COMP219: Artificial Intelligence

Lecture 8: Combining Search Strategies and Speeding Up

Overview

- Last time
  - Basic problem solving techniques:
    - **Breadth-first search**
      - complete but expensive
    - **Depth-first search**
      - cheap but incomplete
- Today
  - Variations and combinations
    - **Limited depth search**
    - **Iterative deepening search**
  - Speeding up techniques
    - Avoiding repetitive states
    - Bi-directional search

- Learning outcome covered today:
  Identify, contrast and apply to simple examples the major search techniques that have been developed for problem-solving in AI

Depth Limited Search

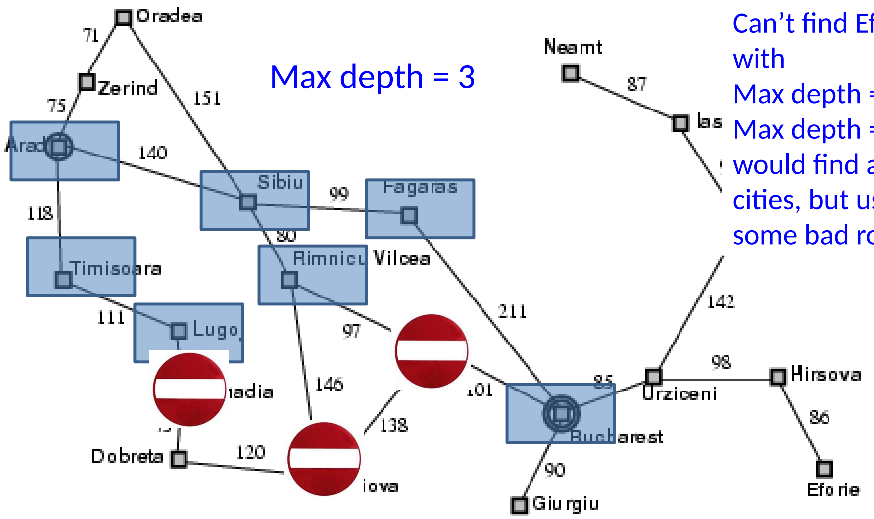
- Depth first search has some desirable properties - space complexity
- But if wrong branch expanded (with no solution on it), then it may not terminate
- Idea: introduce a depth limit on branches to be expanded
- Don't expand a branch below this depth
- Most useful if you know the maximum depth of the solution

Depth Limited Search

```
depth limit = max depth to search to;
agenda = initial state;
  if initial state is goal state then
     return solution
  else
    while agenda not empty do
      take node from front of agenda;
      if depth(node) < depth limit then
      {
        new nodes = apply operations to node;
       add new nodes to front of agenda;
       if goal state in new nodes then
         return solution;
       }
```

Example: Romania Problem

Only 20 cities on the map, so no path longer than 19.
In fact, any city can reach any other in at most 9 steps.

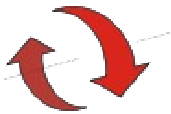


Can't find Eforie with
Max depth = 3;
Max depth = 9 would find all cities, but use some bad routes

Depth Limited Search

- Will always terminate.
- Will find solution if there is one in the depth bound.
- Too small a depth bound misses solutions.
- Too large a depth bound may find poor solutions when there are better ones.

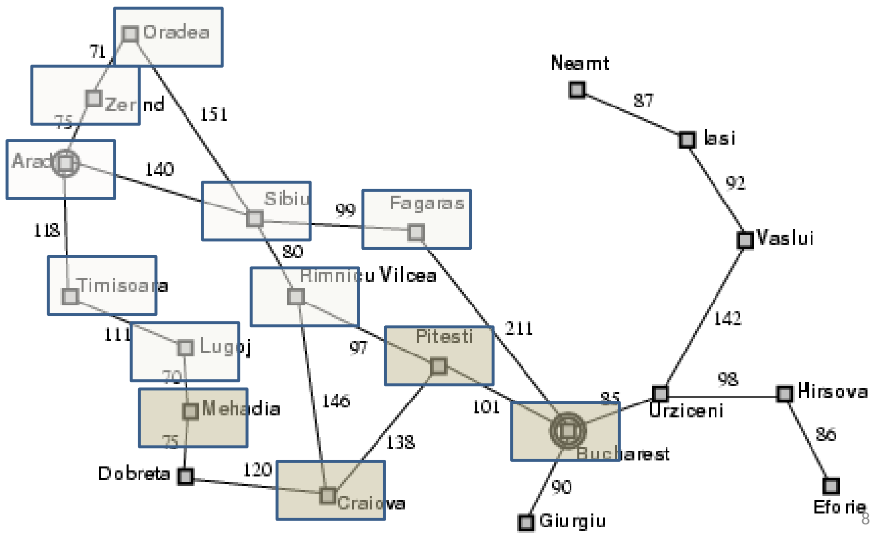
Iterative Deepening



- Problem with choosing depth bound; incomplete or admits poor solutions.
- Iterative deepening is a variation which is complete and finds best solution.
- Basic idea is:
  - do d.l.s. for depth *n* = 0; if solution found, return it;
  - otherwise do d.l.s. for depth *n* = *n* + 1; if solution found, return it, etc;
  - So we repeat d.l.s. for all depths until solution found.
- Useful if the search space is large and the maximum depth of the solution is not known.

Example: Romania Problem

D =0 D =1 D =2 D =3



General Algorithm for Iterative Deepening

```
depth limit = 0;
  repeat
  {result = depth_limited_search
   (max depth = depth limit;
   agenda = initial node; );
    if result contains goal then
       return result;
   depth limit = depth limit + 1;}
  until false; /* i.e., forever */
```

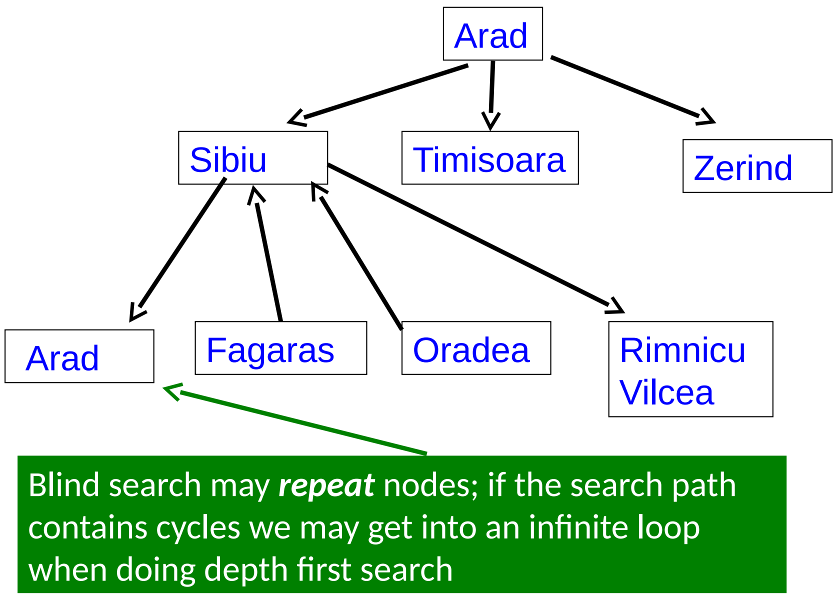
- Calls d.l.s. as subroutine.

IDS Properties

- Note that in iterative deepening, we re-generate nodes *on the fly*.
- Each time we do a call on depth limited search for depth *d*, we need to regenerate the tree to depth $d - 1$.
- Trade off time for memory.

- In general we might take a little more time, but we save a lot of memory.
  - Example: Suppose $b = 10$ and $d = 5$.
  - Breadth first search would require examining 111,110 nodes, with memory requirement of 100,000 nodes.
  - Iterative deepening for same problem: 123,450 nodes to be searched, with memory requirement of only 50 nodes.
  - Takes 11% longer in this case, but savings on memory are immense.

The Search Tree



Avoiding Repeated States

- There are three ways to deal with this (in order of increasing effectiveness and computational overhead):
  - do not return to the state you have just come from
  - do not create paths with cycles in them
  - do not generate any state that was ever generated before
- Note there is a trade-off between the cost of extra search and the cost of checking for repeated states

Branching

- In analyses branching is often assumed to be uniform
- But in practice this is often not so
- This can make a big difference to the search space

Goal vs Data driven search

- We can choose to search from the initial state to the goal (data driven)
- Or from the goal to the initial state (goal driven)
- The branching may be very different, which will affect the search
- Goal driven search is very often very much more efficient (few paths reach the goal)
- Often used in expert systems (and Prolog)

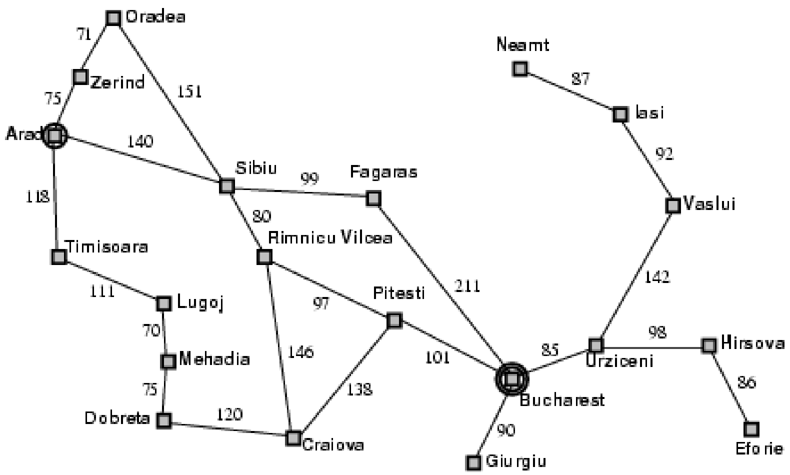
Exercise

Bi-directional Search

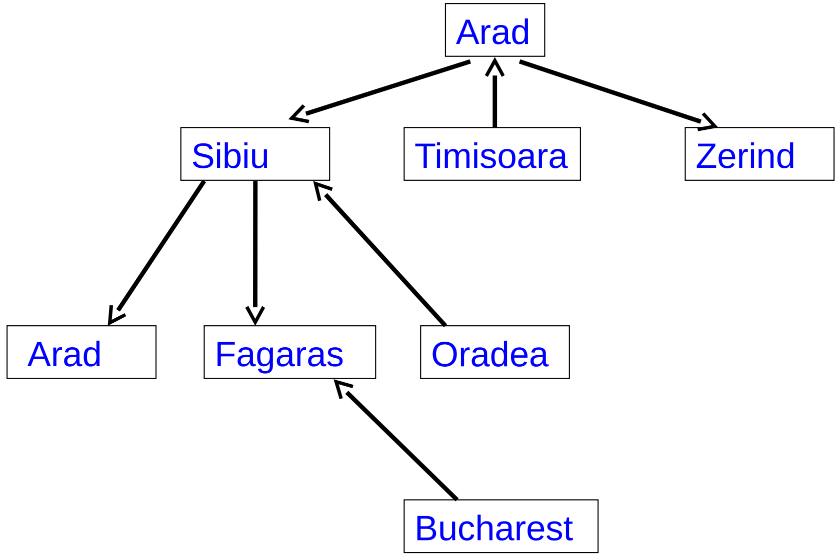
- If we are unsure of the branching factor, then searching from both ends may be best

Example: Romania

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest



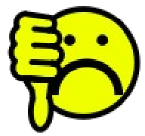
Bi-directional Search



Bi-directional Search: Good

- *Much* more efficient
- Rather than doing one search of b^d, we do *two* $b^{d/2}$ searches
  - Example
    - Suppose $b = 10$, $d = 6$
    - Breadth first search will examine $10^6 = 1{,}000{,}000$ nodes
    - Bidirectional search will examine $2 \times 10^3 = 2{,}000$ nodes
- Can combine different search strategies in different directions

Bi-directional Search: Bad



- Must be able to generate predecessors of states
- There must be an efficient way to check whether each new node appears in the other search
- For large *d*, is still impractical
- For two bi-directional breadth-first searches, with branching factor *b* and depth of the solution *d* we have memory requirement of $b^{d/2}$ for each search

Summary

- More advanced problem-solving techniques
  - Depth-limited search
  - Iterative deepening
  - Bi-directional search
  - Avoiding repeated states
- The above improved on basic techniques like breadth-first and depth-first search
- However, they still aren't always powerful enough to give solutions for realistic problems
- Are there more improvements we can make?

- Next time
  - Lists in Prolog