Informed search algorithms

Russell & Norvig

Section 3.5

Outline

- *Review – Why look at search?*
- Informed search
- Greedy search

What behaviour is “intelligent”?

- Which suggests intelligent behaviour?
  - Adding up a column of numbers
  - Solving a crossword puzzle
  - Calculating the weight of a cup of water
  - Baking a cake
  - Coming up with a new recipe
  - Curing cancer
  - Curing ham

What difference is there?

- What seems to be true about intelligent behaviour compared to not so intelligent?

What difference is there?

- What seems to be true about intelligent behaviour compared to not so intelligent?

- It is often behaviour that solves a problem that doesn't have a "formula" for directly coming to a solution.

What difference is there?

- What seems to be true about intelligent behaviour compared to not so intelligent?
- It is often behaviour that solves a problem that doesn't have a "formula" for directly coming to a solution.
- In short, it involves a **search** for a solution!

What type of problems involve search?

- Solving puzzles
- Playing chess
- Designing new types of machines
- Learning by watching a game
- Proving theorems
- Understanding a foreign accent
- Planning a holiday
- Diagnosing why your car won’t start

Not all search is equally intelligent

- While search is associated with intelligent problem solving, search is not necessarily enough!

- If there is readily available information that *should help* lead to a solution then ignoring it and blindly searching for a solution is not very intelligent.

Search & Intelligence

- Different tasks often use different variations of search.
- These two weeks we will look mainly at the task of solving puzzle-like problems.
- Instead of blindly searching for solutions to these problems we will talk about techniques that use available information to search for their solutions.

Informed (Heuristic) Search

- “Heuristic” can mean many different things, we will look at some of them.
- The core idea is that it guides the problem solver towards the solution
  - like someone crying out “hot”/”cold” in blind man’s bluff.
- Heuristics can be misleading.

What we will be doing

- Our primary goal is to help you to understand how you would write a system that solved problems in a more or less intelligent way.
- Our secondary goal is to indicate how we would do this in a declarative manner.

Developing a Problem Solver

- We will begin with an uninformed problem solver.
- We start with a formal definition of what it means for a sequence of states to be a solution to a problem.
  - Problems are specified by an initial state, a goal state test, & a successor relation.
  - We will incrementally transform this definition into an “informed” problem solver.

Formal Definition of "solution to problem"

solution(+problem(InitState, GoalTest), ?Solution)

Solution is a solution to a problem if and only if
Solution is a non-empty sequence of states
such that Solution's first state is the InitState of the problem,
the last state in Solution satisfies the GoalTest,
& each state in Solution is a successor of its preceding state

Refinement of Definition

- Not all formal definitions are equally useful
- The previous definition is such an example
- The problem is the "*each state in Solution is a neighbor of its preceding state"* part
- There isn't an operation that directly checks this.
- We're going to refine our definition so we have a better idea of how we check this.

Formal Definition of "solution to problem"

solution(+problem(InitState, GoalTest), ?Solution)

Solution is a solution to the problem if and only if
either Solution only contains one state, S,
and S is the InitState and it satisfies the GoalTest
or Solution contains more than one state, e.g., [S, T | RestOfSolution],
and T is a successor of S
and [T | RestOfSolution] is a solution to problem(T, GoalTest)

This is an "inductive" definition of "solution". Inductive definitions *normally* makes it easier to see if the definition is correct.

Does it in this case?

Translation into Prolog

Solution is a solution to the problem if and only if

either Solution only contains one state, S,

and S is the InitState and it satisfied the GoalTest

or Solution contains more than one state, e.g., [S | RestOfSolution],

and T is a successor of S

and RestOfSolution is a solution to problem(T, GoalTest)

*solution(problem(InitState, GoalTest), [InitState] :- GoalTest(InitState)**.*

solution(problem(InitState, GoalTest), [InitState, NextState | RestOfSolution]) :-

successor(InitState, NextState),

solution(problem(NextState, GoalTest), [NextState | RestOfSolution]).

** This is not exactly how you write this in Prolog, instead you write:

"Goal =.. [GoalTest, InitState], call(Goal)."

Example

Domain Definition:
successor(losAngeles, sanFrancisco).
successor(losAngeles, sanDiego).
successor(sanFrancisco, portland).
successor(sanFrancisco, lasVegas).
successor(portland, seattle).

Goal Definition:
reachedHome(seattle).

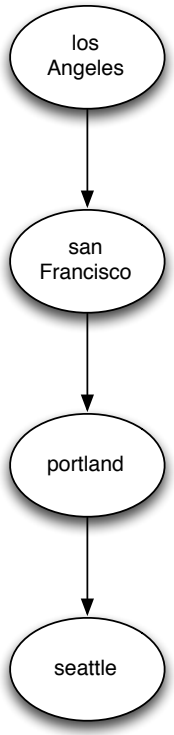
| ?- solution(problem(losAngeles, reachedHome), Solution).

Solution = [losAngeles,sanFrancisco,portland,seattle] ?

Run through example

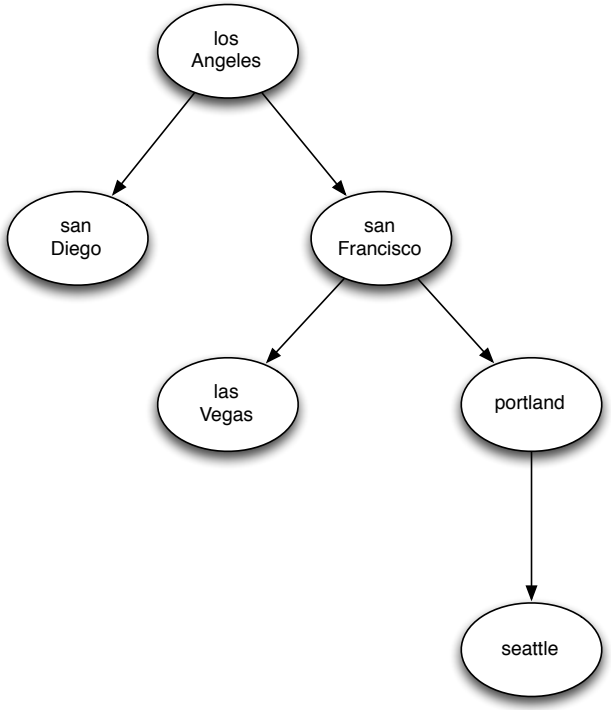
- do example in emacs under SWI Prolog

Search Space



successor(losAngeles, sanFrancisco).
successor(losAngeles, sanDiego).
successor(sanFrancisco, portland).
successor(sanFrancisco, lasVegas).
successor(portland, seattle).

What happens if clauses in different order?



successor(losAngeles, sanDiego).
successor(losAngeles, sanFrancisco).
successor(sanFrancisco, lasVegas).
successor(sanFrancisco, portland).
successor(portland, seattle).

what have we done???

- Seen that a relationship can be defined such that it can be used by prolog to generate instances of that relationship.
- We saw:
  - how that defn can be turned into prolog
  - what files we could create to run this
    - defn of relationship {simple search}
    - problem {example}
    - script to run it {script}

Tree Search

- Keeps record of current path and choice points along path (to visit if current path abandoned).
- [Can check for duplicate states along current path, avoid loops.]
- No global duplicate state checking.
- When goal state is found, solution is simply current path.

Naive *solution* implementation

- Prolog has its own search procedure for executing a program: depth-first search.

- Our naive *solution*'s search strategy is Prolog's and has all the advantages & disadvantages of depth-first search.

Status of ***Naive*** Tree Search

- Advantages:
  - Only needs to store current path
    - Linear memory costs
  - Can use simpler logic (lower costs per node)
- Disadvantages
  - Non-optimal solution (depends on strategy)
  - Repeats search for duplicate states
  - Incomplete (for infinite graphs)

Graph Search

- Primarily, does a type of breadth-first search.
- Does global check for duplicate states.
- Keeps whole search graph in memory.
- When goal state is found, solution needs to be extracted from search graph.

Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Notes:

1.Fringe is the set of leaf nodes

2.Remove-Front is the search strategy

3.Avoids redundant searches from duplicate states

Graph version of *solution*

```
/* solution(+Problem, -Solution) */
solution(problem(InitialState, Goal), Solution) :-
        solution(Goal, [node(InitialState, nil)], [ ], Solution).

/* solution(+Goal, +Fringe, +Closed, -Solution) */
solution(Goal, [node(State, ParentState) | _], Closed, Solution) :-
        call(Goal, State),
        extractSolution(ParentState, Closed, [State], Solution).

solution(Goal, [node(State, Parent) | RestNodes], Closed, Solution) :-
        findall(NeighborNode,
                newNeighborNode(State,  Closed, NeighborNode),
                NeighboringNodes),
        updateClosed(State, Closed, NewClosed),
        orderFringe(RestNodes, NeighboringNodes, NewFringe),
        solution(Goal, NewFringe, [node(State, Parent) | NewClosed], Solution).d
```

Status of Graph Search

- Possible Advantages:
  - Complete
  - Optimal
  - Only searches subspaces once
- These advantages depend upon strategy
- Disadvantages:
  - Exponential memory costs
  - More complex logic

Outline

- **Review**
- *Best-first search*
- Greedy search

Search strategies

- A search strategy is defined by the order of node expansion
- Let *g*(n) be *the cost of n's path from the initial state*.
- Assume all edge costs are 1 then:
  - Depth-first search strategy is pick node with highest g-value.
  - Breadth-first search strategy is pick node with lowest g-value.

Best-first search strategy

- Given a set of nodes on the fringe of a search, which one is best to expand next?
  - Based on what criteria?

Best-first search strategy

- Given a set of nodes on the fringe of a search, which one is best to expand next?
  - Based on what criteria?
- Criteria: expand best nodes first, i.e., those along an optimal solution path
  - How do we do that?

Best-first search strategy

- Given a set of nodes on the fringe of a search, which one is best to expand next?

- Different criteria:
  - Time to find solution
  - Quality of solution
  - Combination of both

Best-first search strategy

- How to order our selection of nodes to find either a quick solution or a good one?
- Need additional information to suggest such nodes.

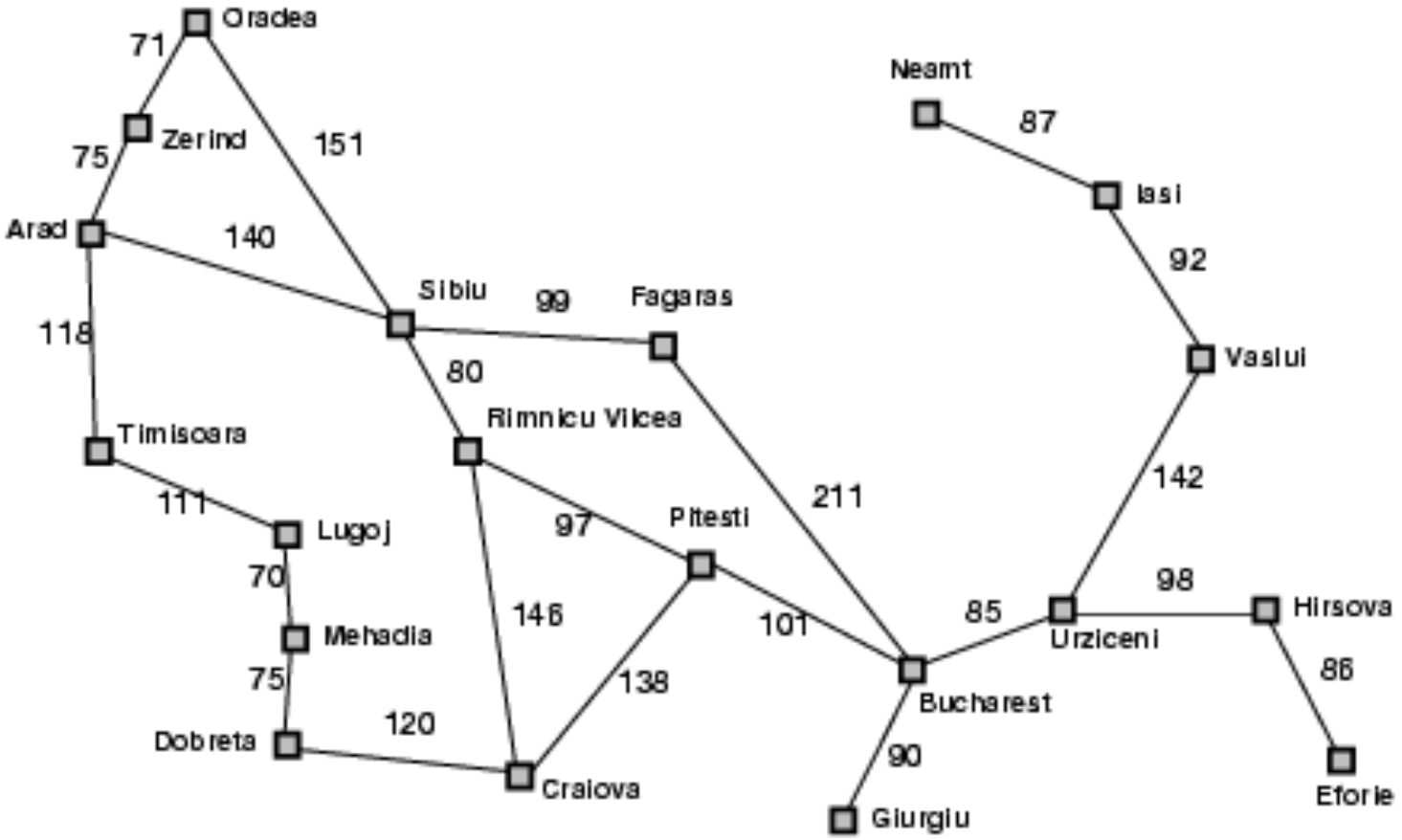
Informed Search Strategies

- *Informed Search Strategies* use info beyond the problem description

- We will first look at functions that "guess" distance from a state to nearest goal state.

- Let *h(n)* be the "function" that guesses how far *n* is from its nearest goal state.

Romania with step costs in km



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Best-first search

- Idea: use a function *f(n)* for each node
  - *f(n)* is an estimate of "desirability" of a node
  - Expand most desirable unexpanded node

- Implementation:

  Order the nodes in fringe in decreasing order of desirability (normally, higher *f* is then less desirable)

- Uninformed Search:
  - Depth-first: *f(n) = -g(n)*
  - Breadth-first: *f(n) = g(n)*

Best-first informed search strategies

- *Greedy Search*
- *A* Search*
- *Iterative Deepening A* (IDA*)*
- *Weighted A* Search*

Outline

- **Review**
- **Best-first search**
- *Greedy search*

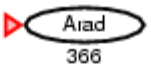
Greedy search

- Evaluation function: $f(n) = h(n)$

- $h(n)$ = estimate of cost from n to *goal*
  - e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest

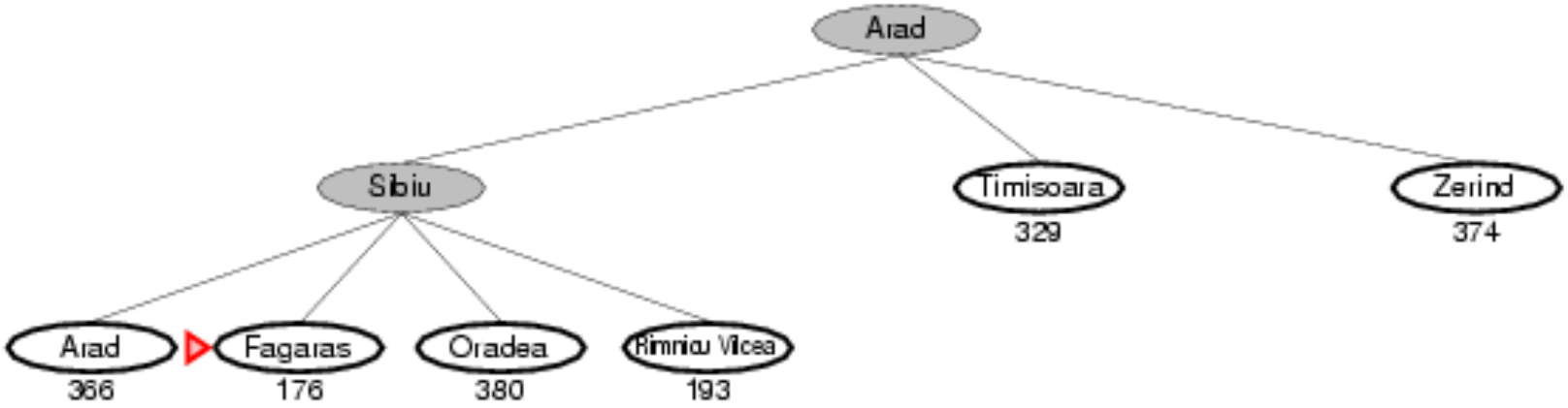
- Greedy search expands the node that *appears* to be closest to goal

Greedy best-first search example

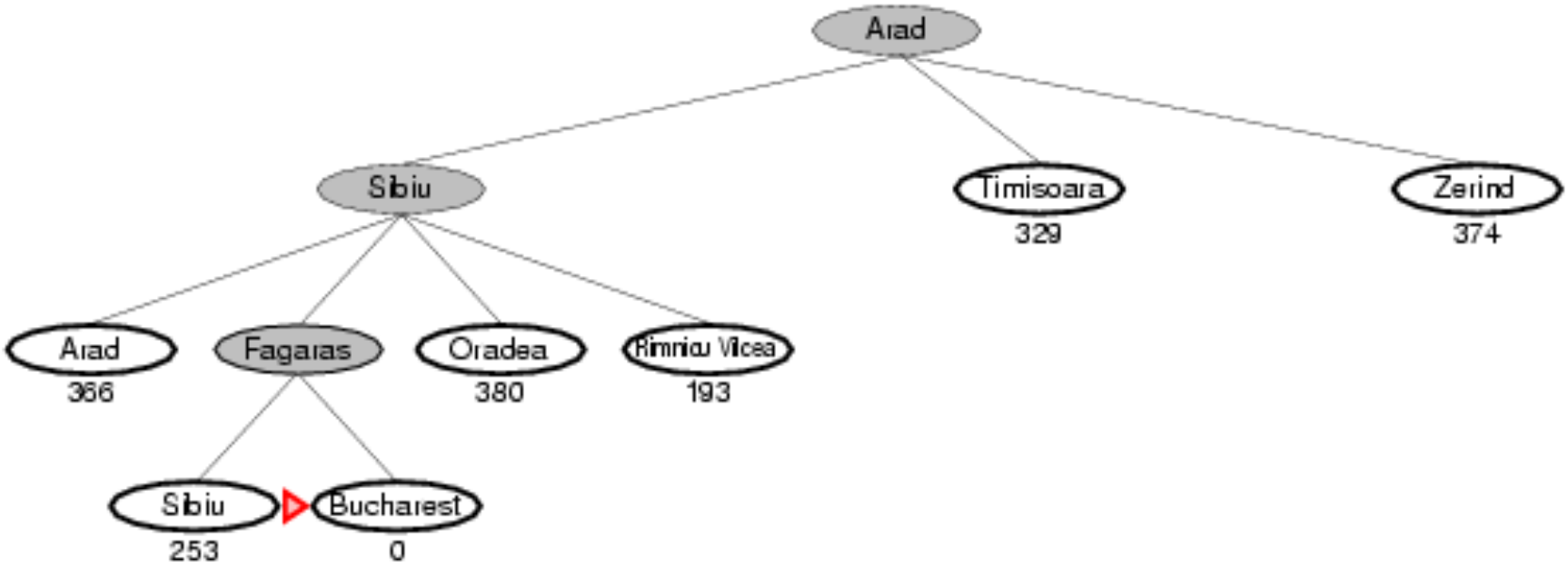


Greedy best-first search example

Greedy best-first search example



Greedy best-first search example



Why greedy search is attractive

- With a decent enough heuristic, goes almost directly to goal.
- Best case: time and space are linear
- So, why not always do greedy search?

Properties of greedy best-first search

- Complete? No, has same problem with infinite graphs as depth-first search
- Time? $O(b^m)$, but a good heuristic can give dramatic improvement
- Space? $O(b^m)$ -- keeps all nodes in memory
- Optimal? No

Greedy Search in Prolog

```
/* solution(+Heuristic, +Goal, +Fringe, +Closed, -Solution) */

solution(_Heuristic, Goal, [Node | _], Closed, Solution) :-
    node(Node, State, ParentState, _FValue),
    test(Goal, State),
    extractSolution(ParentState, Closed, [State], Solution).

solution(Heuristic, Goal, [Node | RestNodes], Closed, Solution) :-
    nodeState(Node, State),
    findall(NeighborNode,
            newNeighborNode(State,     Heuristic,
                                [Node | Closed],      NeighborNode),
            NeighboringNodes),
    orderFringe(RestNodes, NeighboringNodes, NewFringe),
    solution(Heuristic, Goal, NewFringe, [Node | Closed], Solution).
```

Summary

- Intelligent behaviour oft involves search.
- **Search strategy** defines a traversal of the search space, e.g., pick best *f(n)*.
- **Informed** search strategies use information outside of problem description.
- One such type of information is estimated cost to nearest goal: ***h(n)***.
- **Greedy search:** *f(n) = h(n).*

What does these measure?

- Assume *n* is a node in the search space, what do these measure?
  - *f(n)*
  - *g(n)*
  - *h(n)*

Food for thought

- Do you use search in your life to solve problems?
- What sort of information do you use to reduce the amount of search you do?
- What do you aim for, cheapest solution, quickest solution, or a combination?

Challenge

- Can you create state space representations for following domains:
  - scheduling taxi service in Auckland
  - playing chess
  - getting a degree at UofA
  - enjoying your life
- You need to represent states of the world, actions that change states, problems, and solutions.

Next Time

- Look at:
  - A* search
  - IDA*
  - Heuristics