Lecture 10: Search (II)

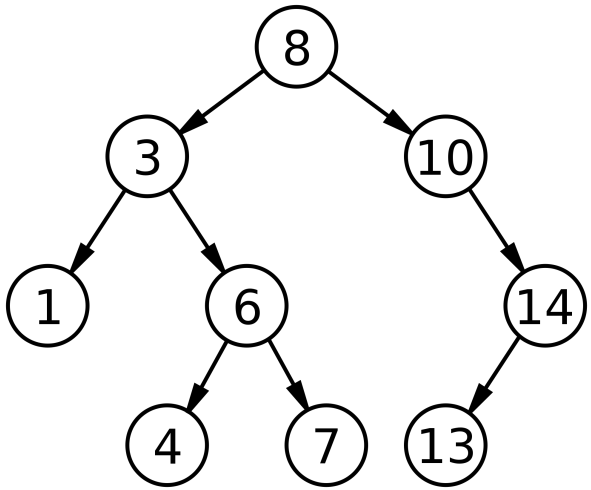
EMAT31530/Feb 2018/Raul Santos-Rodriguez

Have a look at ...

... Russell and Norvig (Ch. 3)

Summary

Objective: find the minimum cost path from s_{start} to an s satisfying $Goal(s)$ =TRUE.



Tree search: Backtracking, BFS, DFS, DFS-ID

State: summary of past actions sufficient to choose future actions optimally

Dynamic programming: backtracking search with memoization (exponential savings)

Dynamic programming only works for acyclic graphs...what if there are cycles?

Question

Suppose we want to travel from city 1 to city n (going only forward). It costs $c_{ij} \geq 0$ to go from i to j. Which of the following algorithms will find the minimum cost path (select all that apply)?

Question

- Depth-first search
- Breadth-first search
- Dynamic programming

Outline

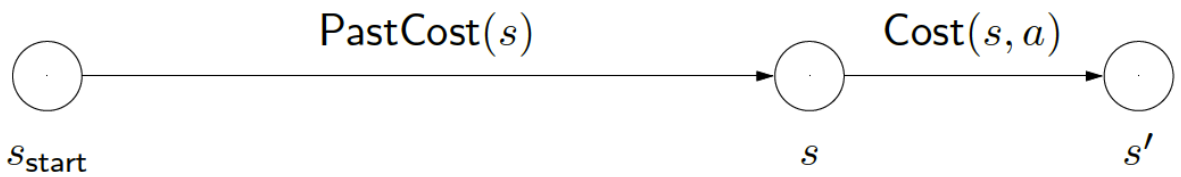
Search or how to find a sequence of actions that achieves a goal when no single action will do. Today we will cover

- Uniform Cost Search
- Informed vs Uninformed search
- Greedy search

Dynamic programming vs Uniform Cost Search

Dynamic programming only works for acyclic graphs... what if there are cycles?

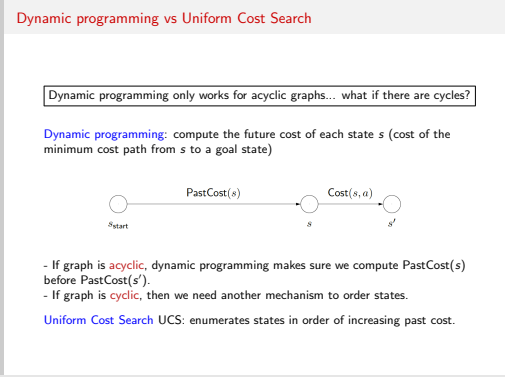
Dynamic programming: compute the future cost of each state s (cost of the minimum cost path from s to a goal state)



- If graph is acyclic, dynamic programming makes sure we compute PastCost(s) before PastCost(s').
- If graph is cyclic, then we need another mechanism to order states.

Uniform Cost Search UCS: enumerates states in order of increasing past cost.

Dynamic programming vs Uniform Cost Search



Recall that we used dynamic programming to compute the future cost of each state s, the cost of the minimum cost path from s to a goal state. We can analogously define *PastCost*(s), the cost of the minimum cost path from the start state to s. If instead of having access to the successors via $Succ(s, a)$, we had access to predecessors (think of reversing the edges in the state graph), then we could define a dynamic program to compute all the *PastCost*(s). Dynamic programming relies on the absence of cycles, so that there was always a clear order in which to compute all the past costs. If the past costs of all the predecessors of a state s are computed, then we could compute the past cost of s by taking the minimum. Note that *PastCost*(s) will always be computed before *PastCost*(s') if there is an edge from s to s'. In essence, the past costs will be computed according to a topological ordering of the nodes. When there are cycles, no topological ordering exists, so we compute the past costs in order of increasing past cost: UCS.

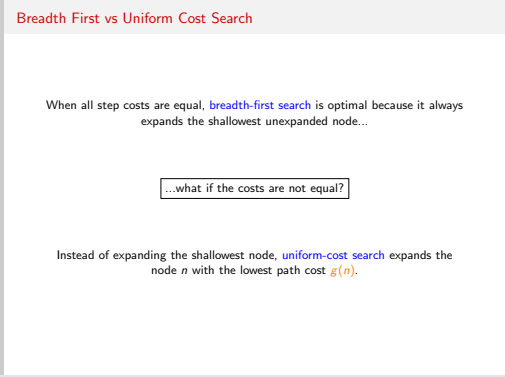
Breadth First vs Uniform Cost Search

When all step costs are equal, breadth-first search is optimal because it always expands the shallowest unexpanded node...

...what if the costs are not equal?

Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost $g(n)$.

Breadth First vs Uniform Cost Search



While breadth first search guarantees to find the solution with the least number of steps, this property becomes less useful for scenarios where all steps are not equal (on a map, for instance, traveling from Bristol to London and traveling from Bristol to Cardiff may both be a 'step', but certainly do not have the same cost in miles!). In this case, a uniform cost search is used instead. Uniform cost searches always expand the node with the lowest total path cost from the initial node. Thus, they are always optimal (since any cheaper solution would already have been found). Their salient characteristic is the fact that they start from the initial start node when they calculate the path cost in the search.

- Note: those of you who have studied algorithms should immediately recognize UCS as Dijkstra's algorithm. Logically, the two are indeed equivalent. There is an important implementation difference: UCS takes as input a search problem, which implicitly defines a large and even infinite graph, whereas Dijkstra's algorithm (in the typical exposition) takes as input a fully concrete graph. The implicitness is important in practice because we might be working with an enormous graph (a detailed map of world) but only need to find the path between two close by points (Bristol to Cardiff). Another difference is that Dijkstras algorithm is usually thought of as finding the shortest path from the start state to every other node, whereas UCS is explicitly about finding the shortest path to a goal state.

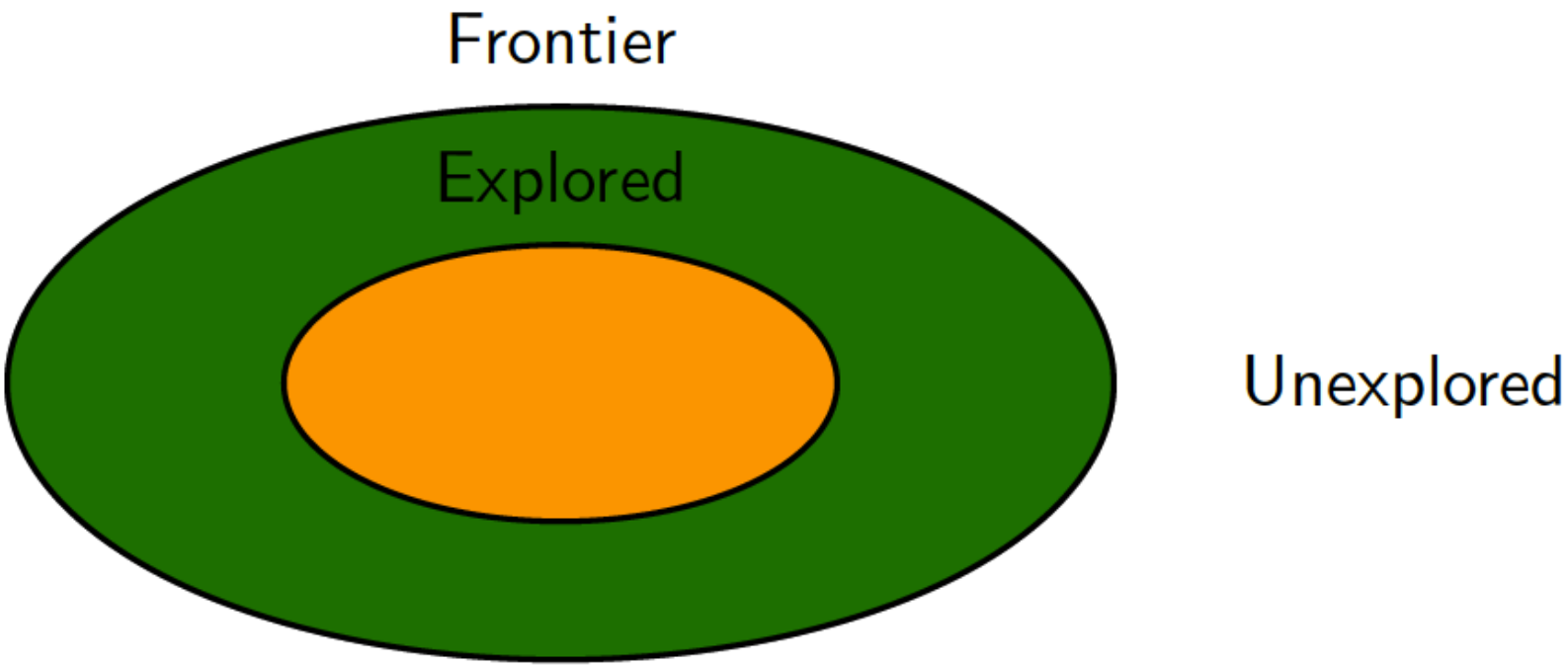
Breadth First vs Uniform Cost Search

Two main differences:

1. The goal test is applied to a node when it is selected for expansion rather than when it is first generated: the first goal node that is generated may be on a suboptimal path.
2. A test is added in case a better path is found to a node currently on the frontier.

Assumption: all action costs are non-negative

Algorithm: intuition

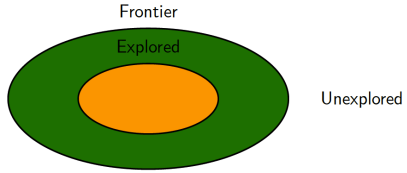


- **Explored**: states we've found the optimal path to
- **Frontier**: states we've seen, still figuring out how to get there cheaply
- **Unexplored**: states we haven't seen

[Liang 2014]

Algorithm: intuition

Algorithm: intuition



- **Explored**: states we've found the optimal path to
- **Frontier**: states we've seen, still figuring out how to get there cheaply
- **Unexplored**: states we haven't seen

[Liang 2014]

The general strategy of UCS is to maintain three sets of nodes: explored, frontier, and unexplored. Throughout the course of the algorithm, we move states from unexplored to the frontier to the explored. The key invariant is that we have computed the minimum cost paths to all the nodes in the explored set. So when the goal state moves into the explored set, then we are done.

UCS: travel from city 1 to city n

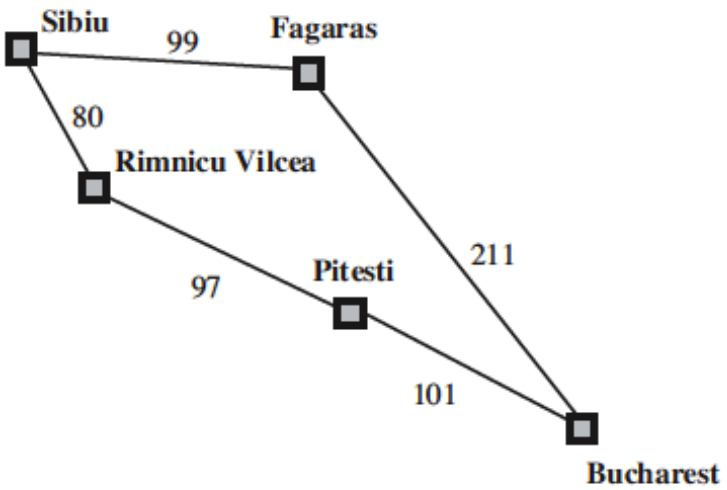


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

[Russell and Norvig]

UCS: travel from city 1 to city n

UCS: travel from city 1 to city n

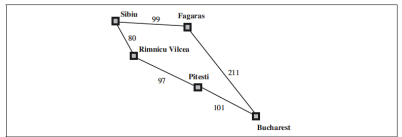
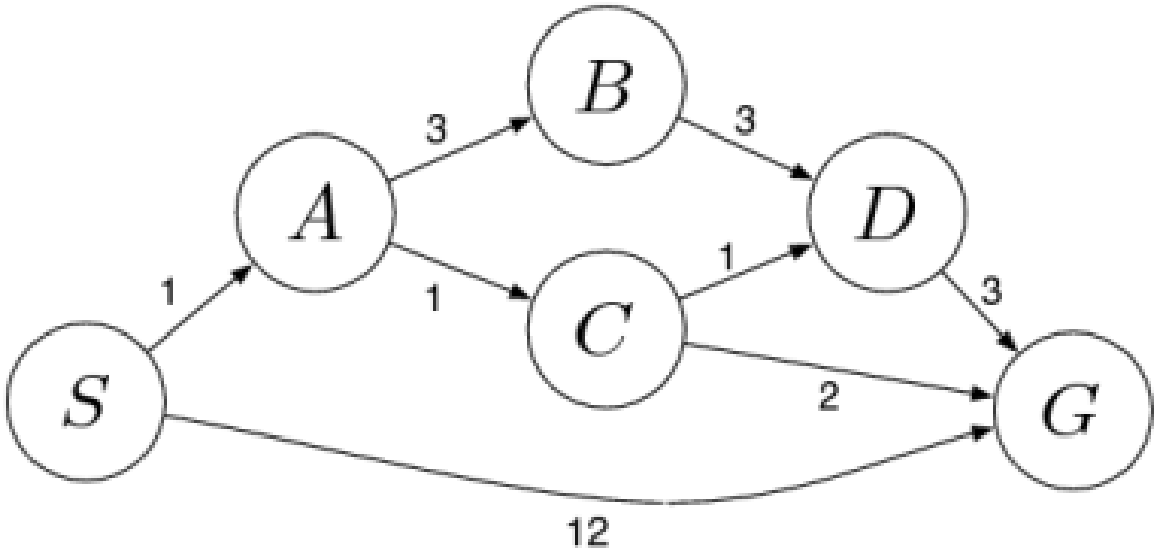


Figure 3.15 Part of the Romania state space, selected to illustrate uniform-cost search.

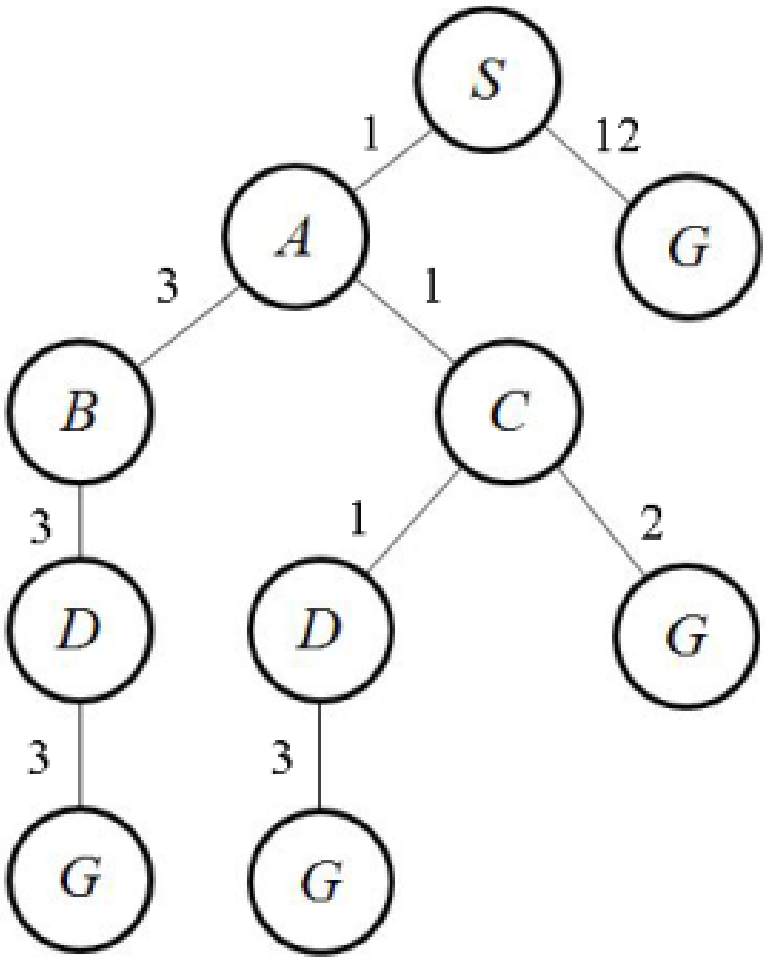
[Russell and Norvig]

The problem is to get from Sibiu to Bucharest. The successors of Sibiu are Rimnicu Vilcea and Fagaras, with costs 80 and 99, respectively. The least-cost node, Rimnicu Vilcea, is expanded next, adding Pitesti with cost 80 + 97=177. The least-cost node is now Fagaras, so it is expanded, adding Bucharest with cost 99+211=310. Now a goal node has been generated, but uniform-cost search keeps going, choosing Pitesti for expansion and adding a second path to Bucharest with cost 80+97+101= 278. Now the algorithm checks to see if this new path is better than the old one; it is, so the old one is discarded. Bucharest, now with g-cost 278, is selected for expansion and the solution is returned.

UCS: example



UCS: example



UCS algorithm

Algorithm

Add s_{start} to *frontier* (priority queue)

Repeat until *frontier* is empty:

Remove s with smallest priority p from frontier

If Goal(s): **return** solution

Add s to *explored*

For each action $a \in Actions(s)$:

Get successor $s' \leftarrow Succ(s, a)$
If s' already in *explored* : **continue**
Else update *frontier* with s' and priority $p + Cost(s, a)$

UCS algorithm

Algorithm

Add s_{start} to *frontier* (priority queue)

Repeat until *frontier* is empty:

  Remove s with smallest priority p from frontier

  If Goal(s): **return** solution

  Add s to *explored*

  For each action $a \in Actions(s)$:

    Get successor $s' \leftarrow Succ(s, a)$
    If s' already in *explored* : **continue**
    Else update *frontier* with s' and priority $p + Cost(s, a)$

At any given point in the execution, the algorithm never expands a node which has a cost greater than the cost of the shortest path in the graph.

Analysis of uniform cost search

- UCS is optimal
- UCS does not care about the number of steps a path has, but only about their total cost: infinite loops!
- Completeness is guaranteed if $c_{ij} \geq \epsilon \; \forall x, y$ for some small constant $\epsilon > 0$
- When all step costs are the same, UCS is similar to BFS, except that BFS stops as soon as it generates a goal, whereas UCS examines all the nodes at the goal's depth to see if one has a lower cost.

Analysis of uniform cost search

- UCS is optimal
- UCS does not care about the number of steps a path has, but only about their total cost: infinite loops!
- Completeness is guaranteed if $c_{ij} \geq \epsilon \ \forall x, y$ for some small constant $\epsilon > 0$
- When all step costs are the same, UCS is similar to BFS, except that BFS stops as soon as it generates a goal, whereas UCS examines all the nodes at the goal's depth to see if one has a lower cost.

It is easy to see that uniform-cost search is optimal in general. First, we observe that whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found (were this not the case, there would have to be another frontier node n' on the optimal path from the start node to n; by definition, n' would have lower g-cost than n and would have been selected first). Then, because step costs are nonnegative, paths never get shorter as nodes are added. These two facts together imply that uniform-cost search expands nodes in order of their optimal path cost. Hence, the first goal node selected for expansion must be the optimal solution. Uniform-cost search does not care about the number of steps a path has, but only about their total cost. Therefore, it will get stuck in an infinite loop if there is a path with an infinite sequence of zero-cost actions. Completeness is guaranteed provided the cost of every step exceeds some small positive constant ϵ.

Summary

- Tree search: memory efficient, suitable for huge state spaces (to the extent anything works)
- State: summary of past actions sufficient to choose future actions optimally
- Graph search: dynamic programming and uniform cost search construct optimal paths

Summary

- Tree search: memory efficient, suitable for huge state spaces (to the extent anything works)
- State: summary of past actions sufficient to choose future actions optimally
- Graph search: dynamic programming and uniform cost search construct optimal paths

The algorithms we have covered in the past two lectures are usually referred to as uninformed search. Tree search algorithms are the simplest methods: just try exploring all possible states and actions. With backtracking search and DFS with iterative deepening, we can scale up to huge state spaces since the memory usage only depends on the number of actions in the solution path. Of course, these algorithms necessarily take exponential time in the worst case. To do better, we presented the idea of a state, which contains all the information about the past to act optimally in the future. With an appropriately defined state, we can apply either dynamic programming or UCS, which have complementary strengths. The former handles negative action costs and the latter handles cycles.

Question

Suppose we want to travel from city 1 to city n (going only forward) and back to city 1 (only going backward). It costs $c_{ij} \geq 0$ to go from i to j. Which of the following algorithms will find the minimum cost path (select all that apply)?

Question

- Depth-first search
- Breadth-first search
- Dynamic programming
- Uniform cost search

Question

Suppose we want to travel from city 1 to city n (going only forward) and back to city 1 (only going backward). It costs $c_{ij} \geq 0$ to go from i to j. Which of the following algorithms will find the minimum cost path (select all that apply)?

Question
- Depth-first search
- Breadth-first search
- Dynamic programming
- Uniform cost search

Recall the various assumptions of the algorithms. DFS won't work because it assumes all edge costs are zero. BFS also won't work because it assumes all edge costs are the same. Dynamic programming will work because the graph is acyclic. Uniform cost search will also work because all the edge costs are non-negative.

Uniformed vs Informed

Uninformed search strategies

- Also known as blind search, uninformed search strategies use no information about the likely direction of the goal node(s)
- Uninformed search methods: Breadth-first, depth-first, depth-limited, uniform-cost, depth-first iterative deepening,...

Informed search strategies

- Also known as heuristic search, informed search strategies use information about the domain to (try to) head in the direction of the goal node(s)
- Informed search methods: Hill climbing, best-first, greedy search, beam search, A, A*,...

Informed search

Goal make UCS faster

Problem UCS orders states by cost from s_{start} to s

Idea take into account cost from s to s_{goal}

Best-first node n is selected for expansion based on an evaluation function $f(n)$

Informed search

Goal make UCS faster

Problem UCS orders states by cost from s_{start} to s

Idea take into account cost from s to s_{goal}

Best-first node n is selected for expansion based on an evaluation function $f(n)$

Now our goal is to make UCS faster. If we look at the UCS algorithm, we see that it explores states based on how far they are away from the start state. As a result, it will explore many states which are close to the start state, but actually moving farther away from the goal. Intuitively, we'd like to bias UCS towards exploring states which are closer to the goal: best-first search. Best-first search is an algorithm in which a node n is selected for expansion based on an evaluation function, $f(n)$. The evaluation function is construed as a cost estimate, so the node with the lowest evaluation is expanded first. The implementation of best-first graph search is identical to that for uniform-cost search, except for the use of f instead of g to order the priority queue. The choice of f determines the search strategy.

Heuristics

Definition

A heuristic $h(n)$ is the estimated cost of the cheapest path from the state at node n to a goal state.

- Unlike $g(n)$, it depends only on the state at that node
- Encode additional knowledge of the problem
- Arbitrary and nonnegative
- Constraint: if n is a goal node, then $h(n) = 0$

Heuristics

Definition

A heuristic $h(n)$ is the estimated cost of the cheapest path from the state at node n to a goal state.

- Unlike $g(n)$, it depends only on the state at that node
- Encode additional knowledge of the problem
- Arbitrary and nonnegative
- Constraint: if n is a goal node, then $h(n) = 0$

Most best-first algorithms include as a component of f a heuristic function, denoted $h(n)$: $h(n)$ is the estimated cost of the cheapest path from the state at node n to a goal state. (Notice that $h(n)$ takes a node as input, but, unlike $g(n)$, it depends only on the state at that node). Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm. We consider them to be arbitrary, nonnegative, problem-specific functions, with one constraint: if n is a goal node, then $h(n) = 0$.

Greedy best-first search

A best-first search that uses h to select the next node to expand: greedy search.

Greedy search

$$f(n) = h(n)$$

Is greedy search

- complete? NO (be careful with loops)
- optimal? NO

Greedy best-first search

A best-first search that uses h to select the next node to expand: greedy search.

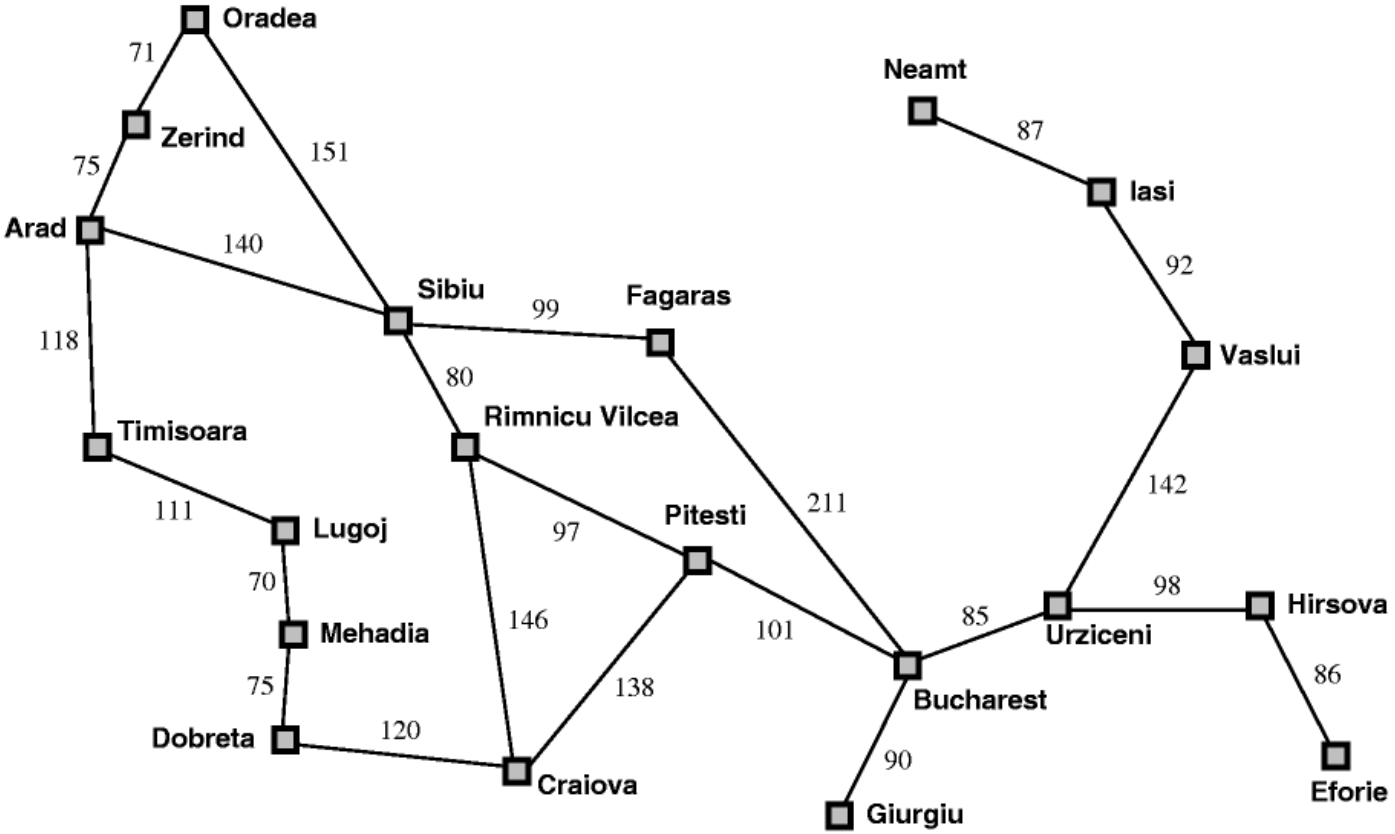
Greedy search

$$f(n) = h(n)$$

Is greedy search
- complete? NO (be careful with loops)
- optimal? NO

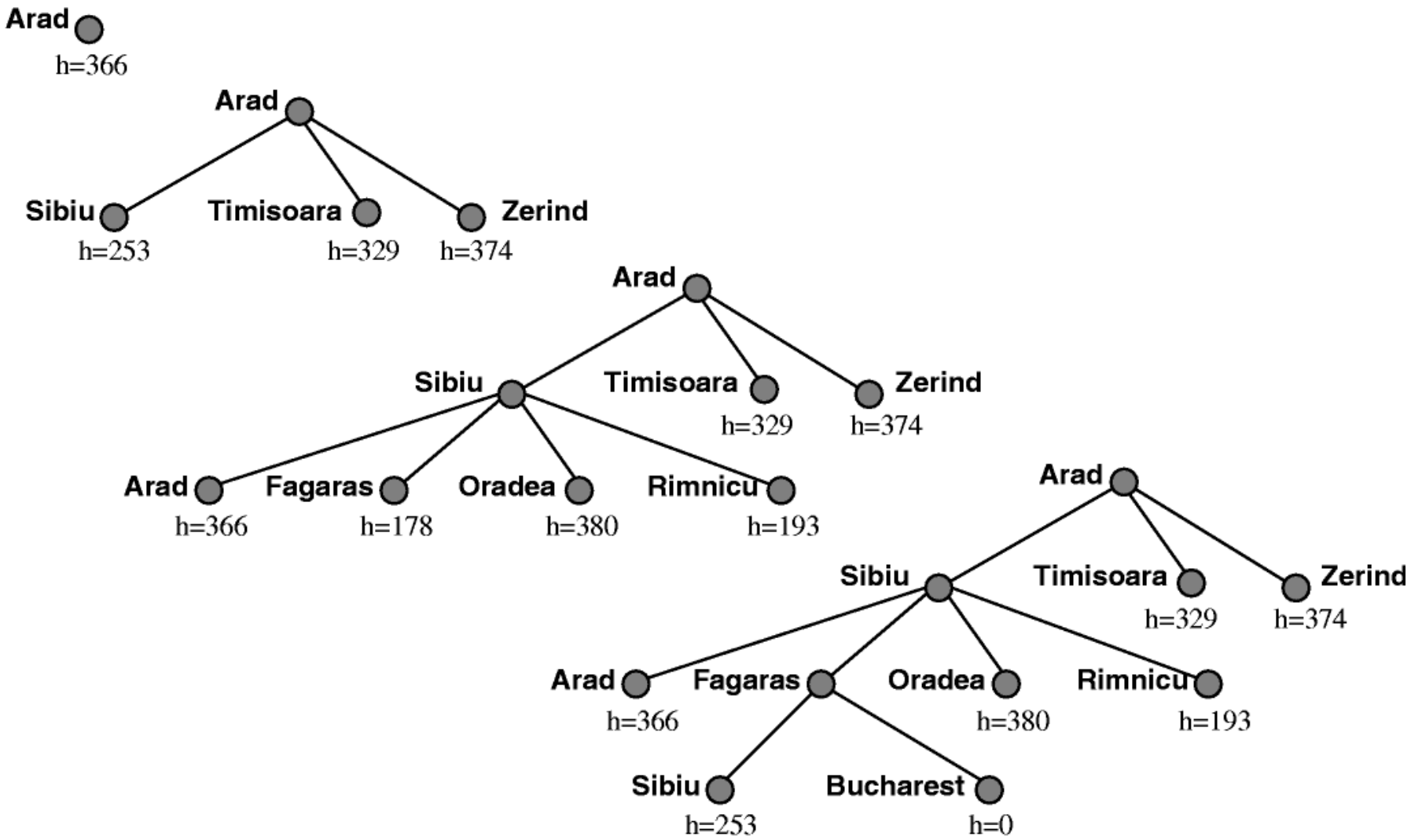
Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.

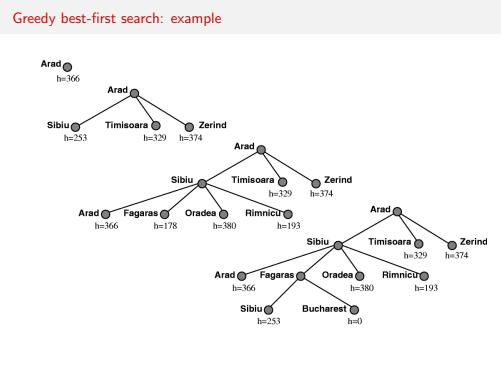
Greedy best-first search: example



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: example





For this problem, greedy search leads to a minimal cost search because no node off the solution path is expanded. However, it does not find the optimal path: the path it found via Sibiu and Fagaras to Bucharest is 32 miles longer than the path through Pimnicu Vilcea and Pitesti. Hence, the algorithm always chooses what looks locally best, rather than worrying about whether or not it will be best in the long run. (This is why its called greedy search.) Greedy search is susceptible to false starts. Consider the problem of getting from Iasi to Fagaras. *h* suggests that Neamt be expanded first, but it is a dead-end. The solution is to go first to Vaslui and then continue to Urziceni, Bucharest and Fagaras. Note that if we are not careful to detect repeated states, the solution will never be found - the search will oscillate between Neamt and Iasi. Greedy search resembles DFS in the way that it prefers to follow a single path to the goal and backup only when a deadend is encountered. It suffers from the same defects as DFS - it is not optimal and it is incomplete because it can start down an infinite path and never try other possibilities.

Next lecture

We will continue with search algorithms:

- A^*
- Relaxation. Heuristics.