Foundations of AI

4. Informed Search Methods

Heuristics, Local Search Methods, Genetic Algorithms

Wolfram Burgard & Bernhard Nebel
Contents

• Best-First Search
• A* and IDA*
• Local Search Methods
• Genetic Algorithms
Best-First Search

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

**Uninformed Search**: Rigid procedure with no knowledge of how “good” a node is.

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

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

```plaintext
function BEST-FIRST-SEARCH\(\text{problem, Eval-FN}\) 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\(\text{problem, Queueing-Fn}\)
```

When \textit{Eval-Fn} is always correct, we don’t need to search!
Greedy Search

A possible way to judge the “worthiness” 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 restriction is that \( h(n) = 0 \) if \( n \) is a goal.

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

Example for *route-finding* problem: \( h = \) straight-line distance between two locations.
Greedy Search Example:
From Arad to Bucharest

Straight-line distance to Bucharest

<table>
<thead>
<tr>
<th>City</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>Rmnicu Vilcea</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
Problems with Greedy Search

• Does find the *suboptimal solutions*
  – Would be *Arad – Sibiu – Rimnicu Vilcea – Pitesti – Bucharest*

• Can be *misleading*
  – What happens if we want to go from *Iasi* to *Fagaras*?

• Can be *incomplete* (if we do not detect duplicates) in the above case
Heuristics

The evaluation function $h$ in greedy searches is also called a *heuristic function* or simply a *heuristic*.

- The word *heuristic* is derived from the Greek word ἡ ἐγκατάστασις (note also: Ἑὐρέως).

- The mathematician Polya introduced the word in the context of problem solving techniques.

- In AI it has two meanings:
  - Heuristics are fast but in certain situations incomplete methods for problem-solving [Newell, Shaw, Simon 1963]
  - Heuristics are methods that focus the search without leading to incompleteness.

→ In all cases, the heuristic is *problem-specific* and *focuses* the search!
A*: Minimization of the Total Estimated Path Costs

A* combines the greedy search with the uniform-search strategy.

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

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

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

Let \( h^*(n) \) be the \textbf{actual cost} of the optimal path from \( n \) to the closest goal.

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

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

We require that for A*, \( h \) is admissible.

(Straight-line distance is admissible)
A* 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>Rimnicu Vilea</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>
A* Search from *Arad* to *Bucharest*
Contours in A*

Within the search space, contours arise in which for the given $f$-value all nodes are expanded.

Contours at $f = 380, 400, 420$
Optimality of A*

**Claim:** The first solution found in *tree search* has the minimum path cost (for *graph search* it is more difficult)

**Proof:** Suppose there exists a goal node G with optimal path cost \( C^* \), but A* has found first another node \( G_2 \) with \( g(G_2) > C^* \), i.e. \( f(G_2) > C^* \).

Let \( n \) be a node on the path from the start to G that has not yet been expanded.

Since \( h \) is admissible, we have

\[
f(n) = g(n) + h(n) \leq C^*.
\]

Since

\[
f(n) \leq C^* < f(G_2),
\]

\( n \) should have been expanded first!
Completeness and Complexity

**Completeness:** If a solution exists, A* will find one, provided that (1) every node has a finite number of successor nodes, and (2) there exists a positive constant $\delta$ such that every operator has at least cost $\delta$.

$\Rightarrow$ Only a finite number of nodes $n$ with $f(n) \leq f^*$.

**Complexity:** In the case where $|h^*(n) - h(n)| \leq O(\log(h^*(n)))$, only a sub-exponential number of nodes will be expanded.

Normally, growth is exponential because the error is proportional to the path costs. So, modify to look for suboptimal solutions and allow non-admissible heuristics!
Iterative Deepening A* Search (IDA*)

Idea: A combination of IDS and A*. All nodes inside a contour are searched in a DFS manner.

```
function IDA*(problem) returns a solution sequence
    inputs: problem, a problem
    static: f-limit, the current f- COST limit
        root, a node

    root ← MAKE-NODE(INITIAL-STATE[problem])
    f-limit ← f- COST(root)
    loop do
        solution, f-limit ← DFS-CONTOUR(root, f-limit)
        if solution is non-null then return solution
        if f-limit = ∞ then return failure; end

function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f- COST limit
    inputs: node, a node
        f-limit, the current f- COST limit
    static: next-f, the f- COST limit for the next contour, initially ∞

    if f- COST[node] > f-limit then return null, f- COST[node]
    if GOAL-TEST[problem](STATE[node]) then return node, f-limit
    for each node s in SUCCESSORS(node) do
        solution, new-f ← DFS-CONTOUR(s, f-limit)
        if solution is non-null then return solution, f-limit
        next-f ← MIN(next-f, new-f); end
    return null, next-f
```
RBFS: Recursive Best-First Search

Avoid re-evaluation of nodes but keep only $O(bd)$ nodes in memory

```
function RBFS(problem, node, f_limit) returns a solution, or failure
    if Goal-Test(problem)(state) then return node
    successors ← Expand(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f[s] ← max(g(s) + h(s), f[node])
    repeat
        best ← the lowest f-value node in successors
        if f[best] > f_limit then return failure, f[best]
        alternative ← the second-lowest f-value among successors
        result, f[best] ← RBFS(problem, best, min(f_limit, alternative))
        if result ≠ failure then return result
```
How to Design a Heuristic

• Simplify the problem (by removing restrictions), creating a **relaxation:**
  – so that it becomes **easy to solve**
  – usually leading to **shorter solutions**
  – and making it easy to **determine optimal solutions** for the relaxation

• Examples:
  – straight line distance
  – simplify movement restrictions in multi-body problems (ignore collisions)
  – ignore negative effects
Example Heuristics

\[ h_1 = \text{the number of tiles in the wrong position} \]
\[ h_2 = \text{the sum of the distances of the tiles from their goal positions} \]
(Manhattan distance)
Empirical Evaluation for IDS vs. A*

- \(d = \text{distance from goal}\)
- \(\text{Average over 100 instances}\)

<table>
<thead>
<tr>
<th>(d)</th>
<th>IDS</th>
<th>(A^*(h_1))</th>
<th>(A^*(h_2))</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>
</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>
</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>
</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>
</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>
</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>
</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>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>1301</td>
<td>211</td>
<td>–</td>
<td>1.45</td>
<td>1.25</td>
</tr>
<tr>
<td>18</td>
<td>–</td>
<td>3056</td>
<td>363</td>
<td>–</td>
<td>1.46</td>
<td>1.26</td>
</tr>
<tr>
<td>20</td>
<td>–</td>
<td>7276</td>
<td>676</td>
<td>–</td>
<td>1.47</td>
<td>1.27</td>
</tr>
<tr>
<td>22</td>
<td>–</td>
<td>18094</td>
<td>1219</td>
<td>–</td>
<td>1.48</td>
<td>1.28</td>
</tr>
<tr>
<td>24</td>
<td>–</td>
<td>39135</td>
<td>1641</td>
<td>–</td>
<td>1.48</td>
<td>1.26</td>
</tr>
</tbody>
</table>
Local Search Methods

- In many problems, it is not possible to explore the search space systematically.
- If a quality measure (or objective function) for states is given, then local search can be used to find solutions.
- Begin with a randomly-chosen configuration/state and improve on it stepwise → Hill Climbing.
- Incomplete, but works for very large spaces.
- Has been used for IC design, scheduling, network optimization, … , 8-queens, …
Hill Climbing

```
function HILL-CLIMBING(problem) returns a solution state
inputs: problem, a problem
static: current, a node
        next, a node

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
    next ← a highest-valued successor of current
    if VALUE[next] < VALUE[current] then return current
    current ← next
end
```
The Landscape: 2D Example
Example: 8 Queens

An 8-queens state with evaluation value 17 (violations), showing the value for all successors (when moving a queen in its column)
Problems with Local Search Methods

• **Local maxima**: The algorithm finds a sub-optimal solution.

• **Plateaus (shoulders, flat local maxima)**: Here, the algorithm can only explore at random (or exhaustively)

• **Ridges**: Similar to plateaus.

**Solutions:**

• **Restart randomly** when no progress is being made.

• “Inject noise” → random walk

• **Tabu search**: Do not apply the last $n$ operators.

Which strategies (with which parameters) prove successful (within a problem class) can usually only empirically be determined.
Simulated Annealing

In the simulated annealing algorithm, “noise” is injected systematically: first a lot, then gradually less.

```plaintext
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
    inputs: problem, a problem
             schedule, a mapping from time to “temperature”
    static: current, a node
             next, a node
             T, a “temperature” controlling the probability of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T=0 then return current
        next ← a randomly selected successor of current
        ΔE ← VALUE[next] − VALUE[current]
        if ΔE > 0 then current ← next
        else current ← next only with probability e^{ΔE/T}
```

Has been used since the early 80’s for VSLI layout and other optimization problems.
Genetic Algorithms

Evolution appears to be very successful at finding good solutions.

Idea: Similar to evolution, we search for solutions by “cross over”, “mutation”, and “selection” successful solutions.

Ingredients:

- Coding of a solution into a string of symbols or bit-string
- A fitness function to judge the fitness of configurations
- A population of configurations

Example: 8-queens problem as a chain of 8 numbers. Fitness is judged by the number of non-attacks. The population consists of a set of arrangements of queens.
Selection, Mutation, and Crossover

- **Population**
  - Selection of individuals according to fitness function
  - Selection
  - Determination of crossover point and recombination
  - Mutation
  - With a fixed small probability, something in the bit string is changed
Example: 8-Queens
Case-Study: Path Planning in Robotic Soccer
Possible Approaches

- **Reactive**: Compute a motor control command based on current observation and goal location
  - try to move towards the goal in a straight line and drive around obstacles
  - May get stuck in local optima
- **Deliberative**: Generate a (optimal) path plan to the goal location
Simplifying Assumptions

• We do not want to / cannot solve the continuous control problem

• Discretization: 10 cm, π/16, …

• Movements of other objects are known (or assumed to be irrelevant)

• Adaptation to dynamic change is achieved by continuous re-planning
Searching in 5D

• Consider the space generated by
  – location \((x,y)\)
  – orientation \((\theta)\)
  – translational velocity \((v)\)
  – Rotational velocity \((\omega)\)

• Search in this space using A* in order to find the fastest way to the goal configuration
  – Computationally too expensive even on current hardware (250 msec for a 2m path, while we needed around 10 msec on a 100 MHz Pentium)
Further simplifications

• Consider only 2D space (location) and search for shortest path (ignoring orientation)
• Assume regular shape: circle
• Reduce robot to point and use obstacle growing
• Apply visibility graph method
• Solve by using $A^*$
Obstacle Growing
Navigating Around Circles

dots

goal
The Visibility Graph: Compute all common visible tangents
Searching in the Visibility Graph

- The visibility map can now be searched as we can search in a road map using straight line distance as the *heuristic estimate*

- Note:
  - State space is *very limited*
  - Optimal solution is not necessarily an *optimal solution* for the original problem
  - Shortest path is neither the *most safe* nor the *fastest* path
**Summary (1)**

- **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**.
Summary (2)

• There are many variations of A*

• **Local search methods** only ever work on one state, attempting to improve it step-wise.

• **Genetic algorithms** imitate evolution by combining good solutions. General contribution not clear yet.

• There are no turnkey solutions, you always have to try and tweak