Review

- Types of Agents
- Problem formulation for goal based agents
    - States
    - Initial State
    - Actions
    - Goal
- Example of searching in the problem space

Search Implementation

- Open set
    - Group of states that haven’t been expanded yet
    - Group of states that are children of nodes that have been expanded
    - Search strategy chooses which open state to expand next
    - “set” as in collection of objects, not strictly mathematical set
    - Data structure depends on search strategy
- Closed set
    - Group of states that *have* been expanded
    - Often neglected (for space reasons), more later

Search Implementation

Pseudocode:

```
1. Initialize Open Set to contain initial state
2. Choose/remove one state from Open
   1. If chosen state is already closed jump to 2.4
   2. Check if chosen state is a goal state
         1. done if so
   3. Get child states of the chosen state using successor function
         1. Insert children into Open
         2. (Optional) Insert original state into Closed
   4. Repeat from (2)
```

Types of Search

- Uninformed search
  - No information about problem (other than its definition)
  - Generate successors
  - Can distinguish a goal state from a non-goal state
  - search strategies are distinguished by the order in which nodes are expanded.

Types of Search

- Uninformed search
  - No information about problem (other than its definition)
  - Generate successors
  - Can distinguish a goal state from a non-goal state
  - search strategies are distinguished by the order in which nodes are expanded.
- Informed search
  - Given some idea of where to look for solutions

Types of Uninformed Search

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

Types of Uninformed Search

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

- Strategies differ by the order in which child nodes are expanded
  - i.e., removed from Open set

Breadth First Search

- Expand root node, then expand all successors, then their successors, and so on

Breadth First Search

- Expand root node, then expand all successors, then their successors, and so on
- Expand shallowest unexpanded node first
    - Left-to-right or right-to-left ordering

Breadth First Search

- Expand root node, then expand all successors, then their successors, and so on
- Expand shallowest unexpanded node first
  - Left-to-right or right-to-left ordering
- All nodes at level d are expanded before level d+1
  - Root node is level d = 0

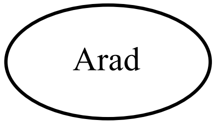
Breadth First Search

- Expand root node, then expand all successors, then their successors, and so on
- Expand shallowest unexpanded node first
  - Left-to-right or right-to-left ordering
- All nodes at level d are expanded before level d+1
  - Root node is level d = 0
- Method finds the shallowest goal state

Breadth First Search

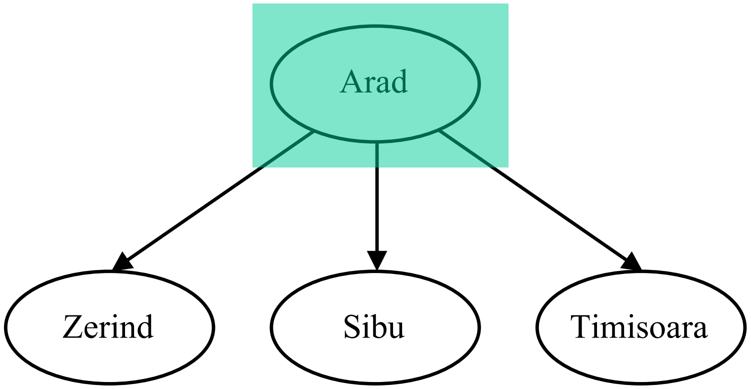
- Expand root node, then expand all successors, then their successors, and so on
- Expand shallowest unexpanded node first
  - Left-to-right or right-to-left ordering
- All nodes at level d are expanded before level d+1
  - Root node is level d = 0
- Method finds the shallowest goal state
- Open set
  - Put new nodes at end, remove from front
  - Queue, first-in first-out (FIFO)

BFS Example



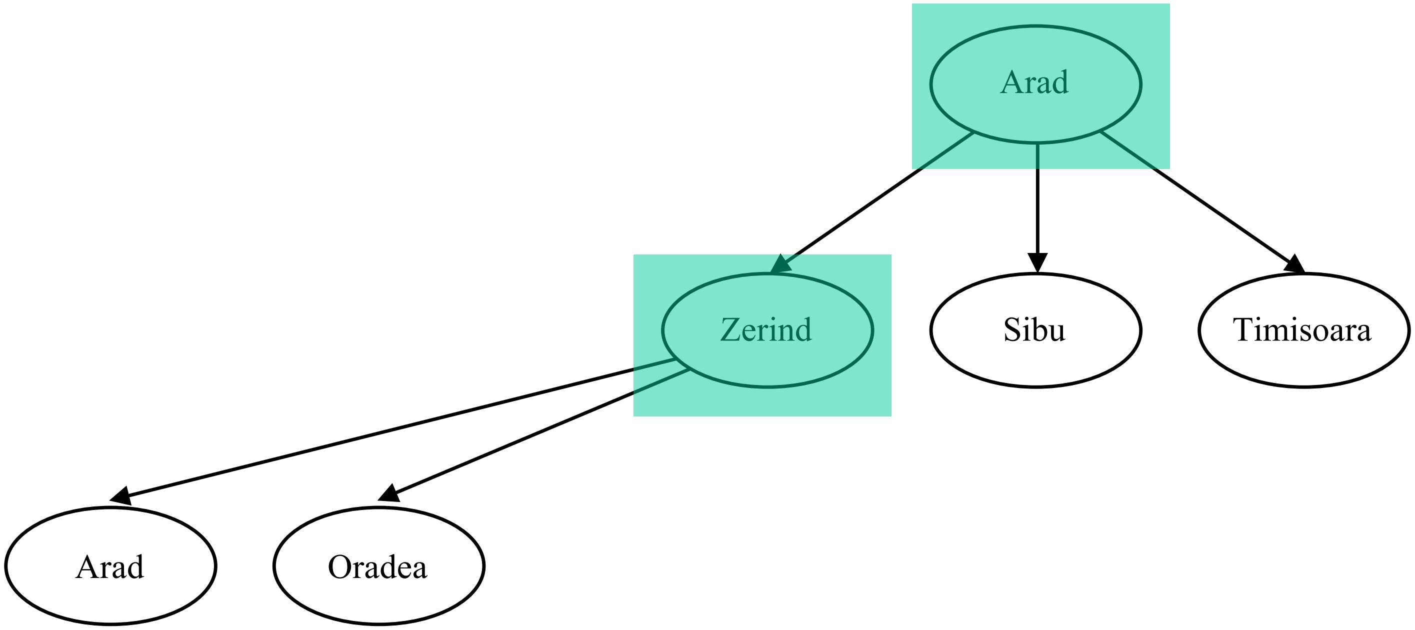
Open = { Arad }
Closed = { }

BFS Example



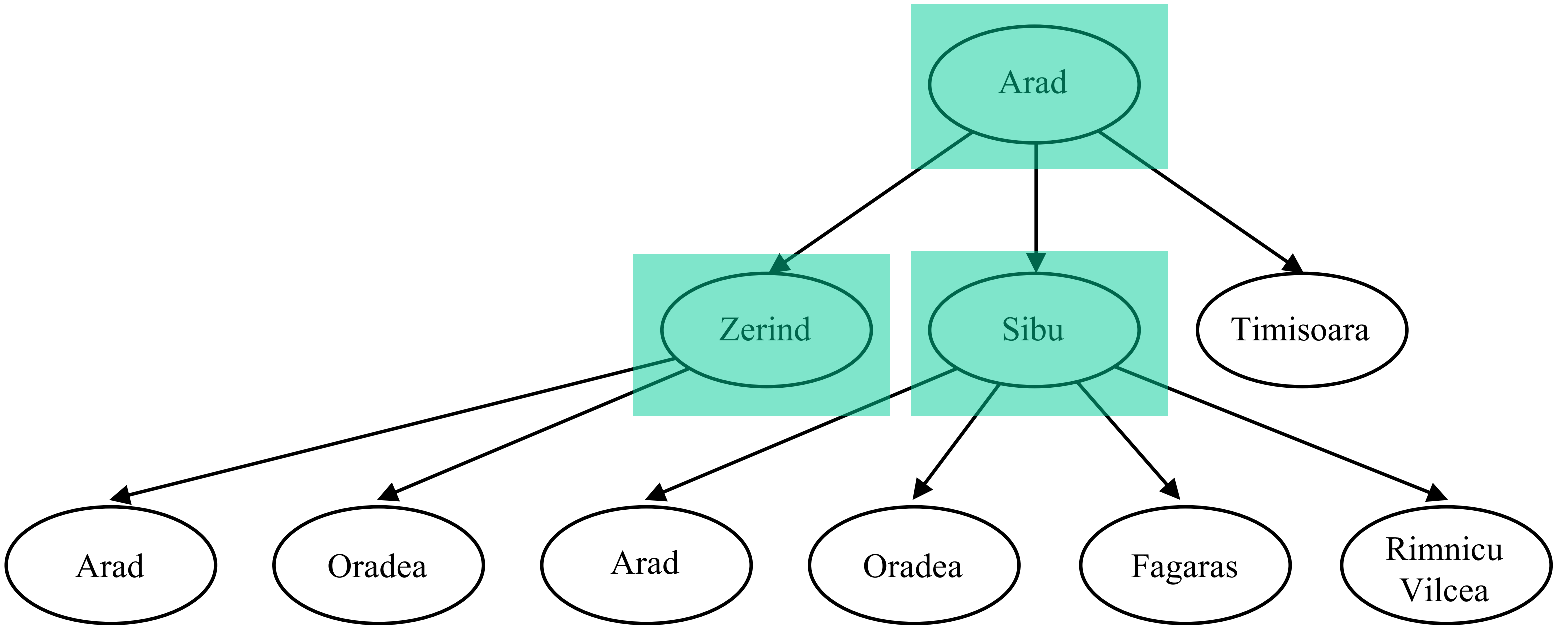
Open = { Zerind, Sibu, Timisoara }
Closed = { Arad }

BFS Example



Open = { Sibu, Timisoara, Arad, Oradea }
Closed = { Arad, Zerind }

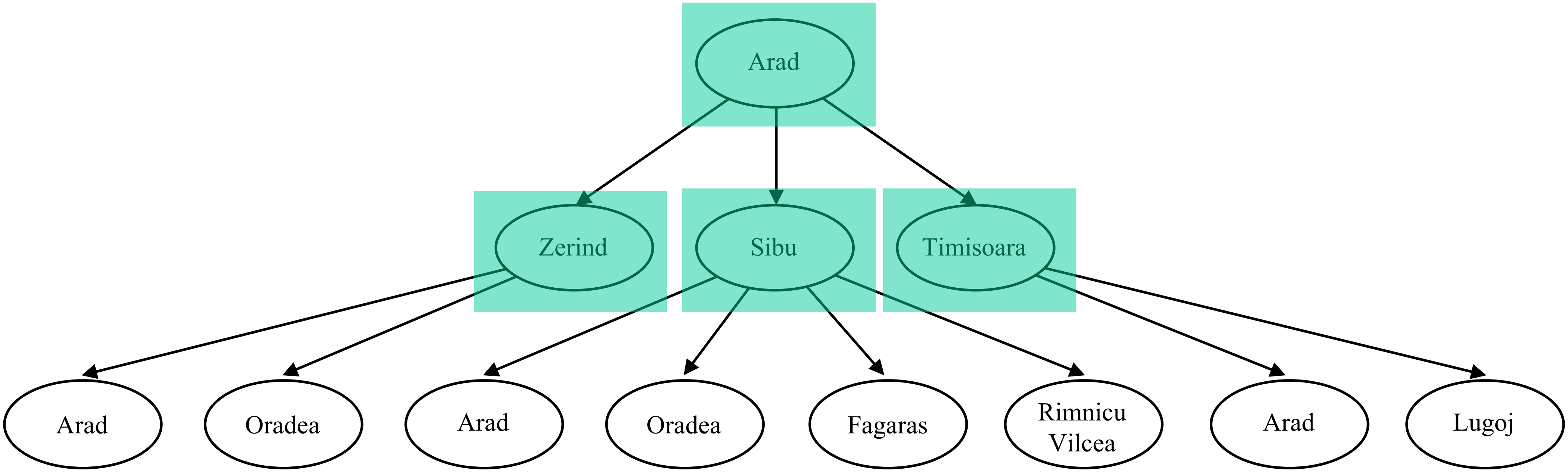
BFS Example



Open = { Timisoara, Arad, Oradea, Arad, Oradea, Fagaras, Rimnicu Vicea }

Closed = { Arad, Zerind, Sibu }

BFS Example



Open = { Arad, Oradea, Arad, Oradea, Fagaras, Rimnicu Vicea, Arad, Lugoj }

Closed = { Arad, Zerind, Sibu, Timisoara }

BFS Properties

- Complete (good)
  - If branching factor *b* is finite

BFS Properties

- Complete (good)
    - If branching factor *b* is finite
- Space – nodes generated (exponential - bad)
    - $O(b^{d+1}) = b + b^2 + \ldots + b^d + \mathbf{(b^{d+1} - b)}$, d = goal depth

BFS Properties

- Complete (good)
  - If branching factor *b* is finite
- Space – nodes generated (exponential - bad)
  - $O(b^{d+1}) = b + b^2 + \ldots + b^d + \mathbf{(b^{d+1} - b)}$, d = goal depth
    - Assume goal is last node (e.g., rightmost) at depth *d*
    - Goal state is not expanded

BFS Properties

- Complete (good)
  - If branching factor *b* is finite
- Space – nodes generated (exponential - bad)
  - $O(b^{d+1}) = b + b^2 + \ldots + b^d + \mathbf{(b^{d+1} - b)}$, d = goal depth
    - Assume goal is last node (e.g., rightmost) at depth *d*
    - Goal state is not expanded
  - Big limitation (need lots of space)
    - Depth=10, branching=10, space=1000 bytes/node ® 101 terabytes!

BFS Properties

- Complete (good)
  - If branching factor *b* is finite
- Space – nodes generated (exponential - bad)
  - $O(b^{d+1}) = b + b^2 + \ldots + b^d + \mathbf{(b^{d+1} - b)}$, d = goal depth
    - Assume goal is last node (e.g., rightmost) at depth *d*
    - Goal state is not expanded
  - Big limitation (need lots of space)
    - Depth=10, branching=10, space=1000 bytes/node ® 101 terabytes!
- Time (bad)
  - Same as space

BFS Properties

- Complete (good)
  - If branching factor *b* is finite
- Space – nodes generated (exponential - bad)
  - $O(b^{d+1}) = b + b^2 + \ldots + b^d + \mathbf{(b^{d+1} - b)}$, d = goal depth
    - Assume goal is last node (e.g., rightmost) at depth *d*
    - Goal state is not expanded
  - Big limitation (need lots of space)
    - Depth=10, branching=10, space=1000 bytes/node ® 101 terabytes!
- Time (bad)
  - Same as space
- Optimality (good)
  - Not in general, shallowest may not be optimal path cost
  - Optimal if path cost non-decreasing function of node depth

Uniform-Cost Search

- Modified breadth-first strategy
- Expand least-cost unexpanded leaf node first

Uniform-Cost Search

- Modified breadth-first strategy
- Expand least-cost unexpanded leaf node first
  - (rather than lowest-depth as in BFS)
  - General additive cost function
  - i.e. cost from initial state to current state
  - Not cost from current state to goal! – not "informed" search!!

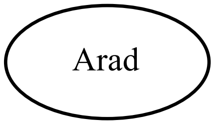
Uniform-Cost Search

- Modified breadth-first strategy
- Expand least-cost unexpanded leaf node first
  - (rather than lowest-depth as in BFS)
  - General additive cost function
  - i.e. cost from initial state to current state
  - Not cost from current state to goal! – not “informed” search!!
- Guaranteed to be the *cheapest* solution
  - Otherwise it would have been expanded later

Uniform-Cost Search

- Modified breadth-first strategy
- Expand least-cost unexpanded leaf node first
  - (rather than lowest-depth as in BFS)
  - General additive cost function
  - i.e. cost from initial state to current state
  - Not cost from current state to goal! – not "informed" search!!
- Guaranteed to be the *cheapest* solution
  - Otherwise it would have been expanded later
- Open set
  - Remove nodes in order of increasing path cost
  - Priority queue

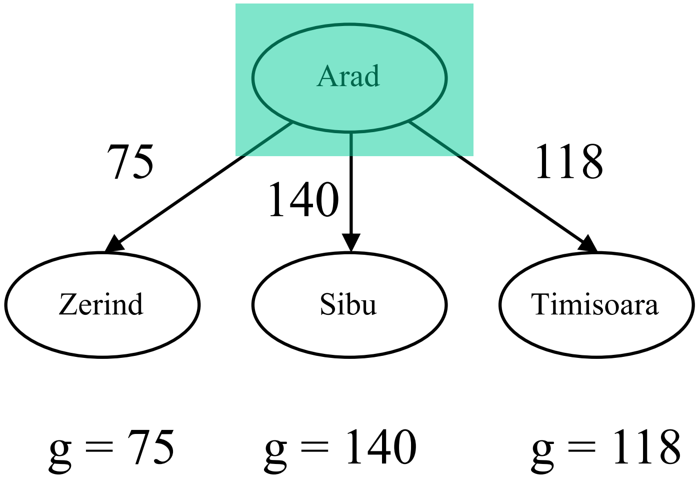
UCS Example



Open = { Arad(0) }

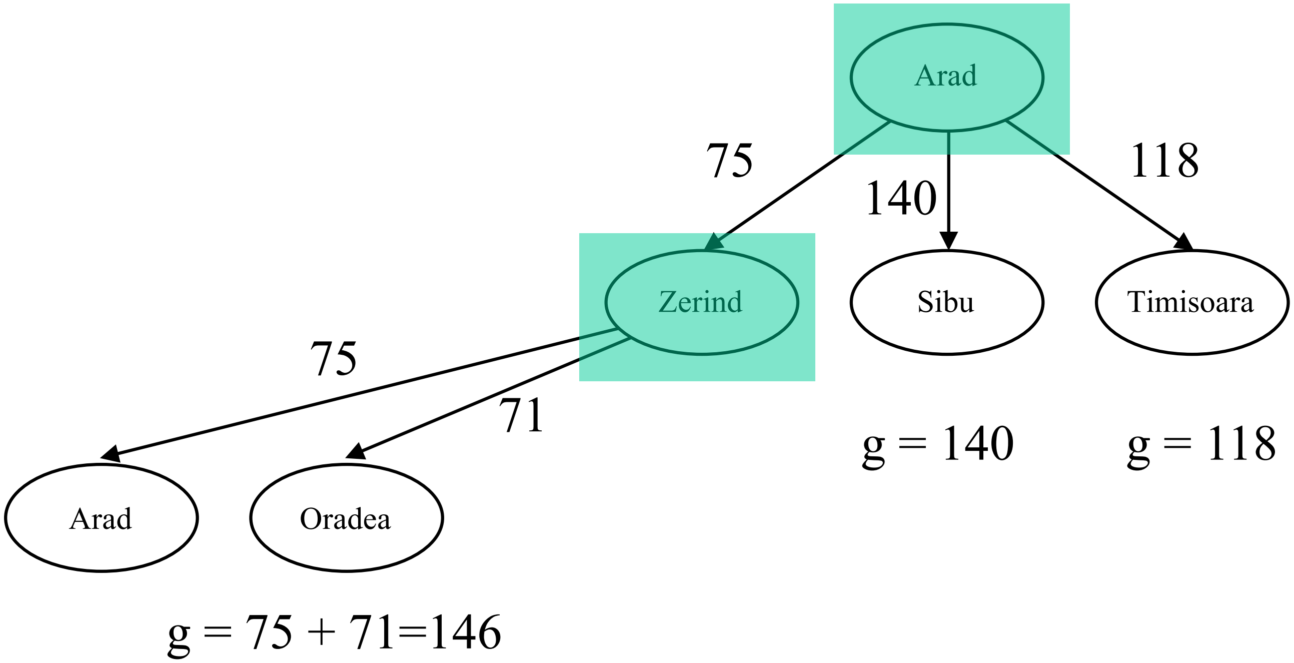
Closed = { }

UCS Example



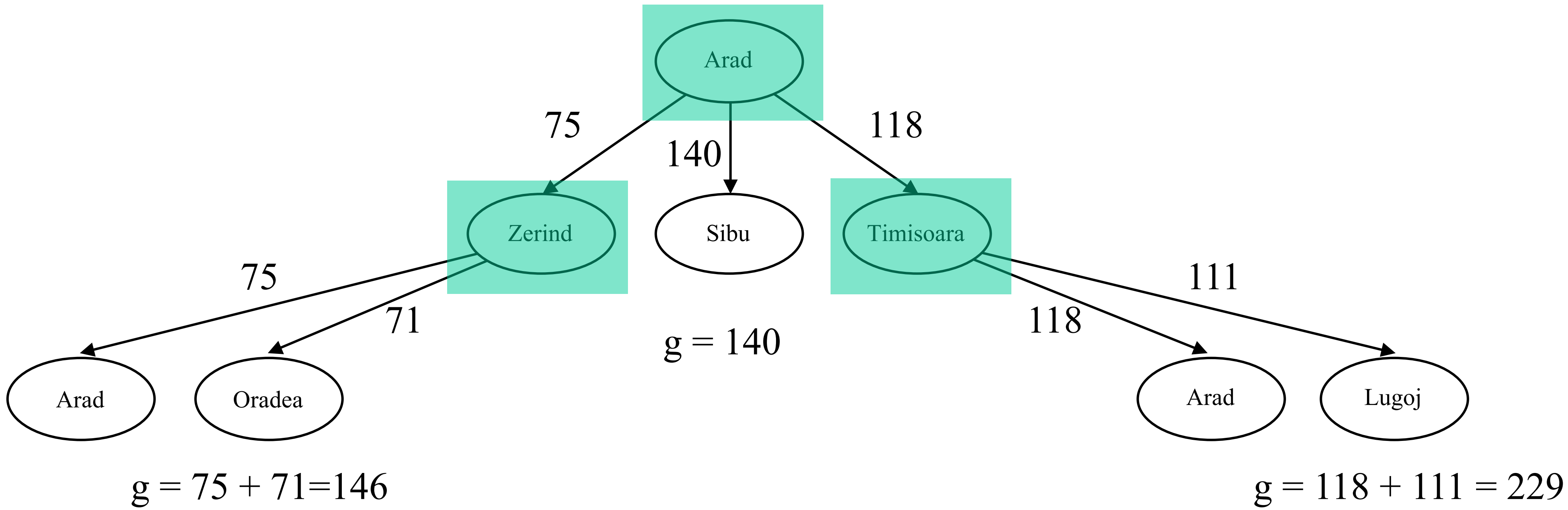
Open = { Zerind(75), Timisoara(118), Sibu(140) }
Closed = { Arad }

UCS Example



Open = { Timisoara(118), Sibu(140), Oradea(146), Arad(150) }
Closed = { Arad, Zerind }

UCS Example



Open = { Sibu(140), Oradea(146), Arad(150), Lugoj(229), Arad(236) }

Closed = { Arad, Zerind, Timisoara }

UCS Properties

- Complete (good)

UCS Properties

- Complete (good)
- Time and space (can be bad)
  - Additional cost of priority queue

UCS Properties

- Complete (good)
- Time and space (can be bad)
  - Additional cost of priority queue
  - Can be much greater than b^d
    - Can explore large subtrees of small steps before exploring large (and perhaps useful) steps

UCS Properties

- Complete (good)
- Time and space (can be bad)
  - Additional cost of priority queue
  - Can be much greater than b^d
    - Can explore large subtrees of small steps before exploring large (and perhaps useful) steps
- Optimal (good)

Depth First Search

- Always expand deepest unexpanded node (on the fringe) first
  - Left-to-right or right-to-left ordering

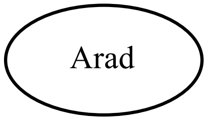
Depth First Search

- Always expand deepest unexpanded node (on the fringe) first
    - Left-to-right or right-to-left ordering
- Only when hit “dead-end” (leaf) does search go back and expand nodes at next shallower level

Depth First Search

- Always expand deepest unexpanded node (on the fringe) first
  - Left-to-right or right-to-left ordering
- Only when hit “dead-end” (leaf) does search go back and expand nodes at next shallower level
- Open set
  - Put new nodes at end, remove from end
  - Stack, last-in first-out (LIFO)

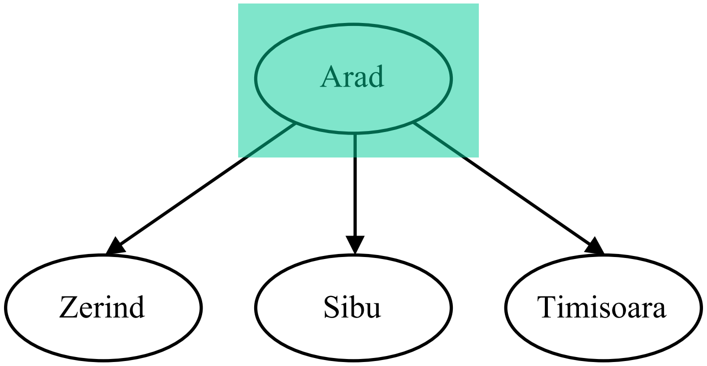
DFS Example



Open = { Arad }

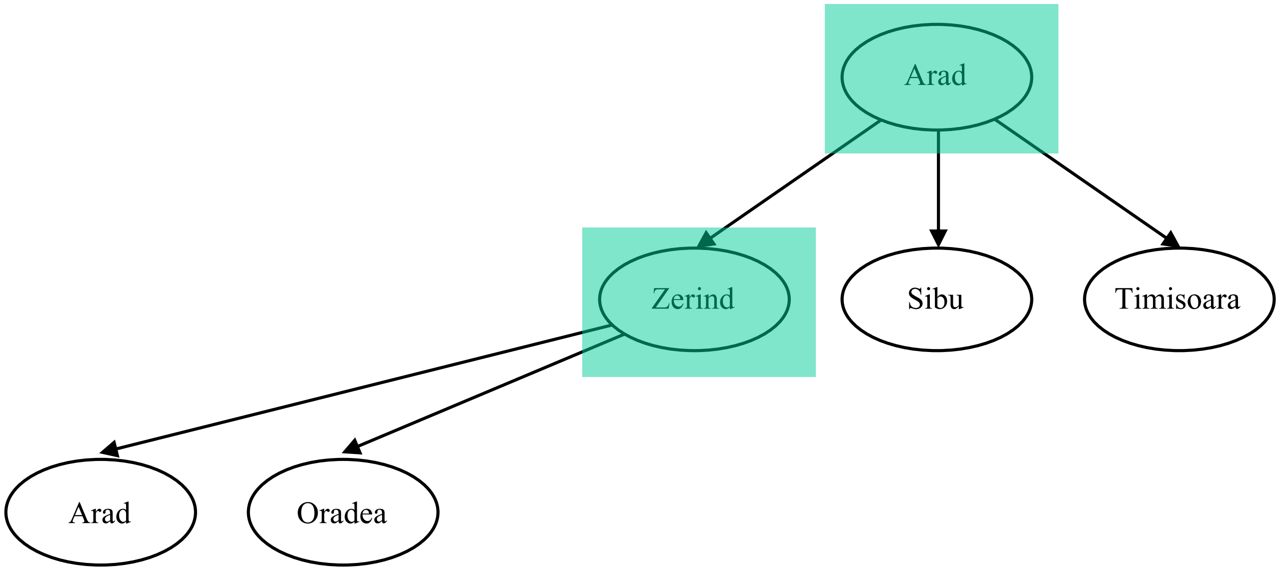
Closed = { }

DFS Example



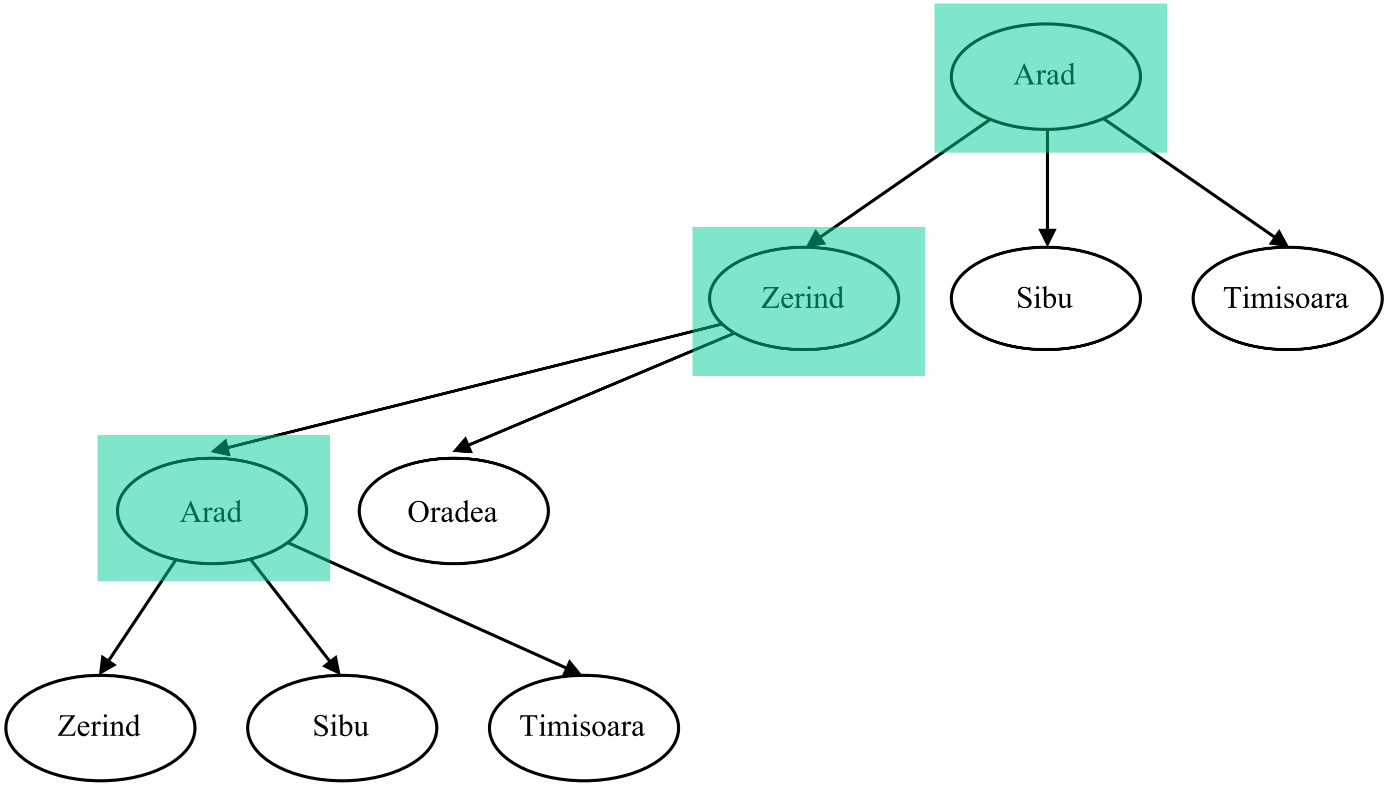
Open = { Timisoara, Sibu, Zerind }
Closed = { Arad }

DFS Example



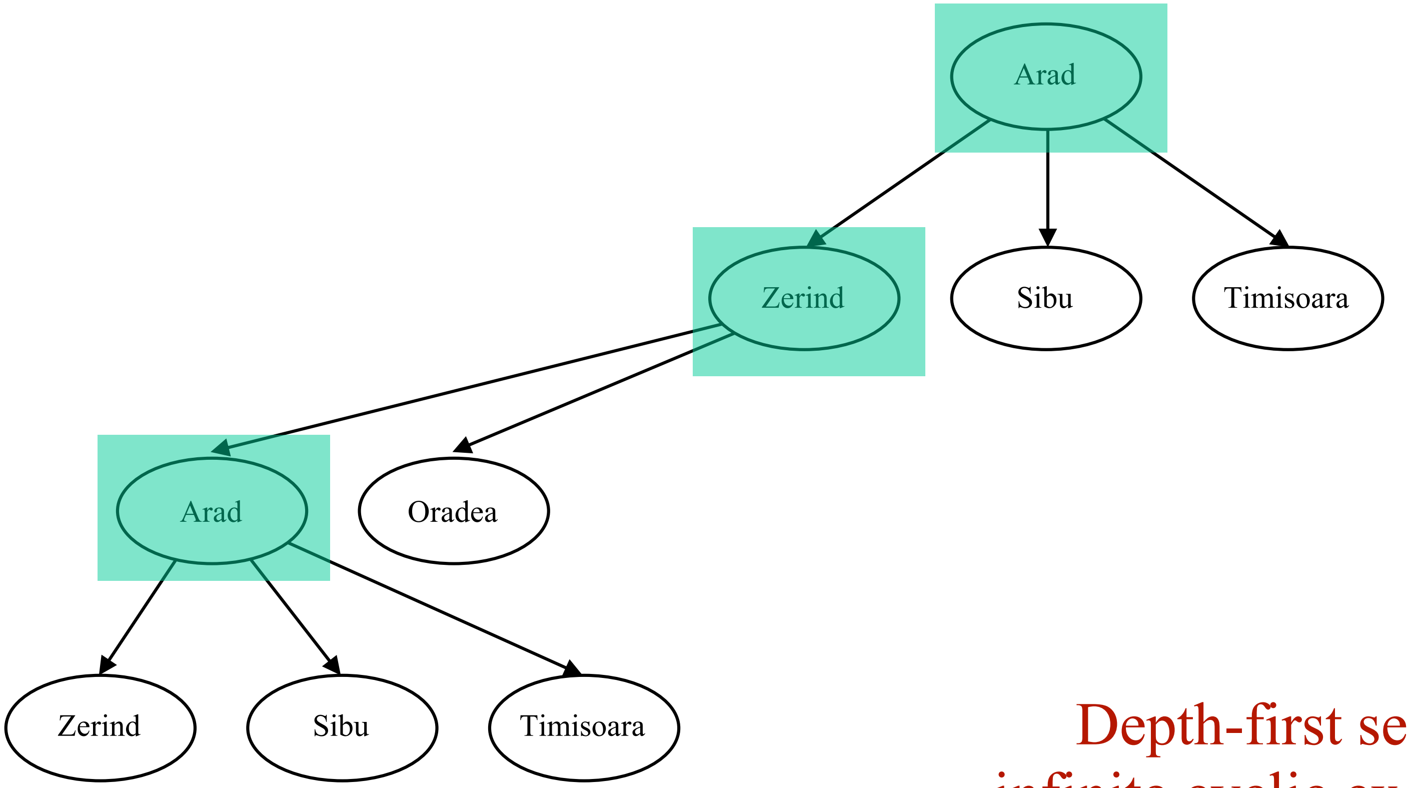
Open = { Timisoara, Sibu, Oradea, Arad }
Closed = { Arad, Zerind }

DFS Example



Open = { Timisoara, Sibu, Oradea, Timisoara, Sibu, Zerind }
Closed = { Arad, Zerind, Arad }

DFS Example



Depth-first search can perform infinite cyclic excursions if not check for repeated-states!

DFS Properties

- Potentially not complete (can be bad)
    - Fails in infinite-depth spaces or with loops

DFS Properties

- Potentially not complete (can be bad)
  - Fails in infinite-depth spaces or with loops
- Time (bad)
  - $O(b^m)$, m=maximum depth
  - Bad if m is larger than depth of goal (d)
  - Good if multiple solutions (hit one)

DFS Properties

- Potentially not complete (can be bad)
  - Fails in infinite-depth spaces or with loops
- Time (bad)
  - $O(b^m)$, m=maximum depth
  - Bad if m is larger than depth of goal (d)
  - Good if multiple solutions (hit one)
- Space (better)
  - $O(mb)$, linear space (keep only leaf nodes "as you go")
    - $<=b$ leaf nodes at each level stored (up to level m)

DFS Properties

- Potentially not complete (can be bad)
  - Fails in infinite-depth spaces or with loops
- Time (bad)
  - *$O(b^m)$, m*=maximum depth
  - Bad if *m* is larger than depth of goal (*d*)
  - Good if multiple solutions (hit one)
- Space (better)
  - *O(mb),* linear space (keep only leaf nodes "as you go")
    - <=*b* leaf nodes at each level stored (up to level *m*)
- Optimality (bad)
  - No, it returns the first deepest solution, so it could miss a shallower solution it has not yet seen (even at low depth)

Depth Limited Search

- Depth-first search with depth limit of *l*

Depth Limited Search

- Depth-first search with depth limit of *l*
- Avoids pitfalls of depth-first search by imposing a cutoff (stop) depth

Depth Limited Search

- Depth-first search with depth limit of *l*
- Avoids pitfalls of depth-first search by imposing a cutoff (stop) depth
- Implementation
  - Depth-first stack with nodes at depth *l* having <u>no</u> successors

Depth Limited Search

- Depth-first search with depth limit of *l*
- Avoids pitfalls of depth-first search by imposing a cutoff (stop) depth
- Implementation
    - Depth-first stack with nodes at depth *l* having <u>no</u> successors
- With 20 cities in the Arad to Bucharest example, know that a solution length < 20 exists

Depth Limited Search

- Depth-first search with depth limit of *l*
- Avoids pitfalls of depth-first search by imposing a cutoff (stop) depth
- Implementation
    - Depth-first stack with nodes at depth *l* having <u>no</u> successors
- With 20 cities in the Arad to Bucharest example, know that a solution length < 20 exists
- Guaranteed to find solution (if exists), but not guaranteed to find shortest solution

Depth Limited Search

- Depth-first search with depth limit of *l*
- Avoids pitfalls of depth-first search by imposing a cutoff (stop) depth
- Implementation
    - Depth-first stack with nodes at depth *l* having <u>no</u> successors
- With 20 cities in the Arad to Bucharest example, know that a solution length < 20 exists
- Guaranteed to find solution (if exists), but not guaranteed to find shortest solution
- Complete (if depth limit big enough), but not optimal

Depth Limited Search

- Depth-first search with depth limit of *l*
- Avoids pitfalls of depth-first search by imposing a cutoff (stop) depth
- Implementation
  - Depth-first stack with nodes at depth *l* having no successors
- With 20 cities in the Arad to Bucharest example, know that a solution length < 20 exists
- Guaranteed to find solution (if exists), but not guaranteed to find shortest solution
- Complete (if depth limit big enough), but not optimal
- Time and space complexity of depth-first search

Iterative Deepening Search

- Sidesteps issue of choosing best depth limit
- Finding good depth limit is difficult for most problems

Iterative Deepening Search

- Sidesteps issue of choosing best depth limit
- Finding good depth limit is difficult for most problems
- Try all possible depth limits
  - First depth 0,
  - then depth 1,
  - then depth 2, …

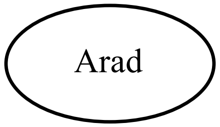
Iterative Deepening Search

- Sidesteps issue of choosing best depth limit
- Finding good depth limit is difficult for most problems
- Try all possible depth limits
  - First depth 0,
  - then depth 1,
  - then depth 2, …
- May seem wasteful, but overhead is not very costly
  - Because most of the nodes are toward <u>bottom</u> of tree
  - Not costly to generate upper nodes multiple times

Iterative Deepening Search

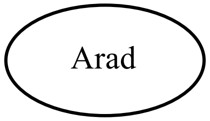
- Sidesteps issue of choosing best depth limit
- Finding good depth limit is difficult for most problems
- Try all possible depth limits
  - First depth 0,
  - then depth 1,
  - then depth 2, …
- May seem wasteful, but overhead is not very costly
  - Because most of the nodes are toward <u>bottom</u> of tree
  - Not costly to generate upper nodes multiple times
- Preferred method with large search space and depth of solution not known
  -

IDS Example [**with l=0]**

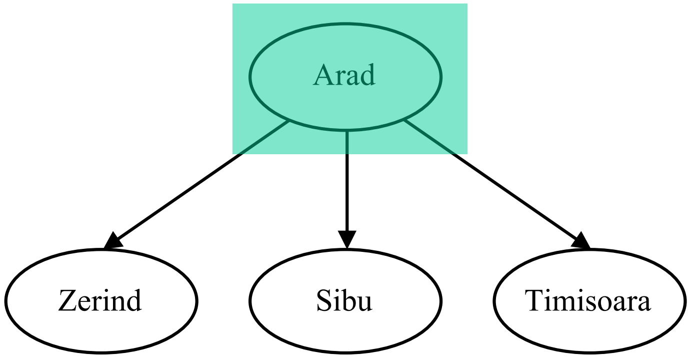


[STOP]

IDS Example [**with l=1**]

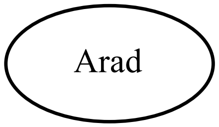


IDS Example [**with l=1**]

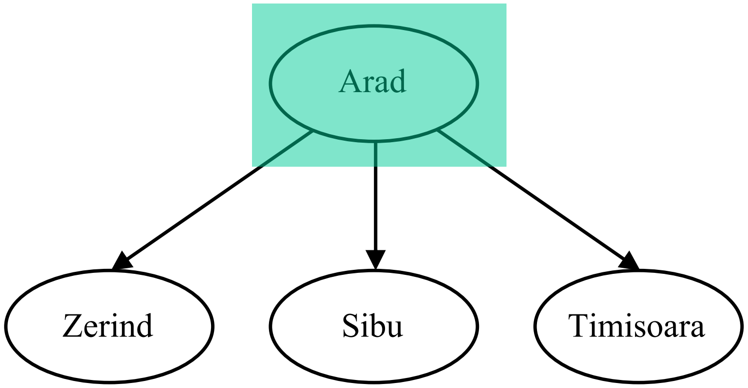


[STOP]

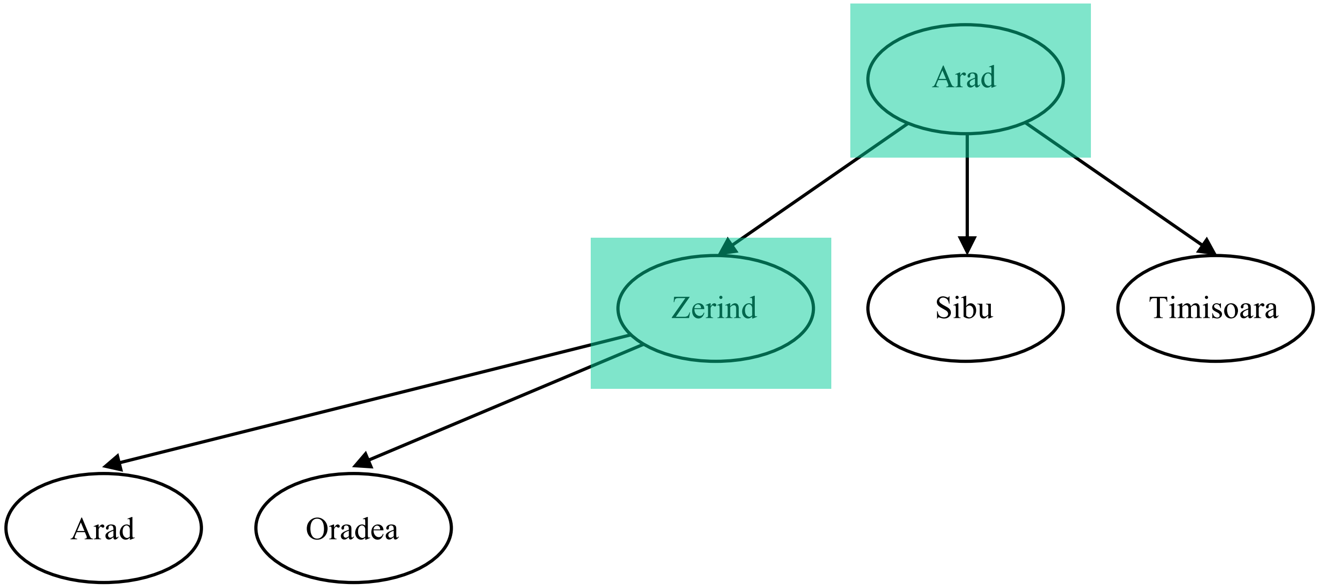
IDS Example [**with l=2**]



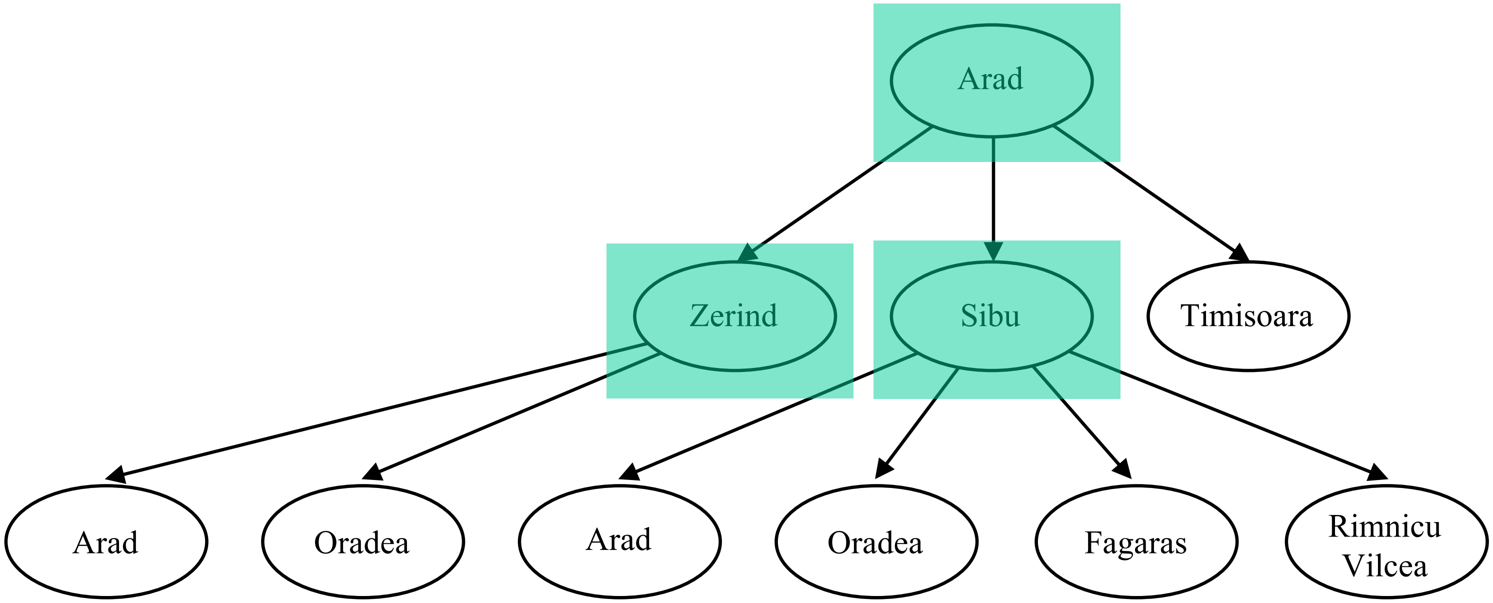
IDS Example [**with l=2**]



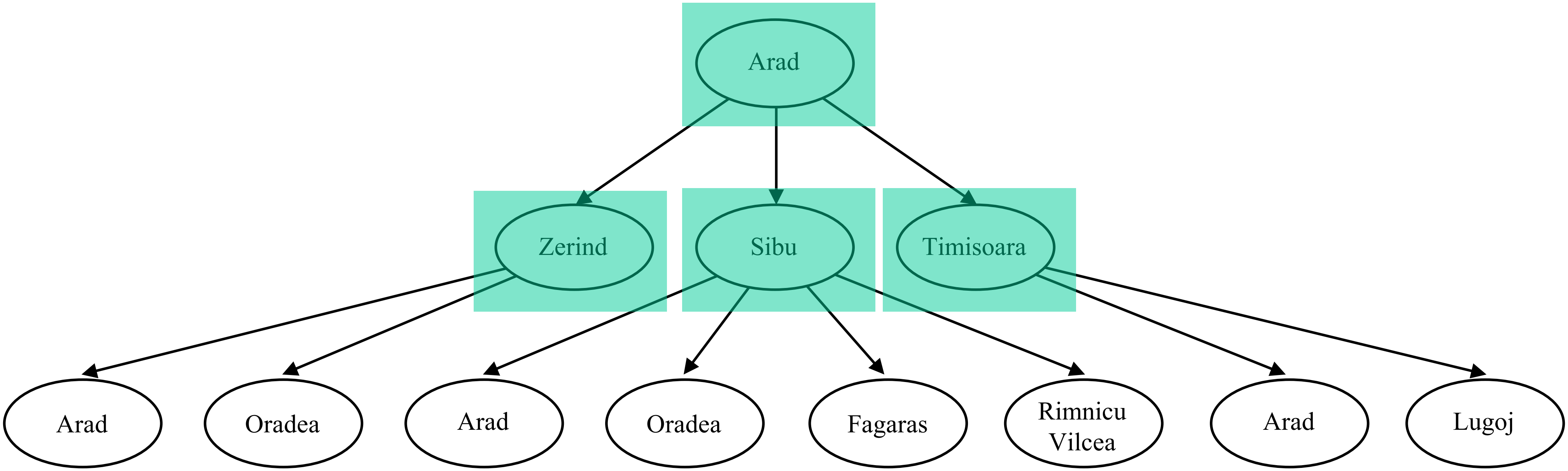
IDS Example [**with l=2**]



IDS Example [**with l=2**]



IDS Example [**with l=2**]



[STOP]

IDS Properties

- Complete (good)

IDS Properties

- Complete (good)
- Time (not too bad)
  - $O(b^d)$
  - where d is depth of shallowest solution

IDS Properties

- Complete (good)
- Time (not too bad)
  - $O(b^d)$
  - where d is depth of shallowest solution
- Breadth-first finds shallowest (good for path cost when is non-decreasing function of the depth)

IDS Properties

- Complete (good)
- Time (not too bad)
  - $O(b^d)$
  - where *d* is depth of shallowest solution
- Breadth-first finds shallowest (good for path cost when is non-decreasing function of the depth)
- Space (good)
  - $O(bd)$ – as in depth-first

IDS Properties

- Complete (good)
- Time (not too bad)
  - $O(b^d)$
  - where *d* is depth of shallowest solution
- Breadth-first finds shallowest (good for path cost when is non-decreasing function of the depth)
- Space (good)
  - $O(bd)$ – as in depth-first
- Optimality (good)
  - Yes (Same as BFS)

IDS Properties

- Complete (good)
- Time (not too bad)
  - $O(b^d)$
  - where *d* is depth of <u>shallowest</u> solution
- Breadth-first finds shallowest (good for path cost when is non-decreasing function of the depth)
- Space (good)
  - *O(bd)* – as in depth-first
- Optimality (good)
  - Yes (Same as BFS)

IDS combines benefits of depth-first and breadth-first search

IDS Properties

- Nodes generated when goal at depth *d*
  - (bottom level; root is depth 0)
  - with branching factor *b*
  - $(d)b + (d\text{-}1)b^2 + \ldots + (1)b^d$ => $O(b^d)$

IDS Properties

- Nodes generated when goal at depth *d*
  - (bottom level; root is depth 0)
  - with branching factor *b*
  - $(d)b + (d\text{-}1)b^2 + \ldots + (1)b^d \quad => \quad O(b^d)$

$\uparrow$

Children of root
generated *d* times

IDS Properties

- Nodes generated when goal at depth *d*
    - (bottom level; root is depth 0)
    - with branching factor *b*
    - $(\underset{\uparrow}{d})b + (d\text{-}1)b^2 + \ldots + \underset{\uparrow}{(1)}b^d \quad => \quad O(b^d)$

Children of root
generated *d* times

Nodes at bottom
generated once

IDS Properties

- Nodes generated when goal at depth *d*
  - (bottom level; root is depth 0)
  - with branching factor *b*
  - $(d)b + (d\text{-}1)b^2 + \ldots + (1)b^d$ => $O(b^d)$

Children of root
generated *d* times

Nodes at bottom
generated once

Note: root node always generated/available

IDS Properties

- Nodes generated when goal at depth *d*
  - (bottom level; root is depth 0)
  - with branching factor *b*
  - $\underbrace{(d)b}_{\text{Children of root generated } d \text{ times}} + (d\text{-}1)b^2 + \ldots + \underbrace{(1)b^d}_{\text{Nodes at bottom generated once}}$ => $O(b^d)$

Note: root node always generated/available

Cost: IDS vs. DLS

b	Worst (Asymp)
2	2x
3	1.5x
4	1.33x

IDS Properties

- Compare to Breadth-first search
  - $b + b^2 + \ldots + b^d + (b^{d+1} - b\ )$ => $O(b^{d+1})$

IDS Properties

- Compare to Breadth-first search
  - $b + b^2 + \ldots + b^d + (b^{d+1} - b) \Rightarrow O(b^{d+1})$

Assume goal is last node (e.g., rightmost) at depth *d*

These are nodes already put in the queue/stack from the non-goal nodes at depth *d* (though not examined later)

Depth-first search does <u>not</u> have these extra nodes

IDS Properties

- Compare to Breadth-first search
  - $b + b^2 + \ldots + b^d + (b^{d+1} - b\ ) \Rightarrow \quad O(b^{d+1})$

Assume goal is last node (e.g., rightmost) at depth *d*

These are nodes already put in the queue/stack from the non-goal nodes at depth *d* (though not examined later)

Depth-first search does not have these extra nodes

- Comparison for $b = 10$ and $d = 5$
  - #nodes(IDS) = 123,450
  - #nodes(BFS) = 1,111,100

When to use IDS

- Preferred uninformed search method when
  - there is a <u>large search space</u> and
  - the <u>depth of the solution is not known</u>.

Notes: Looping

- Loop: Returning to a node previously visited on path
- How to avoid?
  - Keep a list of previously visited nodes → Closed set
- Cost?
  - BFS/UCS: Minimal, Closed set is always smaller than Open
  - DFS: Worsens space cost to similar to BFS

Notes: Exponential Complexity

- So far, describe time/space complexity as exponential in terms of branching factor and depth. E.g., $O(b^d)$
- But if we avoid looping, will only visit each state once!
- In terms of number of states *n*, complexity can be $O(bn)$ or even $O(n)$!!
- For problems with limited number of states, these algorithms are very fast
  - Consider Romania problem, only 20 states

Notes: Recursive DFS

- Possible to implement DFS (and related) as recursive functions
    - Think of child states as “new” initial states
- Reduces space cost to *O(d)*
- Loops
    - Only need to store/check current path (previous recursion levels)
    - No additional space cost
- Recursive implementation is typical

QuestionS