Informed Search

CS457
David Kauchak
Fall 2011

Some material used from :
Sara Owsley Sood and others

Admin

- Q3
  - mean: 26.4
  - median: 27
- Final projects
  - proposals looked pretty good
  - start working
    - plan out exactly what you want to accomplish
    - make sure you have all the data, etc. that you need
    - status 1 still technically due 11/24, but can turn in as late as 11/27
    - status 2 due 12/2 (one day later)

Search algorithms

- Last time: search problem formulation
  - state
  - transitions (actions)
  - initial state
  - goal state
  - costs
- Now we want to find the solution!
- Use search techniques
  - Start at the initial state and search for a goal state
- What are candidate search techniques?
  - BFS
  - DFS
  - Uniform-cost search
  - Depth limited DFS
  - Depth-first iterative deepening

Finding the path: Tree search algorithms

- Basic idea:
  - keep a set of nodes to visit next (frontier)
  - pick a node from this set
  - check if it's the goal state
  - if not, expand out adjacent nodes and repeat

```
def treeSearch(start):
  add start to the frontier
  while frontier isn't empty:
    get the next node from the frontier
        if node contains goal state:
              return solution
       else:
              expand node and add resulting nodes to frontier
```

BFS and DFS

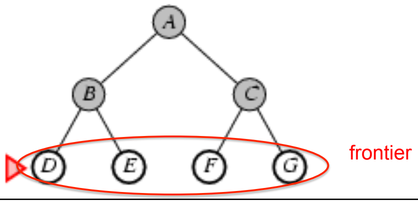
How do we get BFS and DFS from this?

```
def treeSearch(start):
  add start to the frontier
  while frontier isn't empty:
    get the next node from the frontier
        if node contains goal state:
              return solution
       else:
              expand node and add resulting nodes to frontier
```

Breadth-first search

- Expand shallowest unexpanded node
- Nodes are expanded a level at a time (i.e. all nodes at a given depth)

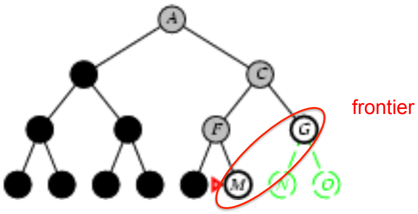
- Implementation:
  - *frontier* is a FIFO (queue), i.e., new successors go at end



Depth-first search

- Expand deepest unexpanded node

- Implementation:
  - *frontier* = LIFO (stack), i.e., put successors at front



Search algorithm properties

- Time (using Big-O)
- Space (using Big-O)
- Complete
  - If a solution exists, will we find it?
- Optimal
  - If we return a solution, will it be the best/optimal solution

- A divergence from data structures
  - we generally won't use V and E to define time and space. Why?
    - Often V and E are infinite (or very large relative to solution)!
    - Instead, we often use the branching factor (*b*) and depth of solution (*d*)

Activity

- Analyze DFS and BFS according to:
  - time,
  - space,
  - completeness
  - optimality

  (for time and space, analyze in terms of *b*, *d* and *m (max depth);* for complete and optimal - simply YES or NO)
  - Which strategy would you use and why?
- Brainstorm improvements to DFS and BFS

BFS

- Time: $O(b^d)$
- Space: $O(b^d)$
- Complete = YES
- Optimal = YES if action costs are fixed, NO otherwise

Time and Memory requirements for BFS

Depth	Nodes	Time	Memory
2	1100	.11 sec	1 MB
4	111,100	11 sec	106 MB
6	10^7	19 min	10 GB
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3,523 years	1 exabyte

BFS with b=10, 10,000 nodes/sec; 10 bytes/node

DFS

- Time: $O(b^m)$
- Space: $O(bm)$
- Complete = YES, if space is finite (and no circular paths), NO otherwise
- Optimal = NO

Problems with BFS and DFS

- BFS
  - doesn't take into account costs
  - memory! ☹
- DFS
  - doesn't take into account costs
  - not optimal
  - can't handle infinite spaces
  - loops

Uniform-cost search

- Expand unexpanded node with the smallest *path* cost, g(x)

- Implementation:
  - *frontier* = priority queue ordered by **path** cost
  - similar to Dijkstra's algorithm
- Equivalent to breadth-first if step costs all equal

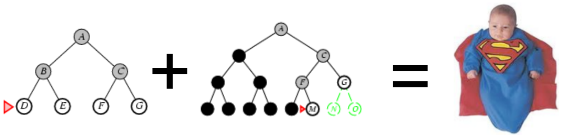
Uniform-cost search

- Time? and Space?
  - dependent on the costs and optimal path cost, so cannot be represented in terms of *b* and *d*
  - Space will still be expensive (e.g. take uniform costs)
- Complete?
  - YES, assuming costs > 0
- Optimal?
  - Yes, assuming costs > 0

- This helped us tackle the issue of costs, but still going to be expensive from a memory standpoint!

Ideas?

Can we combined the optimality and completeness of BFS with the memory of DFS?



Depth limited DFS

- DFS, but with a depth limit **L** specified
  - nodes at depth **L** are treated as if they have no successors
  - we only search down to depth **L**
- Time?
  - O(b^L)
- Space?
  - O(bL)
- Complete?
  - No, if solution is longer than L
- Optimal
  - No, for same reasons DFS isn't

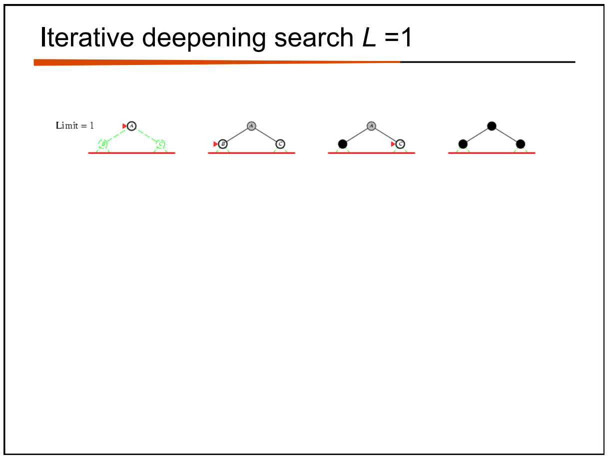
Ideas?

Iterative deepening search

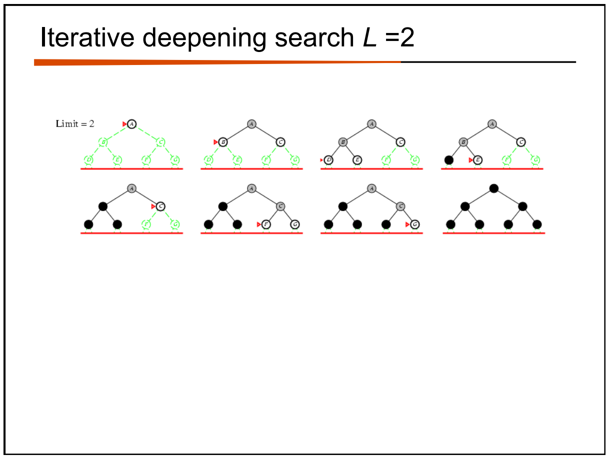
```
For depth 0, 1, …., ∞
    run depth limited DFS
    if solution found, return result
```

- Blends the benefits of BFS and DFS
  - searches in a similar order to BFS
  - but has the memory requirements of DFS
- Will find the solution when **L** is the depth of the shallowest goal

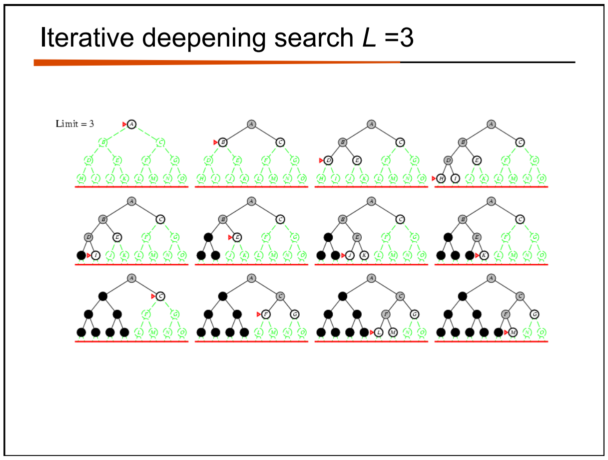
Iterative deepening search *L* =0



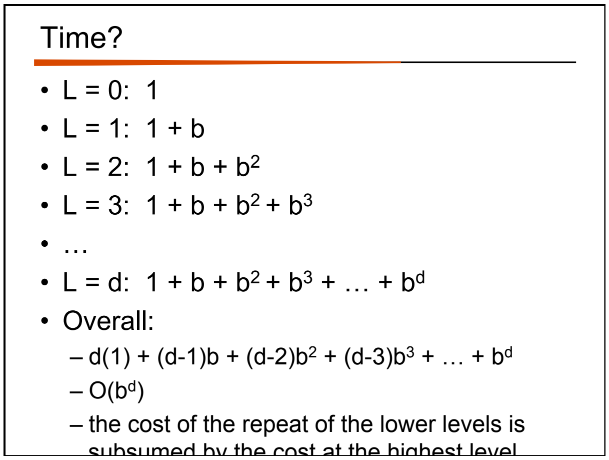
Iterative deepening search *L* =1



Iterative deepening search *L* =2



Iterative deepening search *L* =3



Time?

- L = 0: 1
- L = 1: 1 + b
- L = 2: 1 + b + b^2
- L = 3: 1 + b + b^2 + b^3
- …
- L = d: 1 + b + b^2 + b^3 + … + b^d
- Overall:
  - – d(1) + (d-1)b + (d-2)b^2 + (d-3)b^3 + … + b^d
  - – O(b^d)
  - – the cost of the repeat of the lower levels is subsumed by the cost at the highest level

Properties of iterative deepening search

- Space?
  - *O(bd)*
- Complete?
  - Yes
- Optimal?
  - Yes, if step cost = 1

Summary of algorithms

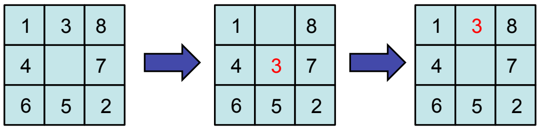
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

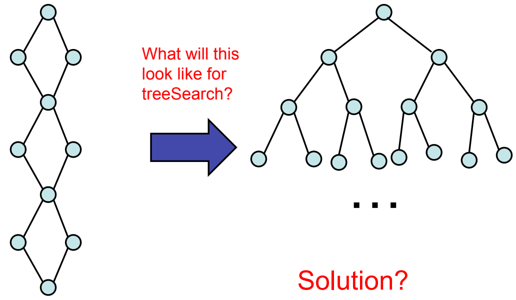
Repeated states

What is the impact of repeated states?



```
def treeSearch(start):
  add start to the frontier
  while frontier isn't empty:
    get the next node from the frontier
       if node contains goal state:
            return solution
      else:
            expand node and add resulting nodes to frontier
```

Can make problems seem harder



What will this look like for treeSearch?

. . .

Solution?

Graph search

- Keep track of nodes that have been visited (explored)
- Only add nodes to the frontier if their *state* has not been seen before

```
def graphSearch(start):
  add start to the frontier
  set explored to empty
  while frontier isn't empty:
    get the next node from the frontier
       if node contains goal state:
              return solution
       else:
              add node to explored set
              expand node and add resulting nodes to frontier,
                if they are not in frontier or explored
```

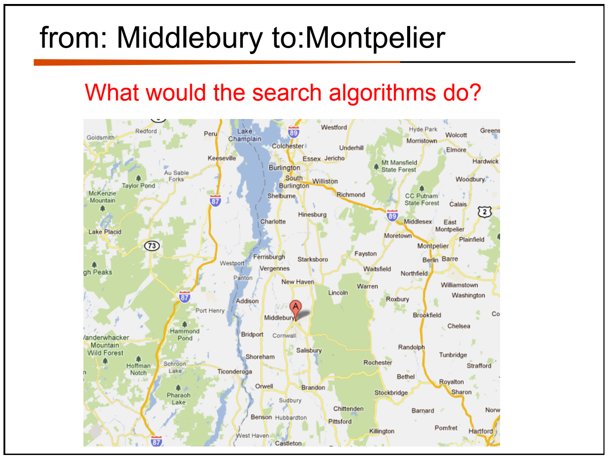
Graph search implications?

- We' re keeping track of all of the states that we've previously seen
- For problems with lots of repeated states, this is a huge time savings
- The tradeoff is that we blow-up the memory usage
  - Space graphDFS?
    - $O(b^m)$
- Something to think about, but in practice, we often just use the tree approach

8-puzzle revisited

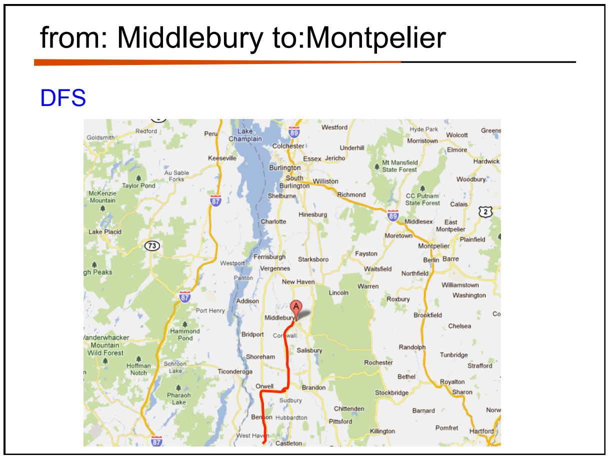
- The average depth of a solution for an 8-puzzle is 22 moves
- What do you think the average branching factor is?
  - ~3 (center square has 4 options, corners have 2 and edges have 3)
- An exhaustive search would require $\sim 3^{22} = 3.1 \times 10^{10}$ states
  - BFS: 10 terabytes of memory
  - DFS: 8 hours (assuming one million nodes/second)
  - IDS: ~9 hours
- Can we do better?

1	3	8
4		7
6	5	2



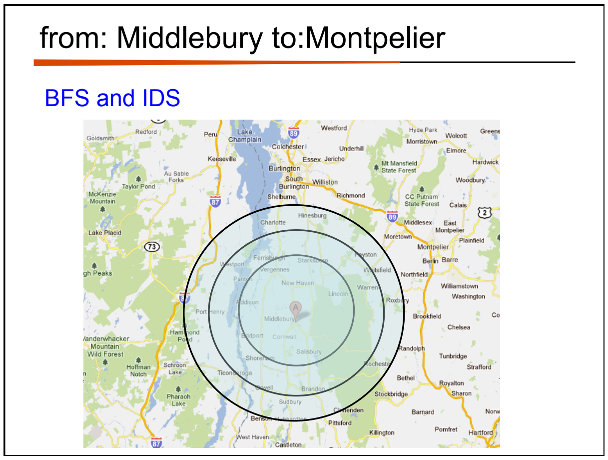
from: Middlebury to:Montpelier

What would the search algorithms do?



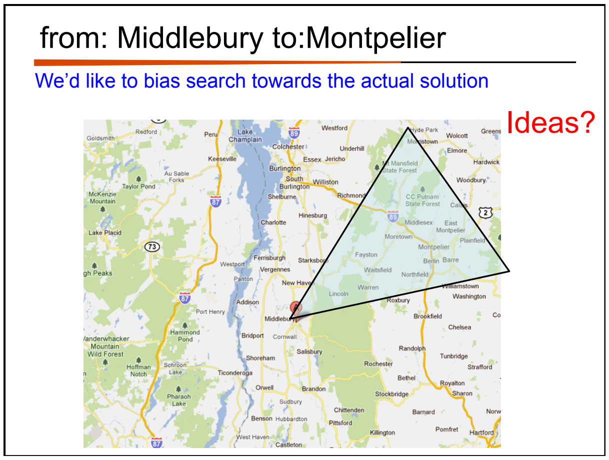
from: Middlebury to:Montpelier

DFS



from: Middlebury to:Montpelier

BFS and IDS



from: Middlebury to:Montpelier

We'd like to bias search towards the actual solution

Ideas?

Informed search

- Order the *frontier* based on some knowledge of the world that estimates how “good” a node is
  - *f(n)* is called an evaluation function
- Best-first search
  - rank the frontier based on *f(n)*
  - take the most desirable state in the frontier first
  - different search depending on how we define *f(n)*

```
def treeSearch(start):
  add start to the frontier
  while frontier isn’t empty:
    get the next node from the frontier
        if node contains goal state:
              return solution
       else:
              expand node and add resulting nodes to frontier
```

Heuristic

Merriam-Webster's Online Dictionary

Heuristic (pron. \hyu-ˈris-tik\): adj. [from Greek *heuriskein* to discover.] involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods

The Free On-line Dictionary of Computing (15Feb98)

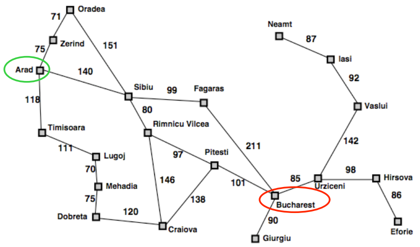
heuristic 1. <programming> A rule of thumb, simplification or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. Unlike algorithms, heuristics do not guarantee feasible solutions and are often used with no theoretical guarantee. 2. <algorithm> approximation algorithm.

Heuristic function: *h(n)*

- An estimate of how close the node is to a goal
- Uses domain-specific knowledge
- Examples
  - Map path finding?
    - straight-line distance from the node to the goal (“as the crow flies”)
  - 8-puzzle?
    - how many tiles are out of place
  - Missionaries and cannibals?
    - number of people on the starting bank

Greedy best-first search

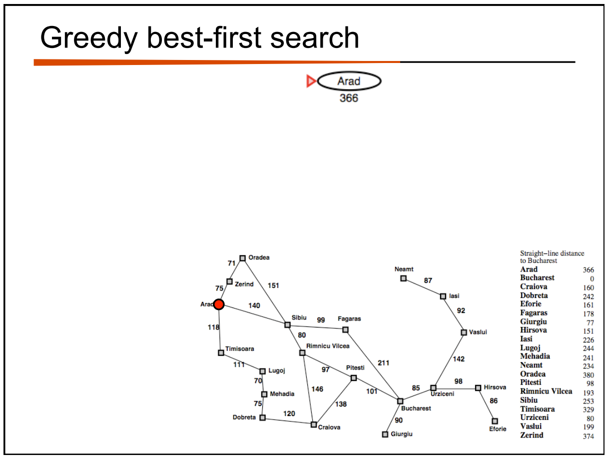
- *f(n) = h(n)*
  - rank nodes by how close we think they are to the goal



Straight−line distance to Bucharest

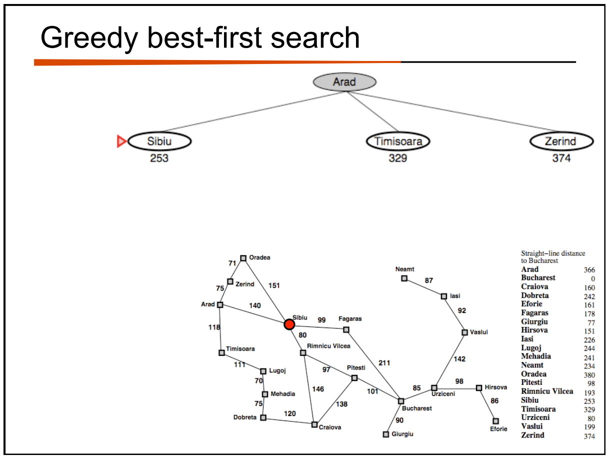
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Arad to Bucharest



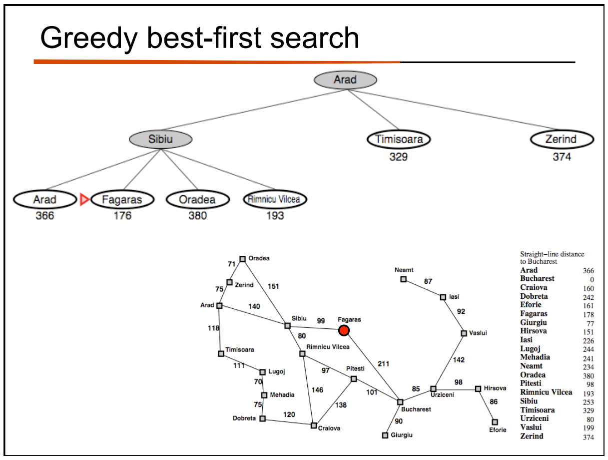
Greedy best-first search

Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



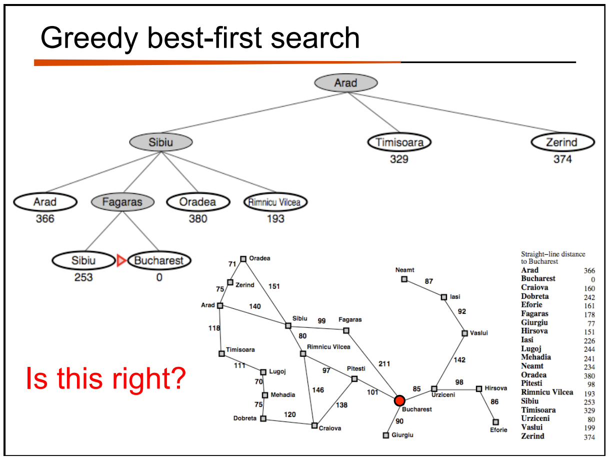
Greedy best-first search

Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Greedy best-first search

Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



Greedy best-first search

Is this right?

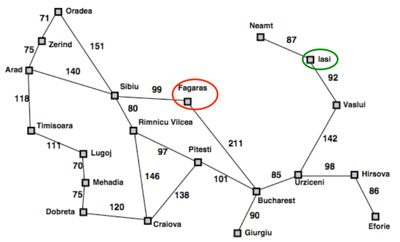
Straight−line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Problems with greedy best-first search

- Time?
  - – O(b^m) – but can be much faster
- Space
  - – O(b^m) – have to keep them in memory to rank
- Complete?

Problems with greedy best-first search

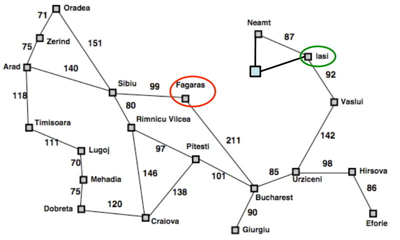
- Complete?
  - – Graph search, yes
  - – Tree search, no



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Problems with greedy best-first search

- Complete?
  - – Graph search, yes
  - – Tree search, no



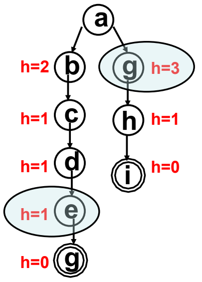
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Problems with greedy best-first search

- Optimal?

Problems with greedy best-first search

- Optimal?
  - no, as we just saw in the map example



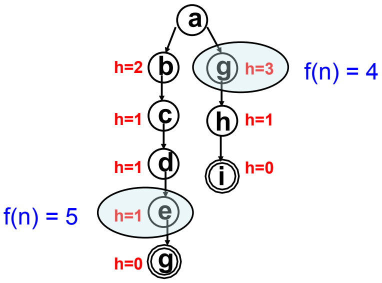
Sometimes
it’s too greedy

What is the problem?

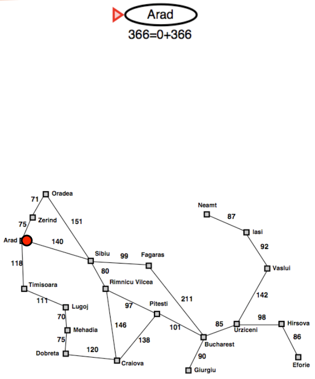
A* search

- Idea:
  - don’t expand paths that are already expensive
  - take into account the path cost!
- *f(n) = g(n) + h(n)*
  - *g(n)* is the path cost so far
  - *h(n)* is our estimate of the cost to the goal
- *f(n)* is our estimate of the total path cost to the goal through *n*

A* search

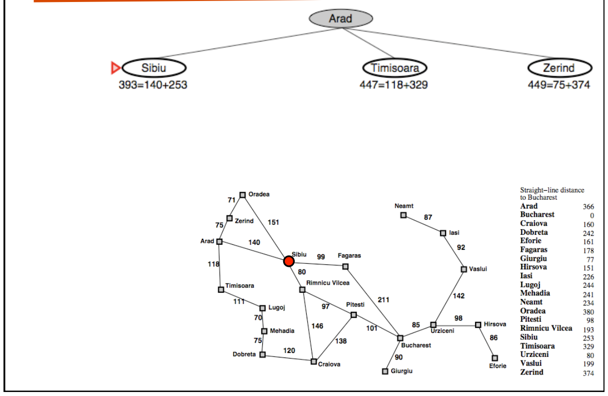


A* search



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

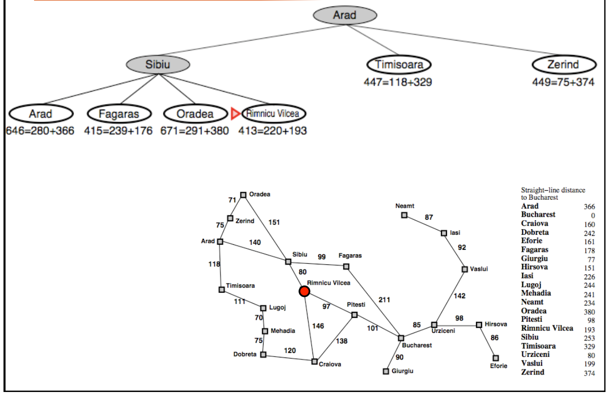
A* search



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

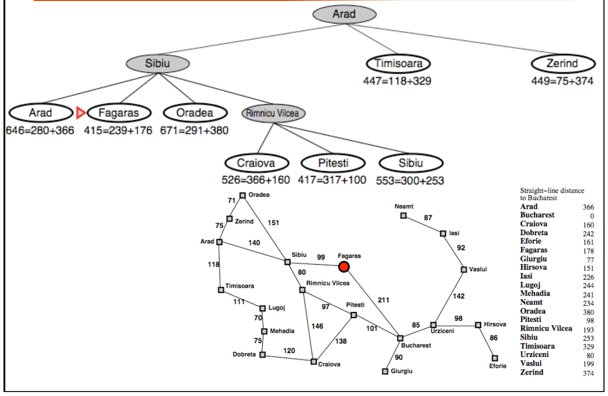
A* search



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

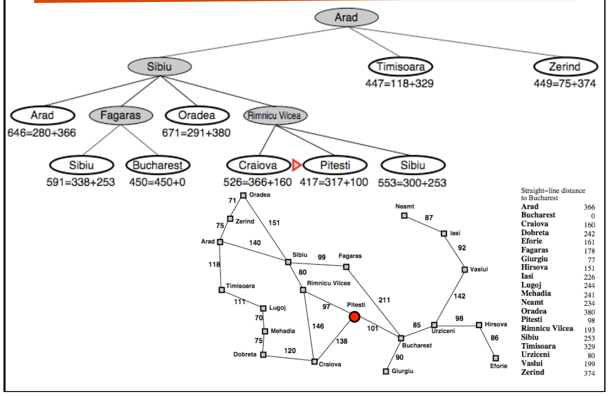
A* search



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

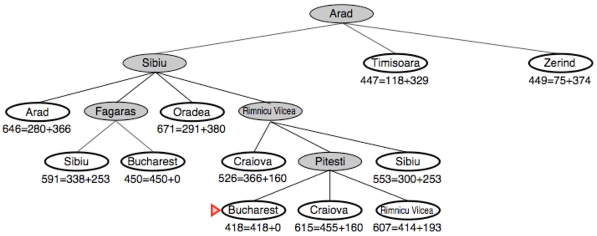
A* search



Straight−line distance
to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

A* search



Admissible heuristics

- A heuristic function is *admissible* if it never overestimates
  - if h*(n) is the actual distance to the goal
  - h(n) ≤ h*(n)
- An admissible heuristic is optimistic (it always thinks the goal is closer than it actually is)
- Is the straight-line distance admissible?

closest to the actual “price”
without going over

A* properties

- Time
  - depends on heuristic, but generally exponential
- Space
  - exponential (have to keep all the nodes in memory/ frontier)
- Complete
  - YES
- Optimal
  - YES, if the heuristic is admissible
  - Why?
    - If we could overestimate, then we could find (that is remove from the queue) a goal node that was suboptimal because our estimate for the optimal goal was too large

A point of technicality

- Technically if the heuristic isn’t admissible, then the search algorithm that uses f(n) = g(n) + h(n) is call “Algorithm A”
- A* algorithm requires that the heuristic is admissible
- That said, you’ ll often hear the later referred to as A*
- Algorithm A is not optimal

Admissible heuristics

- 8-puzzle
  - $h_1(n)$ = number of misplaced tiles?
  - $h_2(n)$ = manhattan distance?

admissible?

$h_1 = 7$
$h_2 = 12$

1	3	8
4		7
6	5	2

$h_1 = 8$
$h_2 = 8$

1	2	5
4		8
3	6	7

	1	2
3	4	5
6	7	8

goal

Admissible heuristics

- 8-puzzle
  - $h_1(n)$ = number of misplaced tiles?
  - $h_2(n)$ = manhattan distance?

which is better?

$h_1 = 7$
$h_2 = 12$

1	3	8
4		7
6	5	2

$h_1 = 8$
$h_2 = 8$

1	2	5
4		8
3	6	7

	1	2
3	4	5
6	7	8

goal

Dominance

- Given two admissible heuristic functions
  - if $h_i(n) \geq h_j(n)$ for all *n*
  - then $h_i(n)$ *dominates* $h_j(n)$
- A dominant function is always better. Why?
  - It always give a better (i.e. closer) estimate to the actual path cost, without going over
- What about?
  - $h_1(n)$ = number of misplaced tiles
  - $h_2(n)$ = manhattan distance

Dominance

- $h_2(n)$ dominates $h_1(n)$

depth of solution	**IDS**	**A*(h1)**	**A*(h2)**
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14		539	113
16		1301	211
18		3056	363
20		7276	676

average nodes expanded for 8-puzzle problems

Combining heuristics

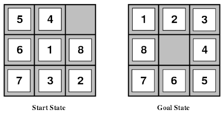
- Sometimes, we have multiple admissible heuristics, but none dominates
- What then?
  - We can take the max of all the heuristics!
- Why?
  - Since they're all admissible, we know none overestimate
  - taking the max gives us a closer/better estimate
  - overall, a better heuristic function

Relaxed problems

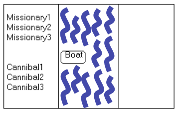
- A problem with fewer restrictions on the actions is called a relaxed problem
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- How might you create a relaxed problem for the 8-puzzle?
  - a tile can move anywhere (same as $h_1(n)$)
  - a tile can move to any adjacent square (same as $h_2(n)$)

Creating Heuristics

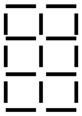
8-Puzzle



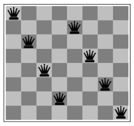
Missionaries and Cannibals



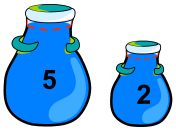
Remove 5 Sticks



N-Queens



Water Jug Problem



Route Planning