Search Algorithms

CSCE 420 – Fall 2024

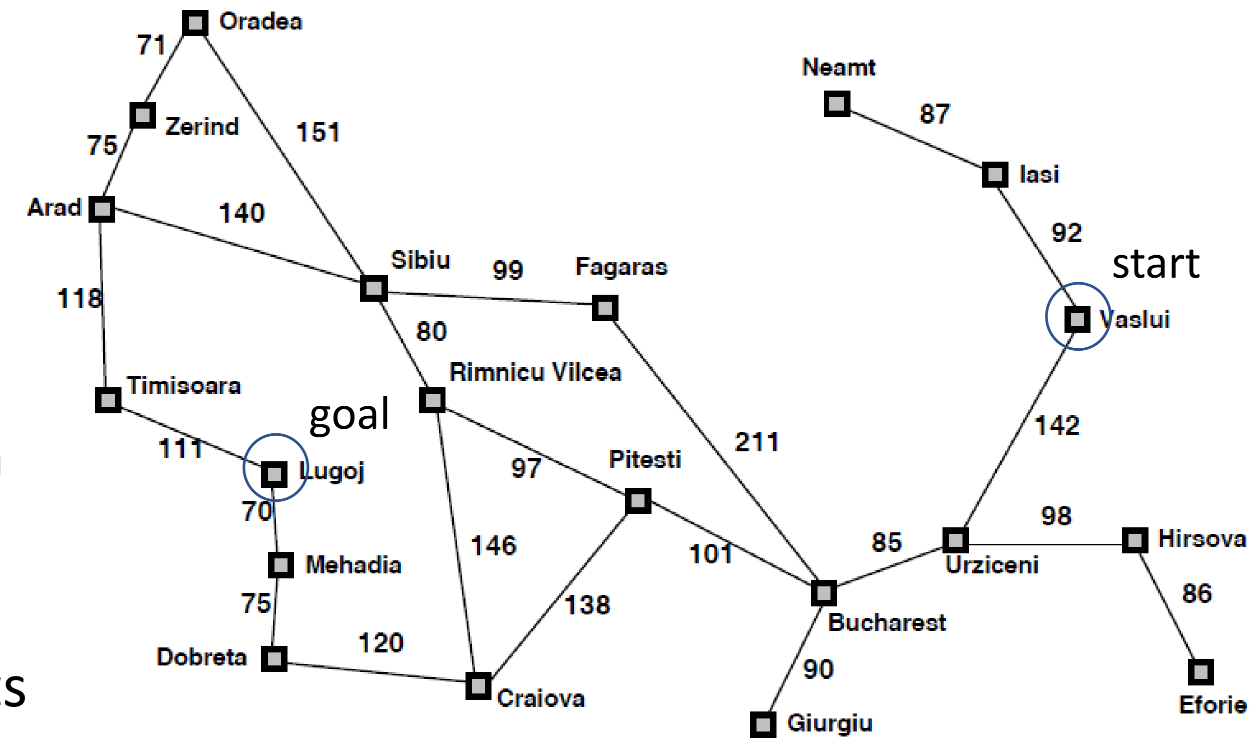
read: Ch. 3

Search as a Model of Problem Solving in AI

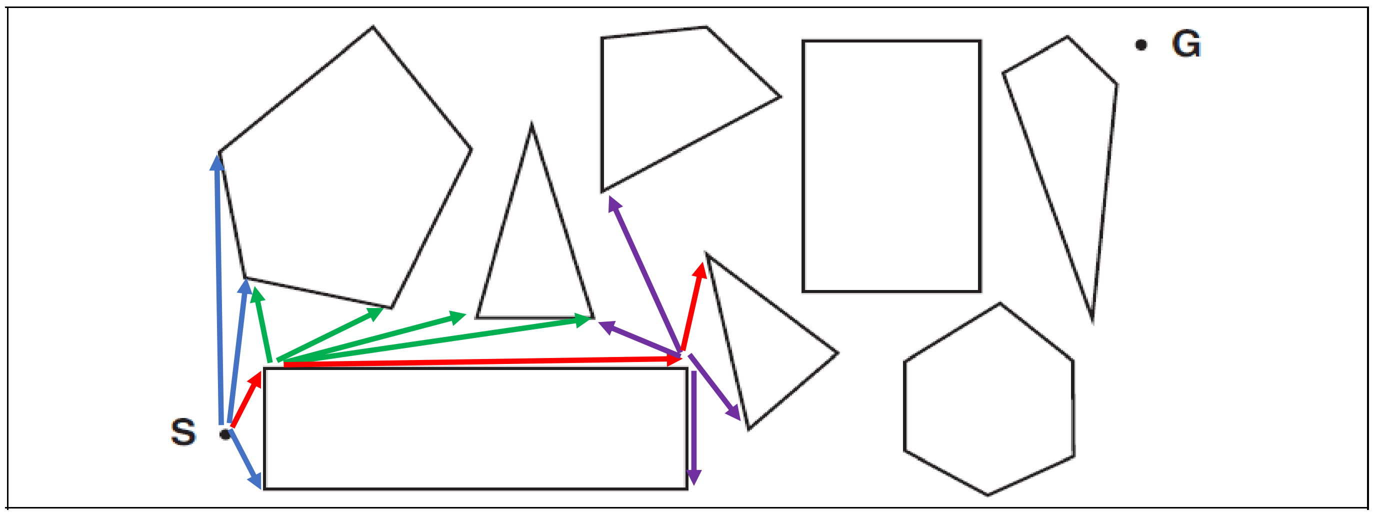
- many AI problems can be formulated as Search
- planning, reasoning, learning...
- define *discrete states* of the world, connected by possible actions
- find a path from the *current state* to a desired *goal state*, producing a *sequence of actions*
- we start by describing generic (un-informed) search algorithms (like DFS)
- then we will extend this to heuristic search algorithms (like A*) which utilize domain knowledge to make the search more efficient

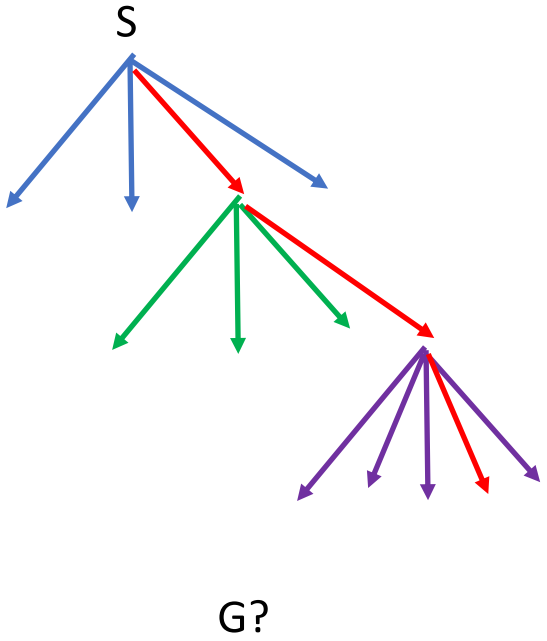
Example: Navigation as Search

- finding a path from an initial location (start) to a desired destination (goal)
- emphasis on discrete moves (city to city, or corner to corner as way-points

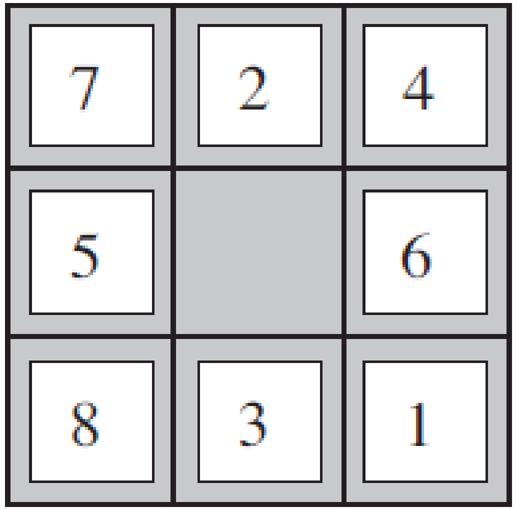


robot moving in workspace with obstacles

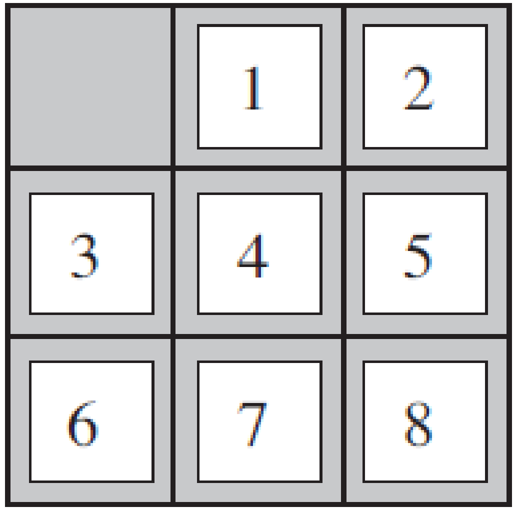




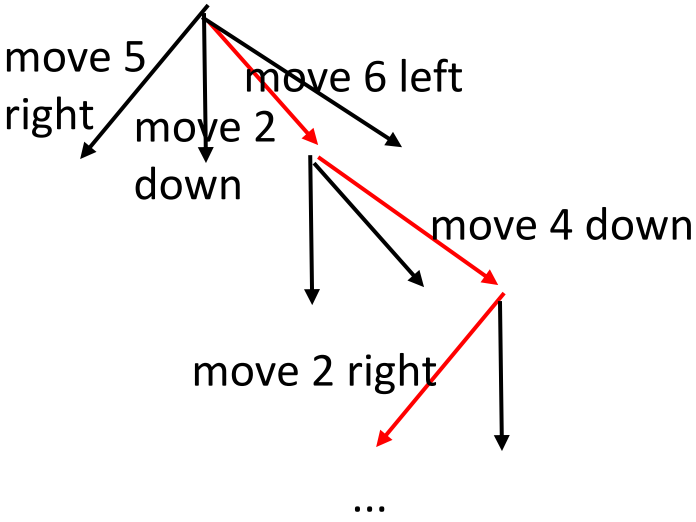
Example: Puzzles as Search



Start State



Goal State



actions = slide a tile up/down/left/right into empty space
a <u>solution path</u> is <u>sequence of actions</u> that transforms start state into the goal

other examples: Rubik's cube, River Crossing Problems, Monkey and Bananas Problem...

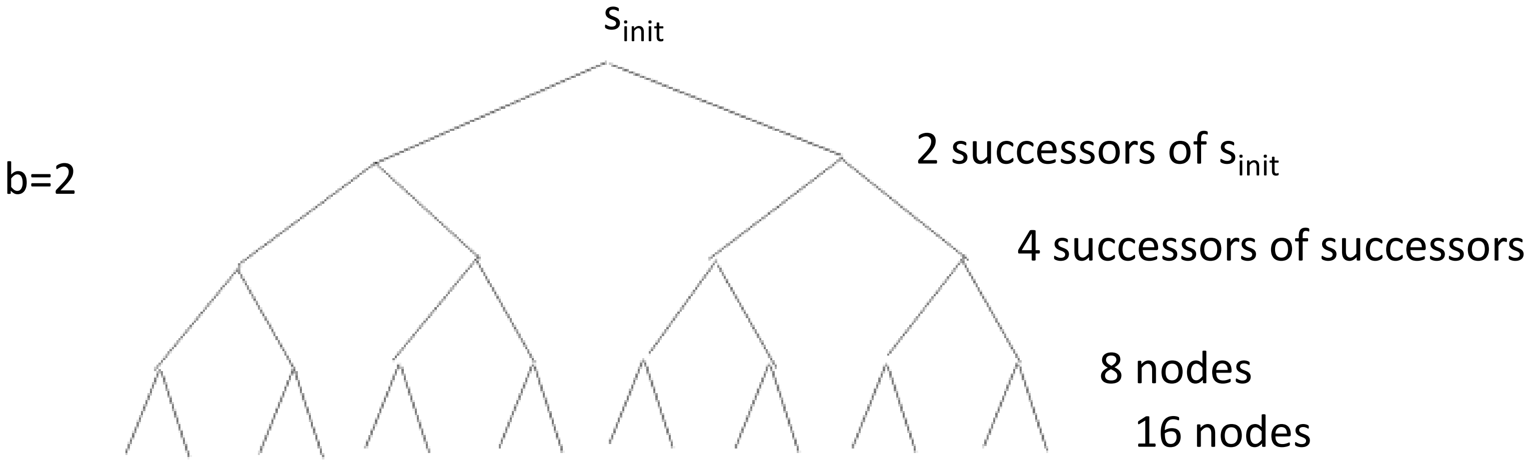
Framework for Formulating Search Problems

- states: a set of discrete representations/configurations of the world
  - this defines the State Space, $S = \{s_1,s_2...\}$
  - could be infinite
- *operator*: a function that generates successor states
  - $S \mid\rightarrow 2^S$ ... mapping from S to powerset of S, i.e. subset of states
  - $\text{oper}(s_i) = \{s_j\} \subset S$
  - this encodes the legal “moves” or “actions” in the space that transform from one state to another (or possibly multiple successors, or none)
  - example: think about moves in tic-tac-toe

$$\text{oper}\left(\begin{array}{|c|c|c|}\hline X & O & O \\ \hline & X & \\ \hline X & & \\ \hline\end{array}\right)=\left\{\begin{array}{|c|c|c|}\hline X & O & O \\ \hline O & X & \\ \hline X & & \\ \hline\end{array}\;,\;\begin{array}{|c|c|c|}\hline X & O & O \\ \hline & X & O \\ \hline X & & \\ \hline\end{array}\;,\;\begin{array}{|c|c|c|}\hline X & O & O \\ \hline & X & \\ \hline X & O & \\ \hline\end{array}\;,\;\begin{array}{|c|c|c|}\hline X & O & O \\ \hline & X & \\ \hline X & & O \\ \hline\end{array}\right\}$$

Search Framework

- the operator, applied recursively to the initial state, s_{init}, generates the State Space (or at least, the reachable part)
- visualize it as a tree (the search tree)
- define *b* as the ‘branching factor’: *average number of successors* for each state
- the size of the tree (nodes on each level) grow exponentially with b



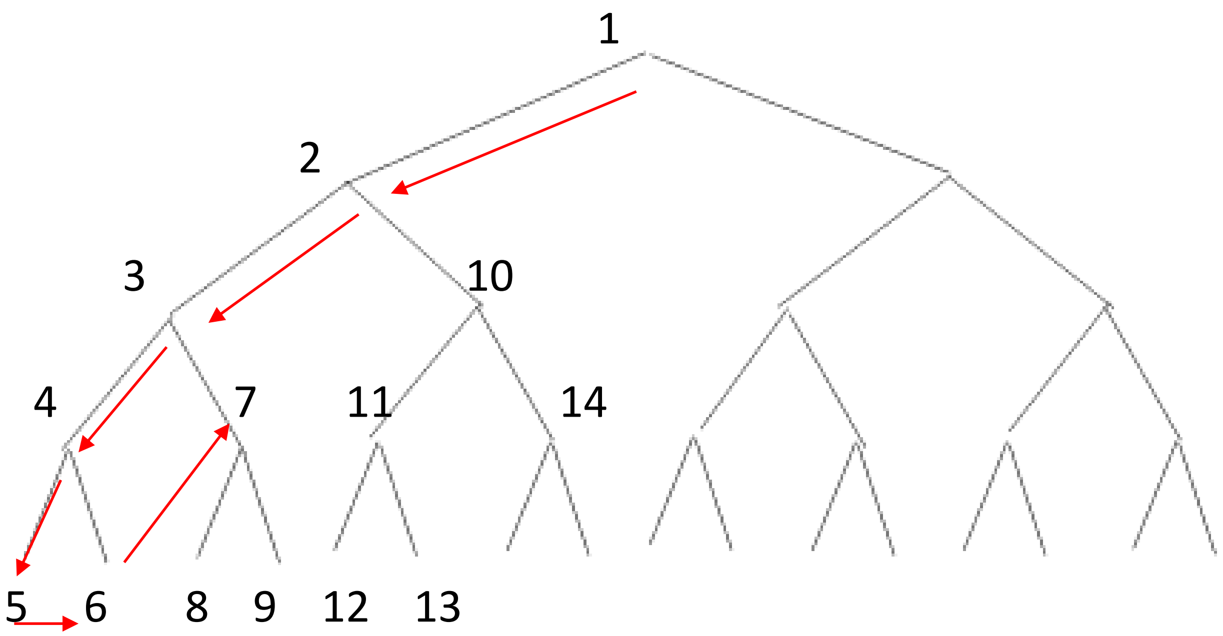
Search Framework

- goals: often specified in a domain-specific way as a set of requirements
  - example: "winning states in tic-tac-toe have 3 X's in a row or column or diagonal"
  - abstractly: we can think of goals as a *subset of states* in the State Space, i.e. $G=\{s_j\} \subset S$
- for many AI problems, we would be happy to find any goal node
  - (doesn't matter which one)
  - we are interested in the path, which is the sequence of actions that transforms the initial state s_{init} into the goal s_{goal}
- in some cases, we might prefer the shortest path (fewest actions required)
- in other cases, if each operator has a different cost, we might be interested in finding the solution with the least path cost
- example: deciding to take a bus instead of a cab as part of a trip in order to minimize cost

$$\text{cost}(s_1..s_n) = \sum_{i=1..n} c(op_i)$$ where s_1=init, s_n=goal, and $s_{i+1} \in op(s_i)$

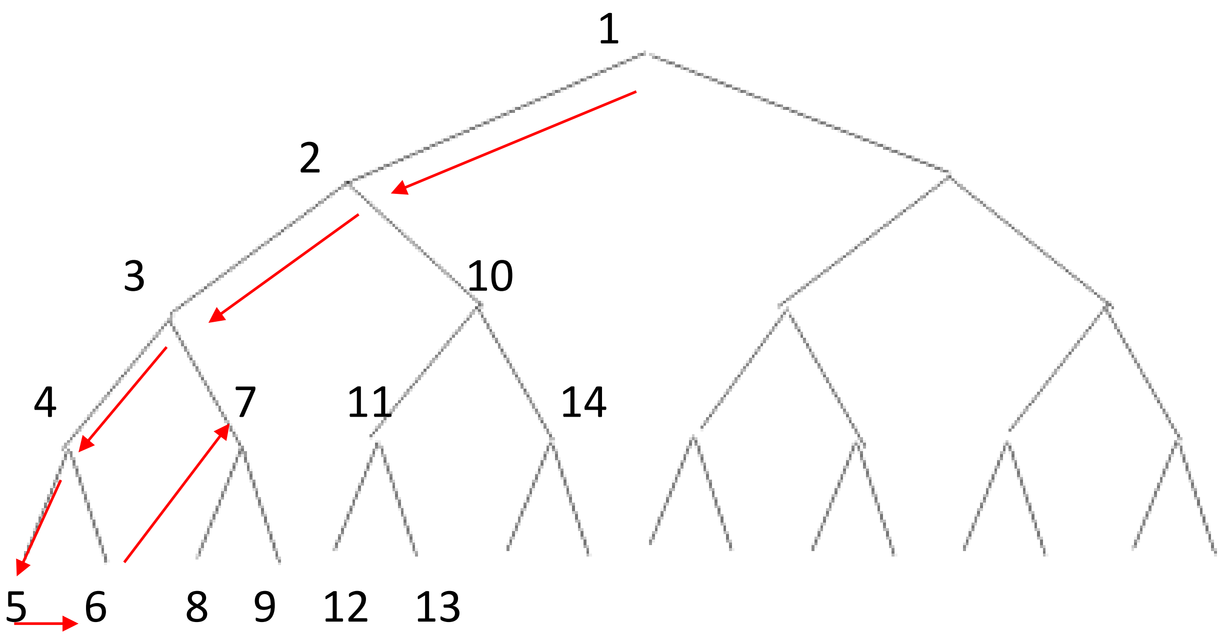
Uninformed Search (‘Weak’ Methods)

- Depth-first Search (DFS) – expand children of children before siblings

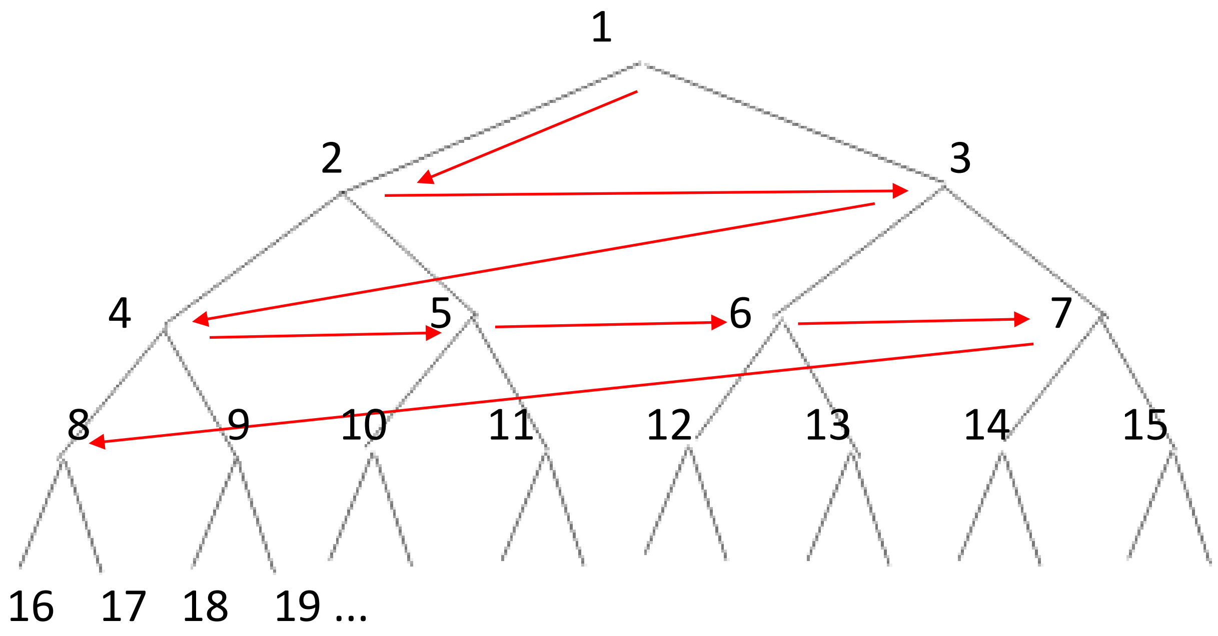


Uninformed Search ('Weak' Methods)

- Depth-first Search (DFS) – expand children of children before siblings

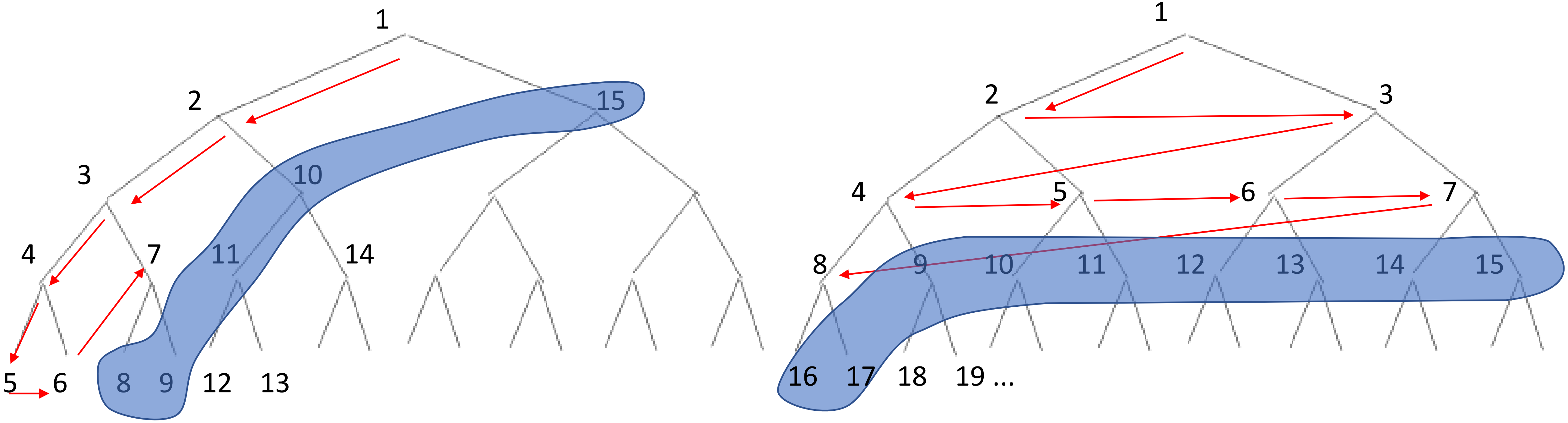


- Breadth-first Search (BFS) – expand children of children AFTER siblings



Uninformed Search (‘Weak’ Methods)

- the ‘frontier’ or ‘agenda’ is the set of nodes that have been expanded but not yet explored, where *expanded* means it is a child of a visited node and *explored* means goal-tested



A Unified Search Algorithm

- although it is easy to write pseudo-code for DFS and BFS separately, they can be unified in an iterative procedure using a data structure to hold the nodes in the frontier
- BFS: frontier = queue (FIFO)
- DFS: frontier = stack (LIFO)

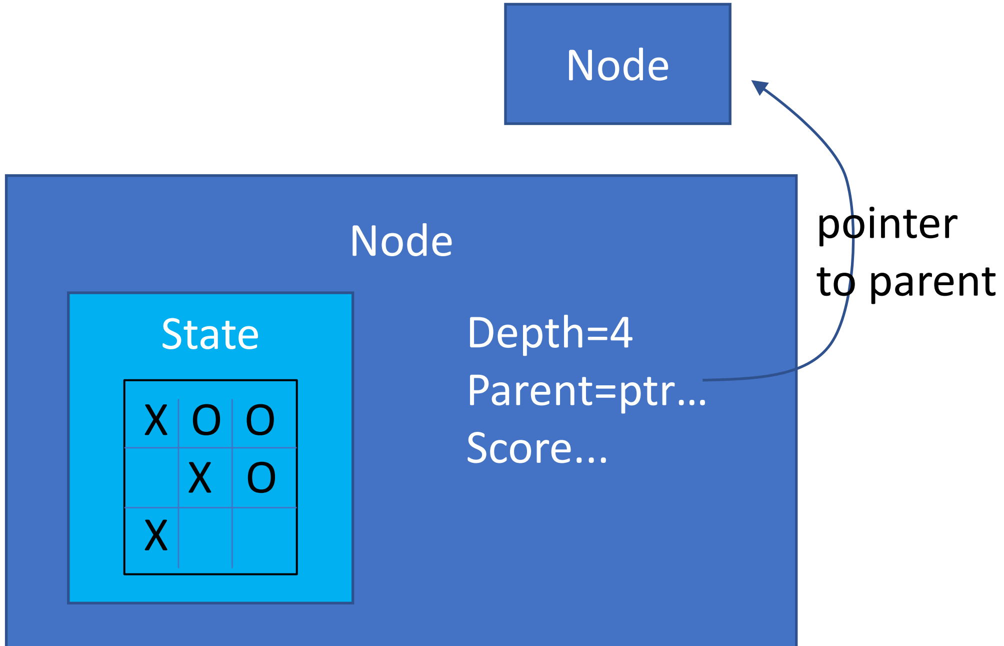
function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution node or *failure*
  node ← NODE(*problem*.INITIAL)
  if *problem*.IS-GOAL(*node*.STATE) **then return** *node*
  frontier ← a FIFO queue, with *node* as an element
  reached ← {*problem*.INITIAL}
  while not IS-EMPTY(*frontier*) **do**
    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*

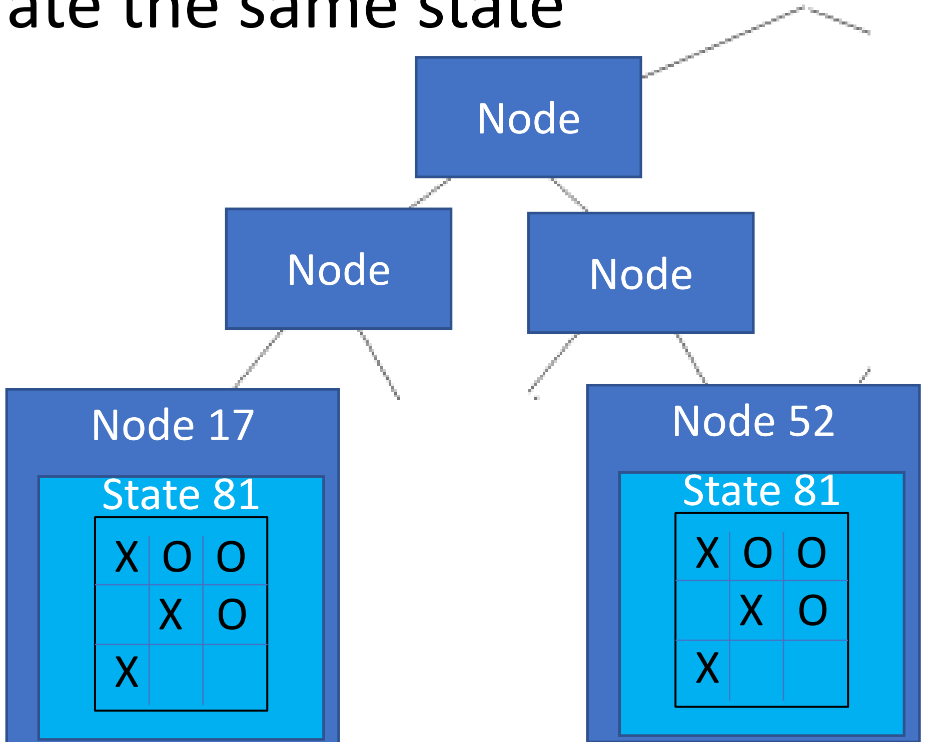
(ignore 'reached' for now;
It is for GraphSearch,
see slides below)

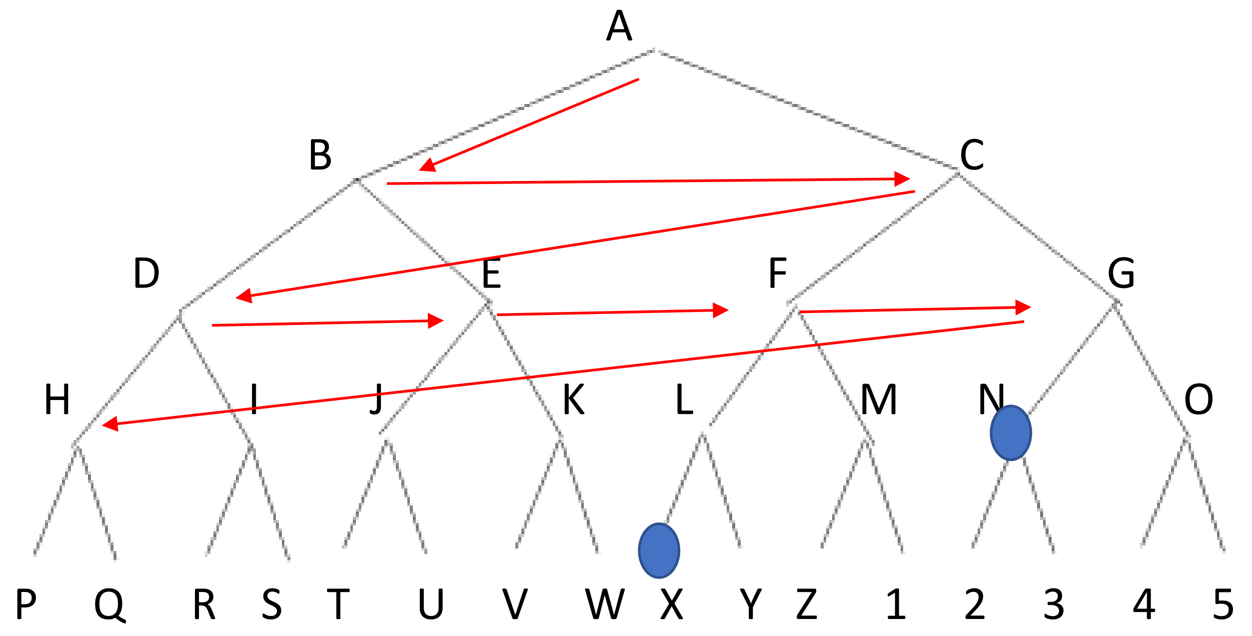
Search Framework

- <u>nodes</u> in the search tree represent <u>states</u> in the state space
- however, they are not quite the same
- a *node* represents a particular path (sequences of actions) to a *state*
- there might be multiple paths that generate the same state

See examples on Navigation slide







- frontier (queue) for BFS:
  - A // [front | A | end]
  - B C // pop A, push children on end
  - // pop B from front
  - // push children D E on end
  - C D E ←
  - D E F G
  - E F G H I // start adding next level
  - F G H I J K
  - G H I J K L M
  - …

```
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
```

to change it to do DFS, all you have to do is replace the frontier with a stack (LIFO):

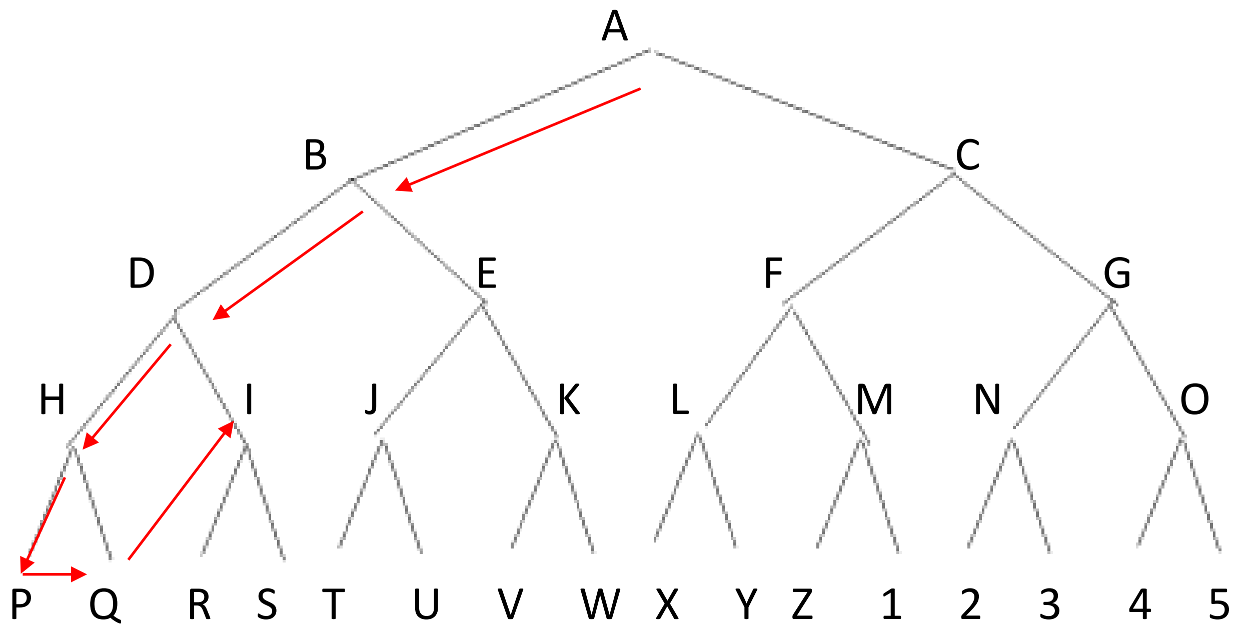
frontier ← **stack,** initialized with start node as first element

function Depth-First Search (*problem*) **returns** a solution node or *failure*

```
node ← NODE(problem.INITIAL)
if problem.IS-GOAL(node.STATE) then return node
frontier ← a LIFO 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
```

to change it to do DFS, all you have to do is replace the frontier with a stack (LIFO): i.e.

frontier ← **stack,** initialized with start node as first element



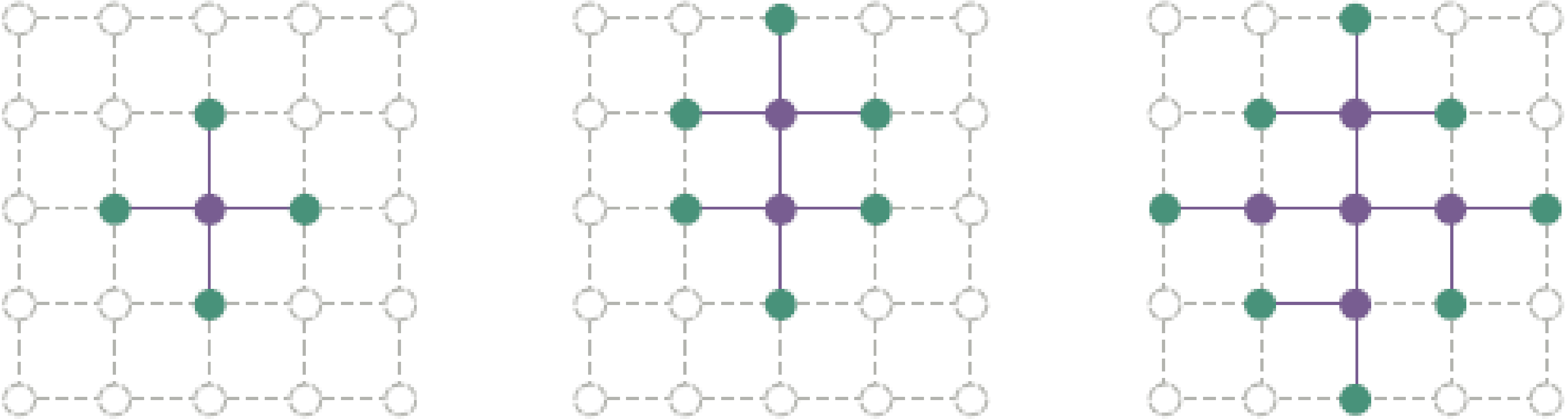
- frontier (stack) for DFS:
  - A
  - // pop A, push children B and C
  - B C
  - // pop B, push D and E on front
  - D E C
  - H I E C // pop D, push H and I
  - P Q I E C // pop H, push P and Q
  - Q I E C // pop P
  - I E C // pop Q
  - R S E C // go to I, push R and S
  - ....

note: when you expand a node, the order in which you push the children makes a difference
In this example, I am pushing the children in *reverse* order, e.g. C before B (as children of A)
what would the search order look like if we pushed the children in alphabetical order?

Graph Search

- in some Search Trees, there are multiple paths to the same state
- example: reversible operators (move, then move back); or think of a map; or think of circular moves in the tile puzzle
- detecting repeated (visited) states can greatly reduce redundancy in the search space
  - if you have already explored children beneath node n, there is no need to do it again
- exception: if you find a shorter/cheaper path to n, you might want to keep track of the best such path found
- ‘reached’: you need a data structure (like a hash table) to keep track of these states

- Graph Search
  - in BFS on a grid, how badly would the size of the search tree scale up if we didn't keep track of reached states?
  - Assume each node has 4 neighbors, so b=4 (worst case) (or b_{avg}=~3)
  - level 0=1 node (initial state, at the center)
  - level 1=4 nodes
  - level 2=16 nodes
  - level 3=64 nodes
  - level 4=256 nodes
  - ...
  - level i: 4^i nodes



  - and yet, there are only 25 distinct states in this space!

Graph Search (=BFS+checking for visited states)

```
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
```

reached is a data structure (e.g. hash table) for keeping track of expanded states to avoid repeating the search

note: that we check reached *before* putting nodes into the frontier, not as we pull them out

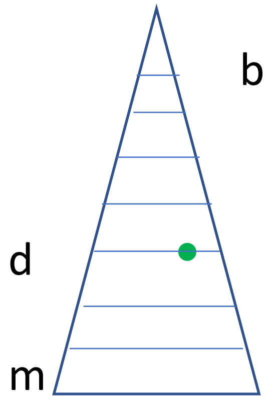
*If s *has* been reached before, you might want to see if a shorter/cheaper path has been discovered and keep track of that…*

- So when should you use DFS, and when should you use BFS?
- On what types of problems would DFS be better, or BFS?
- It depends on properties of the search space...

Computational Complexity

- analysis of computational properties for comparison of DFS and BFS
- **time-complexity**: number of nodes goal-tested (# of loop iterations)
- **space-complexity**: maximum size to which the frontier grows
- **completeness**: if a goal exists, does ALGO guarantee to find it?
- **optimality**: does ALGO guarantee to find the goal node with the minimum path cost?

Computational Complexity of BFS



- time-complexity: number of nodes goal-tested (# of loop iterations)
  - if the shallowest node occurs at depth d, and branching factor is b,
  - then nodes checked (worst case) will be all levels up to and including b
  - $1+b+b^2+....b^d$ = $O(b^{d+1})$
- space-complexity: maximum size to which the frontier grows
  - in worst case, have to store all children at level below goal, $O(b^{d+1})$
- completeness: if a goal exists, does ALGO guarantee to find it?
  - yes (because every goal exists at a finite depth, and BFS explores each level)
- optimality: does ALGO guarantee to find the goal node with the minimum path cost?
  - yes (assuming all operator have equal cost) (but no, if unequal oper costs)
  - in this case, the goal with least path cost is shallowest, and BFS will find it first, because it explores level-by-level)

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=0}^{n} b^i = \left(\frac{b^{n+1} - 1}{b - 1}\right)$$

Computational Complexity of DFS

- time-complexity: number of nodes goal-tested (# of loop iterations)
  - if the maximum depth of the tree is m,
  - the worst case is when goal at depth d is on the right-most branch
  - the nodes checked will be almost all in the tree (even deeper than d): $O(b^m)$
- space-complexity: maximum size to which the frontier grows
  - each time we expand a node, we pop 1 and push b children, (b-1)m = $O(bm)$
- completeness: if a goal exists, does ALGO guarantee to find it?
  - no, in general (i.e. if any branch has infinite depth)
  - yes, only in finite search spaces
- optimality: does ALGO guarantee to find the goal node with the minimum path cost?
  - no (since it is not complete)

Comparison of BFS and DFS

- so which is better? when would we prefer to use one over the other?
- although time-complexity could be exponentially worse for DFS ($O(b^m) \gg O(b^d)$), DFS has <u>linear space-complexity</u>
- in practice, the size of the frontier is what limits AI search
- given modern CPU clock cycles, I can easily search a billion (10^9) nodes (10 μs per loop iteration=17 min), but storing a billion nodes takes too much memory (~100 bytes per node=100 Gb)

	BFS	DFS
time-complexity	$O(b^{d+1})$	$O(b^m)$
space-complexity	$O(b^{d+1})$	$O(bm)$

Iterative Deepening

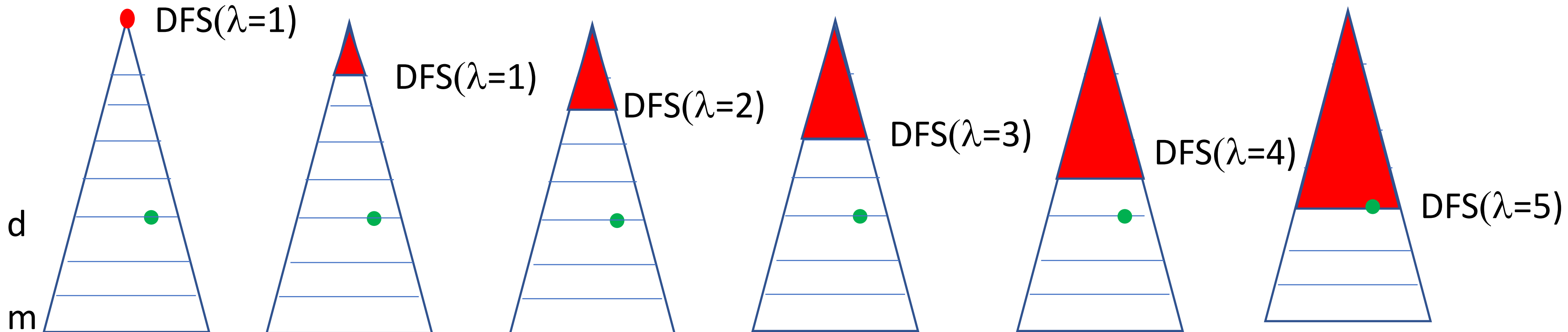
- Is there a way to get the benefits of both BFS and DFS?
- how can we maintain a *linear* frontier size like DFS while still searching level-by-level like BFS?
- how can you maintain the linear space-complexity of DFS while avoiding descending infinitely deep down any single branch?
- answer: depth-limited search
  - do DFS down to depth=1
  - if goal not found, do DFS down to depth=2
  - if goal not found, do DFS down to depth=3
  - ...

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

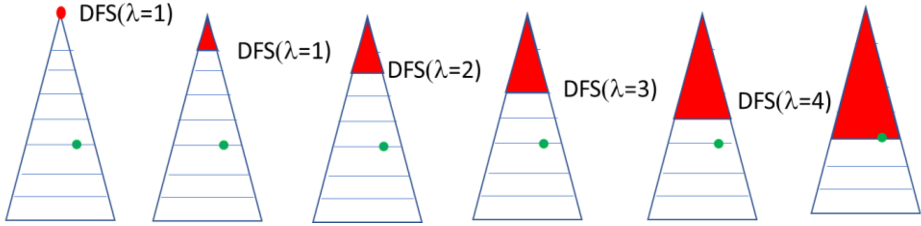
Iterative Deepening

- Complexity analysis:
- since using DFS, the frontier should never get bigger than (b-1)d, hence O(bd)
- and it should be complete and optimal (for equal operator costs)
- what about time complexity?
  - it seems wasteful because you have to re-generate the top part of the search tree each iteration



Iterative Deepening

- time complexity?
  - $1+(1+b)+(1+b+b^2)+(1+b+b^2+b^3)+\ldots+(1+b+\ldots+b^d)$
  - $\leq (1+b+\ldots+b^d)+(1+b+\ldots+b^d)+\ldots\ (1+b+\ldots+b^d)$
  - $\leq d(1+b+\ldots+b^d) \leq d\Sigma b^i \leq d(b^{d+1}-1)/(b-1) = O(b^{d+1})$



- it seems wasteful because you have to re-generate the top part of the search tree each iteration
- why not just "save" the part of the tree generated so far?
- because it will grow exponentially as depth limit increases, negating the benefit of the linear size of the frontier – you have to throw them away
- so it is a <u>tradeoff</u>: you spend a little more time computing (expanding nodes), but you save memory (linear frontier size)

Uniform Cost Algorithm

- suppose we want to find the goal node with the least path cost, when operators have different costs?
- the shortest path (number of actions) is not necessarily the cheapest path (sum of operator costs)
- in this case, BFS is *not* optimal
- however, we can use the same iterative search algorithm, but change the frontier to a *priority queue*
- keep the expanded-but-unexplored nodes sorted in order of increasing path cost
- nodes must keep track of cost; update when generating successors:
  - cost(child) = cost(parent)+cost(op_i)

Uniform Cost Algorithm

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*

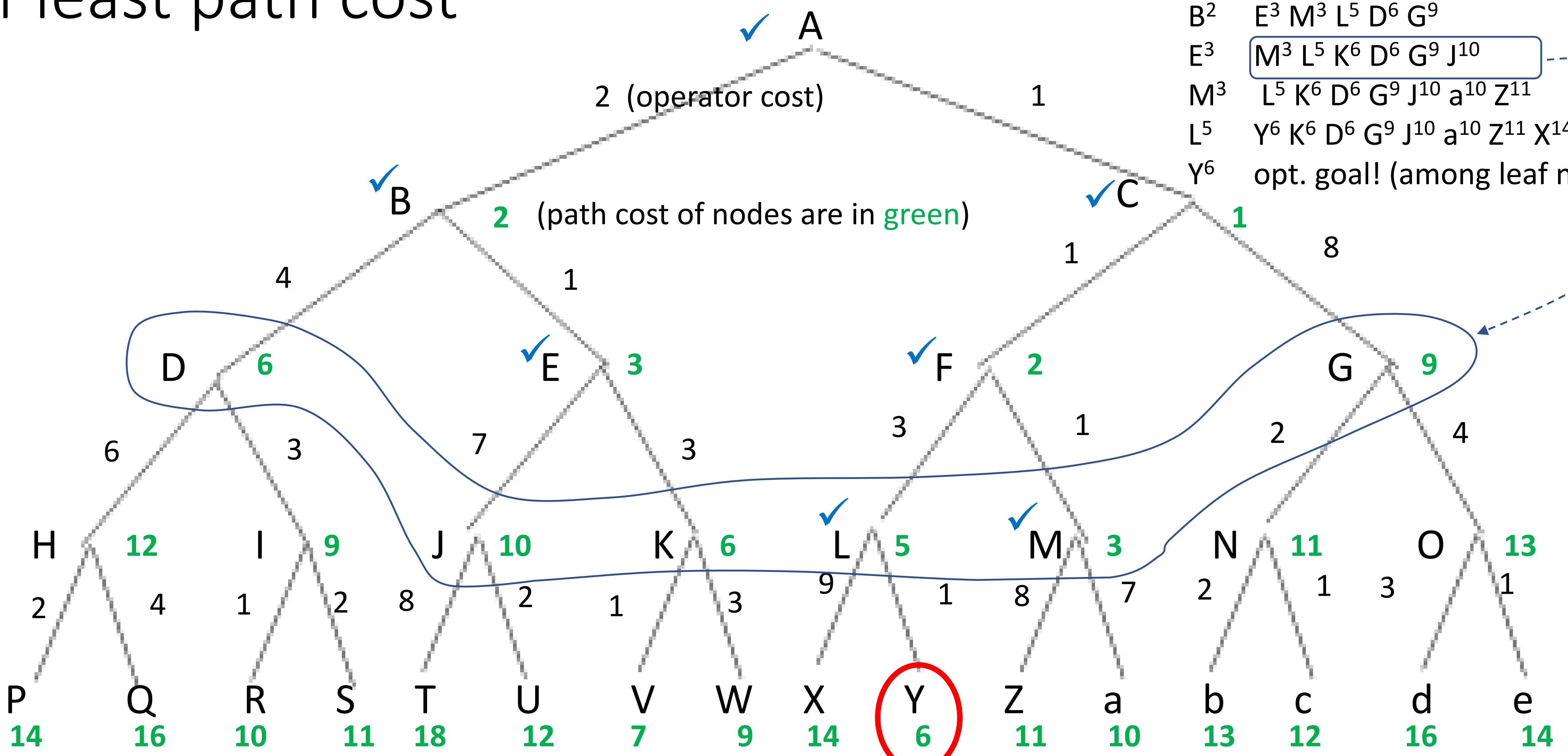
function EXPAND(*problem*, *node*) **yields** nodes
  s ← *node*.STATE
  for each *action* **in** *problem*.ACTIONS(*s*) **do**
    s′ ← *problem*.RESULT(*s*, *action*)
    cost ← *node*.PATH-COST + *problem*.ACTION-COST(*s*, *action*, *s′*)
    yield NODE(STATE=*s′*, PARENT=*node*, ACTION=*action*, PATH-COST=*cost*)

(Note the **'late' goal-test**, which is less efficient than the **'early'** goal test used in **BFS**), but is necessary because we are interested in searching nodes with the lowest path cost first. If there are multiple paths to a node N, put them both in the queue at the same time, and pick whichever has the lowest distance from root.)

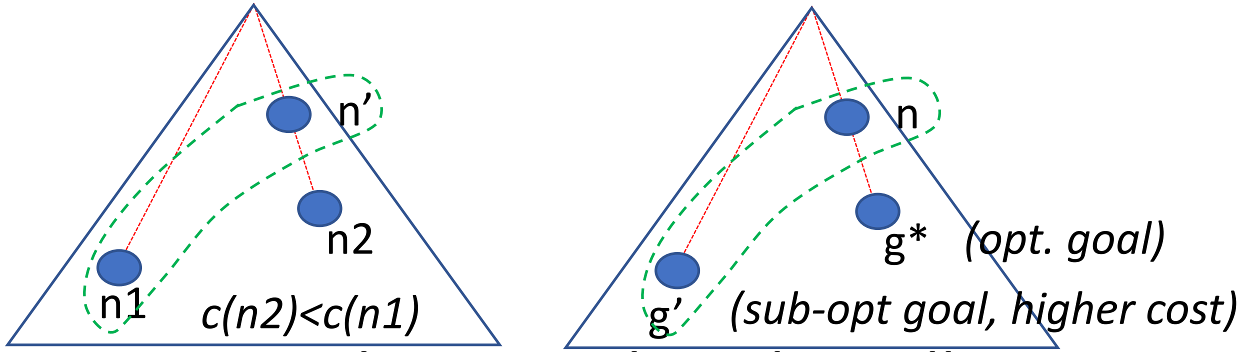
Trace of UC - visits nodes in order of least path cost

at	queue
A^0	C^1 B^2
C^1	F^2 B^2 G^9
F^2	B^2 M^3 L^5 G^9
B^2	E^3 M^3 L^5 D^6 G^9
E^3	$\boxed{M^3\ L^5\ K^6\ D^6\ G^9\ J^{10}}$
M^3	L^5 K^6 D^6 G^9 J^{10} a^{10} Z^{11}
L^5	Y^6 K^6 D^6 G^9 J^{10} a^{10} Z^{11} X^{14}
Y^6	opt. goal! (among leaf nodes)

numbers are path costs



Uniform Cost Algorithm



- sure, every node you pull out of the priority queue has costs less than all other in the priority queue
  - but when you reach a goal, how do you know there is not another cheaper goal out there?
- assumption: all operators have positive costs: $cost(op_i)>0 \geq \varepsilon > 0$
  - therefore, cost of nodes along a path increases monotonically
- Lemma: UC explores nodes in order of increasing total path cost
  - Let pathcost(n1)>pathcost(n2), but suppose n1 is visited first (for sake of contradiction)
  - n2 might not be in the priority queue at same time n1 is popped
  - but there is always some node n' on the path to n2 that is in the priority queue (even it is the initial state/root node), and pathcost(n')<pathcost(n2) since monotonic
  - if n' was in queue when n1 was, then n' would have been popped before n1, because pathcost(n')<pathcost(n2)<pathcost(n1)
- Corollary: when the first node that is a goal, g*, is visited, it has lower cost than any other goal node g', pathcost(g*)≤pathcost(g'), hence g* is optimal

Uniform Cost Algorithm

- Computational properties of UC
  - time-complexity: $O(b^{(1+C^*/\varepsilon)})$
  - where C* is the total path cost of the cheapest solution
  - why? because each step costs at least ε, so goal occurs at depth C^*/ε in the worst case
  - space-complexity: $O(b^{(1+C^*/\varepsilon)})$
  - completeness: yes
  - optimality: yes!

Uniform Cost Algorithm

- comparison to Djikstra's Algorithm
  - UC and Djikstra both solve the single-source shortest-path problem
  - however, an important difference is that Djikstra is based on Dynamic Programming (DP)
  - it uses a data structure (array) to maintain partial path distances from the source to *all vertices* V in the graph
  - you can't do this for most AI problems, especially if they have exponentially large or infinite State Spaces

// from https://en.wikipedia.org/wiki/Dijkstra's_algorithm

```
function Dijkstra(Graph, source):
   create vertex set Q

   for each vertex v in Graph:
      dist[v] ← INFINITY
      prev[v] ← UNDEFINED
      add v to Q
   dist[source] ← 0

   while Q is not empty:
      u ← vertex in Q with min dist[u]
      remove u from Q

      for each neighbor v of u:
         alt ← dist[u] + length(u, v)
         if alt < dist[v]:
            dist[v] ← alt
            prev[v] ← u

   return dist[], prev[]
```

Summary of Computational Properties of Search Algorithms

read for yourself

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

or $O(b^{d+1})$

if $\text{cost}(op_i)$=constant for all operators

except for finite search spaces

or $O(b^{d+1})$

Heuristic Search

- since AI search problems usually have exponential search spaces, the main focus is on how we can exploit *domain knowledge* to improve the efficiency of the search
- *domain knowledge* refers to anything we know about solving these types of problems
  - rules of thumb, common solutions, way to decompose the problem into subgoals, useful sequences of actions, interactions/dependencies between operators...
- in this context, domain knowledge will be encapsulated in a *heuristic function*, h(n)
- it is a 'scoring' function that maps every node (or state) to a real number
- the advantage is using any knowledge we have to *guide* the search toward the goal, and avoid searching 'unproductive' parts the search space

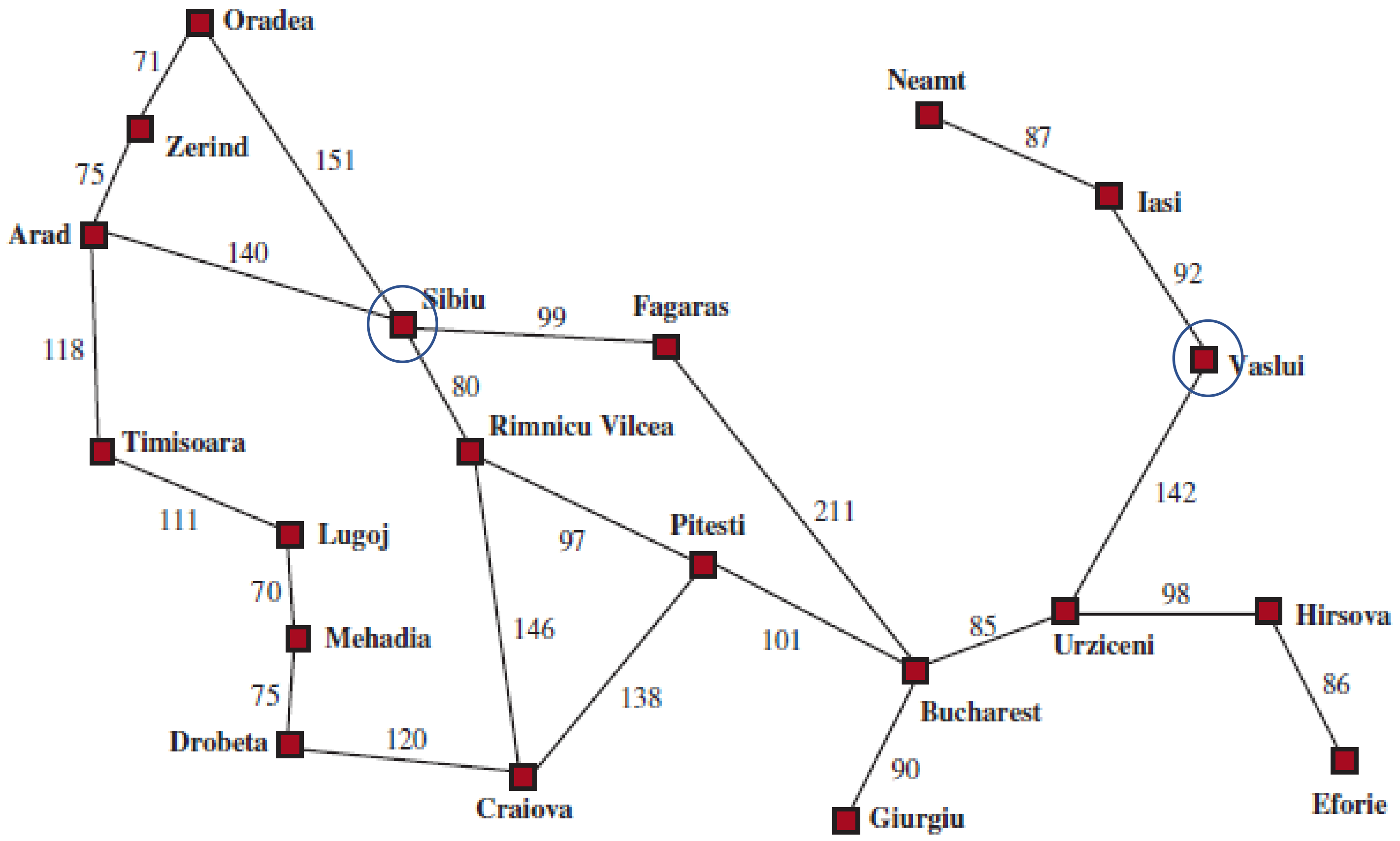
Heuristic Search

- a heuristic function h(n) is an *estimate of the distance (path cost) remaining from n to the closest goal*
- hence it is a mapping from S ⊦> R (State Space to real numbers)
- generally, h(n)≥0, and h(n)=0 for goals
- abstractly, it is a quantification of how close a state is to being solved (higher is farther away)

Heuristic Functions

- Example 1: h_{SLD} for navigation

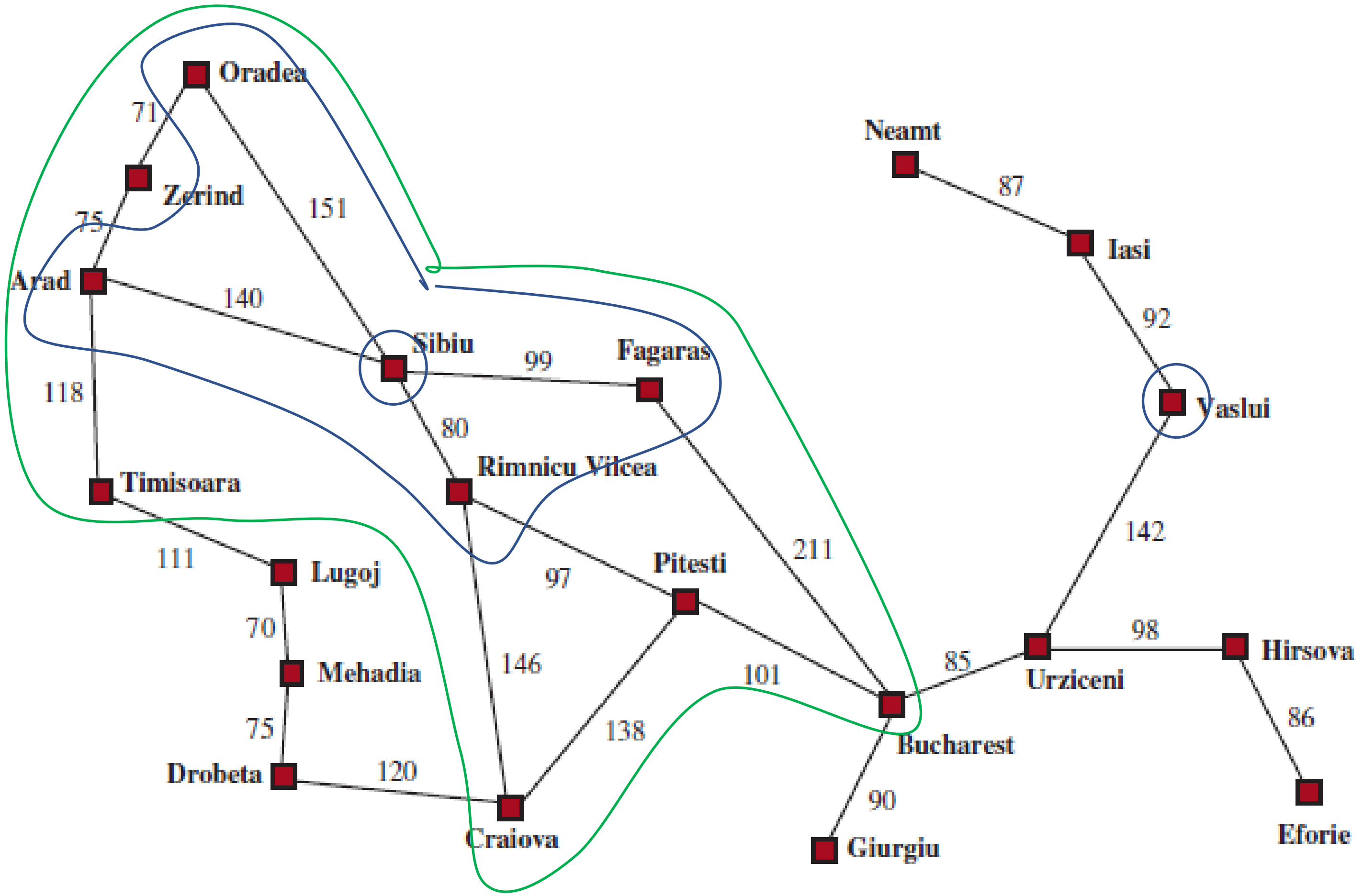
- suppose our goal was to find a route from Sibiu to Vasliu
- compare DFS vs BFS (assuming children are processed in counter-clockwise order)



Heuristic Functions

- Example 1: h_{SLD} for navigation

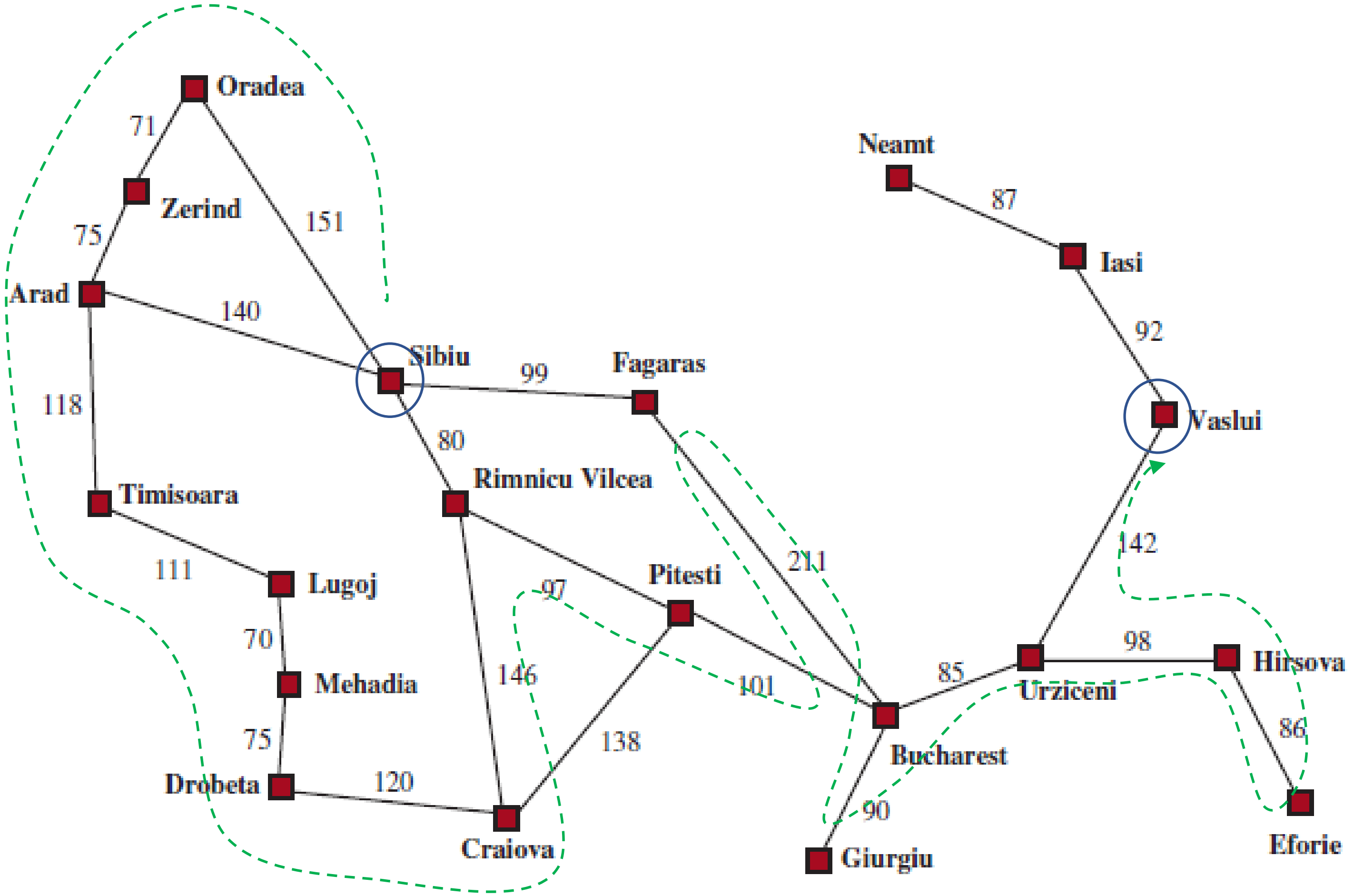
- suppose our goal was to find a route from Sibiu to Vasliu
- compare DFS vs BFS (assuming children are processed in counter-clockwise order)
- **BFS** (FIFO): (expand in levels)
  - frontier at each pass:
  - S | O,A,R,F | Z,T,C,P,B | L,D,U,G | M,H,V



Heuristic Functions

- Example 1: h_{SLD} for navigation

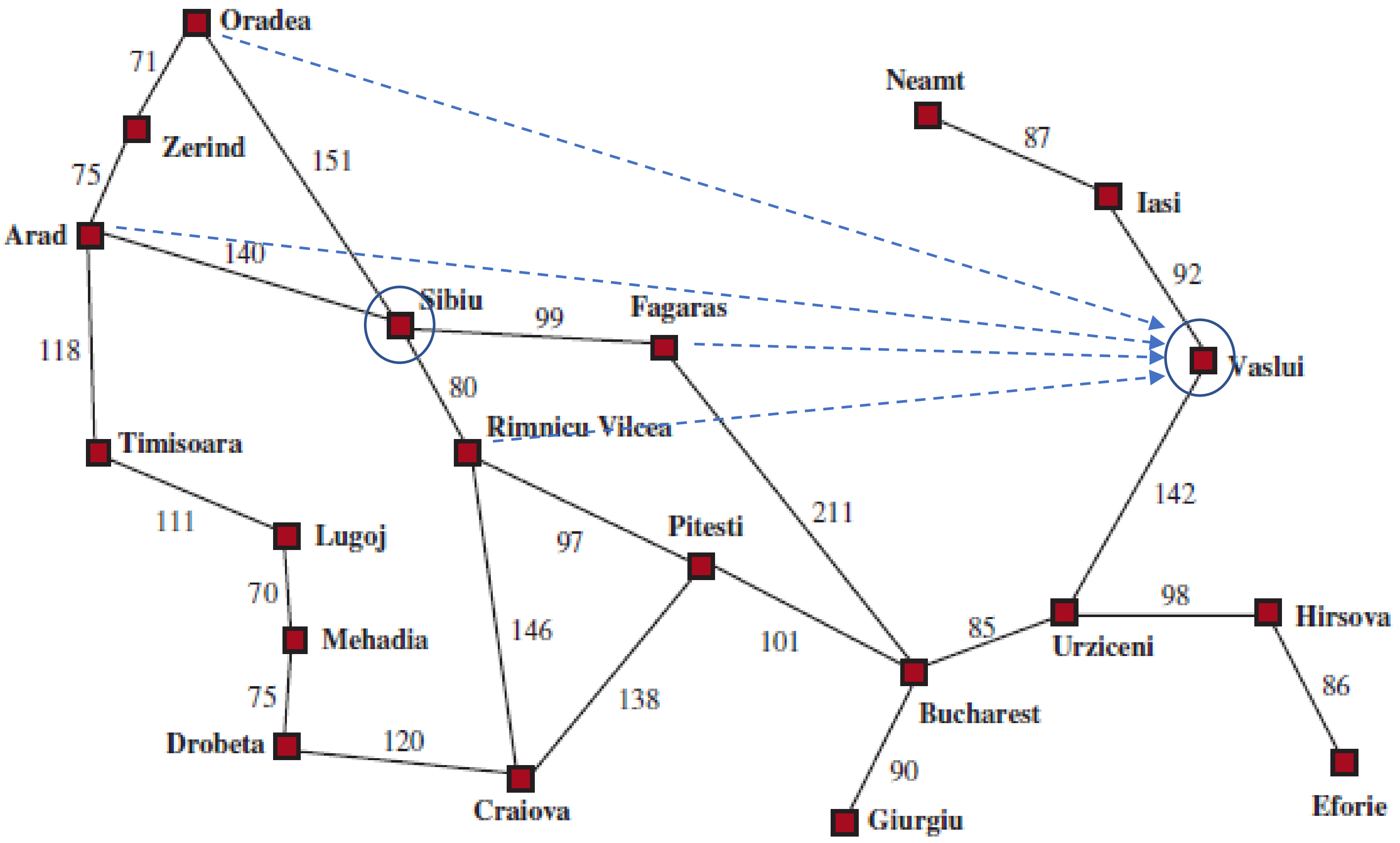
- suppose our goal was to find a route from Sibiu to Vasliu
- compare DFS vs BFS (assuming children are processed in counter-clockwise order)
- BFS (FIFO):
  - frontier at each pass:
  - S | O,A,R,F | Z,T,C,P,B | L,D,U,G | M,H,V
- **DFS** (LIFO): (follows a single path)
  - sequence of states visited:
  - S,O,Z,A,T,L,M,D,C,R,P,B,F,G,U,H,E,V



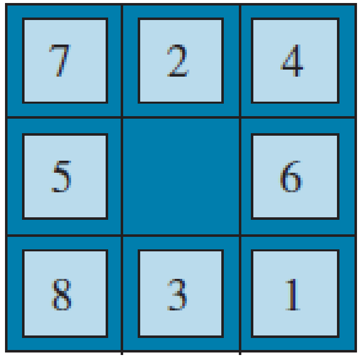
Heuristic Functions

- Example 1: h_{SLD} for Navigation

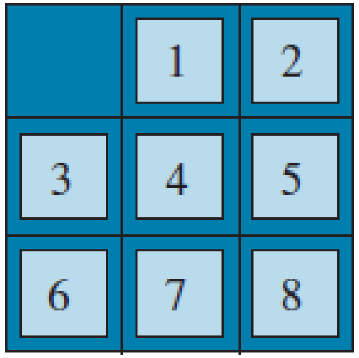
- suppose our goal was to find a route from Sibiu to Vasliu
- compare DFS vs BFS (assuming children are processed in counter-clockwise order)
- BFS (FIFO):
  - frontier at each pass:
  - S | O,A,R,F | Z,T,C,P,B | L,D,U,G | M,H,V
- DFS (LIFO):
  - sequence of states visited:
  - S,O,Z,A,T,L,M,D,C,R,P,B,F,G,U,H,E,V
- **h_{SLD}**: prioritize nodes in frontier based on straight-line distance to goal
  - sequence of states visited: S, F, B, U, V



Heuristic Functions



Start State

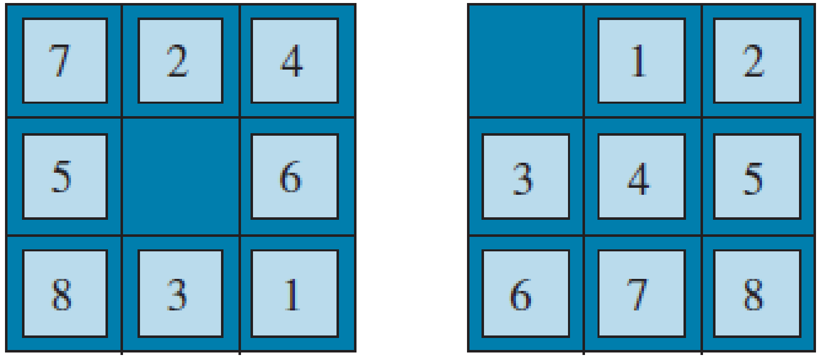


Goal State

- Example 2: heuristic functions for the Tile Puzzle
  - how close is any given state to being solved?
  - $h_1(n)$: # tiles out of place
    - this is an under-estimate because it will take more than move to put each tile in its proper place
    - still, it differentiates states that are almost solved for those that are very jumbled
    - even if 1 block is out of place, it might be close or very far away
  - $h_2(n)$: Manhattan distance
    - for each tile out of place, count number of rows and columns it needs to move
    - still an under-estimate of total moves because moving one tiles can put others out of place
    - ironically, it can also be an over-estimate, because a sequence of moves could put multiple tiles in place

$$h_2(n) = \sum_{i=1}^{9} |currRow(T_i) - goalRow(T_i)| + |currCol(T_i) - goalCol(T_i)|$$

Heuristic Functions



Start State   Goal State

7	2	4
5		6
8	3	1

1	2	
3	4	5
6	7	8

	1	6
3	4	5
2	7	8

h1 = 8
the 1 needs to move 3 steps
the 2 needs to move 1 step
the 3 needs to move 2 steps
...
h2 = 3+1+2+2+2+3+3+2 = 18

h1 = 2
h2 = 2

h1 = 2
h2 = 8

Where Do Heuristics Come From?

- Heuristics encode knowledge you have about the problem
  - rules of thumb
  - common solutions that are often used
  - way to decompose the problem into subgoals
  - useful sequences of actions
  - interactions/dependencies between operators...
- <u>This knowledge has to be formulated into a scoring function *h(n)* that estimates the distance of any state to the goal</u>
- Common strategy: approximate how many steps it would take to solve if we could *relax the constraints*
  - counting tiles out of place implies we can fix them in 1 move
  - Manhattan distance implies we can “slide tiles over each other”
  - for navigation, straight-line distance is shorter than any road, but still useful

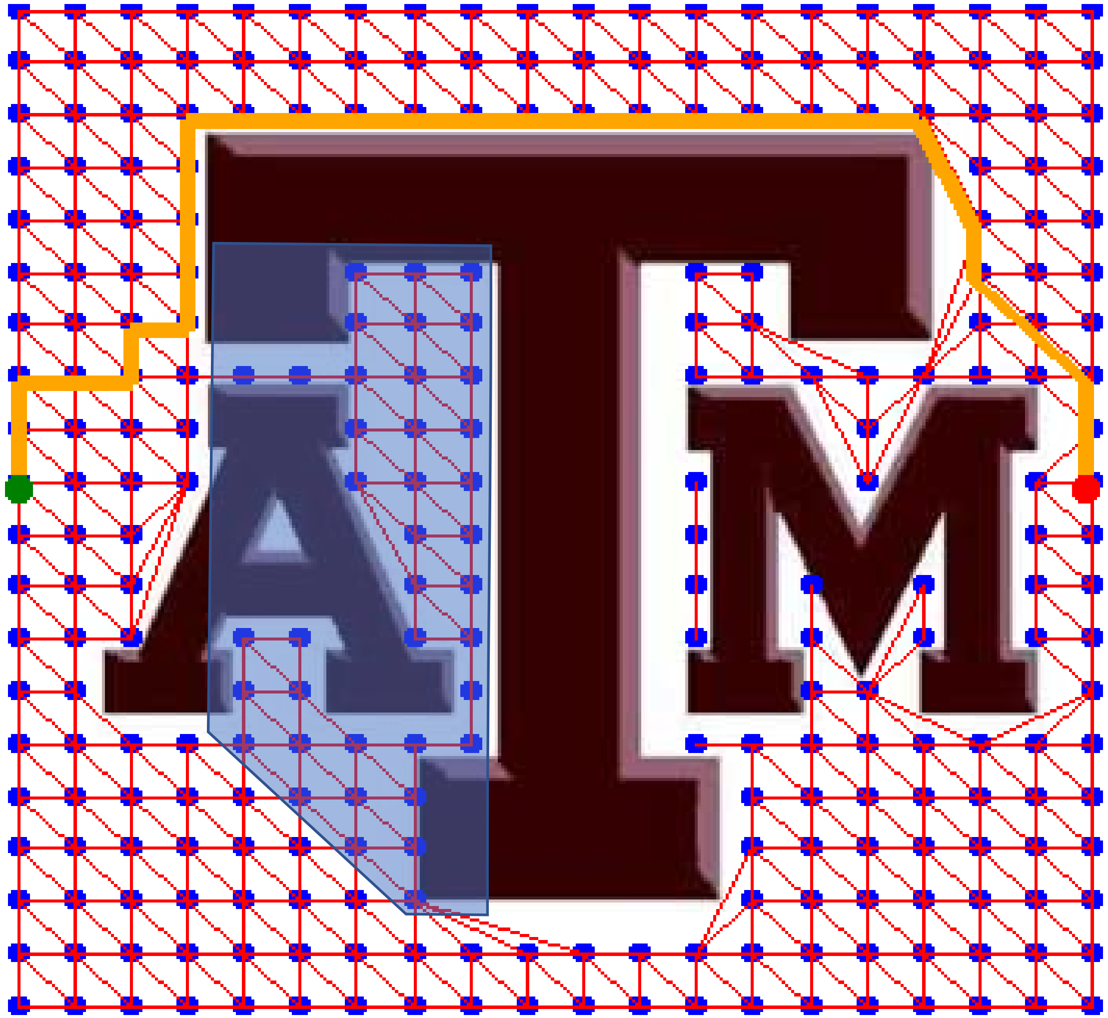
Greedy Search (best-first search with h(n))

- extending the iterative search algorithm to use a heuristic
- use a *priority queue* for frontier; sort nodes based on h(n)

```
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   where f is h(n)
  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
```

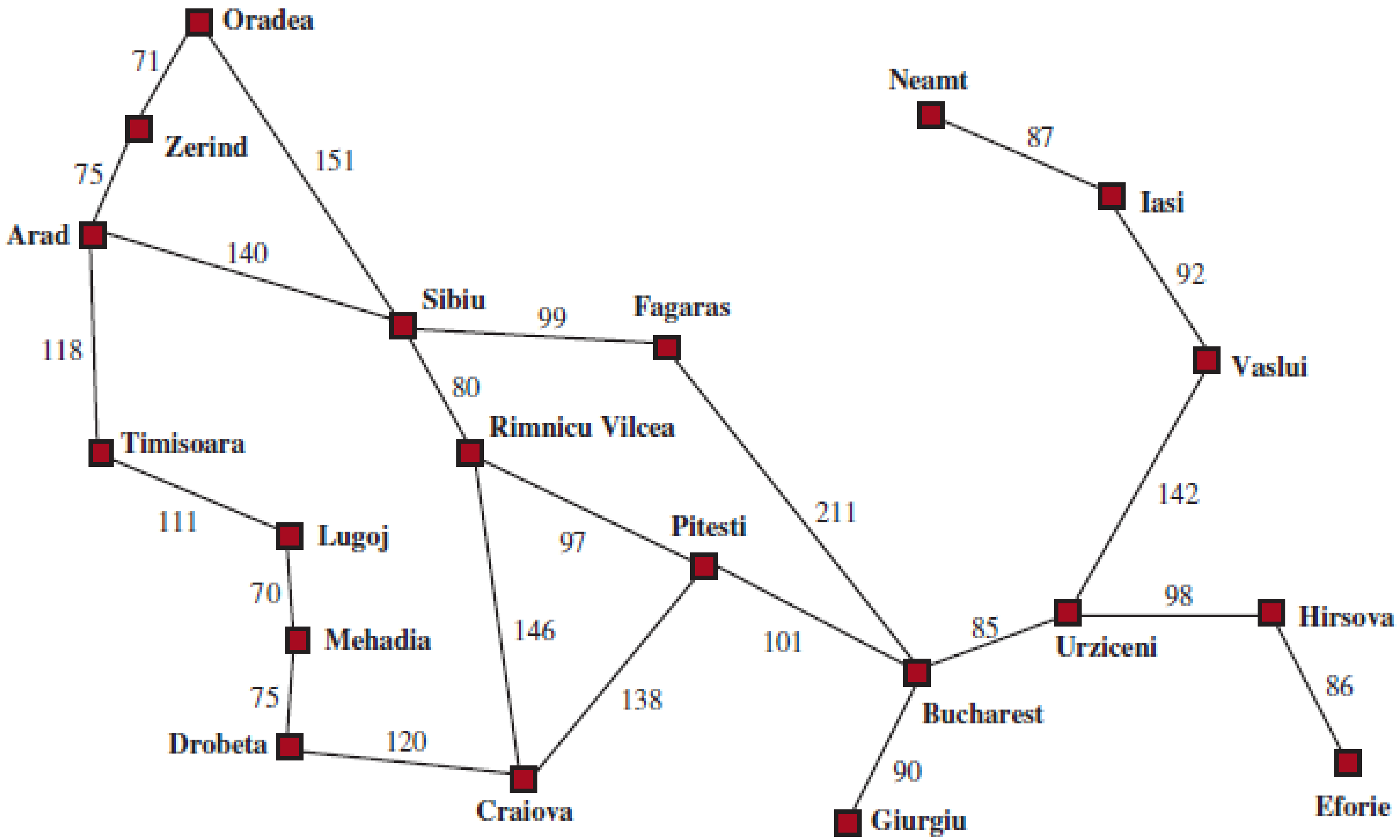
- (go back and review the slide on finding a route from Sibiu to Vasliu using Greedy with h_{SLD}, focusing on the queue)

Greedy Search



- The problem with Greedy Search is that it can be ‘misled’ by the heuristic to go in the wrong direction and waste time searching unproductive regions of the search space
- This is known as the “garden path” problem
- <u>Greedy Search would search the gray-boxed region first</u>, before discovering it has to go around the T to get the goal(red)

- How sub-optimal can it be? (in terms of cities expanded that are not actually on the solution path)
- What's the worst garden-path pair of cities for Romania?
- Can you think of a map and pair of cities that would force Greedy to visit every node before finding a route to the destination?



A* algorithm

- one of the most widely used and practical AI search algorithms
- essentially Best-first search (with priority queue), where nodes in frontier are sorted based on f(n)=g(n)+h(n)
  - where g(n)=path cost so far (from root to n)
  - and h(n)=heuristic estimate of remaining path cost (from n to closest goal)
  - so f(n) is an <u>estimate of total path cost </u>going through n to goal

A* algorithm

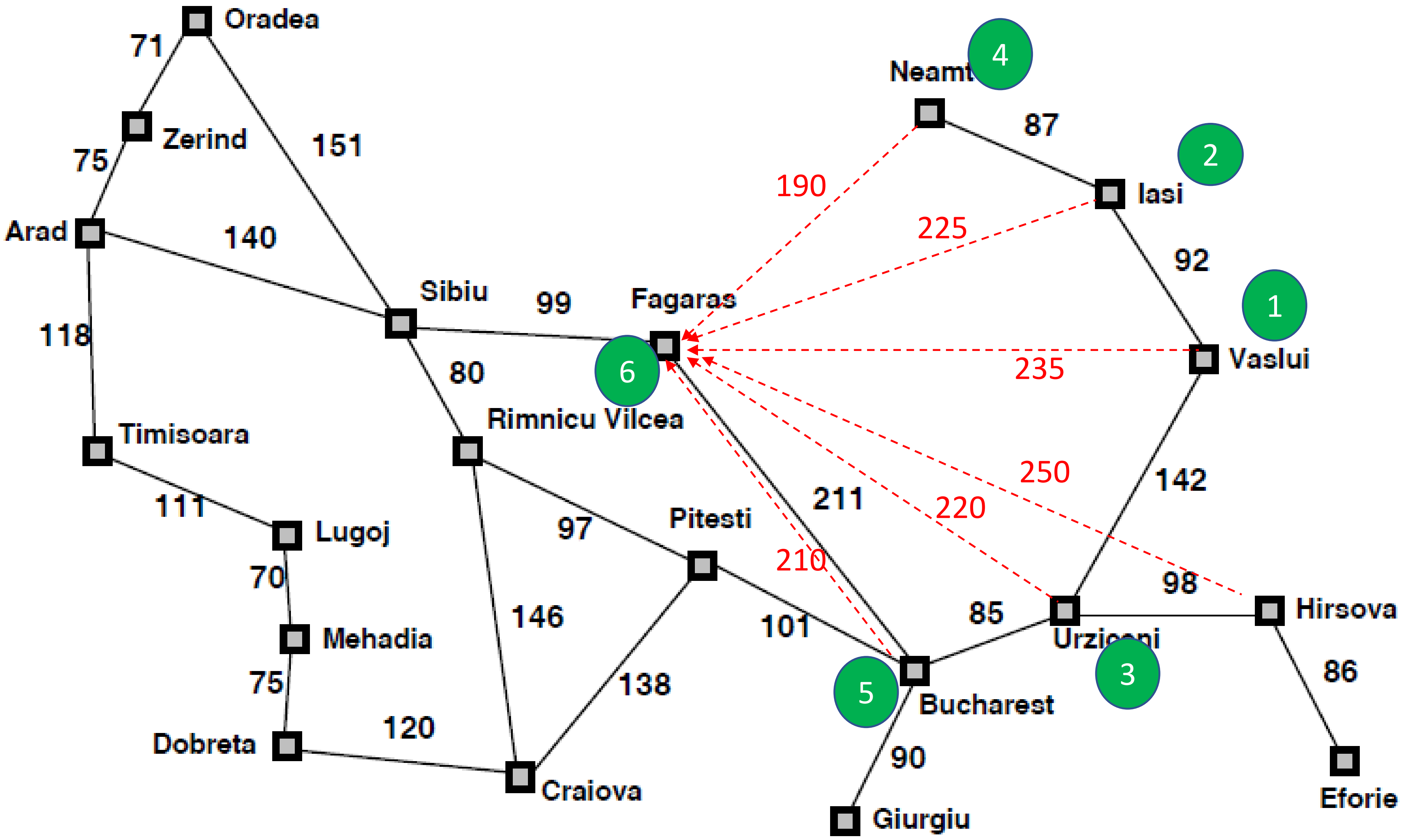
- use a *priority queue* for frontier; sort nodes based on f(n)=h(n)+g(n)

```
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
```

where f=h(n)+g(n)

note the ‘late’ goal test (see slide on UC alg)

A* search of Vasiliu to Fagaras:



frontier:
V(0+235=235)
$<V^{235}>$

1. pop V, push I and U
I(92+225=317)
U(142+220=362)
$<I^{317}, U^{362}>$

2. pop I; push N
N(92+87+190=369)
$<U^{362}, N^{369}>$

3. pop U; push B, H
B(142+85+210=437)
H(142+98+250=490)
$<N^{369}, B^{437}, H^{490}>$

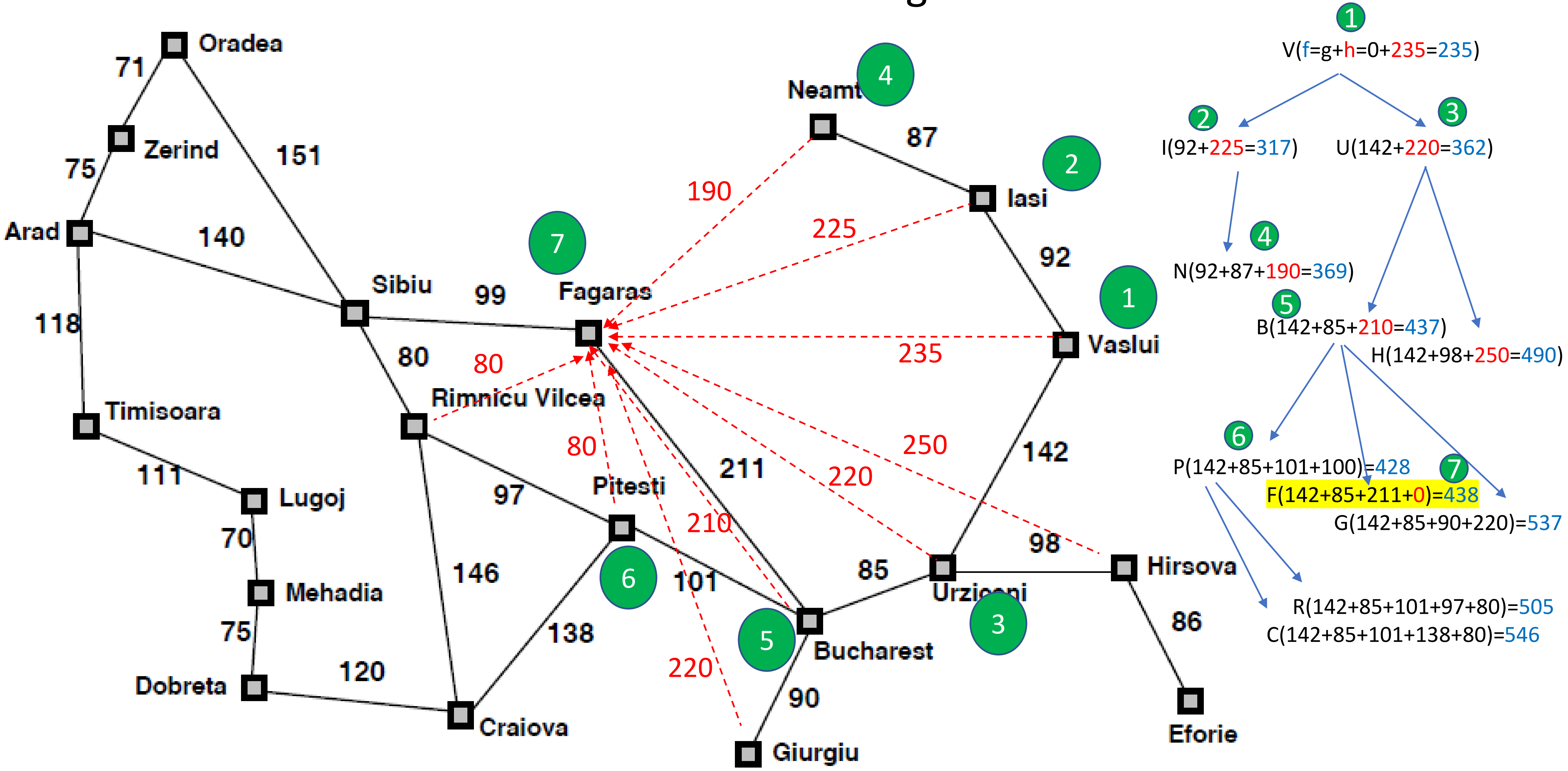
4. pop N; $<B^{437}, H^{490}>$

5. popB; push $F^{438=142+85+211}$
$<F^{438}, H^{490}>$

6. pop F

notice how f(n) for popped nodes keeps increasing:
V(235), I(317), U(362), N(369), B(437), F(438)

A* search of Vasiliu to Fagaras:



notice how f(n) for popped nodes keeps increasing:
V(235), I(317), U(362), N(369), B(437), F(438)

Computational Properties A* Search

- what guarantees about completeness and optimality can we make?
- remember that h(n) could be inaccurate!
  - it could tell us that many nodes down path are getting closer and closer, when in fact there is no way to reach the goal, and back-tracking is required
- first, we need to make an assumption...
- h(n) is *admissible*
  - h(n) <u>never over-estimates the true distance to the goal</u> for any node n
  - $0 \leq h(n) \leq c^*(n)=cost(n…g)$ for all states in the State Space

Computational Properties A* Search

- Theorem: A* is optimal (finds a goal with minimum path cost)
  - although this sounds obvious because the PQ is sorted on f(n), it is deceptive because it only applies to nodes in the frontier, but not all states in the space
  - suppose the optimal goal is g* but greedy returns g first, where c(g)>c(g*)
  - let n* be a node on the optimal path to g* that is in the frontier at same time
  - $f(n^*)=g(n^*)+\underline{h(n^*)} \leq cost(n_0..n^*)+\underline{cost(n^*..g^*)} = cost(n_0..g^*) = c(g^*)$
  - because of <u>admissibility</u>
  - therefore, n* should have been dequeued before g (and so on, down the path to g*)
- Important point: Even though admissibility is desirable, it is not necessary: A* search can be made more efficient with a heuristic even if it is not admissible (however, the solution path found might not be minimal)

Computational Properties A* Search

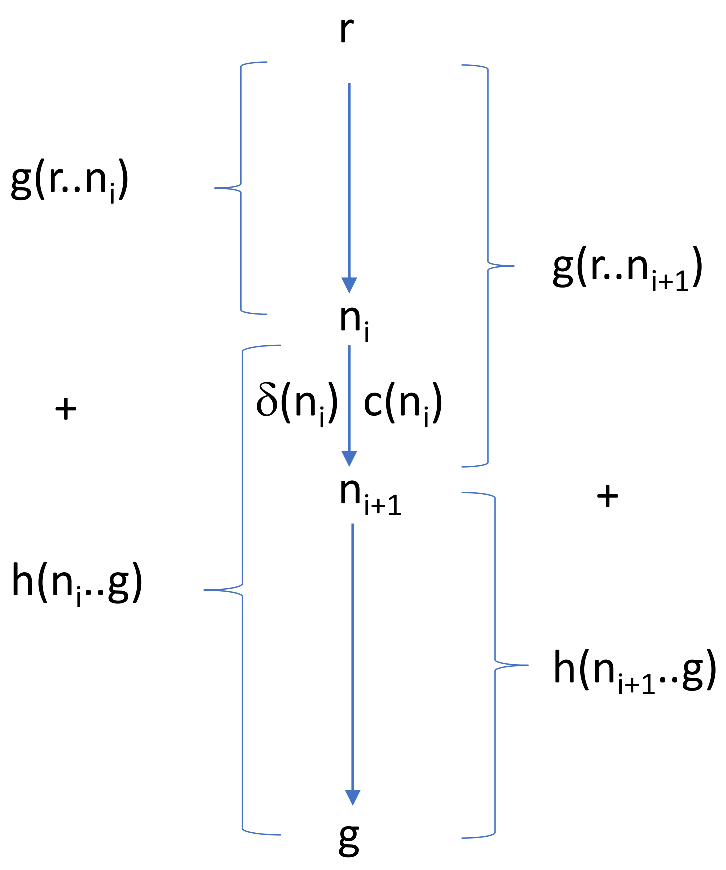
- Lemma: f(n) scores increase *monotonically* down any path from root
  - if a path is <n_0..n_i..g>, then $f(n_0) \leq f(n_i) \leq f(g)$
  - in any step $n_i \rightarrow n_{i+1}$, $h(n_i)$ includes a guess of the cost of op_i, whereas $g(n_{i+1})$ has the actual cost of that step, which could only be higher (by admissibility)
  - also requires *consistency* of heuristic, which is slightly stronger than admissibility (see book)
  - remember that at a goal node, f(g)=c(g) for any goal because f(g)=g(g)+h(g)=c(g)+0
  - so f(n) could be an <u>underestimate of total path length</u> early in a path, but converges to c*(g) as you get closer to the goal
- Theorem: A* explores states in order of increasing *f(n)* (total pathcost)

- estimated pathcost($r..n_i..g$)=$f(r...g)=g(r...n_i)+h(n_i...g)$
- $\delta(n_i)=h(n_i...g)-h(n_{i+1}...g)$
  - "estimated" cost of one action
  - assume δ always less than true cost of operator,
    $\delta(n_i)<c(n_i)$ "consistency" (related to admissibility)
- $g(r...n_{i-1})+h(n_i...g)=g(r...n_{i-1})+\delta(n_i)+h(n_{i+1}...g)$

  $\leq g(r...n_{i-1})+c(n_i)+h(n_{i+1}...g)$

therefore, estimates of total past costs always increase going down path:

  - pathcost($r..n_i..g$)<pathcost($r..n_{i+1}..g$)



Computational Properties A* Search

- analysis of time complexity
  - efficiency of A* is complicated because it depends on accuracy of the heuristic
  - generally speaking, **the more accurate the heuristic is, the faster the search**
    - boundary case 1: h(n)=0 – no help, exponential time like Uniform Cost, $O(b^{1+C^*/\varepsilon})$
    - boundary case 2: h(n)=c*(n) – a heuristic that perfectly predicts the true distance to the goal for any node will lead A* right to it (in time linear in the path length)

Computational Properties A* Search

- analysis of time complexity
  - if the inaccuracy of the heuristic is bounded, search will be sub-exponential
    - define "relative error" $\Delta = |h-h^*|/h^*$ (max over all nodes in the State Space)
    - then time complexity of A* is $O(b^{\Delta \cdot L(g)})$ where L is the path length to the goal g
    - if $|h-h^*|=O(\log(h^*))$ for all n, then A* will search a sub-exponential number of nodes before finding the optimal goal
    - however, this is rarely achievable in practice
  - one can also think of heuristic as making A* search more efficient by *reducing the effective branching factor* (for example, by half, if Δ=1/2)