Problem Solving: Informed (Heuristic) Search

- **Uninformed (blind)** search algorithms can find an (optimal) solution to the problem, but they are usually not very efficient.

- **Informed (heuristic)** search algorithms can find solutions more efficiently thanks to exploiting problem-specific knowledge.

  - How to use heuristics in search?
    - BFS, A*, IDA*, RBF, SMA*
  
  - How to build heuristics?
    - relaxation, pattern databases

Information in search

- Recall that we are looking for (the shortest) path from the initial state to some goal state.

- Which information can help the search algorithm?
  - For example, the length of path to some goal state.
  - However such information is usually not available (if it is available then we do not need to do search). Usually some **evaluation function** \( f(n) \) is used to evaluate „quality“ of node \( n \) based on the length of path to the goal.

- **best-first search**
  - The node with the smallest value of \( f(n) \) is used for expansion.
  - There are search algorithms with different views of \( f(n) \). Usually the part of \( f(n) \) is a **heuristic function** \( h(n) \) estimating the length of the shortest (cheapest) path to the goal state.
  - Heuristic functions are the most common form of additional information given to search algorithms.
  - We will assume that \( h(n) = 0 \iff n \) is goal.

Greedy best-first search

- Let us try to expand first the node that is closest to some goal state, i.e. \( f(n) = h(n) \).
  - greedy best-first search algorithm

Example (path Arad → Bucharest):
- We have a table of direct distances from any city to Bucharest.
- Note: this information was not part of the original problem formulation!

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>0</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Cluj-Napoca</td>
<td>340</td>
</tr>
<tr>
<td>Debraș</td>
<td>523</td>
</tr>
<tr>
<td>Timisoara</td>
<td>67</td>
</tr>
</tbody>
</table>

Nejkratší cesta?
Greedy best-first search: analysis

- We already know that the greedy algorithm may not find the optimal path.
- Can we at least always find some path?
  - If we expand first the node with the smallest cost then the algorithm may not find any solution.
    - Example: path Iasi → Fagaras
      - Go to Neamt, then back to Iasi, Neamt, ...
      - We need to detect repeated visits in cities!

- Time complexity $O(b^m)$, where $m$ is the maximal depth
- Memory complexity $O(b^m)$
- A good heuristic function can significantly decrease the practical complexity.

Properties of A*

What about completeness and optimality of A*?
First a few definitions:
- admissible heuristic $h(n)$
  - $h(n) = \text{the cost of the cheapest path from n to goal}$
  - an optimistic view (the algorithm assumes a better cost than the real one)
  - function $f(n)$ in A* is a lower estimate of the cost of path through n
- monotous (consistent) heuristic $h(n)$
  - let $n'$ be a successor of $n$ via action $a$ and $c(n,a,n')$ be the transition cost
  - $h(n) \leq c(n,a,n') + h(n')$
  - this is a form of triangle inequality

Monotonous heuristic is admissible.

Let $n_1, n_2, ..., n_k$ be the optimal path from $n_1$ to goal $n_k$, then
$h(n_1) + h(n_{i+1}) \leq c(n_i,a,n_{i+1})$, via monotony
$h(n_k) = \sum_{i=1}^{k-1} c(n_i,a,n_{i+1})$, after „sum”

For a monotous heuristic the values of $f(n)$ are not decreasing over any path.
Let $n'$ be a successor of $n$, i.e. $g(n') = g(n) + c(n,a,n')$, then
$f(n') = g(n') + h(n') = g(n) + c(n,a,n') + h(n') \leq g(n) + h(n) = f(n)$

Algorithm A*:

- Let us now try to use $f(n) = g(n) + h(n)$
  - Recall that $g(n)$ is the cost of path from root to $n$
  - probably the most popular heuristic search algorithm
  - $f(n)$ represents the cost of path through $n$
  - the algorithm does not extend already long paths

- If $h(n)$ is an admissible heuristic then the algorithm A* in TREE-SEARCH is optimal.
  - in other words – the first expanded goal is optimal
  - Let $G_2$ be sub-optimal goal from the fringe and $C^*$ be the optimal cost
    - $f(G_2) = g(G_2) + h(G_2) = g(G_2) > C^*$, because $h(G_2) = 0$
  - Let $n$ be a node from the fringe and being on the optimal path
    - $f(n) = g(n) + h(n) \leq C^*$, via admissibility of $h(n)$
    - together
      - $f(n) \leq C^* < f(G_2)$,
      - i.e., the algorithm must expand $n$ before $G_2$ and this way it finds the optimal path.
**Algorithm A*: optimality**

- If \( h(n) \) is a monotonous heuristic then the algorithm A* in GRAPH-SEARCH is optimal.
  - Possible problem: reaching the same state for the second time using a better path – classical GRAPH-SEARCH ignores this second path!
  - A possible solution: selection of better from both paths leading to a close node (extra bookkeeping) or using monotonous heuristic.
    - for monotonous heuristics, the values of \( f(n) \) are not decreasing over any path
    - A* selects for expansion the node with the smallest value of \( f(n) \), i.e., the values \( f(m) \) of other open nodes \( m \) are not smaller, i.e., among all “open” paths to \( n \) there cannot be a shorter path than the path just selected (no path can shorten)
  - hence, the first closed goal node is optimal

**Algorithm A*: properties**

- For non-decreasing function \( f(n) \) we can draw contours in the state graph (the nodes inside a given contour have \( f \)-costs less than or equal to the contour value.
  - for \( h(n) = 0 \) we obtain circles around the start
  - for more accurate \( h(n) \) we use, the bands will stretch toward the goal state and become more narrowly focused around the optimal path.
    - A* expands all nodes such that \( f(n) < C^* \) on the contour
    - A* can expand some nodes such that \( f(n) = C^* \)
    - the nodes \( n \) such that \( f(n) > C^* \) are never expanded
    - the algorithm A* is optimality efficient for any given consistent heuristic

**Time complexity:**
- A* can expand an exponential number of nodes
  - this can be avoided if \( |h(n) - h^*(n)| \leq O(\log h^*(n)) \), where \( h^*(n) \) is the cost of optimal path from \( n \) to goal

**Space complexity:**
- A* keeps in memory all expanded nodes
- A* usually runs out of space long before it runs out of time

**Iterative-deepening A***

- A simple way to decrease memory consumption is iterative deepening.

**Algorithm IDA***

- the search limit is defined using the cost \( f(n) \) instead of depth
- for the next iteration we use the smallest value \( f(n) \) of node \( n \) that exceeded the limit in the last iteration
- frequently used algorithm

**Recursive best-first search**

- Let us try to mimic standard best-first search, but using only linear space.
  - the algorithm stops exploration if there is an alternative path with better cost \( f(n) \)
  - when the algorithm goes back to node \( n \), it replaces the value \( f(n) \) using the cost of successor’s (remembers the best leaf in the forgotten subtree)
- If \( h(n) \) is an admissible heuristic then the algorithm is optimal.
- Space complexity \( O(bd) \)
- Time complexity is still exponential (suffers from excessive node re-generation)
Recursive BFS - example

1. After expansion of Arad, Sibiu, Rimnicu Vilcea.

2. The path from Rimnicu Vilcea now seems too expensive, go back to the closest neighbour – Fagaras.

3. The path through Fagaras is now worse, go back to Rimnicu Vilcea and expand the best successor.

Simplified memory-bounded A* - example

• Assume memory for three nodes only.
• If there is enough memory to store an optimal path then SMA* find optimal solution.
• Otherwise it finds the best path with available memory.
  – If the cost of J would be 19, then this is optimal goal, but the path to it can be stored in memory.

IDA* and RBFS do not exploit available memory!
• This is a pity as the already expanded nodes are re-expanded again (waste of time)
• Let us try to modify classical A*

Looking for heuristics

How to find admissible heuristics?

Example: 8-puzzle
• 22 steps to goal in average
• branching factor around 3
• (complete) search tree: $3^{22} \approx 3.1 \times 10^{10}$ nodes
• the number of reachable states is only $9!/2 = 181440$
• for 15-puzzle there are $10^{13}$ states
• We need some heuristic, preferable admissible
  – $h_1 =$ „the number of misplaced tiles“ = 8
  – $h_2 =$ „the sum of the distances of the tiles from the goal positions“ = $3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$
    a so called Manhattan heuristic
    – the optimal solution needs 26 steps

function SMA(problem) returns a solution sequence:
inputs: problem, a problem
static: Queue is a queue of nodes ordered by f-cost
Queue = MAKEQUEUE(MAKE-INITIAL-STATE(problem))
loop:
  if Queue is empty then return failure
  s = dequeue from Queue
  if Goal(s) then return sequence
  s = NEXT SUCCESSORS(s)
  f(s) = cost of s + heuristic(s)
  if f(s) < f(parent(s))
    insert s into Queue
  else
    if memory is full then
      drop the worst node (if there are such nodes)
      update f-value of forgotten node to its parent
  Path from root to this non-goal node can be stored in memory, hence no optimal path through this node can be found.

Simplified memory-bounded A* example
• Assume memory for three nodes only.
• If there is enough memory to store an optimal path then SMA* find optimal solution.
• Otherwise it finds the best path with available memory.
  – If the cost of J would be 19, then this is optimal goal, but the path to it can be stored in memory!
Performance of heuristics

How to characterize the quality of a heuristic?

Effective branching factor $b^*$

- Let the algorithm needs $N$ nodes to find a solution in depth $d$
- $b^*$ is a branching factor of a uniform tree of depth $d$ containing $N+1$ nodes

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

Example:

- 15-puzzle
- the average over 100 instances for each of various solution lengths

<table>
<thead>
<tr>
<th>Search Cost</th>
<th>Effective Branching Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.47</td>
</tr>
<tr>
<td>2</td>
<td>2.87</td>
</tr>
<tr>
<td>3</td>
<td>3.06</td>
</tr>
<tr>
<td>4</td>
<td>3.26</td>
</tr>
<tr>
<td>5</td>
<td>3.45</td>
</tr>
<tr>
<td>6</td>
<td>3.64</td>
</tr>
<tr>
<td>7</td>
<td>3.84</td>
</tr>
<tr>
<td>8</td>
<td>4.03</td>
</tr>
<tr>
<td>9</td>
<td>4.23</td>
</tr>
<tr>
<td>10</td>
<td>4.43</td>
</tr>
<tr>
<td>11</td>
<td>4.64</td>
</tr>
<tr>
<td>12</td>
<td>4.84</td>
</tr>
<tr>
<td>13</td>
<td>5.05</td>
</tr>
<tr>
<td>14</td>
<td>5.25</td>
</tr>
</tbody>
</table>

Dominance

- Is $h_2$ (from 8-puzzle) always better than $h_1$ and how to recognize it?
  - notice that $\forall n h_2(n) \geq h_1(n)$
  - We say that $h_2$ dominates $h_1$
  - A* with $h_2$ never expands more nodes than A* with $h_1$
    - A* expands all nodes such that $f(n) < C^*$, i.e. $h(n) < C^* - g(n)$
    - In particular if it expands a node using $h_2$, then the same node must be expanded using $h_1$

- It is always better to use a heuristic function giving higher values provided that
  - the limit $C^* - g(n)$ is not exceeded (then the heuristic would not be admissible)
  - the computation time is no too long

Can an agent construct admissible heuristics for any problem?

Yes via problem relaxation!

- relaxation is a simplification of the problem such that the solution of the original problem is also a solution of the relaxed problem (even if not necessarily optimal)
- we need to be able to solve the relaxed problem fast
- the cost of optimal solution to a relaxed problem is a lower bound for the solution to the original problem and hence it is an admissible (and monotonic) heuristic for the original problem

- Example (8-puzzle)
  - A tile can move from square A to square B if:
    - A is horizontally or vertically adjacent to B
    - B is blank
  - possible relaxations (omitting some constraints to move a tile):
    - a tile can move from square A to square B if A is adjacent to B (Manhattan distance)
    - a tile can move from square A to square B if B is blank
    - a tile can move from square A to square B (heuristic $h_1$)

Relaxation

Another approach to admissible heuristics is using a pattern database

- based on solution of specific sub-problems (patterns)
- by searching back from the goal and recording the cost of each new pattern encountered
- heuristic is defined by taking the worst cost of a pattern that matches the current state
- Beware! The "sum" of costs of matching patterns need not be a admissible (the steps for solving one pattern may be used when solving another pattern).

If there are more heuristics, we can always use the maximum value from them (such a heuristic dominates each of used heuristics).