State space search
A* Algorithm and way to it via Breath-first search and Dijkstra algorithms

Václav Hlaváč
Czech Technical University in Prague (ČVUT)
Czech Institute of Informatics, Robotics, and Cybernetics (CIIRC)
Prague 6, Zikova 4, Czech Republic
hlavac@ciirc.cvut.cz
http://people.ciirc.cvut.cz/hlavac/

Courtesy: Antonín Vobecký and authors of several presentation on the web
Motivation

- Many analytical tasks can be solved by searching through a space of possible states.
- Starting from an initial state, we try reaching a goal state.
- Sequence of actions leading from initial to goal state is the solution to the problem.
- The issues: large number of states and many choices to make in each state.
- Search has to be performed in a systematic manner.
Typical search tasks
State space search, the basic idea

- State space search amounts to a search through a directed graph.
  - graph nodes = states
  - arcs (directed edges) = transitions between states.

- Graph may be defined explicitly or implicitly.

- Graph may contain cycles.

- If we also need the transition costs, we work with a weighted directed graph.

V. Hlaváč, State space search
Size of the search space

- The state space can be HUGE! (Combinatorial explosion)
- Right representation helps.
  - Eight puzzle: 181,440
  - Draughts / Checkers / in Czech dáma: $10^{40}$
  - Chess: $10^{120}$ (in an average length game)
  - Theorem Proving: Infinite!
- Control strategy helps choose which operators to apply:
  - Small # of operators: general, but bushy tree.
  - Large #: perhaps overly specific, but less bushy trees.
Search tree

- By searching through a directed graph, we gradually construct a search tree.

- We do this by expanding one node after the other: we use the successor function to generate the descendants of each node.

- **Open nodes** or “the frontier”: nodes that have been generated, but have not yet been expanded.

- **Closed nodes**: already expanded nodes.

- **Search strategy** is defined by the order in which the nodes are expanded. Different orders yield different strategies.
State space vs. search tree

- Search tree is created while searching through the state space.
- Search tree can be infinite even if the state space is finite. E.g. if the state space contains cycles $\rightarrow$ search tree is infinite.
Open nodes, pictorial illustration
The basic search algorithm

Initialize: put the start node into OPEN

while OPEN is not empty
  take a node N from OPEN
  if N is a goal node, report success
  put the children of N onto OPEN

Report failure

- If OPEN is a stack, this is a depth-first search.
- If OPEN is a queue, this is a breadth-first search.
- If OPEN is a priority queue, sorted according to most promising first, we have a best-first search (Dijkstra algorithm).
Breadth-first search

(abbrev. BFS)

Implementation:

- Pick and remove a location from the **OPEN** (frontier).
- Mark the location as visited so that we know not to process it again.
- Expand it by looking at its neighbors. Any neighbors we haven’t seen yet we add to the frontier.
Breadth-first search (2)
Breadth-first search (3)

- visits all reachable places
- efficiency:
  - time: $O(b^d)$
  - space: $O(b^d)$
  - $b=$ branching factor, $d=$ depth of goal
- no priority
- possible improvements:
  - early exit = search stops when the goal is reached
  - movement cost → Dijkstra algorithm
Dijkstra algorithm

- Adding movement cost to Breath-first search algorithm, expands in all directions
- Using priority queue
  - Choosing move with the lowest cost
- Time efficiency: $O(|E| + |V| \log |V|)$, $V =$ number of nodes, $E =$ number of edges
Dijkstra algorithm vs. BFS
Greedy best first search

- better for finding path to one exact location
- use of heuristics:
  - distance to the goal
  - e.g.:
    ```python
    def heuristics(a, b):
        return abs(a.x - b.x) + abs(a.y + b.y)
    ```
- time/space efficiency: $O(b^m)$
  - good heuristics can give huge improvements
- priority queue
  - priority = distance to goal
Greedy best-first search - examples

Breadth First Search

Greedy Best-First Search
Greedy best-first search - examples

- Problem with obstacles.
- May not find the shortest path.
**A* algorithm (read “A star”)**

- Using the best of both Dijkstra and Greedy algorithms, worst time/space: $O(b^d)$
- Expanding based on:
  - distance from start
  - distance to goal (=heuristics)
A* algorithm

V. Hlaváč, State space search
Map of Manhattan

- How would you find a path from S to G?
Best-First Search

- The *Manhattan distance* \((\Delta x + \Delta y)\) is an estimate of the distance to the goal
  - It is a heuristic function

- Best-First Search
  - Order nodes in priority queue to minimize estimated distance to the goal \(h(n)\)

- Compare: Dijkstra
  - Order nodes in priority queue to minimize distance from the start
Best First in action

- How would you find a path from S to G?
Problem 1: Led astray

- Eventually will expand vertex to get back on the right track
Problem 2: Optimality

- With Best-first search, are you *guaranteed* a shortest path is found when
  - goal is first seen?
  - when goal is removed from priority queue (as with Dijkstra?)
Sub-optimal solution

- No! Goal is by definition at distance 0: will be removed from priority queue immediately, even if a shorter path exists!
Synergy?

- Dijkstra / Breadth First guaranteed to find *optimal* solution
- Best First often visits *far fewer* vertices, but may not provide optimal solution

*Can we get the best of both?*
A*, heuristics

Order vertices in priority queue to minimize
(distance from start) + (estimated distance to goal)

\[ f(n) = g(n) + h(n) \]

- \( f(n) \) = priority of a node
- \( g(n) \) = true distance from start
- \( h(n) \) = heuristic distance to goal
Optimality

- Suppose the estimated distance \((h)\) is *always* less than or equal to the *true* distance to the goal
  - heuristic is a *lower bound on true distance*
  - heuristic is *admissible*

- Then: *when the goal is removed* from the priority queue, we are *guaranteed* to have found a shortest path!
A* in action

<table>
<thead>
<tr>
<th>vertex</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>52\textsuperscript{nd} &amp; 9\textsuperscript{th}</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>
A* in action

The diagram shows a grid with streets and avenues as follows:

- **52nd St**
- **51st St**
- **50th St**

The grid has avenues labeled as 9th Ave, 8th Ave, 7th Ave, 6th Ave, 5th Ave, and 4th Ave.

A vertex **S** is marked on 52nd St, and a vertex **G** is marked on 51st St. The text indicates that there are 5 blocks between these two vertices.

There is a table showing the values for vertices **52nd & 4th**, **51st & 9th**:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>52nd &amp; 4th</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>51st &amp; 9th</td>
<td>1</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

This suggests that the cost function f(n) for A* algorithm is being demonstrated.
A* in action

<table>
<thead>
<tr>
<th>vertex</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>52\text{nd} &amp; 4\text{th}</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>51\text{st} &amp; 8\text{th}</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>50\text{th} &amp; 9\text{th}</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>
A* in action

<table>
<thead>
<tr>
<th>vertex</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>52$<em>{nd}$ &amp; 4$</em>{th}$</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>51$<em>{st}$ &amp; 7$</em>{th}$</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>50$<em>{th}$ &amp; 9$</em>{th}$</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>50$<em>{th}$ &amp; 8$</em>{th}$</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
</tbody>
</table>

(5 blocks)
A* in action

<table>
<thead>
<tr>
<th>vertex</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>52\textsuperscript{nd} &amp; 4\textsuperscript{th}</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>51\textsuperscript{st} &amp; 6\textsuperscript{th}</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>50\textsuperscript{th} &amp; 9\textsuperscript{th}</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>50\textsuperscript{th} &amp; 8\textsuperscript{th}</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>50\textsuperscript{th} &amp; 7\textsuperscript{th}</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
A* in action

<table>
<thead>
<tr>
<th>vertex</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>52nd &amp; 4th</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>51st &amp; 5th</td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>50th &amp; 9th</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>50th &amp; 8th</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>50th &amp; 7th</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>
A* in action

52nd St

51st St

50th St

9th Ave 8th Ave 7th Ave 6th Ave 5th Ave 4th Ave

(5 blocks)

S

G

<table>
<thead>
<tr>
<th>vertex</th>
<th>g(n)</th>
<th>h(n)</th>
<th>f(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>52nd &amp; 4th</td>
<td>5</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>50th &amp; 9th</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>50th &amp; 8th</td>
<td>3</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>50th &amp; 7th</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

DONE!
What would Dijkstra have done?
## Importance of Heuristics

- **h₁ = number of tiles in the wrong place**
- **h₂ = sum of distances of tiles from correct location**

<table>
<thead>
<tr>
<th>D</th>
<th>IDS</th>
<th>A*(h₁)</th>
<th>A*(h₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>39</td>
<td>25</td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
<td>39</td>
</tr>
<tr>
<td>12</td>
<td>364404</td>
<td>227</td>
<td>73</td>
</tr>
<tr>
<td>14</td>
<td>3473941</td>
<td>539</td>
<td>113</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>3056</td>
<td>363</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>39135</td>
<td>1641</td>
</tr>
</tbody>
</table>
Summary

Finding path to ALL locations:

- Same cost → Breadth-first search
- Costs vary → Dijkstra algorithm

Finding path to ONE location:

- Preferably use A* algorithm