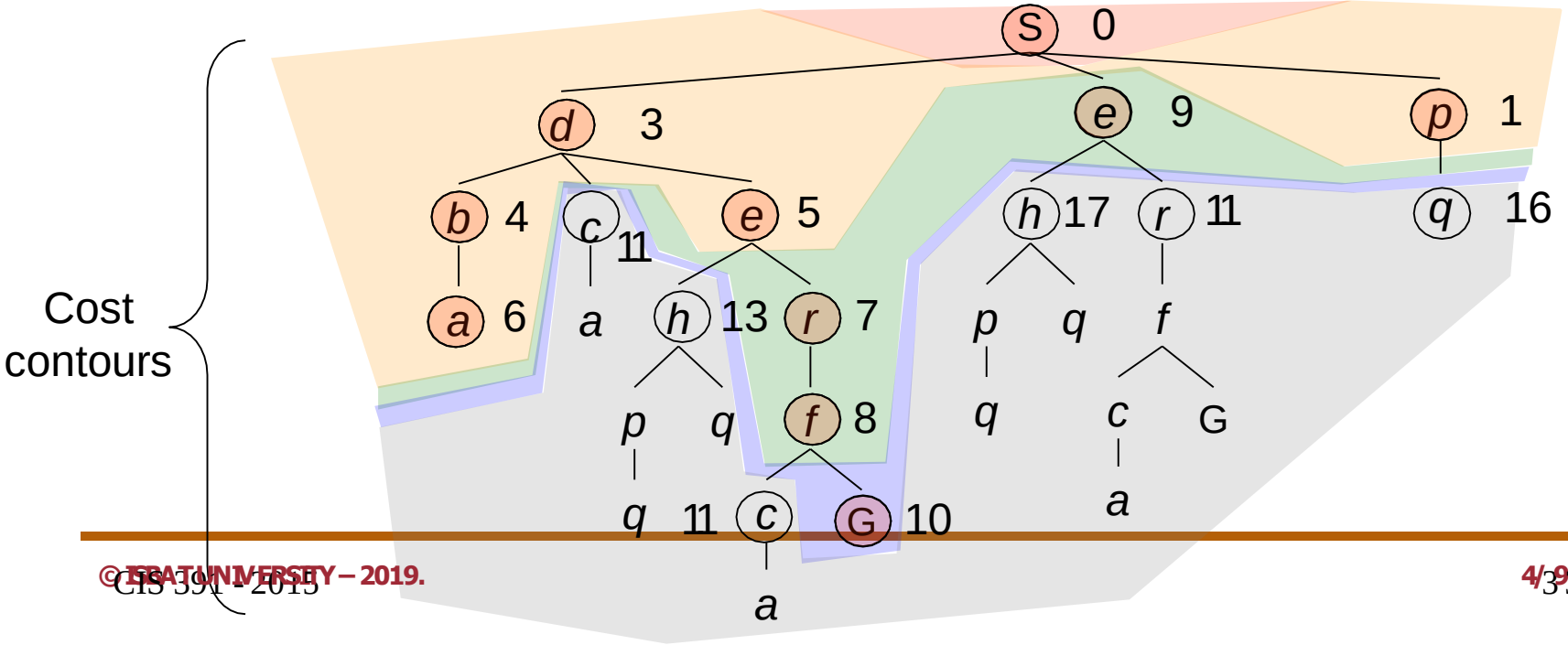
Expand cheapest node first:

Frontier is a priority queue

No longer ply at a time, but follows cost contours

Therefore: Must be optimal





Complexity of UCS

- **Complete!**
- **Optimal!**
  - if the cost of each step exceeds some positive bound ε
- **Time complexity:** $O(b^{1 + C^*/\varepsilon})$
- **Space complexity:** $O(b^{1 + C^*/\varepsilon})$

where C is the cost of the optimal solution*

(if all step costs are equal, this becomes $O(b^{d+1})$

NOTE: Dijkstra's algorithm just UCS without goal

Summary of algorithms (for notes)

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete ?	**YES**	**YES**	**NO**	**NO**	**YES**	**YES**
Time	b^d	$b^{1+C^*/e}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	$b^{1+C^*/e}$	bm	bl	bd	$b^{d/2}$
Optimal?	**YES**	**YES**	**NO**	**NO**	**YES**	**YES**

Assumes b is finite

RECAP OF UNINFORMED SEARCH ALGORITHMS

BFS

Idea: Expand shallowest unexpanded node
Implementation: Frontier is FIFO (First-In-First-Out) and implements a Queue
Dimensions for evaluation:
Completeness : COMPLETE, finite search space
Optimality: OPTIMAL for all actions having the same cost, finite search space
Time Complexity: O(b^d)
Space Complexity: O(b^d) – Has memory constraints

RECAP OF UNINFORMED SEARCH ALGORITHMS

DFS

Idea:

Dimensions for evaluation

Completeness : NO fails in infinite-depth spaces, spaces with loops

Optimality : NO

Time Complexity: O(b^m) **: terrible if *m* is much larger than *d***

- but if solutions are dense, may be much faster than breadth-first

$T(n)= 1+ n^2+ n^3 +.....+ n^m=O(n^m)$

Space Complexity O(b*m)

RECAP OF UNINFORMED SEARCH ALGORITHMS

Uniform cost search:

Dimensions for evaluation:

Completeness - It is a complete algorithm, therefore, it will find a solution if one exists in a finite search space.

2. **Optimality** – It is an optimal algorithm, assuming all edge costs are equal to or greater than zero, and the search space is finite

3. Time Complexity - $O(b^{1+C^*/e})$

4. Space Complexity - $O(b^{1+C^*/e})$

RECAP OF UNINFORMED SEARCH ALGORITHMS

Uniform cost search:

3. Time Complexity - $O(b^{1+C^*/e})$

b" is the branching factor, which is the maximum number of successor nodes generated from any given node.

"C*" is the cost of the optimal solution (i.e., the minimum path cost to the goal).

"e" is a positive constant representing the minimum edge cost in the search space.

RECAP OF UNINFORMED SEARCH ALGORITHMS

Uniform cost search:

3. Time Complexity - O($b^{1+C^*/e}$)

"O(b^1 + C*/e)" means that it explores nodes at each level of the search tree (b^1) and may need to reach a certain depth (C*/e) to find the optimal solution.

RECAP OF UNINFORMED SEARCH ALGORITHMS

Depth limited search

Dimensions for evaluation:
Completeness: NO, stops when the depth limit comes in
Optimality: NO
Time Complexity: O(b^l)
Space Complexity: O(bl)

RECAP OF UNINFORMED SEARCH ALGORITHMS

Iterative deepening depth first search

Dimensions for evaluation:
Completeness : YES if search space is finite
Optimality: YES if step costs are all identical
Time Complexity: O(b^d)
Space Complexity: O(bd)

RECAP OF UNINFORMED SEARCH ALGORITHMS

Bidirectional Search

Idea:
Implementation:

Dimensions for evaluation:
Completeness: YES if b is finite, if both directions use the same search strategy like BFS.
Optimality: YES if step costs are identical
Time Complexity: O(b^d/2)
Space Complexity: O(b^d/2)

RECAP OF UNINFORMED SEARCH ALGORITHMS

Read about:

THE MISSIONARIES AND CANNIBALS PROBLEM

GROUP ASSIGNMENTS

2. Define Search algorithms, describe the properties of search algorithms, why are search algorithms useful in AI, types of search algorithms with examples of each.

3. Breadth first search and uniform cost search

1. Depth first search

4. Depth limit search and iterative deepening search

5. Bidirectional search

(key points, how it works, advantages and disadvantages, dimensions of evaluation)