

Problem Solving and Search

Geraint A. Wiggins
Professor of Computational Creativity
Department of Computer Science
Vrije Universiteit Brussel

What is problem solving?



- An agent can act by
 - establishing goals that result in solving a problem
 - considering sequences of actions that might achieve those goals
- To automate this, we need to formulate the problem
- The process of considering action sequences within a problem formulation is called search
- This part of the module is about
 - problem-solving agents
 - problem types
 - problem formulation
 - example problems
 - search strategies

An Agent on Holiday



Example

- An agent is in Arad, Romania, at the end of a touring holiday. It has a ticket to fly out of Bucharest the following day. The ticket is non-refundable, the agent's visa is about to expire and there are no seats available on flights for six weeks.
- The agent's performance measure includes
 - cost of ticket
 - avoidance of arrest and/or deportation
 - improvement of suntan
 - seeing more sights
- Lots of actions are possible to score highly on the performance measure, but the seriousness of the agent's situation means that the agent should adopt the *goal* of getting to Bucharest
 - Agent can reject any actions that fail to meet this goal

A problem-solving agent



- Goals help to organise behaviour
 - goal formulation is the first step to problem solving
 - a goal is a desired state of (part of) the world
 - though it does not usually specify the state of the whole world
 - alternatively, a goal can be thought of as the set of world states in which it is satisfied
- Actions cause transitions between world states
 - ▶ The agent has to work out which action(s) will get it to the goal state
- Problem formulation is exhaustive analysis of possible actions and states
- The agent needs to consider possible actions and the resulting states
 - put left foot forward 18 inches??
 - get cab to next town??

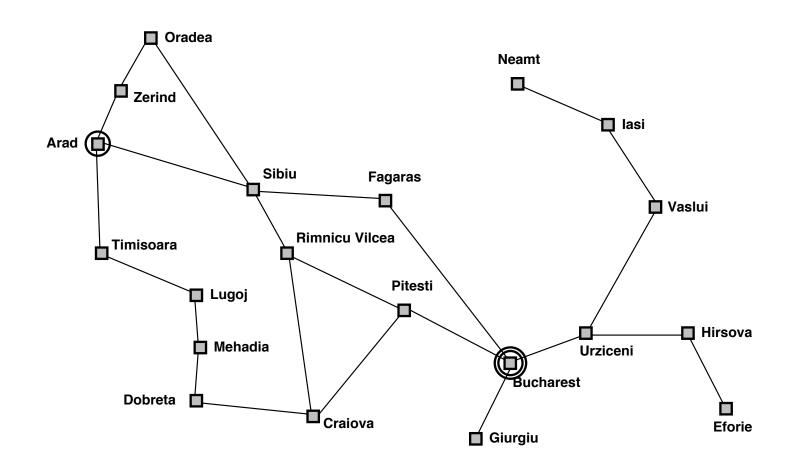
A problem-solving agent



- Our holiday agent has adopted the goal of driving to Bucharest
- There are three roads out of Arad: to Sibiu, Timisoara and Zerind
 - ▶ None of these reaches the goal. Which road should the agent follow?
- Without additional knowledge about Romania, the agent cannot choose the best road to take: it can only make a random choice
- However, if the agent has a map of Romania, it knows which states it can get to and which actions it can perform

Exercise: route planning on a map





- Find a route from Arad to Bucharest.
 - Is this the optimal route?
 - ▶ How did you go about finding the route?

A Problem-Solving Agent



- The agent can use the map information
 - consider subsequent stages of a hypothetical journey through each of the 3 nearest towns, to try to find a route that eventually leads to Bucharest
- Once it has found a path on the map from Arad to Bucharest, it can achieve its goal locally by driving to each town on the route
- More generally, with several immediate options of unknown value, the agent can decide what to do by searching
 - assessing different possible sequences of actions that lead to states of known utility (goal states)
 - choosing the best one

Design Process



- To design a problem-solving program
 - ▶ FORMULATE: Formulate problem, goal and starting conditions
 - ▶ SEARCH: Use search algorithm to find action sequence that realises goal from starting conditions
 - ▶ EXECUTE: Use solution to guide actions
- To design a problem-solving agent
 - Do the above in the context of a changing world

THE ART OF AI PROBLEM SOLVING IS DECIDING WHAT GOES INTO THE DESCRIPTION OF THE OPERATORS AND STATES, AND WHAT IS LEFT OUT

Search Algorithms



- A search algorithm takes a problem as input, formulated as
 - a set of goals to be achieved
 - a set of possible actions to be applied
- It returns a solution in the form of an action sequence or plan
- Once a solution is found, the chosen actions can be carried out
 - ▶ the execution phase
- In the agent context, this becomes complicated because of interaction between agents and other dynamic things in the world
 - so we consider the search on its own first

Well-defined problems



- Basic problem elements are states and actions
- Formally, we need
 - initial state: the world the agent starts in
 - operators: describe actions in terms of states reached by applying them
 - goal state: the world that the agent is trying to be in
- The initial state and operators define the state space of the problem
 - the set of all states reachable from the initial state and any sequence of actions
- To determine when the goal state is reached, we need
 - goal test: a boolean function over states, true if input is the goal state

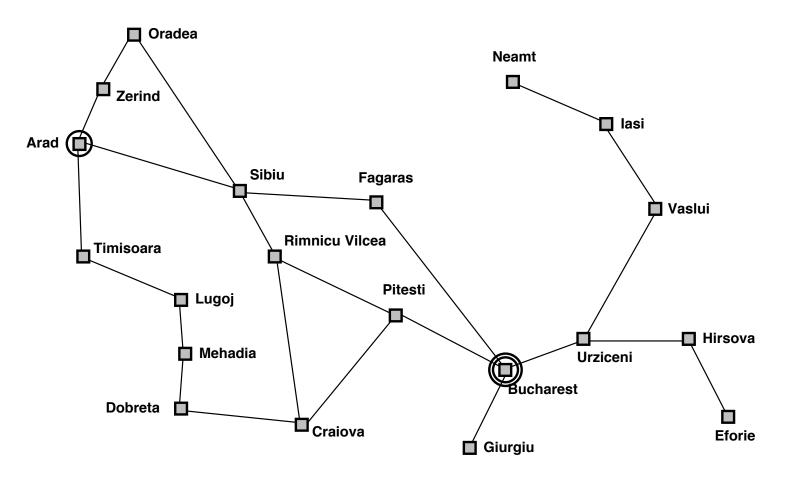
Searching for solutions



- A solution is a sequence of operators leading from start to goal state
- Given a problem formulation, we have to map out the state space
 - while we are searching the state space it's sometimes called the search space
- Generate action sequences, using an agenda:
 - start at initial state
 - check whether in goal state
 - if yes, we're done
 - otherwise, consider other states by applying the operators to the present state to generate a new set of states
 - This is called expanding the state



- It's helpful to think of the state space as structured into a tree
 - actions correspond with arcs
 - states correspond with nodes

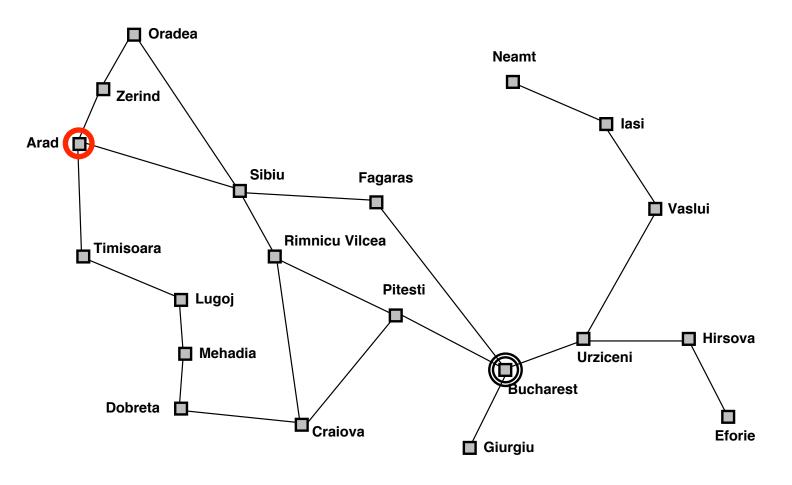




- It's helpful to think of the state space as structured into a tree
 - actions correspond with arcs

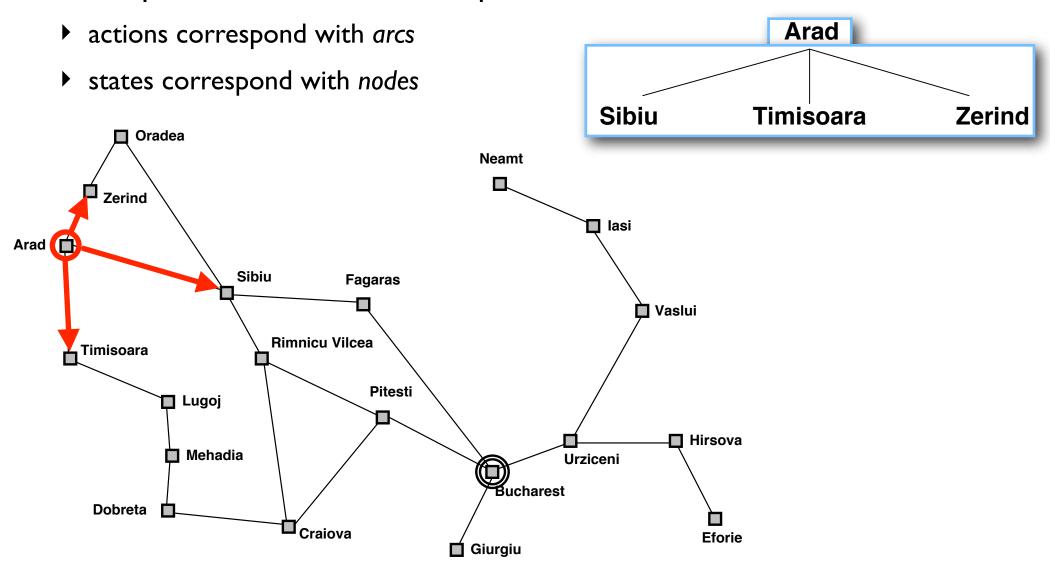
Arad

states correspond with nodes



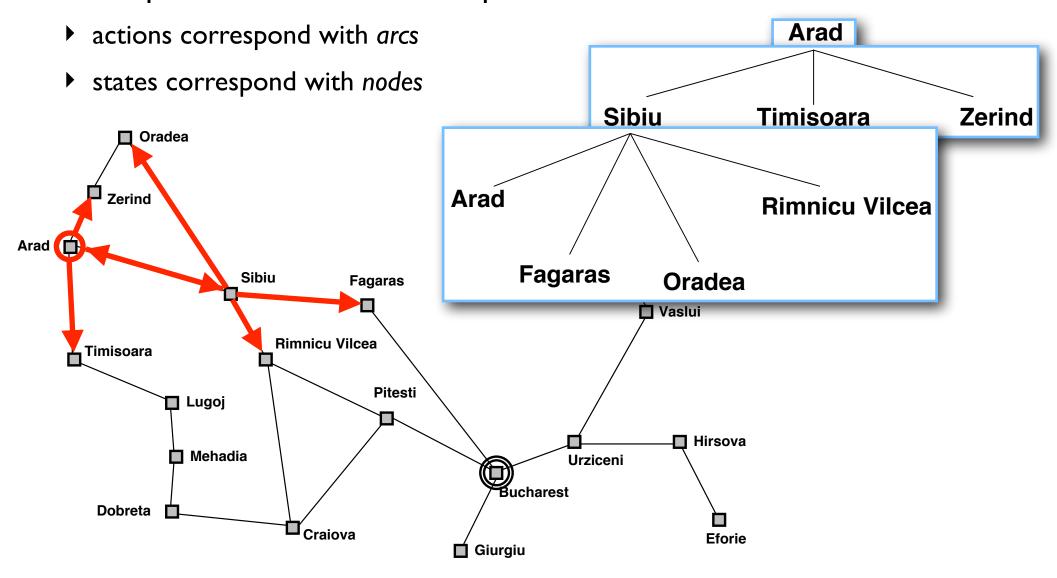


• It's helpful to think of the state space as structured into a tree



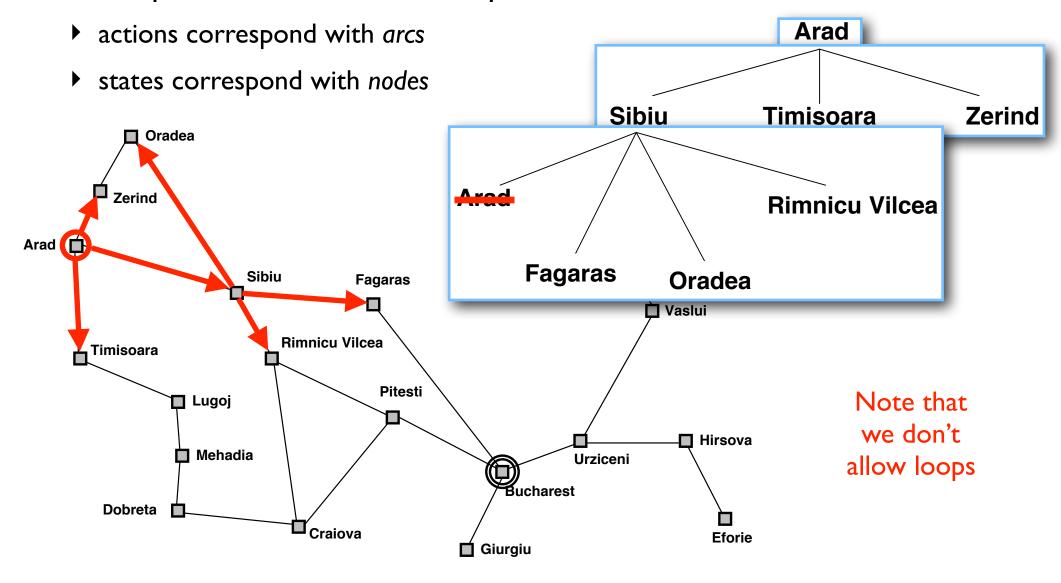


• It's helpful to think of the state space as structured into a tree

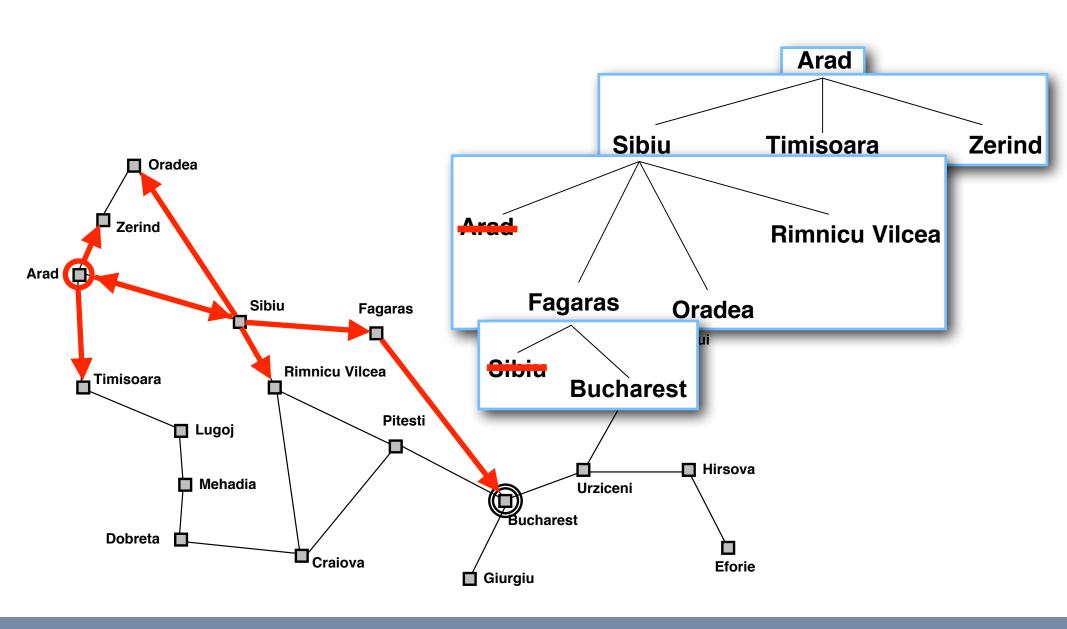




• It's helpful to think of the state space as structured into a tree

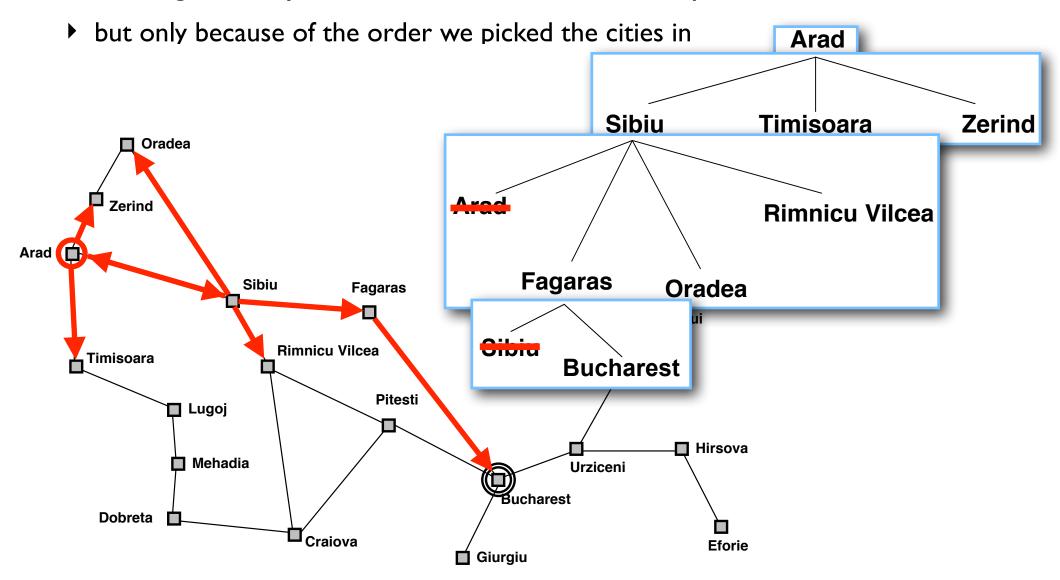








• Searching this way takes us to Bucharest in 3 steps



Clear thinking: state ≠ node!



- States of the world and nodes in the search tree are not the same thing
 - A state is a (representation of) a problem configuration
 - A node is a data structure used by a search algorithm
 - constitutes part of a search tree
 - includes parent, action, children, depth (for IDS), path cost (for UCS)
 - States do not formally have parents, actions, children, depth, or path cost!
- The Expand function creates new *nodes*, filling in the various fields and using the operators of the problem formulation to approximate the corresponding *states*

Search Algorithm Framework



- The Queuing-Fn function determines what kind of search we are doing
- ▶ The Expand function produces a set of states achieved by applying all possible actions to the given state
- This is generally called Agenda-based search

Search Algorithm Framework



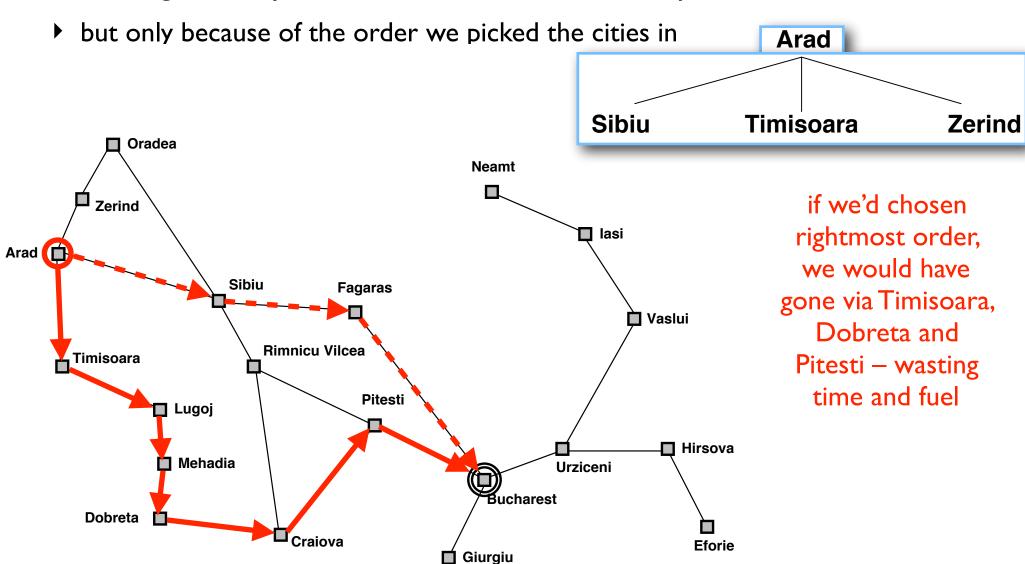
• Alternatively, in Prolog:

```
agendaSearch( [First|_], [First] ) :-
    goalTest( First ).
agendaSearch( [First|Rest], [First|Answer] ) :-
    expand( First, MidAgenda ),
    queuingFn( Rest, MidAgenda, Agenda ),
    agendaSearch( Agenda, Answer ).
```

- ▶ The Queuing-Fn function determines what kind of search we are doing
- ▶ The Expand function produces a set of states achieved by applying all possible actions to the given state
- ▶ This is generally called Agenda-based search



Searching this way takes us to Bucharest in 3 steps





- This is an example of Depth-First Search (DFS)
 - we construct the search tree by going down it, only going across it when going down fails
 - depth-first search can fail when the tree is infinitely deep
 - the space contains cycles or is generated from continuous values
 - depth-first search depends (crucially) on the order in which operators are applied/nodes are expanded
 - it may not find things that are present
 - it may find long action sequences before short ones (usually, that's unhelpful)
- In terms of agendas, we implement DFS with a Queuing-Fn that
 - removes the current node from the agenda
 - appends its children to the front of the agenda
 - Inote that the agenda is rotating all the time, always dealing with its first item

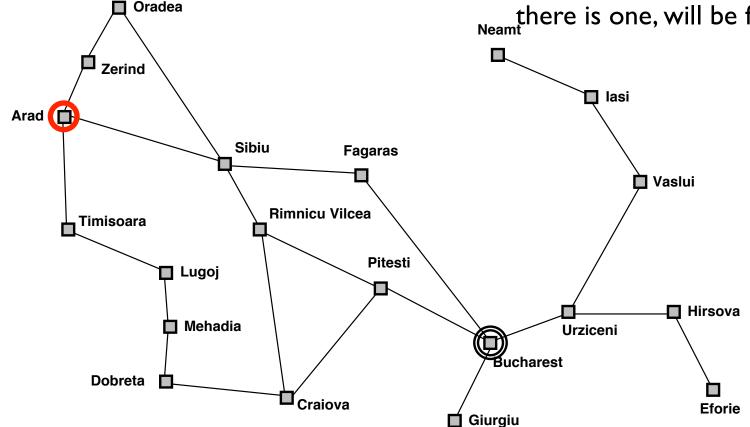


- An alternative is Breadth-First Search (BFS)
 - we construct the search tree by going across it, repeatedly, replacing each node with its children
 - breadth-first search guarantees to find the solution if there is one
 - breadth-first search guarantees to find the shortest action sequence
 - but it causes a combinatorial explosion of nodes in the agenda, and so is unsuitable for very large search spaces
 - most real AI problems have very large search spaces
- In terms of agendas, we implement BFS with a Queuing-Fn that
 - removes the current node from the agenda
 - appends its children to the back of the agenda
 - note that the agenda is rotating all the time, always dealing with its first item



• Here the different depths are given different colours

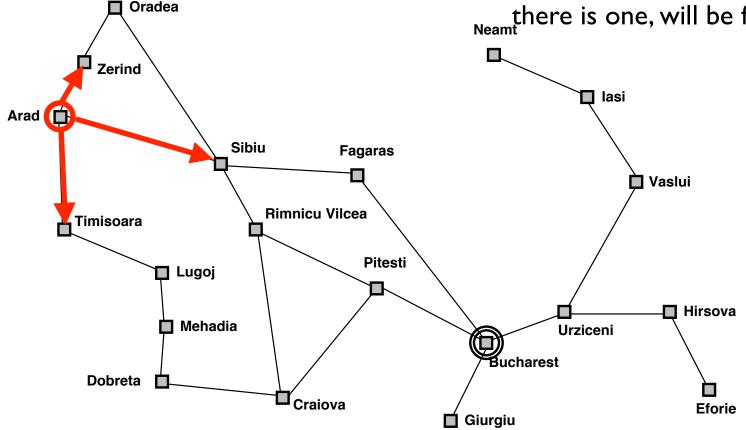
we can see the different layers of the tree as the search proceeds





• Here the different depths are given different colours

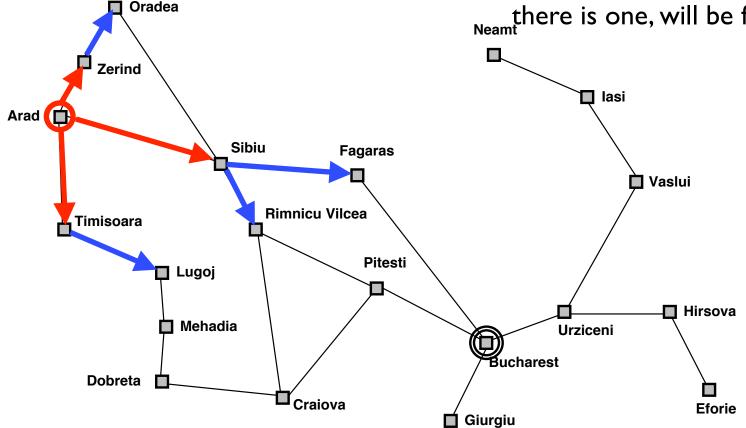
we can see the different layers of the tree as the search proceeds





• Here the different depths are given different colours

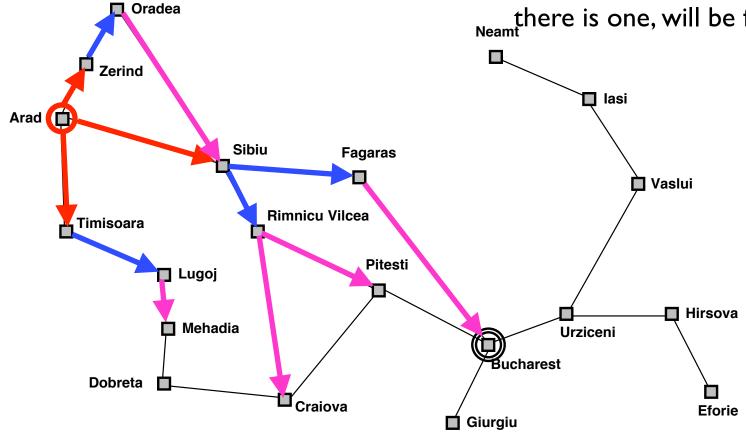
we can see the different layers of the tree as the search proceeds





• Here the different depths are given different colours

we can see the different layers of the tree as the search proceeds



Measuring search effectiveness

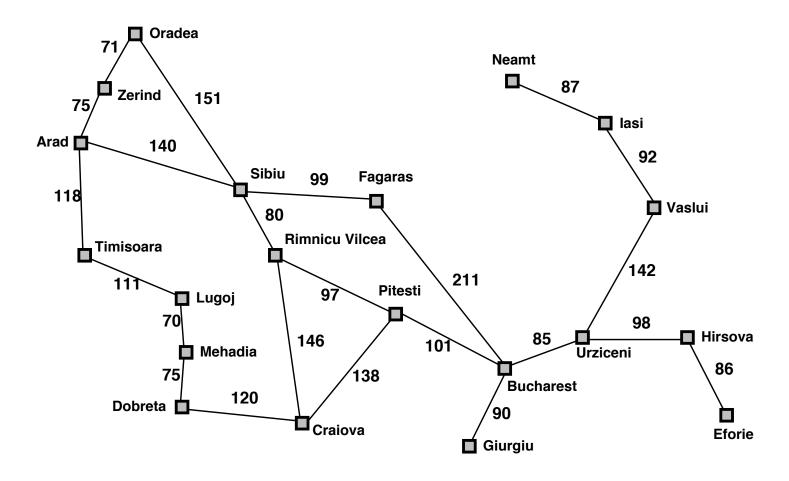


- To determine the utility of a particular path, we need
 - cost: a function that assigns a cost to a path (to differentiate between solutions)
- To evaluate a search strategy or algorithm, need to ask
 - does it find a solution?
 - is it a good solution (one with a low path cost)?
 - is it the best solution (optimal)?
 - how much time and memory are required to find a solution?
- Effectiveness of search depends on the objectives. Compare
 - searching for a route to Bucharest (many routes may be satisfactory)
 - searching for the next prime number (must be a true prime, and there's only I)

Measuring search effectiveness



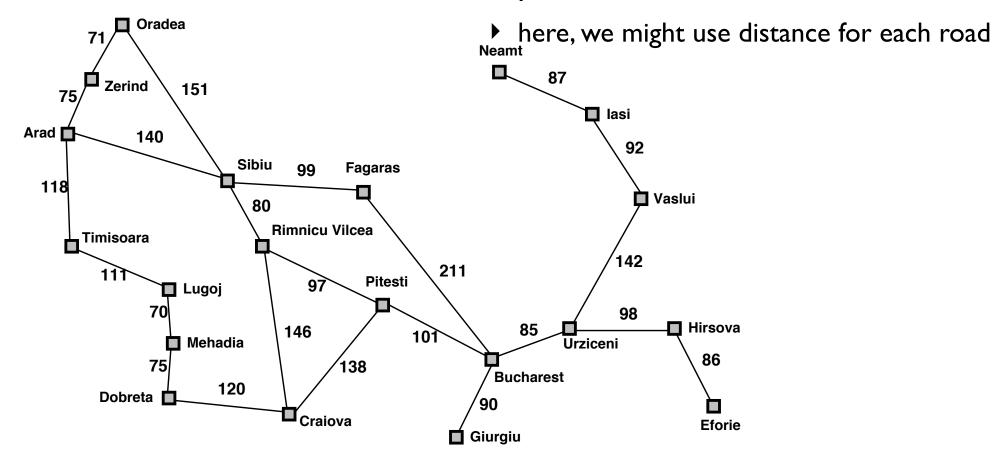
- Now consider the efficiency of the planned route
 - going via Sibiu and Fagaras has the fewest towns
 - but going via Sibiu and Pitesti is shorter



Measuring search effectiveness



 We can compute the *utility* of a route by a function which implements the performance measure, in terms of the problem formulation

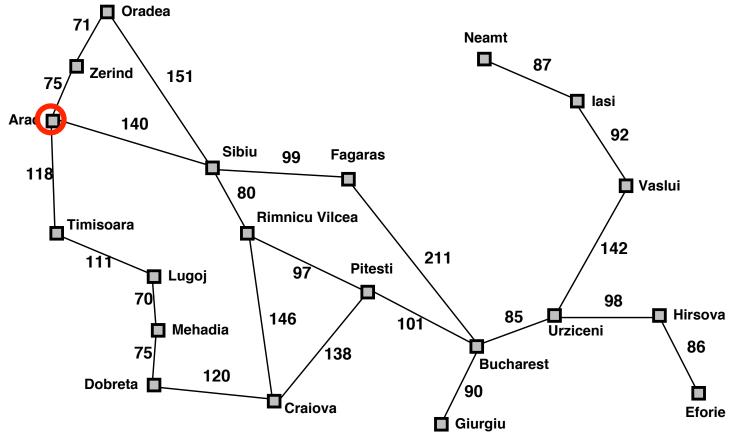




- We can implement this using Uniform-Cost Search (UCS)
 - Also called Best-First Search
 - we construct the search tree by picking the best node in the agenda after each expansion, according to the utility function, expanding it, and so on
 - best-first search guarantees to find the solution if there is one
 - best-first search guarantees to find the best action sequence in terms of utility
 - but it cannot guarantee to use less memory than BFS
 - most real Al problems have very large search spaces
- In terms of agendas, implement UCS/BestFS with a Queuing-Fn that
 - removes the current node from the agenda
 - appends its children to the agenda and then sorts the whole agenda by the utility function
 - and stores the cost-so-far at each node in the agenda, to avoid recalculation
 - note that the agenda is rotating all the time, always dealing with its first item

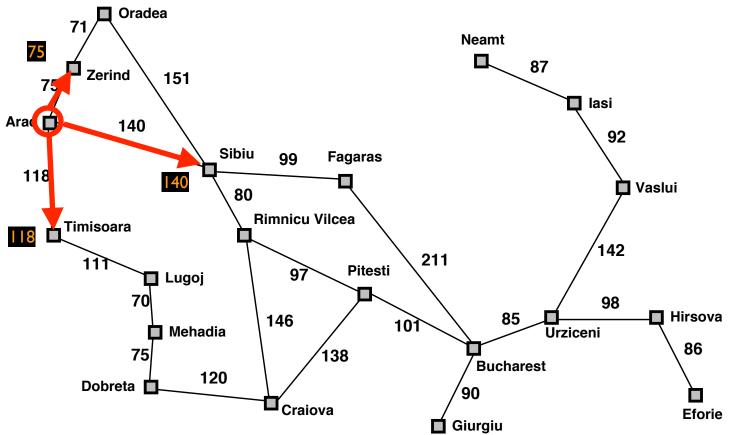


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



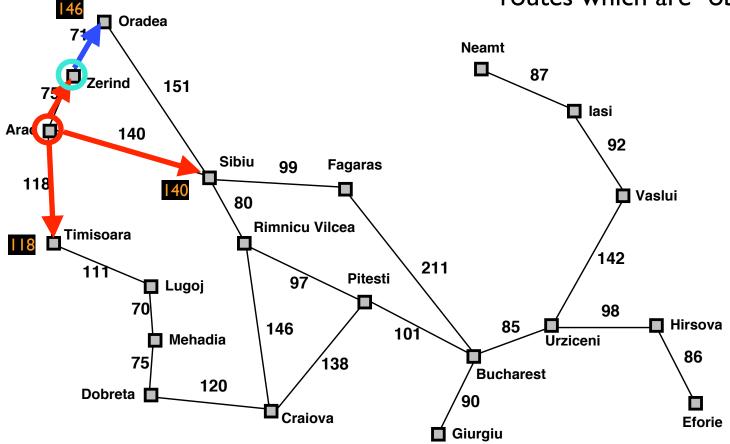


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



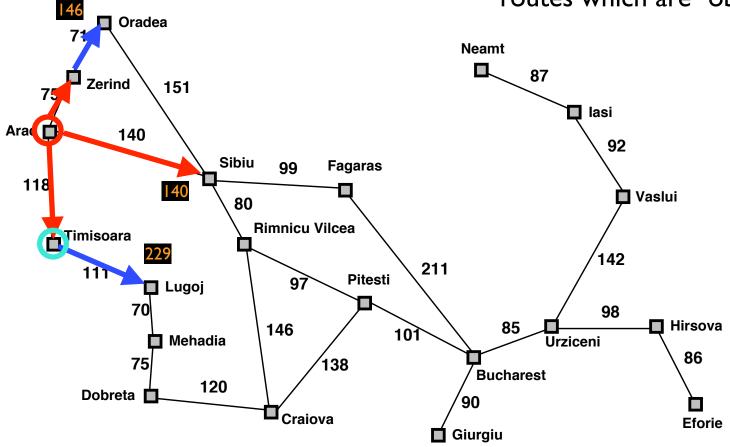


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



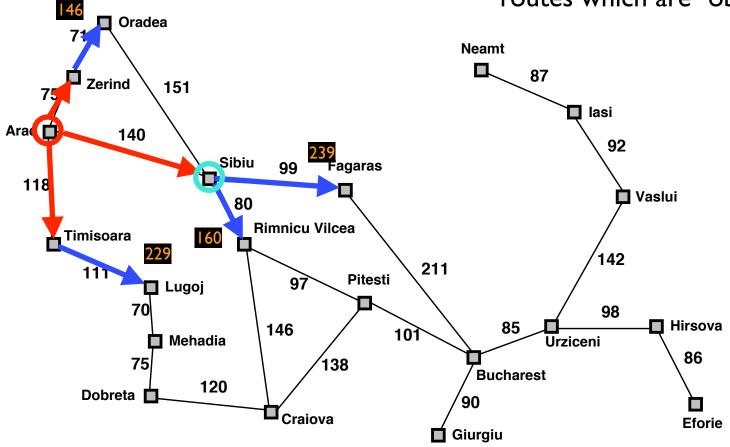


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



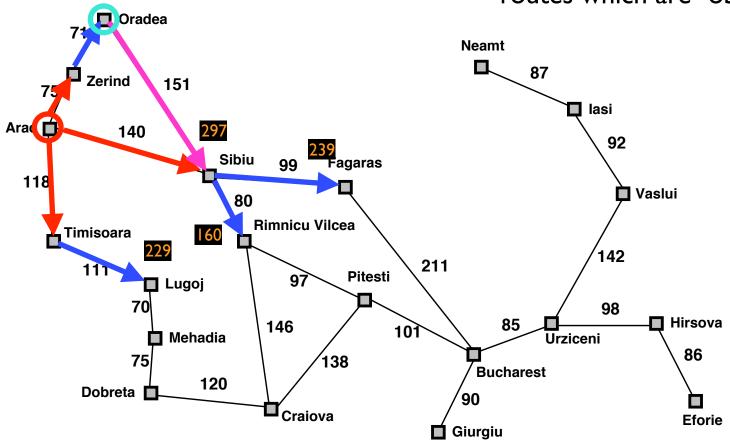


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



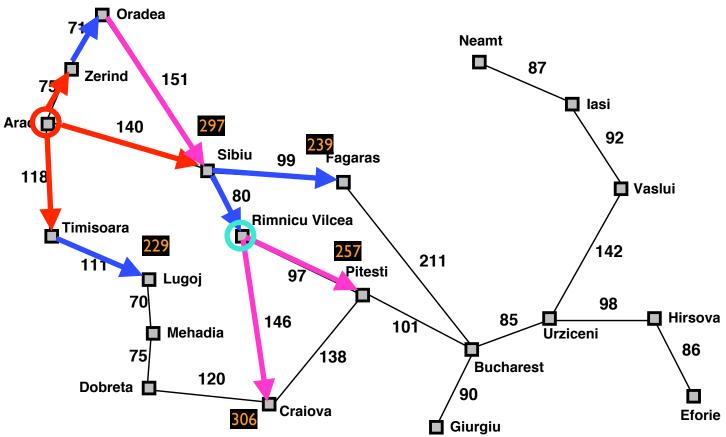


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



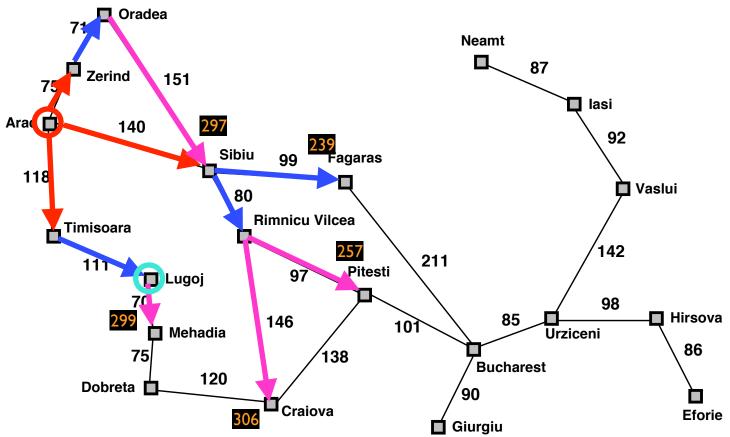


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



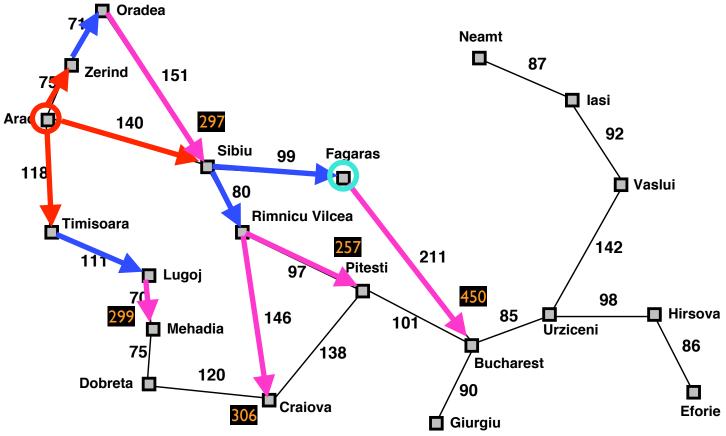


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



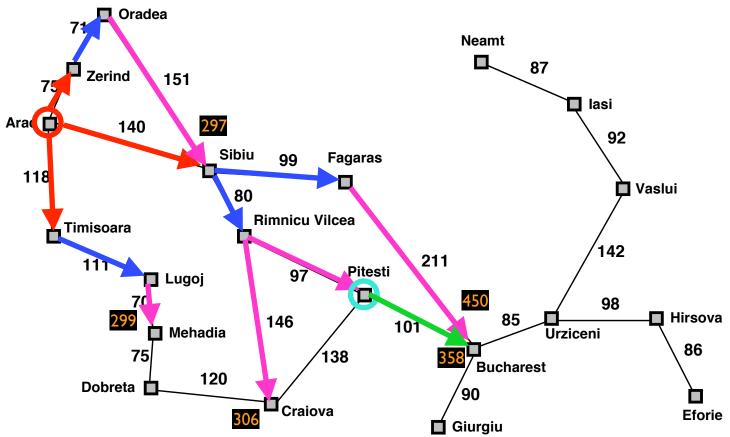


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



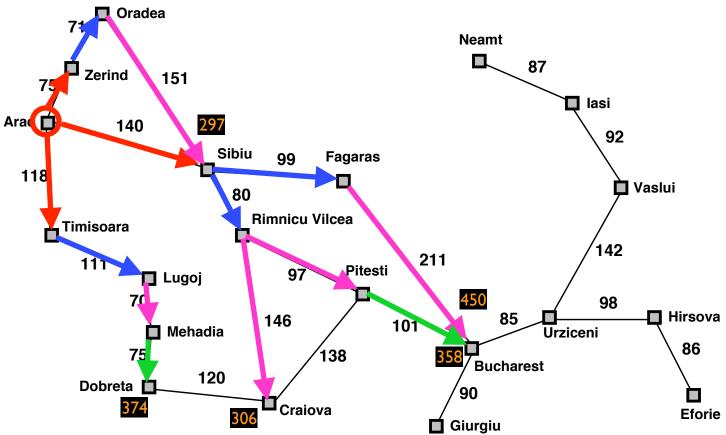


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



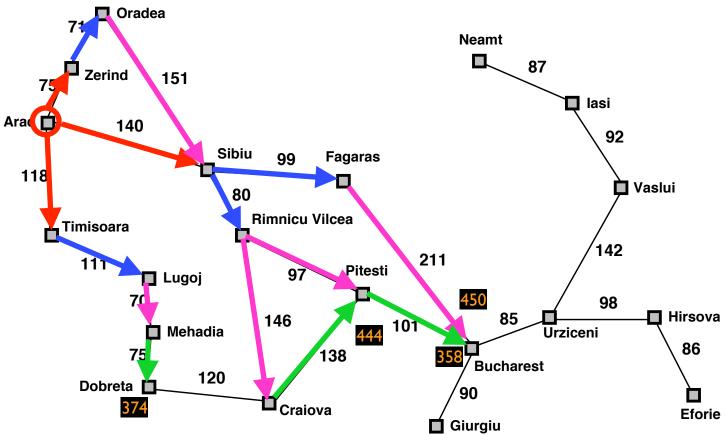


- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong





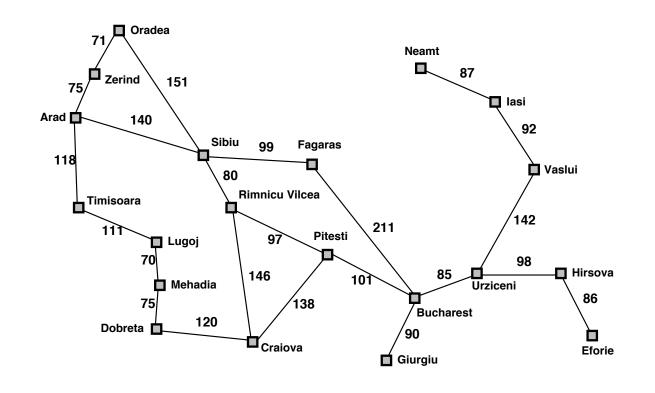
- Using UCS to consider real distances
 - ▶ Here BFS would get us a wrong answer
 - UCS is correct, but spends time searching routes which are "obviously" wrong



Formulating the problem



- Now let's look at the full specification of this problem
 - Initial state: in Arad
 - Operator: drive between two cities
 - ▶ State space: cities (20)
 - Goal test: in Bucharest?
- Path cost: one of
 - total milage
 - total expected travel time
 - step count in search
- Many details are left out
 - abstraction



Analysing search spaces



- A strategy is defined by its order of node expansion
- Uninformed strategies use only information in the problem formulation
 - Sometimes called blind search or brute-force search
 - Breadth-first search
 - Depth-first search
 - Best-first search/Uniform-cost search (this "feels" informed, but isn't, by this definition)
- Informed or heuristic search strategies a quality measure not strictly in the problem formulation to guide the order in which nodes are expanded
 - e.g., An informed agent might notice that the goal state, Bucharest, is southeast of Arad, and that only Sibiu is in that direction, so it is likely to be the best choice
 - Algorithm A and Algorithm A*

Analysing search spaces for cost



A strategy is evaluated along the following dimensions

completeness does it always find a solution if one exists?

time complexity number of nodes generated

space complexity maximum number of nodes in memory at once

• optimality does it always find a least-cost solution?

- Time and space complexity are measured in terms of the search tree
 - b maximum branching factor of the search tree ("bushiness" of tree)
 - ▶ d depth of the least-cost solution
 - ▶ m maximum depth of the state space (may be ∞)
- Even the best strategies need to avoid repeated nodes
 - trade-off between time and space (see Russell and Norvig)

Analysing search algorithms for cost Queen Mary University of London



- So far, we've looked at Depth first, Breadth first, and Uniform Cost search in practice
 - now, the theory

Analysing search algorithms for cost



- Depth first search (DFS)
 - ▶ Complete? No, DFS fails in infinite depth search spaces and spaces with loops
 - ▶ Time? $O(b^m)$: terrible if m is much larger than d, but if search tree is "bushy", may be much faster than breadth first
 - ▶ Space? *O(bm)*, i.e., space linear in length of action sequence! Need only store a single path from the root to the leaf node, along with remaining unexpanded sibling nodes for each node on path
 - ▶ Optimal? No
 - Pros: For problems with many solutions, DFS may be faster than BFS because it has a good chance of finding a solution after exploring only a small portion of the whole space
 - Cons: can get stuck down wrong search path

Analysing search algorithms for cost Queen Mary



- Breadth first search
 - ▶ Complete? Yes, all nodes are examined if b is finite
 - ▶ Time? $1 + b + b^2 + b^3 + ... + b^d = O(b^d)$, i.e., exponential in b
 - Space? $O(b^d)$ keeps every node in memory
 - Optimal? Only if cost = I per step, not in general
 - If there are several solutions it will find the shallowest goal first
 - Drawbacks: Memory requirement to store nodes is a big problem
 - \bullet Assuming b = 10, expansion of 1000 nodes/sec, and 100 bytes/node:

Depth	Nodes	Time	Memory	
0	1	1 millisecond	100 bytes	
2	111	0.1 seconds	11 kilobytes	
4	11,111	11 seconds	1 megabyte	
6	10^{6}	18 minutes	111 megabytes	
8	10^{8}	31 hours	11 gigabytes	
10	10^{10}	128 days	1 terabyte	
12	10^{12}	35 years	111 terabytes	
14	10^{14}	3500 years	11,111 terabytes	

Analysing search algorithms for cost



- Uniform-cost (Best first) search
 - Shares dynamic properties of DFS and BFS
 - Requires step cost information in problem formation and path cost in agenda
 - ▶ Time? Number of nodes with path cost less than optimal solution
 - ▶ Space? Number of nodes with path cost less than optimal solution
 - ▶ Optimal? Yes finds the cheapest solution as long as step cost > 0
 - First solution reported (but not necessarily encountered) is always the cheapest
 - does not explore the whole tree and avoids looping
 - but can explore the same path multiple times if there is more than one route through a node

Depth bounded search: IDS

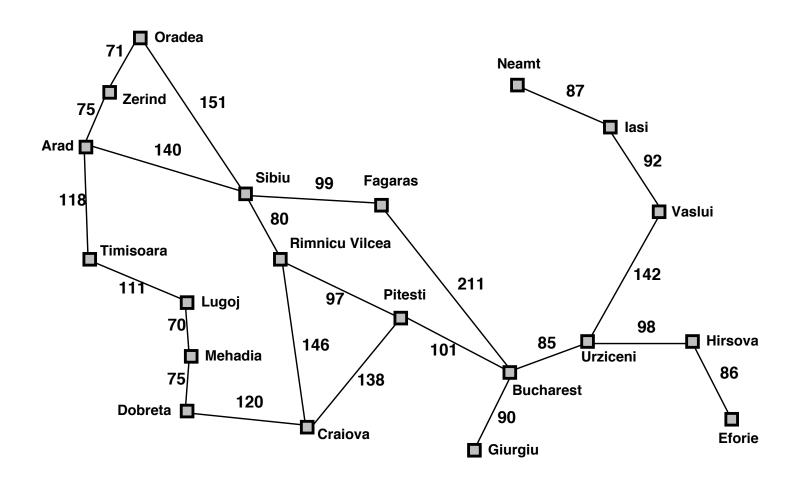


- Iterative-deepening search is a compromise between DFS and BFS
 - ▶ Do DFS, but include a depth bound, forcing BFS-like behaviour
 - In the agenda-based algorithm, we keep track of the depth of each node in the agenda, then
 - Start with depth-cutoff = I
 - Search, expanding only nodes with depth < depth-cutoff</p>
 - if unsuccessful, increment depth and repeat

Exercise: IDS on holiday



• Execute IDS for the Romanian holiday problem, on paper



Analysing search algorithms for cost Queen Mary



- Iterative Deepening Search
 - Complete? Yes
 - ▶ Time? $(d+1)b^0 + db^1 + (d-1)b^2 + ... + b^d = O(b^d)$
 - Space? $O(b^d)$
 - Optimal? Yes, if step cost = I
 - Pros: combines the benefits of BFS and DFS
 - optimal and complete (like BFS)
 - modest memory requirements (like DFS)
 - can be modified to explore uniform-cost tree
 - Cons: some (perhaps many) states are expanded multiple times
 - Although this appears wasteful, the overhead is actually quite small for most cases

Informed Search

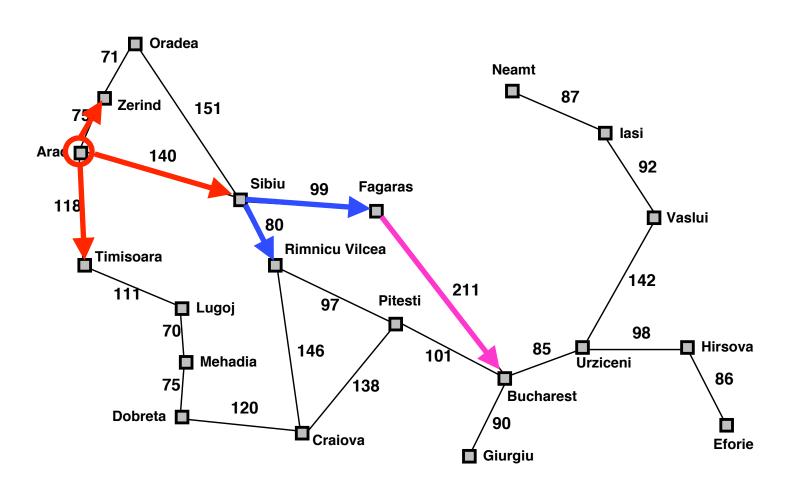


- Informed search methods use problem specific information about the state space to generate solutions more effectively
 - Sometimes called Best First Search
- Extra knowledge is provided by a heuristic which returns a number describing the desirability of expanding a given node
 - an approximation since we usually don't know best node to expand
 - nodes expanded in order of desirability (like UCS)
 - most desirable processed first
- The aim is to find low cost solutions
 - typically evaluate estimated cost of solution and try to minimise it
 - measure incorporates estimate of cost from a state to the closest goal state
 - try to expand the node closest to the goal
 - try to expand the node on the least cost solution path

Greedy, or hill-climbing, search



- The simplest kind of informed search estimates the cost from the current node to the solution and uses that to sort its agenda
 - ▶ note the difference with UCS, where we use the actual cost so far



Straight–line distanto Bucharest	ice
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
	517

Greedy, or hill-climbing, search



- Most greedy search has an agenda of length I
 - lack all but the best node is thrown away, at every cycle
 - but it can be implemented with an agenda of any length
 - the shorter the agenda, the more likely the search is to get caught in local minima, where there is a temporary dip in the solution quality
 - throwing away solutions means you can't recover if you go astray
- In the route-finding problem, a good heuristic is the straight-line distance to the goal
 - "as the crow flies"
 - this idea, that a geometrical distance estimates a distance along a graph, is the basis of many heuristics
- But because it doesn't account for cost so far, it can lead us astray

Algorithm A



- Algorithm A combines the best of UCS and Greedy search
 - UCS had "cost so far"
 - Greedy had "estimated cost remaining" or heuristic
 - Algorithm A has "cost so far" + "estimated cost remaining"
- This is usually notated as f(n) = g(n) + h(n)
 - f is the overall cost/evaluation/utility function
 - g is the cost-so-far function
 - ▶ *h* is the heuristic
- Algorithm A
 - is complete
 - but can be sub-optimal in node expansion



Algorithm A*



- Algorithm A* uses a heuristic which is admissible
 - this means that it never over-estimates the distance to the closest solution
 - in any graph representation of space, geometric distance has this property
- A* is optimal
 - In the sense that we are guaranteed a solution if there is one
 - in the sense that the fewest possible nodes will be expanded, given this problem formulation
 - NB re-formulating may produce better search

Exercise

- \blacktriangleright Using the data on the earlier slide, apply A* to the Arad-Bucharest route problem
 - in what order are the nodes expanded?

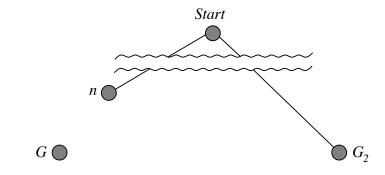
Analysing search algorithms for cost



- Proving optimality of A*
 - Suppose there is a suboptimal goal G_2 in the agenda
 - \bullet Let n be an unexpanded node on a shortest path to an optimal goal G

•
$$f(G_2) = g(G_2) + h(G_2)$$

- but $h(G_2) = 0$, as it's a goal
- - \bullet because G_2 is sub-optimal
- $g(G) \geq f(n)$
 - because h is admissible



- ▶ Because $f(G_2) \ge f(n)$, A* will never select G_2 for expansion
- The more accurate h is, the fewer unnecessary nodes will be expanded

Analysing search algorithms for cost Queen Mary



- One way to characterise the quality of a heuristic is the effective branching factor (b*)
 - ▶ if
 - the total number of nodes expanded by A^* for a particular problem is N
 - the solution depth is d
 - then
 - \bullet b* is the branching factor that a uniform tree of depth d would have to have in order to contain N nodes
 - $N = 1 + b^* + (b^*)^2 + ... + (b^*)^d$
 - \bullet e.g., if A* finds a goal at depth 5 using 52 nodes, the effective branching factor is 1.91
- Usually the effective branching factor exhibited by a given heuristic is fairly constant over a large range of problems
 - This can be used for empirical evaluation of a heuristic's usefulness

Exercise: The 8s-Puzzle



- Choosing a heuristic function for a more abstract problem
 - the 8s-puzzle
 - typical solution ≈ 20 steps
 - branching factor \approx 3 (2 if space in corner, 4 in middle, 3 otherwise)
 - exhaustive search to depth 20 examines $320 \approx 3.5 \times 109$ nodes
 - ▶ A* Heuristic must never over-estimate number of steps to goal
 - h_1 = number of tiles in wrong position
 - h_2 = sum of Manhattan/city-block distances of tiles from their goals

Choosing a heuristic for 8s-puzzle



	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
$\mid 4 \mid$	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	_	1301	211	_	1.45	1.25
18	_	3056	363	_	1.46	1.26
20	-	7276	676	_	1.47	1.27
22	_	18094	1219	_	1.48	1.28
24	_	39135	1641	_	1.48	1.26

Table comparing the search costs and effective branching factors for IDS and A* using h_1 and h_2 . Data are averaged over 100 instances of the 8-puzzle, for various solution lengths.

- h_2 is better than h_1 which is better than uninformed search
- Always use a heuristic with higher values, so long as it doesn't overestimate

Further Reading



- In Chapters 3 & 4 of the textbook you can read more about search including
 - Bidirectional search
 - Searching with partial information
 - ▶ Hill-climbing search and simulated annealing
 - Genetic algorithms
 - Online search agents