4. Informed Search Methods

Heuristics, Local Search Methods, Genetic Algorithms

Wolfram Burgard, Andreas Karwath, Bernhard Nebel, and Martin Riedmiller

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

```plaintext
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)
```

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

Route-finding problem: \( h = \text{straight-line distance between two locations} \).

Greedy Search Example

Heuristics

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

- The word practical is derived from the Greek word εὑρίσκειν (note also: εὑρήκας!)
- 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] (The greedy search is actually generally incomplete).
  - Heuristics are methods that improve the search in the average-case.

\[ \text{In all cases, the heuristic is problem-specific and focuses the search!} \]
A*: Minimization of the estimated path costs

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

\[ g(n) = \text{actual cost from the initial state to } n. \]
\[ h(n) = \text{estimated cost from } n \text{ to the next goal.} \]
\[ f(n) = g(n) + h(n), \text{ the estimated cost of the cheapest solution through } n. \]

Let \( h^*(n) \) be the actual 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 A*, \( h \) is admissible (straight-line distance is admissible).

A* Search Example

Contours in A*

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

Contours at \( f = 380, 400, 420 \)
Example: Path Planning for Robots in a Grid-World

Optimality of A*

Claim: The first solution found has the minimum path cost.

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

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

\[ f(n) \leq f^*. \]

Since $n$ was not expanded before $G_2$, the following must hold:

\[ f(G_2) \leq f(n) \]

and

\[ f(G_2) \leq f^*. \]

It follows from $h(G_2) = 0$ that

\[ g(G_2) \leq f^*. \]

→ Contradicts the assumption!

Completeness and Complexity

Completeness:
If a solution exists, $A^*$ will find it 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 $\delta$.

\[ \rightarrow \text{Only a finite number of nodes } n \text{ with } f(n) \leq f^*. \]

Complexity:
In the case where $|h^*(n) - h(n)| \leq O(\log(h^*(n)))$, only a sub-exponential number of nodes will be expanded – provided the search space is a tree and there is only one goal state. This, however, is a quite unrealistic assumption [Helmert & Roeger, 2008] (best AAAI paper 2008)

 Normally, growth is exponential because the error is proportional to the path costs.
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>Search Cost</th>
<th>Effective Branching Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$</td>
<td>A*(h_1)</td>
<td>A*(h_2)</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>39</td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
</tr>
<tr>
<td>12</td>
<td>364404</td>
<td>227</td>
</tr>
<tr>
<td>14</td>
<td>3473941</td>
<td>539</td>
</tr>
<tr>
<td>16</td>
<td>370100</td>
<td>1301</td>
</tr>
<tr>
<td>18</td>
<td>-</td>
<td>3056</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>7276</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>18094</td>
</tr>
<tr>
<td>24</td>
<td>-</td>
<td>39135</td>
</tr>
</tbody>
</table>

Iterative Deepening A* Search (IDA*)

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

```
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-COUNTOR(root, f-limit)
    if solution is non-null then return solution
    if f-limit = \infty then return failure; end
```

```
function DFS-COUNTOR(node, f-limit) returns a solution sequence and a new f-COST limit
inputs: node, a node
static: next_f, the f-COST limit for the next contour, initially \infty
if f-COST(node) \leq f-limit then return null, f-COST[node]
if GOAL-TEST(problem[node]) then return node, f-limit
for each node n in SUCCESSORS(node) do
    solution, next_f = DFS-COUNTOR(n, f-limit)
    if solution is non-null then return solution, f-limit
next_f = MIN(next_f, new_f); end
return null, next_f
```

Local Search Methods

In many problems, it is unimportant how the goal is reached – only the goal itself matters (8-queens problem, VLSI Layout, TSP).

If in addition a quality measure for states is given, a local search can be used to find solutions.

Idea: Begin with a randomly-chosen configuration and improve on it stepwise \rightarrow Hill Climbing.
Hill Climbing

```python
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
```

Example: 8-Queens Problem

Selects a column and moves the queen to the square with the fewest conflicts.

Problems with Local Search Methods

- **Local maxima**: The algorithm finds a sub-optimal solution.
- **Plateaus**: Here, the algorithm can only explore at random.
- **Ridges**: Similar to plateaus.

Solutions:

- **Start over** when no progress is being made.
- “Inject smoke” $\rightarrow$ random walk
- **Tabu search**: Do not apply the last $n$ operators.

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

Simulated Annealing

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

```python
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 $t$ in 1 to $\infty$
        $T$ <- schedule($t$)
        if $T > 0$ then return current
        next <- a randomly selected successor of current
        $\Delta$ <- VALUE(next) - VALUE(current)
        if $\Delta > 0$ then current <- next
        else current <- next only with probability $e^{\Delta/T}$
```

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 “crossing”, “mutating”, and “selecting” successful solutions.

Ingredients:
- Coding of a solution into a string of symbols or bit-string
- A fitness function to judge the worth 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.

Summary

- **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.
- **IDA** is a combination of the iterative-deepening and A* searches.
- **Local search methods** only ever work on one state, attempting to improve it step-wise.
- **Genetic algorithms** imitate evolution by combining good solutions.