4. Search Algorithms and Path-finding

Uninformed & informed search, online search, ResQ Freiburg path planner

Alexander Kleiner, Bernhard Nebel
Contents

• Problem-Solving Agents
• General Search (Uninformed search)
• Best-First Search (Informed search)
  – Greedy Search & A*
• Online Search
  – Real-Time Adaptive A*
• Case Study: ResQ Freiburg path planner
• Conclusion
Problem-Solving Agents

→ Goal-based agents

Formulation: *goal* and *problem*

Given: *initial state*

Task: To reach the specified goal (a state) through the execution of appropriate actions.

→ Search for a suitable action sequence and execute the actions
A Simple Problem-Solving Agent

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation

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

• **Goal formulation**
  World states with certain properties

• Definition of the **state space**
  important: only the relevant aspects → abstraction

• Definition of the **actions** that can change the world state

• Determination of the **search cost** (search costs, offline costs) and the execution costs (path costs, online costs)

**Note:** The type of problem formulation can have a big influence on the difficulty of finding a solution.
Problem Formulation for the Vacuum Cleaner World

- **World state space**: 2 positions, dirt or no dirt → 8 world states

- **Successor function (Actions)**: Left (L), Right (R), or Suck (S)

- **Goal state**: no dirt in the rooms

- **Path costs**: one unit per action
The Vacuum Cleaner State Space

States for the search: The world states 1-8.
Example: Missionaries and Cannibals

Informal problem description:

- Three missionaries and three cannibals are on one side of a river that they wish to cross.
- A boat is available that can hold at most two people and at least one.
- You must never leave a group of missionaries outnumbered by cannibals on the same bank.

→ Find an action sequence that brings everyone safely to the opposite bank.
Formalization of the M&C Problem

State space: triple \((x,y,z)\) with \(0 \leq x,y,z \leq 3\), where \(x\), \(y\), and \(z\) represent the number of missionaries, cannibals and boats currently on the original bank.

Initial State: \((3,3,1)\)

Successor function: From each state, either bring one missionary, one cannibal, two missionaries, two cannibals, or one of each type to the other bank.

Note: Not all states are attainable (e.g., \((0,0,1)\)), and some are illegal.

Goal State: \((0,0,0)\)

Path Costs: 1 unit per crossing
General Search

From the initial state, produce all successive states step by step → search tree.

(a) initial state

(b) after expansion of (3,3,1)

(c) after expansion of (3,2,0)
Implementing the Search Tree

*Data structure for nodes in the search tree:*

State: state in the state space

Node: Containing a state, pointer to predecessor, depth, and path cost, action

Depth: number of steps along the path from the initial state

Path Cost: Cost of the path from the initial state to the node

Fringe: Memory for storing expanded nodes. For example, a stack or a queue

*General functions to implement:*

Make-Node(state): Creates a node from a state

Goal-Test(state): Returns true if state is a goal state

Successor-Fn(state): Implements the successor function, i.e. expands a set of new nodes given all actions applicable in the state

Cost(state, action): Returns the cost for executing action in state

Insert(node, fringe): Inserts a new node into the fringe

Remove-First(fringe): Returns the first node from the fringe
General Tree-Search Procedure

**function TREE-SEARCH**(problem, fringe) **returns** 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) **returns** 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

Uninformed or blind searches:

No information on the length or cost of a path to the solution.

- breadth-first search, uniform cost search, depth-first search,
- depth-limited search, Iterative deepening search, and
- bi-directional search.

In contrast: informed or heuristic approaches
Criteria for Search Strategies

Completeness:
Is the strategy guaranteed to find a solution when there is one?

Time Complexity:
How long does it take to find a solution?

Space Complexity:
How much memory does the search require?

Optimality:
Does the strategy find the best solution (with the lowest path cost)?
Breadth-First Search (1)

Nodes are expanded in the order they were produced. fringe = Enqueue-at-end() (FIFO).

- Always finds the shallowest goal state first.
- Completeness.
- The solution is optimal, provided the path cost is a non-decreasing function of the depth of the node (e.g., when every action has identical, non-negative costs).
Breadth-First Search (2)

The costs, however, are very high. Let \( b \) be the maximal branching factor and \( d \) the depth of a solution path. Then the maximal number of nodes expanded is

\[
b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) \in O(b^{d+1})
\]

Example: \( b = 10 \), 10,000 nodes/second, 1,000 bytes/node:

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1,100</td>
<td>.11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>4</td>
<td>111,100</td>
<td>11 seconds</td>
<td>106 megabytes</td>
</tr>
<tr>
<td>6</td>
<td>(10^7)</td>
<td>19 minutes</td>
<td>10 gigabytes</td>
</tr>
<tr>
<td>8</td>
<td>(10^9)</td>
<td>31 hours</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>10</td>
<td>(10^{11})</td>
<td>129 days</td>
<td>101 terabytes</td>
</tr>
<tr>
<td>12</td>
<td>(10^{13})</td>
<td>35 years</td>
<td>10 petabytes</td>
</tr>
<tr>
<td>14</td>
<td>(10^{15})</td>
<td>3,523 years</td>
<td>1 exabyte</td>
</tr>
</tbody>
</table>

Note: One could easily perform the goal test BEFORE expansion, then the time & space complexity reduces to \( O(b^d) \)
Uniform Cost Search

Modification of breadth-first search to always expand the node with the lowest-cost $g(n)$.

Always finds the cheapest solution, given that $g(\text{successor}(n)) \geq g(n)$ for all $n$. 
Depth-First Search

Always expands an unexpanded node at the greatest depth

$fringe = \text{Enqueue-at-front (LIFO)}$.

Example (Nodes at depth 3 are assumed to have no successors):
Iterative Deepening Search (1)

- Combines depth- and breadth-first searches
- Optimal and complete like breadth-first search, but requires less memory

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution sequence
  inputs: problem, a problem

  for depth ← 0 to ∞ do
    if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
  end
  return failure
```

Iterative Deepening Search (2)

Example

Limit = 0

Limit = 1

Limit = 2

Limit = 3

......
Iterative Deepening Search (3)

Number of expansions

<table>
<thead>
<tr>
<th>Method</th>
<th>Expansion Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterative Deepening Search</td>
<td>((d)b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d)</td>
</tr>
<tr>
<td>Breadth-First-Search</td>
<td>(b + b^2 + \ldots + b^{d-1} + b^d + b^{d+1} - b)</td>
</tr>
</tbody>
</table>

Example: \(b = 10, d = 5\)

<table>
<thead>
<tr>
<th>Method</th>
<th>Expansion</th>
<th>Calculation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breadth-First-Search</td>
<td>(10 + 100 + 1,000 + 10,000 + 999,990)</td>
<td>(= 1,111,100)</td>
<td></td>
</tr>
<tr>
<td>Iterative Deepening Search</td>
<td>(50 + 400 + 3,000 + 20,000 + 100,000)</td>
<td>(= 123,450)</td>
<td></td>
</tr>
</tbody>
</table>

For \(b = 10\), only 11\% of the nodes expanded by breadth-first-search are generated, so that the time complexity is considerably lower.

Time complexity: \(O(b^d)\)  
Memory complexity: \(O(b \cdot d)\)

→ Iterative deepening in general is the preferred uninformed search method when there is a large search space and the depth of the solution is not known.
Bidirectional Search

As long as forwards and backwards searches are symmetric, search times of $O(2 \cdot b^{d/2}) = O(b^{d/2})$ can be obtained.

E.g., for $b=10$, $d=6$, instead of 111111 only 2222 nodes!
Comparison of Search Strategies

Time complexity, space complexity, optimality, completeness

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
<th>Bidirectional (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes(^a)</td>
<td>Yes(^{a,b})</td>
<td>No</td>
<td>No</td>
<td>Yes(^a)</td>
<td>Yes(^{a,d})</td>
</tr>
<tr>
<td>Time</td>
<td>O((b^{d+1}))</td>
<td>O((b[ C^* / \varepsilon ]))</td>
<td>O((b^m))</td>
<td>O((b^l))</td>
<td>O((b^d))</td>
<td>O((b^{d/2}))</td>
</tr>
<tr>
<td>Space</td>
<td>O((b^{d+1}))</td>
<td>O((b[ C^* / \varepsilon ]))</td>
<td>O((bm))</td>
<td>O((bl))</td>
<td>O((bd))</td>
<td>O((b^{d/2}))</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes(^c)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes(^c)</td>
<td>Yes(^{c,d})</td>
</tr>
</tbody>
</table>

\(b\)  branching factor
\(d\)  depth of solution,
\(m\)  maximum depth of the search tree,
\(l\)  depth limit,
\(C^*\) cost of the optimal solution,
\(\varepsilon\) minimal cost of an action

Superscripts:
\(a\)  \(b\) is finite
\(b\)  if step costs not less than \(\varepsilon\)
\(c\)  if step costs are all identical
\(d\)  if both directions use breadth-first search
Problems With Repeated States

• Tree search ignores what happens if nodes are repeatedly visited
  – For example, if actions lead back to already visited states
  – Consider path planning on a grid
• Repeated states may lead to a large (exponential) overhead

(a) State space with $d+1$ states, were $d$ is the depth
(b) The corresponding search tree which has $2^d$ nodes corresponding to the two possible paths!
(c) Possible paths leading to A
Graph Search

• Add a *closed* list to the tree search algorithm
• **Ignore** newly expanded state if already in *closed* list
• *Closed list* can be implemented as a hash table
• Potential problems
  – Needs a lot of memory
  – Can ignore better solutions if a node is visited first on a suboptimal path (e.g. IDS is not optimal anymore)
Best-First Search

Search procedures differ in the way they determine the next node to expand.

**Uninformed Search:** Rigid procedure with no knowledge of the cost of a given node to the goal.

**Informed Search:** Knowledge of the cost of a given node to the goal is in the form of an *evaluation function* $f$ or $h$, which assigns a real number to each node.

**Best-First Search:** Search procedure that expands the node with the “best” $f$- or $h$-value.
General Algorithm

When $h$ is always correct, we do not need to search!

```
function BEST-FIRST-SEARCH(problem, EVAL-FN) returns a solution sequence
    inputs: problem, a problem
             Eval-Fn, an evaluation function
    Queueing-Fn ← a function that orders nodes by EVAL-FN
    return GENERAL-SEARCH(problem, Queueing-Fn)
```
Greedy Search

A possible way to judge the “worth” of a node is to estimate its distance to the goal.

\[ h(n) = \text{estimated distance from } n \text{ to the goal} \]

The only real condition is that \( h(n) = 0 \) if \( n \) is a goal.

A best-first search with this function is called a greedy search.

The evaluation function \( h \) in greedy searches is also called a heuristic function or simply a heuristic.

→ In all cases, the heuristic is problem-specific and focuses the search!

Route-finding problem: \( h = \) straight-line distance between two locations.
Greedy Search Example

Straight-line distance to Bucharest

<table>
<thead>
<tr>
<th>Location</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobrota</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>178</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadia</td>
<td>241</td>
</tr>
<tr>
<td>Neamt</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitesti</td>
<td>98</td>
</tr>
<tr>
<td>Râmnicu Vâlcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
Greedy Search from *Arad* to *Bucharest*

However: *Arad*→*Sibiu*→*Fagaras*→*Bucharest* = 450
*Arad*→*Sibiu*→*Rimnicu*→*Pitesti*→*Bucharest* = 418!
**A*: Minimization of the estimated path costs**

A* combines the greedy search with the uniform-cost-search, i.e. taking costs into account.

\[ g(n) = \text{actual cost from the initial state to } n. \]

\[ h(n) = \text{estimated cost from } n \text{ to the next goal.} \]

\[ f(n) = g(n) + h(n), \text{ the estimated cost of the cheapest solution through } n. \]

Let \( h^*(n) \) be the true cost of the optimal path from \( n \) to the next goal.

\( h \) is admissible if the following holds for all \( n \) :

\[ h(n) \leq h^*(n) \]

We require that for optimality of A*, \( h \) is admissible (straight-line distance is admissible).
A* Search Example

Straight-line distance to Bucharest

- Arad: 366
- Bucharest: 0
- Craiova: 160
- Dobrota: 242
- Eforie: 181
- 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
A* Search from Arad to Bucharest

\[ f = 220 + 193 = 413 \]
Heuristic Function Example

\[ h_1 = \] the number of tiles in the wrong position
\[ h_2 = \] the sum of the distances of the tiles from their goal positions \((Manhatten\ distance)\)
Empirical Evaluation

- $d =$ distance from goal
- Average over 100 instances

<table>
<thead>
<tr>
<th>$d$</th>
<th>IDS</th>
<th>$A^*(h_1)$</th>
<th>$A^*(h_2)$</th>
<th>$d$</th>
<th>IDS</th>
<th>$A^*(h_1)$</th>
<th>$A^*(h_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>2.45</td>
<td>1.79</td>
<td>1.79</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
<td>12</td>
<td>2.87</td>
<td>1.48</td>
<td>1.45</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
<td>18</td>
<td>2.73</td>
<td>1.34</td>
<td>1.30</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>39</td>
<td>25</td>
<td>2.80</td>
<td>1.33</td>
<td>1.24</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
<td>39</td>
<td>2.79</td>
<td>1.38</td>
<td>1.22</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>364404</td>
<td>227</td>
<td>73</td>
<td>2.78</td>
<td>1.42</td>
<td>1.24</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3473941</td>
<td>539</td>
<td>113</td>
<td>2.83</td>
<td>1.44</td>
<td>1.23</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>1301</td>
<td>211</td>
<td>–</td>
<td>1.45</td>
<td>1.25</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>–</td>
<td>3056</td>
<td>363</td>
<td>–</td>
<td>1.46</td>
<td>1.26</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>–</td>
<td>7276</td>
<td>676</td>
<td>–</td>
<td>1.47</td>
<td>1.27</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>–</td>
<td>18094</td>
<td>1219</td>
<td>–</td>
<td>1.48</td>
<td>1.28</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>–</td>
<td>39135</td>
<td>1641</td>
<td>–</td>
<td>1.48</td>
<td>1.26</td>
<td></td>
</tr>
</tbody>
</table>
A* Implementation Details

• How to code A* efficiently?
• Costly operations are:
  – Insert & lookup an element in the closed list
  – Insert element & get minimal element (f-value) from open list
• The closed list can efficiently be implemented as a hash set
• The open list is typically implemented as a priority queue, e.g. as
  – Fibonacci heap, binomial heap, k-level bucket, etc.
  – binary-heap with $O(\log n)$ is normally sufficient
• Hint: see priority queue implementation in the “Java Collection Framework”
Online search

- Intelligent agents usually don`t know the state space (e.g. street map) exactly in advance
  - Environment can dynamically change!
  - True travel costs are experienced during execution
- Planning and plan execution are interleaved
- Example: RoboCup Rescue
  - The map is known, but roads might be blocked from building collapses
  - Limited drivability of roads depending on traffic volume
- Important issue: How to reduce computational cost of repeated A* searches!
Online search

• **Incremental heuristic search**
  – Repeated planning of the complete path from current state to goal
  – Planning under the free-space assumption
  – Optimized versions reuse information from previous planning episodes:
    • Focused Dynamic A* (D*) [Stenz95]
      – Used by DARPA and NASA
    • D* Lite [Koenig et al. 02]
      – Similar as D* but a bit easier to implement (claim)
  – In particular, these methods reuse closed list entries from previous searches
  – All Entries that have been compromised by weight updates (from observation) are adjusted accordingly

• **Real-Time Heuristic search**
  – Repeated planning with limited look-ahead (agent centered search)
  – Solutions can be suboptimal but faster to compute
  – Updated of heuristic values of visited states
    • Learning Real-Time A* (LRTA*) [Korf90]
    • Real-Time Adaptive A* (RTAA*) [Koenig06]
Real-Time Adaptive A* (RTAA*)

- Executes A* plan with limited lookahead
- Learns better informed heuristic $H(s)$ from experience (initially $h(s)$, e.g. Euclidian distance)
- Lookahed defines trade-off between optimality and computational cost

```plaintext
while ($s_{curr} \not\in$ GOAL)
    $astar($lookahead$);
    if ($s' = FAILURE$) then
        return FAILURE;
    for all $s \in$ CLOSED do
        $H(s) := g(s')+h(s')-g(s)$;
    end;
    execute($plan$);
end;
return SUCCESS;
```

$s'$: last state expanded during previous A* search
After first A* planning with lookahead until $s'$:
$g(s')=7$, $h(s')=6$, $f(s')=13$
$g(s)=2$, $h(s)=3$

Update of each element in CLOSED list, e.g.:
$H(s) = g(s') + h(s') - g(s)$
$H(s) = 7 + 6 - 2 = 11$
Real-Time Adaptive A* (RTAA*)

A* vs. RTAA*

A* expansion

RTAA* expansion (inf. Lookahead)
Case Study: ResQ Freiburg path planner

Requirements

- Rescue domain has some special features:
  - Interleaving between planning and execution is within large time cycles
  - Roads can be merged into “longroads”

- Planner is not used only for path finding, also for task assignment
  - For example, prefer high utility goals with low path costs
  - Hence, planner is frequently called for different goals

- Our decision: Dijkstra graph expansion on longroads
Case Study: ResQ Freiburg path planner

Longroads

- RoboCup Rescue maps consist of buildings, nodes, and roads
  - Buildings are directly connected to nodes
  - Roads are inter-connected by crossings
- For efficient path planning, one can extract a graph of longroads that basically consists of road segments that are connected by crossings
Case Study: ResQ Freiburg path planner

Approach

• Reduction of street network to longroad network
• Caching of planning queries (useful if same queries are repeated)
• Each agent computes two Dijkstra graphs, one for each nearby longroad node
• Selection of optimal path by considering all 4 possible plans
• Dijkstra graphs are recomputed after each perception update (either via direct sensing or communication)
• Additional features:
  – Parameter for favoring unknown roads (for exploration)
  – Two more Dijkstra graphs for sampled time cost (allows time prediction)
Case Study: ResQ Freiburg path planner
Dijkstra’s Algorithm (1)

Single Source Shortest Path, i.e. finds the shortest path from a single node to all other nodes.

Worst case runtime $O(|E| \log |V|)$, assuming $E>V$, where $E$ is the set of edges and $V$ the set of vertices.
  - Requires efficient priority queue
Case Study: ResQ Freiburg path planner
Dijkstra’s Algorithm (2)

Graph expansion

```plaintext
function Dijkstra(Graph, source):
    for each vertex v in Graph:        // Initializations
        dist[v] := infinity            // Unknown distance function from source to v
        previous[v] := undefined      // Previous node in optimal path from source
    dist[source] := 0                   // Distance from source to source
    Q := the set of all nodes in Graph  // All nodes in the graph are unoptimized - thus are in Q
    while Q is not empty:               // The main loop
        u := node in Q with smallest dist[]
        remove u from Q
        for each neighbor v of u:        // where v has not yet been removed from Q.
            alt := dist[u] + dist_between(u, v)  // be careful in 1st step - dist[u] is infinity yet
            if alt < dist[v]:             // Relax (u,v)
                dist[v] := alt
                previous[v] := u
        return previous[]
```

Pseudo code taken from Wikipedia

Extracting path to target

```plaintext
S := empty sequence
u := target
while defined previous[u]
    insert u at the beginning of S
    u := previous[u]
```

Pseudo code taken from Wikipedia
Summary

• Before an agent can start searching for solutions, it must formulate a goal and then use that goal to formulate a problem.

• A problem consists of five parts: The state space, initial situation, actions, goal test, and path costs. A path from an initial state to a goal state is a solution.

• A general search algorithm can be used to solve any problem. Specific variants of the algorithm can use different search strategies.

• Search algorithms are judged on the basis of completeness, optimality, time complexity, and space complexity.

• Heuristics focus the search

• Best-first search expands the node with the highest worth (defined by any measure) first.

• With the minimization of the evaluated costs to the goal $h$ we obtain a greedy search.

• The minimization of $f(n) = g(n) + h(n)$ combines uniform and greedy searches. When $h(n)$ is admissible, i.e., $h^*$ is never overestimated, we obtain the A* search, which is complete and optimal.

• Online search provides method that are computationally more efficient when planning and plan execution are tightly coupled
Literature

• On my homepage:

• Homepage of Tony Stentz:

• Homepage of Sven Koenig:

• Harder to find, also explained in the AIMA book (2nd ed.):
  – Demo search code in Java on the AIMA webpage http://aima.cs.berkeley.edu/