Spezielle Themen der Künstlichen Intelligenz

1. Termin: Einführung & Wiederholung

Dr. Stefan Kopp
Center of Excellence „Cognitive Interaction Technology“
AG Sociable Agents

Administrativa

Sociable Agents
http://www.techfak.uni-bielefeld.de/ags/soa/

Dr.-Ing. Stefan Kopp

- skopp@techfak.uni-bielefeld.de
- Sprechstunde: Fr 13-14, Q1-144
- Tel: (106-)12144

Semesterapparat: Universitätsbibliothek, FB Informatik, „Kopp“

Webseite: www.techfak.uni-bielefeld.de/~skopp/Lehre/STdKI_SS10

Übungen:

- Thies Pfeiffer
- Ramin Yaghoubzadeh

Leistungspunkte

Vorlesung: 6 LPs für

- regelmäßige Teilnahme an der Vorlesung
- regelmäßige Teilnahme an den Übungen
- erfolgreiches Bearbeiten der Übungsaufgaben
- erfolgreiche Abschlussprüfung/Klausur → benotete EL

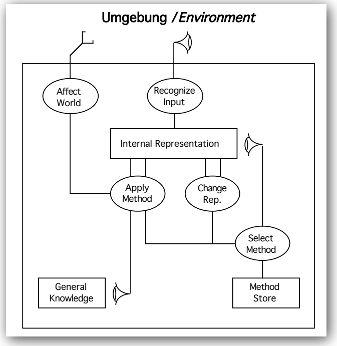
Modul „Vertiefung Künstliche Intelligenz“ = 10 LP

- +4 LP und EL aus weiterem Seminar

Methoden der der KI

Grundlagen und Überblicke in

- symbolischer Wissensrepräsentation
  - Logik, Frames, semantische Netze, KL-ONE
- Suche
  - blinde und informierte
  - Means-Ends-Analysis, Goal-Trees, CSP
- Logik und Inferenz
  - Prädikatenlogik, Resolution, Skolemisierung, Unifikation, Indexing
- spezielle Schlußverfahren
  - abduktive und induktive
  - probabilistische und nicht-monotone
  - räumliche und temporale



General Intelligent Agent

Spezielle Methoden der KI

Fortgeschrittene Techniken zur Realisierung künstlichen intelligenten Verhaltens in der Realität

Reale Domänen schwierig weil oft nachteilig in Bezug auf

- Größe
- Struktur
- Unbekanntheit & Vagheit
- Beobachtbarkeit
- Beeinflussbarkeit
- Dynamik & Vorhersagbarkeit

Spezielle Methoden der KI

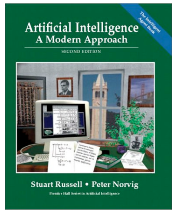
Fortgeschrittene Techniken zur Realisierung künstlichen intelligenten Verhaltens in der Realität

Vorlesung: Methoden geeignet für verschiedene Domänen

- Search, Reasoning & Planning
- Constraint Satisfaction
- Game-playing
- Uncertainty & Bayesian Belief Networks
- (Partially Observable) Markov Decision Problems
- Learning

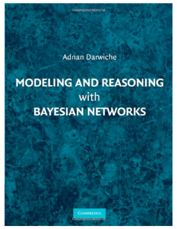
...with applications in actual research projects

Literatur



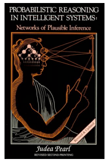
Russell & Norvig: Artificial Intelligence: A Modern Approach. Prentice Hall, 2nd Edition, 2003

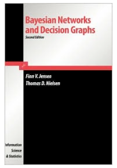
(~2nd part, Ch.11-18)

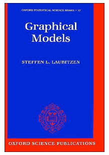


Darwiche: Modeling and Reasoning with Bayesian Networks. Cambridge Univ. Press, 2009

Weiterführende Literatur


PROBABILISTIC REASONING IN INTELLIGENT SYSTEMS
Networks of Plausible Inference
Judea Pearl





Judea Pearl, Probabilistic reasoning in intelligent systems, Morgan Kaufmann, 1989

Finn V. Jensen, Bayesian networks and decision graphs, , Springer, 2001

Steffen L. Lauritzen, Graphical models, Oxford, 2002

Günther Görz (Ed.), Handbuch der künstlichen Intelligenz, 4. Auflage, Oldenbourg, 2003

Search & Exploration (recap´)

Dr. Stefan Kopp
Center of Excellence „Cognitive Interaction Technology“
AG Sociable Agents

Search problem

Defined by:

- Specification of <u>start state</u>
- Specification of <u>goal state</u>
- Set of <u>operators</u> to go from one state into another

Solution:

- specific state meeting the specification of goal state
- *or:* sequence of operators that lead from state state into goal state (path in search space)

Different requirements

- finding one solution
- finding all solutions
- finding optimal solution
- proving no solution to exist

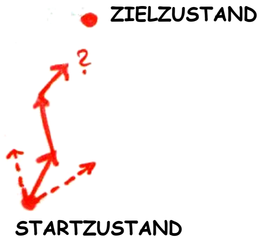
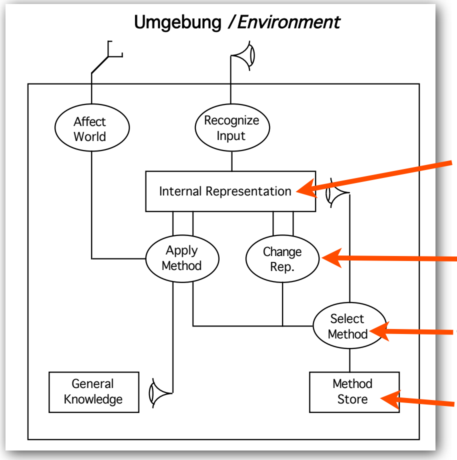


Image © 2007 GeoContent

Problem-solving by searching



database (complete, offline) or world (incomplete, online)

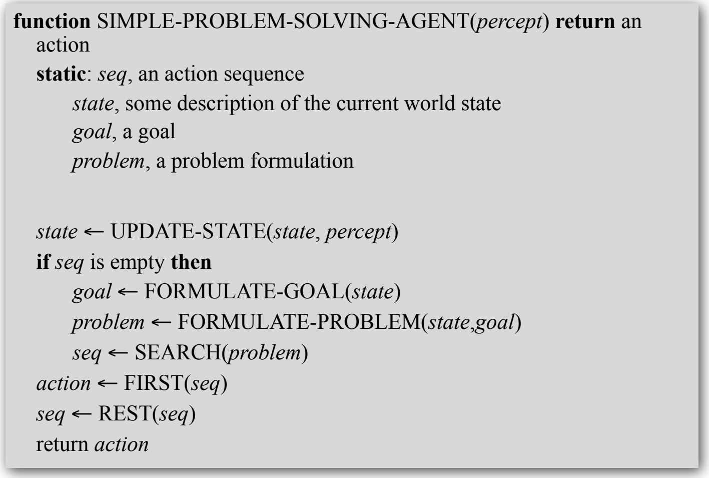
`as-if´-acting

control strategy

operators

(Newell & Simon)

Problem-solving agent



```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) return an action
    static: seq, an action sequence
        state, some description of the current world state
        goal, a goal
        problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state,goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Problem types

Single-state problem

- Environment is static, deterministic, and fully observable
- Agent knows exactly which state it is now and will be in
- Solution: sequence of action that need to be executed (open-loop)

Sensorless (conformant) problem

- Partial knowledge of states, but known actions
- Agent may have no idea where it is, each action may lead to one of several possible states
- Solution (if any): sequence of action that will do the job in any case

Problem types

Contingency problem

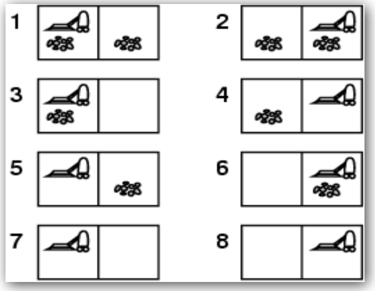
- Environment is non-deterministic, i.e. actions are uncertain, or partially observable
- Each percept provides new, but partial information after each action (contingency that must be planned for)
- Solution: no fixed action sequence, interleave search and execution (closed-loop)

Exploration problem

- Environment and actions are unknown up-front
- Agent must act to discover states and actions
- Extreme case of contingency problem

Example: vacuum world

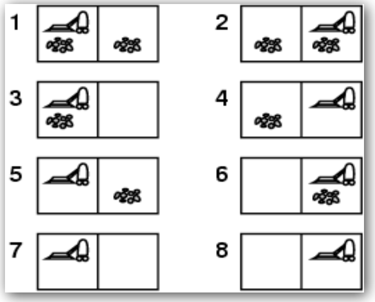
- Single-state, start in #5.
  Solution?



Task: Clean the room (#7 or #8)

Example: vacuum world

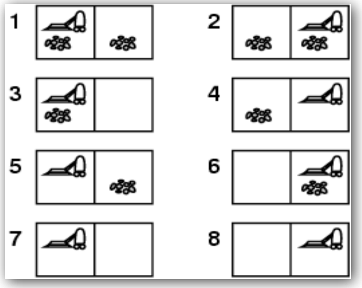
- Single-state, start in #5.
  Solution? *[Right, Suck]*
- Sensorless, start in one of {*1,2,3,4,5,6,7,8*}, e.g. *Right* goes to {*2,4,6,8*} and *[Right, Suck]* to {*4,8*}
  Solution?



Task: Clean the room (#7 or #8)

Example: vacuum world

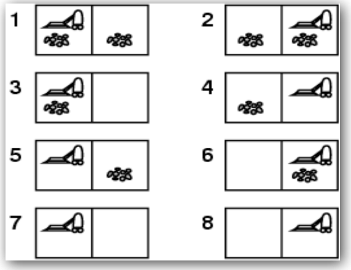
- Sensorless, start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]
  Search in *sets* of states (=*belief states*)



- Contingency problem
  - Non-deterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location
  - Percept: *[L, Clean],* i.e., start in #5 or #7
    Solution?

Example: vacuum world

- Sensorless start in {*1,2,3,4,5,6,7,8*} e.g., *Right* goes to {*2,4,6,8*}
  Solution?
  [Right,Suck,Left,Suck]

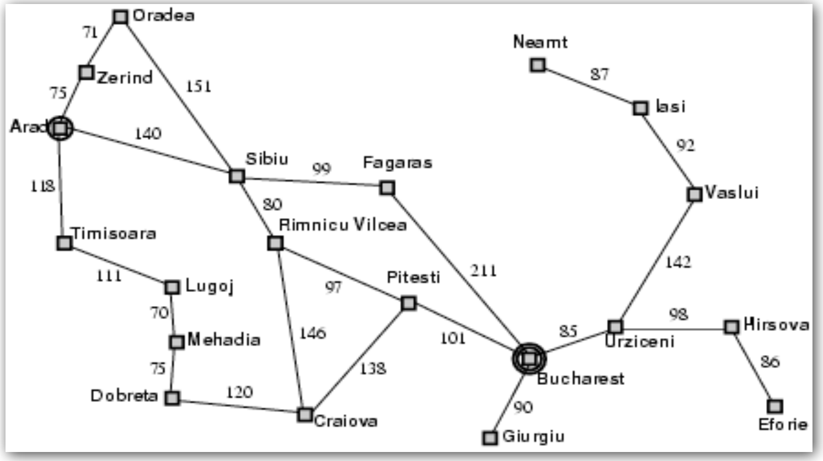


- Contingency
  - Nondeterministic: *Suck* may dirty a clean carpet
  - Partially observable: location, dirt at current location.
  - Percept: *[L, Clean],* i.e., start in #5 or #7 or ??
    Solution? *[Right,* ***if*** *dirt* ***then*** *Suck, Left,* ***if*** *dirt* ***then*** *Suck]*
    actions based on contingencies arising during execution

Example: Romania

- Problem:
  - on holiday in Romania; currently in Arad; flight leaves tomorrow from Bucharest
- Formulate goal:
  - be in Bucharest in time
- Formulate problem:
  - states: various cities
  - actions: drive between cities
- Find solution:
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Single-state problem formulation

A problem is defined by four items:

State space

1. initial state e.g., *In(Arad)*
2. actions or successor function at state *x*:
   S(x) = set of action–state pairs
   - e.g., *S(In(Arad))* = {<*Go(Zerind)*, *In(Zerind)*>, … }
3. goal test, is given state *x* goal state?
   - explicit, e.g., *x* = *In(Bucharest)*
   - implicit, e.g., *HasAirport(x)*
4. path cost (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - step cost *c(x,a,y)* of getting from *x* to *y* by action *a*, assumed to be ≥ 0

- Solution = action sequence leading from initial to goal state

Formulating the state space

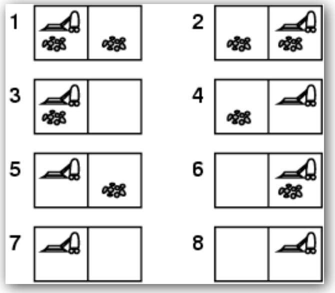
Real world is usually too complex ➔ state space must be abstracted

- (Abstract) state = *set* of real/virtual states/properties
- (Abstract) action = *complex combination* of real/virtual actions
  - abstraction *valid* if path between (abstract) search space states reflected in the world (*realizability*)
- (Abstract) solution = set of paths that reflect actual solutions in the real/virtual world

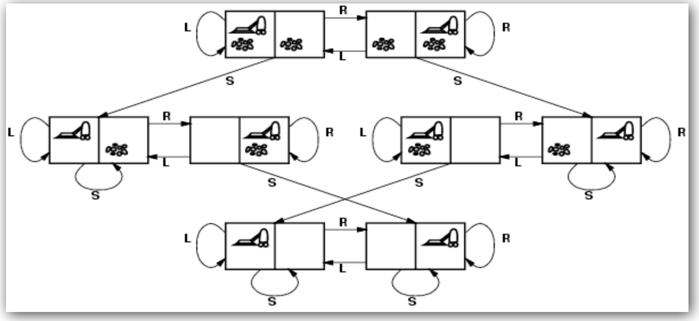
Needless to say, each abstract state-space formulation should be easier than the real problem to enable searching

Vacuum world state space graph

- states?
- actions?
- goal test?
- path cost?



Vacuum world state space graph



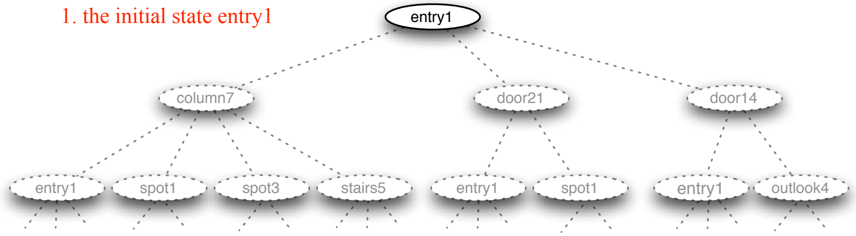
- states? integer dirt and robot location ($n*2^n$ states)
- actions? *Left*, *Right*, *Suck*
- goal test? no dirt at all locations
- path cost? 1 per action (step cost)

Simple tree search algorithm (pseudo-code)

```
function TREE-SEARCH(problem, strategy) return a solution or failure
  Initialize search tree to the initial state of the problem
  loop do
     if no candidates for expansion then return failure
     choose leaf node for expansion according to strategy
     if node contains goal state then return solution
     else expand the node and add resulting nodes to the search tree
  end
```

Exampe: simple tree search

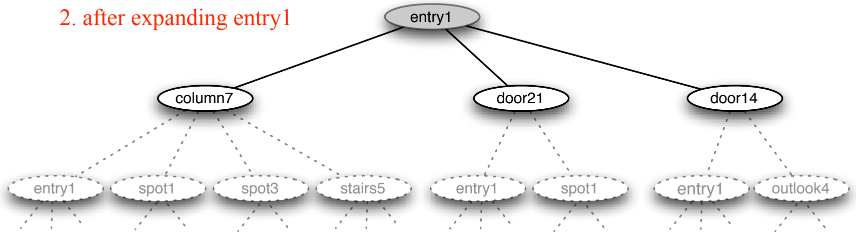
1. the initial state entry1



```
function TREE-SEARCH(problem, strategy) return a solution or failure
  Initialize search tree to the initial state of the problem
  loop do
     if no candidates for expansion then return failure
     choose leaf node for expansion according to strategy
     if node contains goal state then return solution
     else expand the node and add resulting nodes to the search tree
  end
```

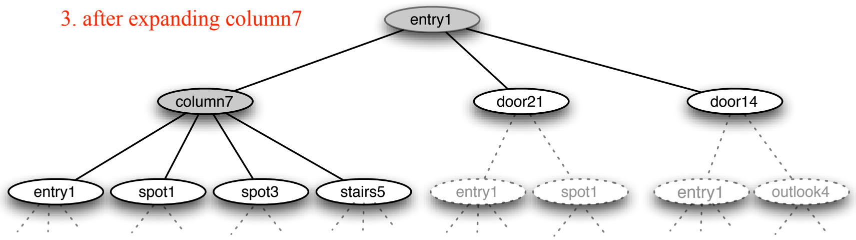
Simple tree search example

2. after expanding entry1



```
function TREE-SEARCH(problem, strategy) return a solution or failure
  Initialize search tree to the initial state of the problem
  loop do
     if no candidates for expansion then return failure
     choose leaf node for expansion according to strategy
     if node contains goal state then return solution
     else expand the node and add resulting nodes to the search tree
  end
```

Simple tree search example


3. after expanding column7

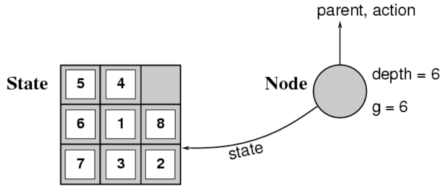
```
function TREE-SEARCH(problem, strategy) return a solution or failure
   Initialize search tree to the initial state of the problem
   loop do
       if no candidates for expansion then return failure
       choose leaf node for expansion according to strategy
       if node contains goal state then return solution
       else expand the node and add resulting nodes to the search tree
   end
```

State space vs. search tree

state: (representation of) a world configuration

node: data structure to represent part of the search tree

- includes state, parent node, action, path cost *g(x)*, depth
- fringe set of generated nodes not yet expanded



An **Expand** function creates new nodes, filling in the various fields and using the **SuccessorFn** of the problem to create the corresponding states

Tree search algorithm

```
function TREE-SEARCH(problem,fringe) return a solution or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
       if EMPTY?(fringe) then return failure
       node ← REMOVE-FIRST(fringe)
       if GOAL-TEST[problem] applied to STATE[node] succeeds
               then return SOLUTION(node)
       fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node,problem) return a set of nodes
   successors ← the empty set
   for each <action, result> in SUCCESSOR-FN[problem](STATE[node]) do
       s ← a new NODE
       STATE[s] ← result
       PARENT-NODE[s] ← node
       ACTION[s] ← action
       PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action,s)
       DEPTH[s] ← DEPTH[node]+1
       add s to successors
   return successors
```

Search strategies

The search strategy defines the order of node expansion

Evaluated along the following dimensions:

- completeness: does it always find a solution if one exists?
- optimality: does it always find a least-cost solution?
- time complexity: how long does it take? (#nodes expanded)
- space complexity: how much memory is needed? (#nodes stored)

Time and space complexity depend on problem size, measured in terms of

- *b*: branching factor or maximum #successors of any node
- *d*: depth of the least-cost solution (root node at d=0)
- *m*: maximum depth of any path in state space (may be ∞)

Uninformed search strategies

Use only information available in problem definition (blind search)

- generate successors, distinguish goal from non-goal state
- when strategy can determine whether one non-goal state is better than another non-goal state → informed search

Categories defined by expansion algorithm (and fringe organization):

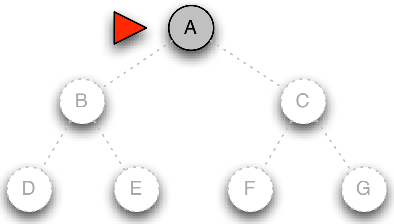
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search.
- Bidirectional search

Breadth-First (BF) search

Expand *shallowest* unexpanded node

Implementation:

- fringe is a FIFO queue, i.e., new successors go at end

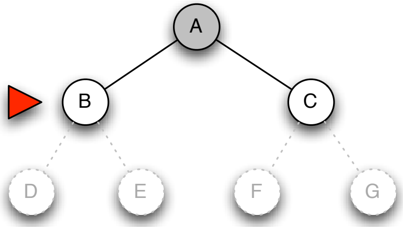


Breadth-First (BF) search

Expand *shallowest* unexpanded node

Implementation:

- fringe is a FIFO queue, i.e., new successors go at end

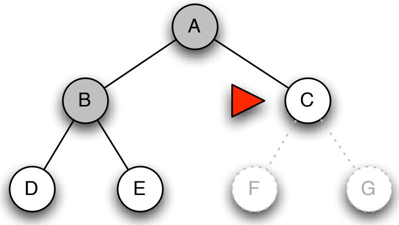


Breadth-First (BF) search

Expand *shallowest* unexpanded node

Implementation:

- fringe is a FIFO queue, i.e., new successors go at end

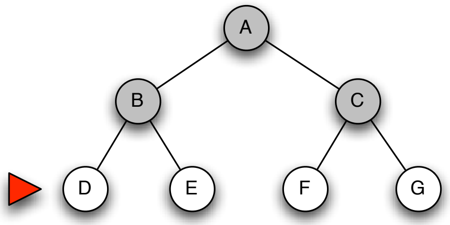


Breadth-First (BF) search

Expand *shallowest* unexpanded node

Implementation:

- fringe is a FIFO queue, i.e., new successors go at end



Properties of BF search

Complete? Yes (if *b* is finite)

Time? $1+b+b^2+b^3+\ldots+b^d+(b^{d+1}-b) = O(b^{d+1})$

Space? $O(b^{d+1})$ (keeps every node in memory)

Optimal? Yes (if step costs grow with depth → shallowest node is optimal)

DEPTH	NODES	TIME	MEMORY
2	1100	0.11 seconds	1 megabyte
4	111100	11 seconds	106 megabytes
6	10^7	19 minutes	10 gigabytes
8	10^9	31 hours	1 terabyte
10	10^{11}	129 days	101 terabytes
12	10^{13}	35 years	10 petabytes
14	10^{15}	3523 years	1 exabyte

b = 10
10.000 nodes/sec
1.000 byte/node

- Space is the bigger problem
- Exponential search problems cannot be solved by uninformed search methods for any but the smallest instances

Uniform-cost search

Expand node with *lowest* total path cost *g(n)*

fringe = queue ordered by path cost

- equivalent to breadth-first if step costs are all equal

Complete? Yes, if every step cost ≥ ε > 0

Optimal? Yes – nodes expanded in increasing order of *g(n)*

Time? ~ #nodes with cost $g \leq$ cost of optimal solution C^*

- at depth of about $C^*/\varepsilon \rightarrow O(b^{ceiling(C^*/\varepsilon)})$

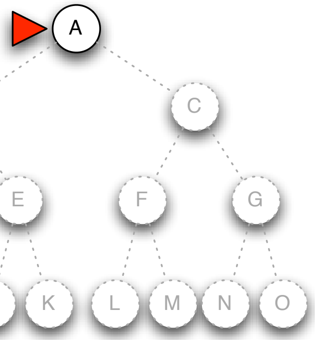
Space? $O(b^{ceiling(C^*/\varepsilon)})$

Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue, i.e., new successors go at front (=stack)

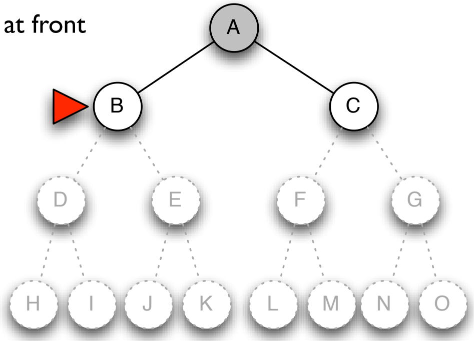


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

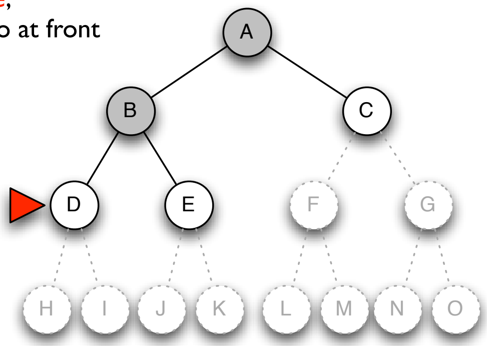


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

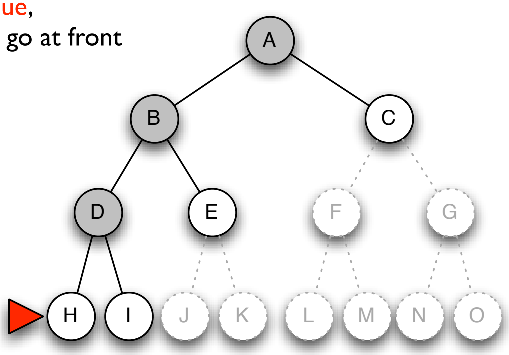


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

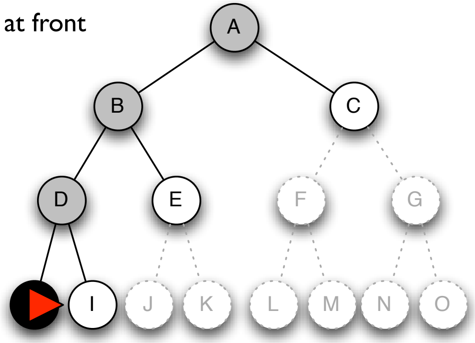


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

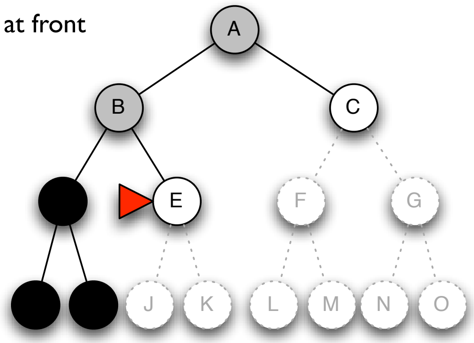


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

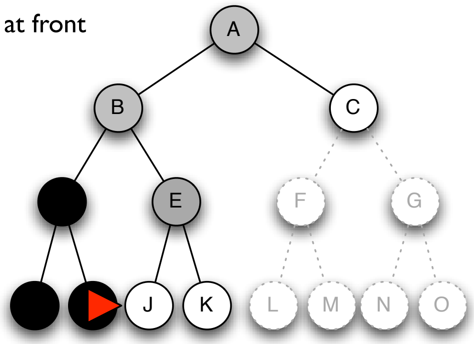


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

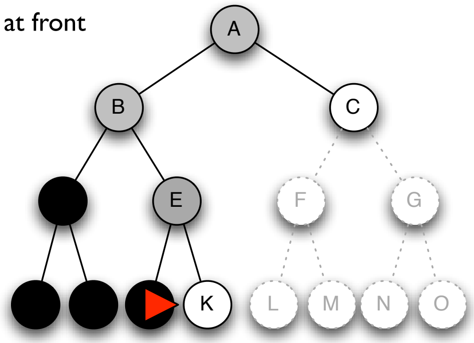


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

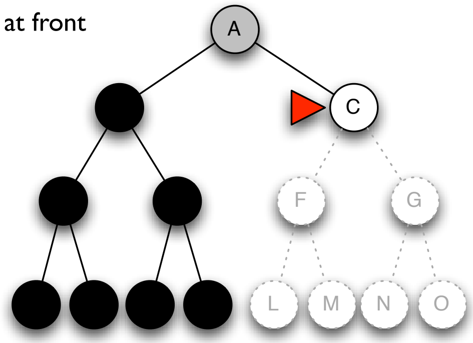


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

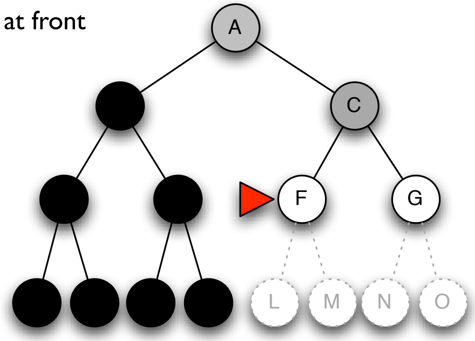


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

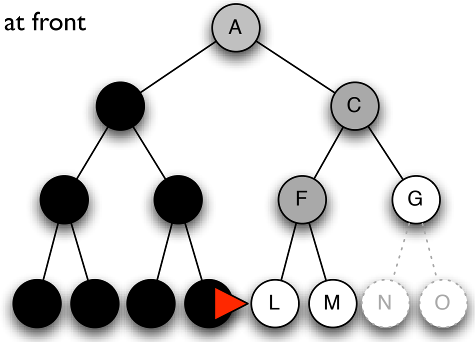


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front

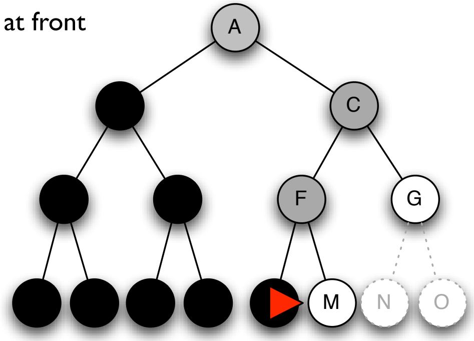


Depth-First (DF) search

Expand *deepest* unexpanded node

Implementation:

- fringe is a LIFO queue,
  i.e., new successors go at front



Properties of DF search

Complete? No, fails in infinite-depth spaces or spaces with loops

- modify to avoid repeated states along path makes it complete in finite spaces

Time? $O(b^m)$, i.e. all nodes expanded in worst case

- but if solutions are dense, may be much faster than breadth-first

Space? $O(bm)$, i.e. linear space complexity

- Backtracking search uses even less memory
  - One successor instead of all *b*.

Optimal? No, returns left-most goal state

Depth-limited search (DLS)

is DF-search with depth limit *l*

- i.e. nodes at depth *l* treated as if having no successors
- problem knowledge can be used to define good limits

solves the infinite-path problem, but adds incompleteness

- If $l < d$ then incompleteness results
- If $l > d$ then complete, but still not optimal

Time complexity: $O(b^l)$

Space complexity: $O(bl)$

Can be directly implemented in a recursive fashion

Recursive depth-limited search algorithm

```
function DEPTH-LIMITED-SEARCH(problem,limit) return a solution or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]),problem,limit)

function RECURSIVE-DLS(node, problem, limit) return a solution or failure/cutoff
    cutoff_occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] == limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result == cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
    if cutoff_occurred? then return cutoff else return failure
```

Iterative deepening search (IDS)

A general strategy to find best depth limit *l*

Goal is found at depth *d*, the depth of the shallowest goal-node

Combines benefits of DF-search and BF-search

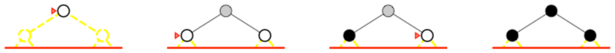
```
function ITERATIVE_DEEPENING_SEARCH(problem) return a solution or failure
    inputs: problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED_SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

IDS-search example

limit = 0

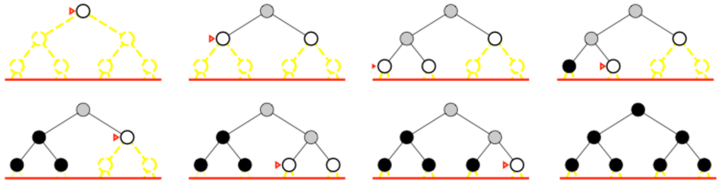
IDS-search example

limit = 1



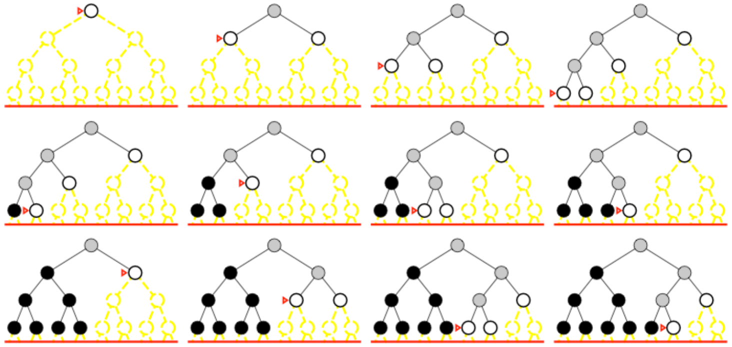
IDS-search example

limit = 2



IDS-search example

limit = 3



Properties of IDS

Complete? Yes, if *b* is finite

Time? sub-optimal because nodes are generated multiple times, but this is not so costly since most nodes are in the bottom level

➡ $(d+1)1 + d\, b + (d-1)b^2 + \ldots\ 2\, b^{(d-1)} + 1\, b^d = O(b^d)$

Space? $O(bd)$

Optimal? Yes, if path cost monotonically increases with depth

Properties of IDS vs. BFS

Number of nodes generated in a breadth-first search to depth *d* with branching factor *b*:

- $N_{BFS} = b^0 + b^1 + b^2 + \ldots + b^{d-1} + b^d + (b^{d+1}-b) = O(b^{d+1})$

Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

- $N_{IDS} = (d+1)1 + d\, b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d = O(b^d)$

Example for *b* = 10, *d* = 5:

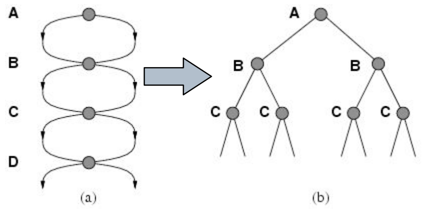
- $N_{BFS} = 10 + 100 + 1.000 + 10.000 + 100.000 + 999.999 = 1.111.111$
- $N_{IDS} = 50 + 400 + 3.000 + 20.000 + 100.000 = 123.450$

➡ IDS preferred search method for large search spaces and unknown depth of solution

Repeated states

Failure to detect repeated states can turn solvable problems into unsolvable ones

Example: simple state space generates an exponentially larger search tree



state space of size d+1

search tree with repeated states of size 2^d

Tree search ➔ Graph search algorithms

```
function TREE-SEARCH(problem,fringe) return a solution or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if EMPTY?(fringe) then return failure
      node ← REMOVE-FIRST(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 ← INSERT-ALL(EXPAND(node, problem), fringe)
```

closed list stores all expanded nodes

Summary of *uninformed* algorithms

Criterion	Breadth-First	Uniform-cost	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
Complete?	YES*	YES*	NO	YES, if *limit* ≥ d	YES	YES*
Time	b^{d+1}	$b^{C^*/e}$	b^m	b^l	b^d	$b^{d/2}$
Space	b^{d+1}	$b^{C^*/e}$	bm	bl	bd	$b^{d/2}$
Optimal?	YES*	YES*	NO	NO	YES	YES

Informed search

General approach of informed search:

- „best-first search“: node *n* is selected for expansion based on an evaluation function *f(n)*.
- use problem-specific knowledge beyond problem definition

idea: evaluation function *hints* to costs of the solution, i.e. the path from start to goal via node *n*

- Choose node which *appears* best („seemingly-best-first“)

Implementation:

- *fringe* is queue sorted in increasing order of evaluation f(n)

special cases: Greedy search, A* search

Heuristic evaluation function

Heuristic [dictionary]:
“A rule of thumb, simplification, or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood.”

most common and easy way to impart additional problem knowledge to a search algorithm

h(n) = estimated cost of the cheapest path from node *n* to goal node

- constraint: if *n* is goal, then *h(n)=0*

Greedy best-first search

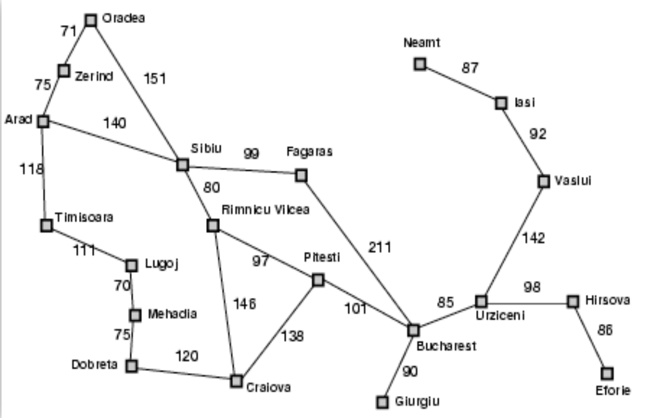
Evaluation function *f(n) = h(n)*

= estimate of cost from current state *n* to goal state

Greedy best-first search expands the node that *appears to be closest* to goal, i.e. executes the action that takes away as much as possible of the remaining costs (hence *greedy*)

Example:
$h_{SLD}(n)$:= straight-line distance from *n* to Bucharest

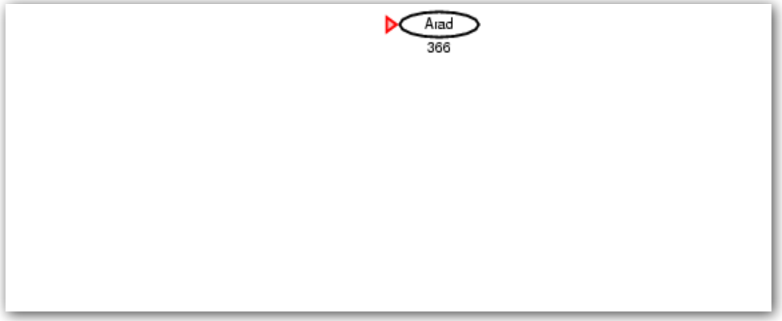
Romania with step costs in km



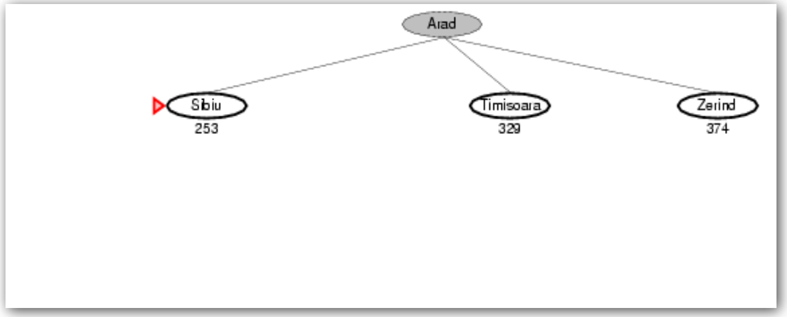
Straight–line distance to Bucharest

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

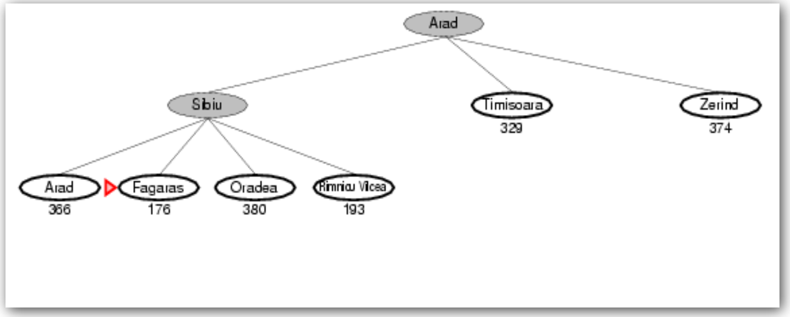
Greedy best-first search example



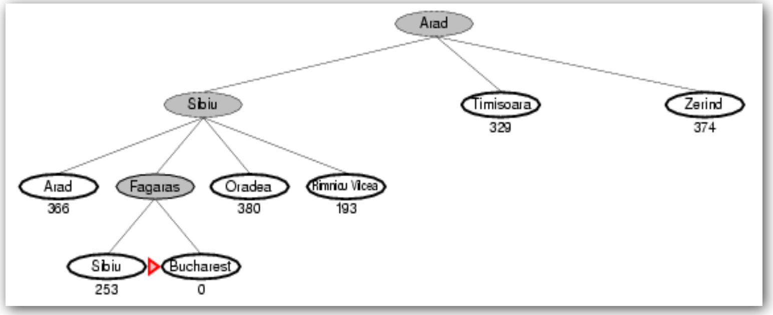
Greedy best-first search example



Greedy best-first search example



Greedy best-first search example



Finds solution without expanding a node not part of the solution, i.e. search costs are minimal

Properties of greedy best-first search

Complete?
no–can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt → ...

Time?
$O(b^m)$, but a good heuristic can give dramatic improvement

Space?
$O(b^m)$ -- keeps all nodes in memory

Optimal?
no! path via Sibiu and Fagaras is 32km longer than path through Rimnicu Vilcea and Pitesti.

A* search

Best-known form of best-first search

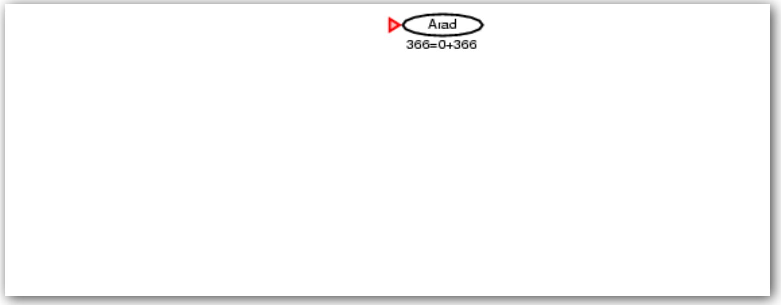
Idea:

- use adequate heuristics
- avoid expanding paths that are already expensive

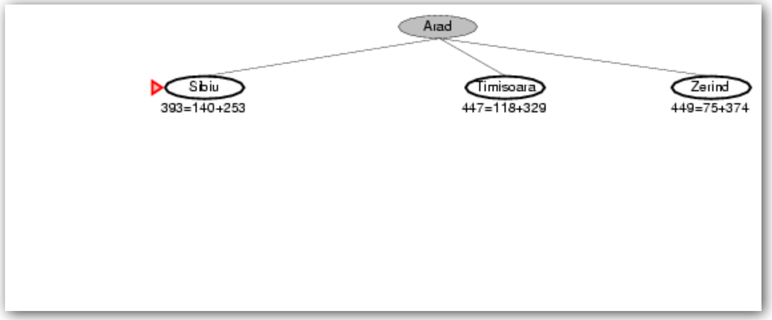
Evaluation function $f(n)=g(n) + h(n)$

- $g(n)$ the cost (so far) to reach the node
- $h(n)$ estimated cost to get from the node to the goal
- $f(n)$ estimated total cost of path through n to goal

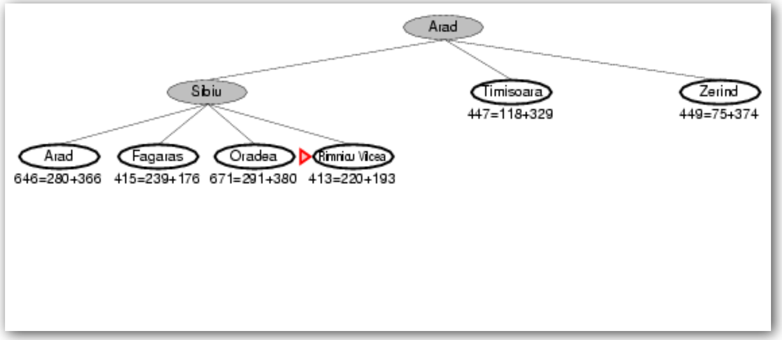
A* search example



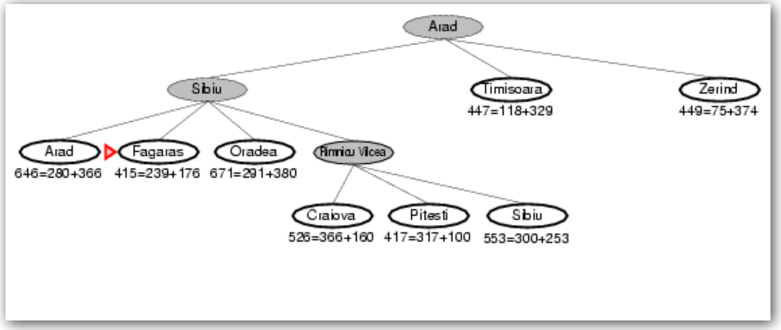
A* search example



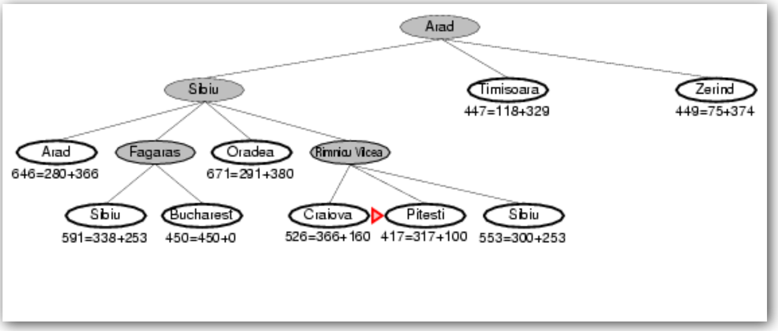
A* search example



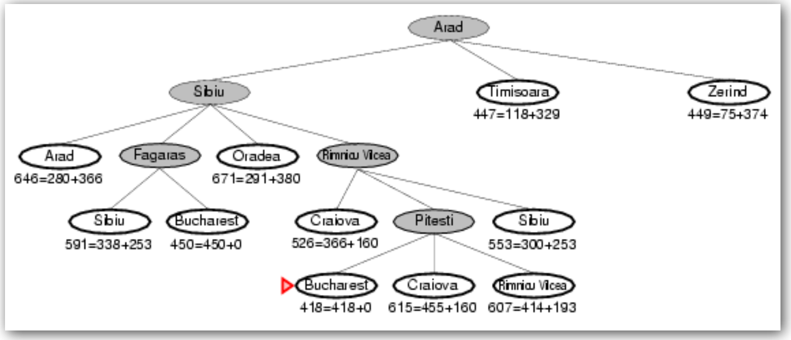
A* search example



A* search example



A* search example



A* uses an *admissible* heuristic

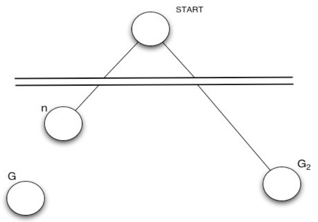
heuristic *h(n)* is admissible if for every node *n*: $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.

an admissible heuristic never over-estimates the cost to reach the goal, i.e., it is optimistic

Example: $h_{SLD}(n)$ (never overestimates the actual road distance)

Theorem: If *h(n)* is admissible, A* using `TREE-SEARCH` is optimal

Optimality of A* - standard proof



Suppose suboptimal *goal G2* generated, in the fringe

Let *n* be an unexpanded node on a shortest path to optimal goal *G*.

$f(G_2) = g(G_2)$, since $h(G_2)=0$

$f(G_2) > C$, with C cost of optimal solution

$f(n) = g(n)+h(n) <= C$, since h(n) admissible

thus $f(n) <= C < f(G_2)$, so *n* will be expanded before G_2

BUT ... A* graph search?

Because repeated states are prevented in graph search, can discard optimal path to a *repeated* state if not the first one generated

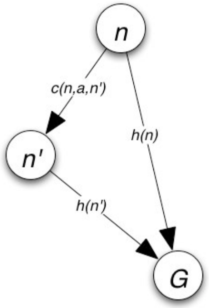
Two solutions:

- add extra book-keeping, i.e., remove the more expensive of two paths found to the same node
- ensure that optimal path to any repeated state is always the first one followed
  - ➡ holds with extra requirement on h(n): consistency

Consistency (a.k.a. monotonicity)

A heuristic is consistent if for every node *n* and every successor *n'* of *n* generated by any action *a*:

$$h(n) \leq c(n,a,n') + h(n')$$



Theorem: If *h(n)* is consistent, A* using `GRAPH-SEARCH` is optimal

If *h(n)* is consistent, the values of *f(n)* along any path are non-decreasing

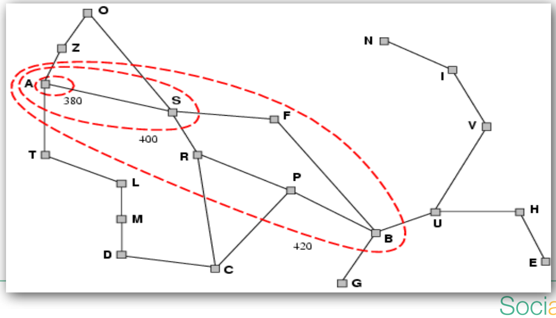
$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n,a,n') + h(n') \\ &\geq g(n) + h(n) \\ &\geq f(n) \end{aligned}$$

Optimality of A*

A* expands nodes in order of increasing *f* value, gradually adds "*f*-contours" of nodes

- contour *i* has all nodes with $f<=f_i$, where $f_i < f_{i+1}$
- uniform-cost search = A* with $h(n)=0$: contours are circles

the more correct the heuristics, the more the contours „focus“ on optimal path



Properties of A*

Complete? Yes (unless there are infinitely many nodes with $f <= f(G)$)

Time? exponential with path length

Space? all nodes are stored

Optimal? Yes

- Cannot expand f_{i+1} until f_i is finished.
- A* expands all nodes with $f(n) < C^*$ (cost of optimal solution)
- A* expands some nodes with $f(n) = C^*$ (on „goal contour“)
- A* expands no nodes with $f(n) > C^*$

A* is optimally efficient for given heuristic, no other algorith expands fewer nodes (except from ties)

Outlook

Further search algorithms

- IDA*: Iterative deepening A*
  - *f*-cost used as cut-off (instead of depth)
- RBFS: Recursive best-first search
  - recursive DF search with best alternative *f*-cost as limit for back-tracking
- MA* / SMA*: (Simplified) Memory bounded A*
  - limited memory
  - if memory is full, drops worst leaf node (highest *f*-cost) and backs up value of forgotten node to its parent
  - regenerates subtree not until all other paths turned out to be worse
  - ...can become a problem for computation time, if required often