Seminar: Search and Optimization

4. Basic Search Algorithms

Martin Wehrle

Universität Basel

October 4, 2012

Seminar: Search and Optimization

October 4, 2012 — 4. Basic Search Algorithms

4.1 Basics

State Spaces

Definition (State Space)

A state space (or transition system) is a 6-tuple $\mathcal{S} = \langle S, A, cost, T, s_0, S_\star \rangle$ where

- S finite set of states
- A finite set of actions
- $cost : A \to \mathbb{R}_0^+$ action costs
- $T \subseteq S \times A \times S$ transition relation; deterministic in $\langle s, a \rangle$
- $s_0 \in S$ initial state
- $S_\star \subseteq S$ set of goal states

Representation of State Spaces

How to get the state space into the computer?

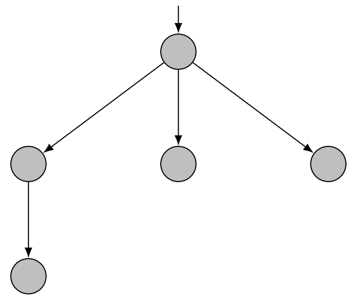
State space $\mathcal{S} = \langle S, A, cost, T, s_0, S_\star \rangle$ as black box:

- init(): creates initial state
  Returns: the state s_0
- is-goal(s): tests if state s is goal state
  Returns: **true** if $s \in S_\star$; **false** otherwise
- succ(s): lists all applicable actions and successors of s
  Returns: List of tuples $\langle a, s' \rangle$ with $s \xrightarrow{a} s'$
- cost(a): determines action cost of action a
  Returns: the non-negative number $cost(a)$

Search Algorithms

Start with initial state. In every step, expand a state through generating its successors.

$\rightsquigarrow$ search space



Terminology

- Search node
  Represents a state + additional information during the search
- Node expansion
  Generating the successor nodes of a node n through applying the applicable actions in n
- Open list or Frontier
  Set of nodes that are candidates for expansion
- Closed list
  Set of nodes that are already expanded
- Search strategy
  Determines which node to expand next

Properties of Search Algorithms

Completeness: Guarantee to find a solution is a solution exists.
Guarantee to terminate if no solution exists.

Optimality: Guarantee to find optimal solutions

Complexity: Time: How long does it take to find a solution?
(measured in generated nodes)
Space: How much memory is used?
(measured in nodes)

Parameters:

- b: branching factor (= max. number of successors of a state)
- d: search depth (length of longest path in search space)

4.2 Blind Search Algorithms

Blind Search Algorithms

Blind (or Uninformed) Search Algorithms

Use no additional information about the state space beyond the problem definition

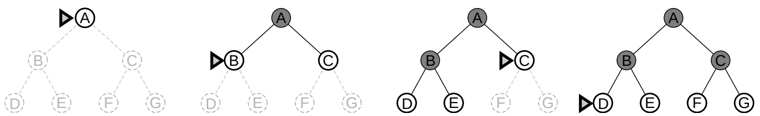
- Breadth-first search
- Depth-first search
- Uniform cost search, iterative depth-first search, . . . (not considered in this talk)

In contrast to

heuristic search algorithms ($\rightsquigarrow$ introduced later)

Breadth-First Search

Nodes are expanded in the order they have been generated (FIFO)
$\rightsquigarrow$ open list implemented as, e. g., a double-ended queue (deque)



- searches the state space layer by layer
- complete
- always finds a shallowest goal state first
- optimal in case all actions have the same costs

Breadth-First Search: Pseudo-Code

BFS: Pseudo-Code (inefficient!)

n_0 := make-root-node(init())
if is-goal(n_0.state):
  return extract-solution(n_0)
open := **new** FIFO queue with n_0 as the only element
closed := $\emptyset$
loop do
  if *open*.empty():
    return none
  n = *open*.pop-front()
  closed.insert(n)
  for each $\langle a, s' \rangle \in$ succ(n.state):
    if $s' \notin open \cup closed$:
      n' := make-node(n, a, s')
      if is-goal(s'):
        return extract-solution(n')
      open.push-back(n')

Breadth-First Search: Complexity

Proposition: Time Complexity

Let b be the branching factor and d the minimal solution length in the generated state space. Let $b \geq 2$.

Then the time complexity of breadth-first search is

$$1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$$

Recall: we measure time complexity as number of generated nodes

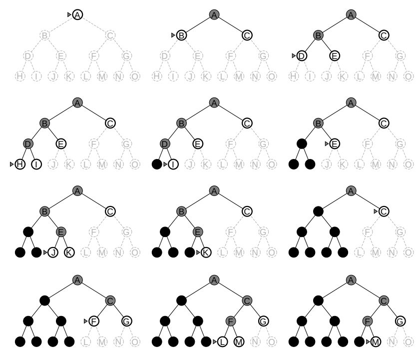
It follows that (for $b \geq 2$) also the space complexity of breadth-first search is $O(b^d)$.

Depth-First Search

Nodes that are generated last are expanded first (LIFO)
⇝ nodes with highest depth are expanded first

- Open list implemented as a stack

Example: (Assumption: nodes in depth 3 have no successors)



Depth-First Search: Properties and Implementation

Properties:

- neither complete nor optimal (Why?)
- complete if the state space is acyclic

Implementation:

- common and efficient: depth-first search as recursive function

⇝ use stack of programming language/CPU as open list

Depth-First Search: Pseudo-Code

Pseudo-Code: Main Procedure

n_0 := make-root-node(init())
solution := recursive-search(n_0)
if *solution* ≠ **none**:
    return *solution*
return unsolvable

function recursive-search(n):

    if is-goal(n.state):
        return extract-solution(n)
    for each $\langle a, s' \rangle \in$ succ(n.state):
        n' := make-node(n, a, s')
        solution := recursive-search(n')
        if *solution* ≠ **none**:
            return *solution*
    return none

Depth-First Search: Complexity

Time Complexity:

- If there exist paths of length m in the state space, then depth-first search can generate $O(b^m)$ nodes.
- However, in the best case, a solution of length l can be found by generating only $O(bl)$ nodes.

Depth-First Search: Complexity

Space Complexity:

- Only maintains nodes in memory along the path from initial node to currently expanded node (no duplicate elimination!) ("along the path" = nodes on this path and their successors)
- Therefore, if m is the maximal depth of the search, the space complexity is $O(bm)$
- Low space complexity ⇝ depth-first search is interesting despite its disadvantages

4.3 Best-First Search

Heuristic Search Algorithms

- So far: blind search algorithms (no additional properties of the problem are used to guide the search)
- Drawback: Limited scalability (even for small problems)
- Idea: find criteria to estimate which states are "good" and which states are "bad" ⇝ prefer good states

⇝ heuristic search algorithms

Heuristics

Definition (Heuristic)

Let $\mathcal{S}$ be a state space with set of states S.
A heuristic function or heuristic for $\mathcal{S}$ is a function

$$h : S \to \mathbb{N}_0 \cup \{\infty\},$$

that maps states to natural numbers (or ∞).

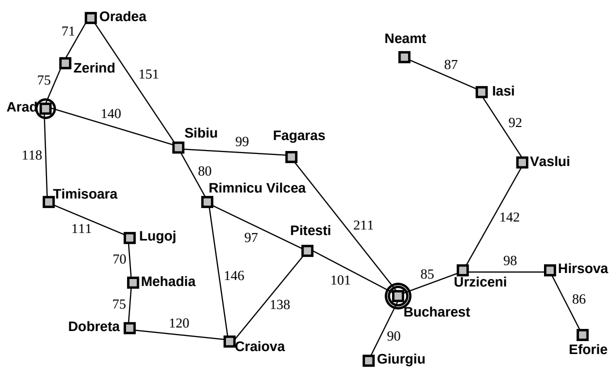
Idea: $h(s)$ estimates distance of s to goal

- Intuition: the better h approximates the real goal distance, the more efficient the search

Notation: we write $h(n)$ as an abbreviation for $h(n.\text{state})$

Example: Route Planning in Romania

Example heuristic: straight-line distance to Bucharest



Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Best-First Search

Best-first search represents a class of heuristic search algorithms that expand in every step the "best" candidate node.

Best-First Search

Algorithms based on best-first search

- use a heuristic to compute an evaluation function f
- evaluate every node n with f (i. e., compute $f(n)$)
- expand node with minimal f value next
- different definitions of f
  $\rightsquigarrow$ different search algorithms

Best-First Search: Pseudo-Code

Best-First Search (delayed duplicate elimination, no re-opening)

```
open := new priority queue, ordered by f
open.insert(make-root-node(init()))
closed := ∅
while not open.empty():
    n = open.pop-min()
    if n.state ∉ closed:
        closed := closed ∪ {n.state}
        if is-goal(n.state):
            return extract-solution(n)
        for each ⟨a, s'⟩ ∈ succ(n.state):
            if h(s') < ∞:
                n' := make-node(n, a, s')
                open.insert(n')
return unsolvable
```

Important Best-Search Algorithms

Important Best-First Search Algorithms

- Greedy best-first search
  - $f(n) := h(n)$
  - Quality of node is determined solely by the heuristic
- A*
  - $f(n) := g(n) + h(n)$
  - Combination of path costs $g(n)$ (from init to n) and heuristic

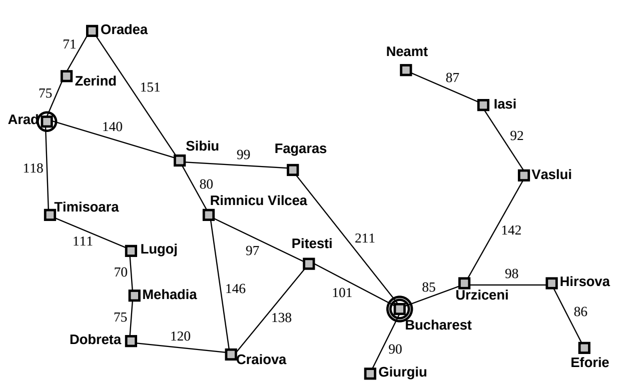
⇝ In the following: discussion of greedy best-first search and A*

Greedy Best-First Search

Greedy Best-First Search

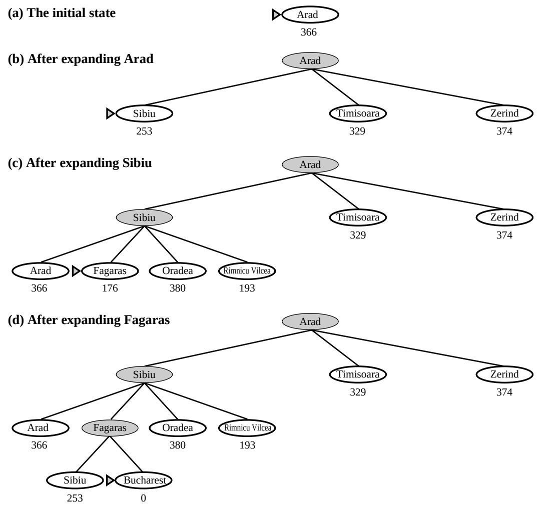
Only take heuristic into account: $f(n) := h(n)$

Example: Greedy Best-First Search for Route Planning



Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example: Greedy Best-First Search for Route Planning



Greedy Best-First Search: Properties

Greedy Best-First Search is

- complete for heuristics h with the property that $h(s) = \infty$ implies that no solution starts in s (safe heuristics)
- suboptimal (solution can be arbitrarily bad)
- often one of the best search algorithms in practice if optimality isn't a requirement

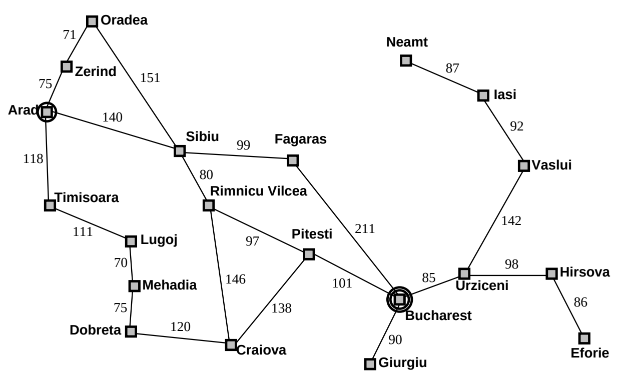
A^*

A^*

In addition to greedy best-first search, take the path costs into account: $f(n) = g(n) + h(n)$

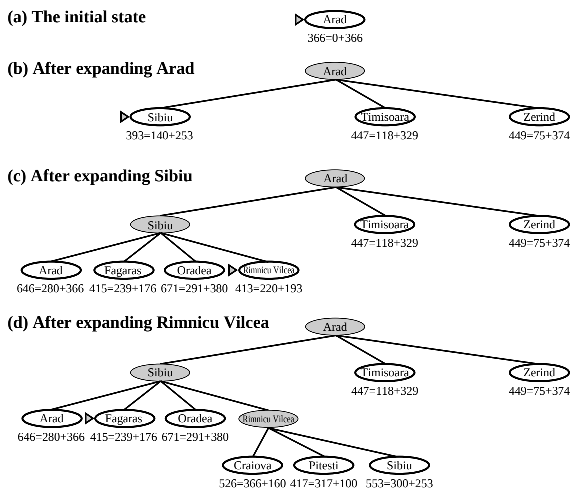
- Balance path costs and estimated proximity to goal
- $f(n)$ estimates costs of cheapest solution from initial state through n to the goal

Example: A^* for Route Planning

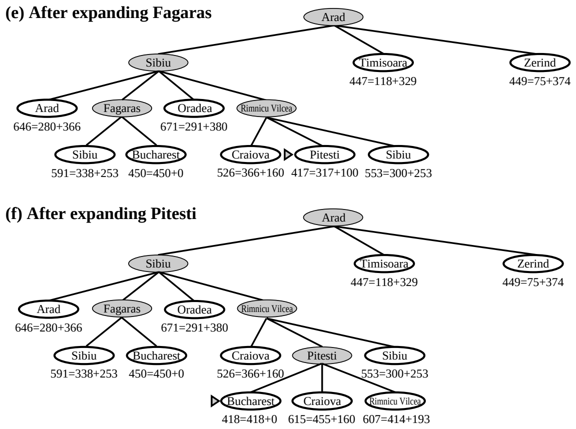


Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example: A^* for Route Planning



Example: A^* for Route Planning



A^*: Properties

- Most important advantage of A^* compared to greedy best-first search: optimal under appropriate requirements to heuristic (mainly: admissibility)
- Important result!

4.4 Summary

Summary

Blind Search Algorithms

- No additional problem properties used to guide the search
- Often limited scalability even for small problems
- Examples: breadth-first search and depth-first search

Heuristic Search Algorithms

- Use heuristics to guide the search
- Often much more efficient than blind search
- Examples: greedy best-first search and A^*