Search

Search strategies determined by choice of node (in queue) to expand

Uninformed search:
– Distance to goal not taken into account

Informed search:
– Information about cost to goal taken into account

Aside: “Cleverness” about what option to explore next, almost seems a hallmark of intelligence. E.g., a sense of what might be a good move in chess or what step to try next in a mathematical proof. We don’t do blind search...

Informed Search Methods

- How can we make use of other knowledge about the problem to improve searching strategy?
- Map example:
  - Heuristic: Expand those nodes closest in “straight-line” distance to goal
- 8-puzzle:
  - Heuristic: Expand those nodes with the most tiles in place

How to take information into account? Best-first search.

Idea: use an evaluation function for each node
– Estimate of “desirability” of node
– Expand most desirable unexpanded node first (“best-first search”)
– Heuristic Functions:
  - $f$: States $\rightarrow$ Numbers
  - $f(n)$ expresses the quality of the state $n$
  - Allows us to express problem-specific knowledge,
  - Can be imported in a generic way in the algorithms.
– Implementation: The same as the uniform-cost search, but use $f(n)$ instead of path cost $g(n)$.
– Queuing based on $f(n)$:
  - Order the nodes in fringe in decreasing order of desirability

Special cases:
- Greedy best-first search
- A* search

Greedy Best-First Search

- Create evaluation function which returns estimated “value” of expanding node
- Estimate cost of cheapest path from node $n$ to goal
- $h(n) =$ “as the straight-line distance”

Romania with step costs in km

Greedy Search

<table>
<thead>
<tr>
<th>City</th>
<th>$h(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Fagaras</td>
<td>178</td>
</tr>
<tr>
<td>Riminiciu</td>
<td>193</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
</tbody>
</table>
Greedy Best-First Search

- Expand the node with smallest \( h \)
- Similar to depth-first search
  - Follows single path all the way to goal, backs up when dead end
- Worst case time:
  - \( O(b^m) \), \( m \) = depth of search space
- Worst case memory:
  - \( O(b^m) \), needs to store all nodes in memory to see which one to expand next

A* Search

- Greedy best-first search minimizes
  - \( h(n) \) = estimated cost to goal
- Uniform cost search minimizes
  - \( g(n) \) = cost to node \( n \)
  - Example of each on map
- A* search minimizes
  - \( f(n) = g(n) + h(n) \)
  - \( f(n) \) = best estimate of cost for complete solution through \( n \)

A* search example

Using: \( f(n) = g(n) + h(n) \)
A* search example

Bucharest appears on the fringe but not selected for expansion since its cost (450) is higher than that of Pitesti (417).

Important to understand for the proof of optimality of A*

A* search example

What happens if h(Pitesti) = 150?

Claim: Optimal path found!

1) Can it go wrong?
2) What’s special about “straight distance” to goal?
3) What if all our estimates to goal are 0? Eg h(n) = 0
4) What if we overestimate?
5) What if h(n) is true distance (h*(n))?

Arad --- Sibiu --- Rimnicu --- Pitesti --- Bucharest

Shortest dist. through n --- perfect heuristics --- no search

Note: Best first

Arad --- Sibiu --- Fagaras --- Bucharest

Uniform cost search

A* properties

Under some reasonable conditions for the heuristics, we have:

Complete
  Yes, unless there are infinitely many nodes with f(n) < f(Goal)

Time
  Sub-exponential grow when \(|h(n) - h^*(n)| \leq O(\log h^*(n))\)
  So, a good heuristics can bring exponential search down significantly!

Space
  Fringe nodes in memory. Often exponential. Solution: IDA*

Optimal
  Yes (under admissible heuristics; discussed next)
  Also, optimal use of heuristics information!

Widely used. After almost 40 yrs, still new applications found.
Also, optimal use of heuristic information.
Provably: Can’t do better!

Heuristics: (1) Admissibility

A heuristic \(h(n)\) is admissible if for every node \(n\),

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

where \(h^*(n)\) is the true cost to reach the goal state from \(n\).

An admissible heuristic never overestimates the cost to reach the goal,
I.e., it is optimistic. (But no info of where the goal is if set to 0.)

Example: \(h_{SLD}(n)\) (never overestimates the actual road distance)

Note: it follows that \(h(\text{goal}) = 0\).

Evaluation function \(f(n) = g(n) + h(n)\)

Note: less optimistic heuristic push nodes to be expanded later. Can prune a lot more.
Heuristics: (2) Consistency

A heuristic is consistent (or monotone) if for every node $n$, every successor $n'$ of $n$ generated by any action $a$,

$$h(n) \leq c(n,a,n') + h(n')$$

(form of the triangle inequality)

If $h$ is consistent, we have

$$f(n') = g(n') + h(n')$$

$$\geq g(n) + c(n,a,n') + h(n')$$

$$= g(n) + h(n)$$

i.e., $f(n)$ is non-decreasing along any path.

Note: Consistency is a stronger condition than admissibility. Any consistent heuristic is also admissible.

Intuition: Contours of $A^*$

$A^*$ expands nodes in order of increasing $f$-value. Gradually adds "$f$-contours" of nodes.

$A^*$ expands all nodes with $f(n) < C^*$ where $f(n) = g(n) + h(n)$ expands in circles.

Note: with uniform cost ($h(n)=0$) the bands will be circular around the start state.

Proof:

1. If $h(n)$ is consistent, then the values of $f(n)$ along any path are non-decreasing. See consistent heuristics slide.

2. Whenever $A^*$ selects a node $n$ for expansion, the optimal path to that node has been found. Why?

   Assume not. Then, the optimal path, $P$, must have some not yet expanded node $n'$ on the current frontier (because of graph separation; fig. 3.9; frontier separates explored region from unexplored region). But, because $f$ is nondecreasing along any path, $n'$ would have a lower $f$-cost than $n$ and would have been selected first for expansion before $n$. Contradiction.

From (1) and (2), it follows that the sequence of nodes expanded by $A^*$ using Graph-Search is in non-decreasing order of $f(n)$. Thus, the first goal node selected must have the optimal path, because $f(n)$ is the true path cost for goal nodes ($h(\text{Goal}) = 0$), and all later goal nodes have paths that are at least as expensive. QED

A* Search: Optimality

Theorem:

$A^*$ used with a consistent heuristic ensures optimality with graph search.

Proof:

1. If $h(n)$ is consistent, then the values of $f(n)$ along any path are non-decreasing. See consistent heuristics slide.

2. Whenever $A^*$ selects a node $n$ for expansion, the optimal path to that node has been found. Why?

   Assume not. Then, the optimal path, $P$, must have some not yet expanded node $n'$ on the current frontier (because of graph separation; fig. 3.9; frontier separates explored region from unexplored region). But, because $f$ is nondecreasing along any path, $n'$ would have a lower $f$-cost than $n$ and would have been selected first for expansion before $n$. Contradiction.

From (1) and (2), it follows that the sequence of nodes expanded by $A^*$ using Graph-Search is in non-decreasing order of $f(n)$. Thus, the first goal node selected must have the optimal path, because $f(n)$ is the true path cost for goal nodes ($h(\text{Goal}) = 0$), and all later goal nodes have paths that are at least as expensive. QED

Note: Termination / Completeness

Termination is guaranteed when the number of nodes with $f(n) \leq f^*$ is finite.

None-termination can only happen when

- There is a node with an infinite branching factor, or

- There is a path with a finite cost but an infinite number of nodes along it.
  - Can be avoided by assuming that the cost of each action is larger than a positive constant $d$

A* Optimal in Another Way

It has also been shown that $A^*$ makes optimal use of the heuristics in the sense that there is no search algorithm that could expand fewer nodes using the heuristic information (and still find the optimal / least cost solution).

So, $A^*$ is “the best we can get” (in this setting).

Note: We’re assuming a search based approach with states/nodes, actions on them leading to other states/nodes, start and goal states/nodes.
8-Puzzle

Slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration.

What's the branching factor? (slide “empty space”)

- About 3, depending on location of empty tile:
  - middle → 4; corner → 2; edge → 3

The average solution cost for a randomly generated 8-puzzle instance is about 22 steps. So, search space to depth 22 is about \(3^{22} \approx 10^{10}\) states.

Reduced to by a factor of about 170,000 by keeping track of repeated states (9!/2 = 181,440 distinct states). Note: 2 sets of disjoint states. See exercise 3.4

But: 15-puzzle is about \(10^{13}\) distinct states!

We'd better find a good heuristic to speed up search! Can you suggest one?

Note: “Clever” heuristics now allow us to solve the 15-puzzle in a few milliseconds!

Admissible heuristics

E.g., for the 8-puzzle:

- \(h_1(n)\) = number of misplaced tiles
- \(h_2(n)\) = total Manhattan distance (i.e., sum of steps from desired location of each tile)

\[ h_1(\text{Start}) = 8 \]
\[ h_2(\text{Start}) = 3 + 1 + 2 + 2 + 2 + 3 + 2 = 18 \]
\[ \text{True cost} = 26 \]

Why are heuristics admissible?

Which is better?

How can we get the optimal heuristics? (Given \(H_{\text{opt}}(\text{Start}) = 26\). How would we find the exact board on the optimal path to the goal?)

Desired properties heuristics:

1. Consistent (admissible)
2. As close to optimal as we can get (sometimes go a bit over…)
3. Easy to compute! We want to explore many nodes.

Effective Branching Factor, \(b^*\)

- If \(A^*\) generates \(N\) nodes to find the goal at depth \(d\)
- \(b^* = \text{branching factor such that a uniform tree of depth } d \text{ contains } N \text{ nodes (we add one for the root node that wasn’t included in } N)\)
- \(N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d\)

E.g., if \(A^*\) finds solution at depth 5 using 52 nodes, then the effective branching factor is 1.92.

- \(b^* \) close to 1 is ideal:
  - because this means the heuristic guided the \(A^*\) search is closer to ideal (linear).
  - If \(b^*\) were 100, on average, the heuristic had to consider 100 children for each node.
  - Compare heuristics based on their \(b^*\)

**Dominating heuristics**

- \(h_2\) is always better than \(h_1\)
  - Because for any node, \(n\), \(h_2(n) \geq h_1(n)\). (Why?)

  We say \(h_2\) dominates \(h_1\)

  It follows that \(h_1\) will expand at least as many nodes as \(h_2\).

  Because:

  Recall all nodes with \(f(n) < C^*\) will be expanded.

  This means all nodes, \(h(n) + g(n) < C^*\), will be expanded.

  So, all nodes \(n\) where \(h(n) < C^* - g(n)\) will be expanded.

  All nodes \(h_2\) expands will also be expanded by \(h_1\) and because \(h_1\) is smaller, others may be expanded as well.

Inventing admissible heuristics: Relaxed Problems

Can we generate \(h(n)\) automatically?

- Simplify problem by reducing restrictions on actions.

  A problem with fewer restrictions on the actions is called a relaxed problem.

Comparison of heuristics

<table>
<thead>
<tr>
<th>ID</th>
<th>h0(n)</th>
<th>h1(n)</th>
<th>h2(n)</th>
<th>Effective Branching Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>1.00</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>16</td>
<td>16</td>
<td>1.00</td>
</tr>
<tr>
<td>4</td>
<td>152</td>
<td>12</td>
<td>12</td>
<td>1.14</td>
</tr>
<tr>
<td>5</td>
<td>680</td>
<td>16</td>
<td>16</td>
<td>1.00</td>
</tr>
<tr>
<td>6</td>
<td>1344</td>
<td>29</td>
<td>29</td>
<td>1.17</td>
</tr>
<tr>
<td>7</td>
<td>2173</td>
<td>36</td>
<td>36</td>
<td>1.14</td>
</tr>
<tr>
<td>8</td>
<td>3140</td>
<td>36</td>
<td>36</td>
<td>1.11</td>
</tr>
<tr>
<td>9</td>
<td>4656</td>
<td>36</td>
<td>36</td>
<td>1.06</td>
</tr>
<tr>
<td>10</td>
<td>6493</td>
<td>36</td>
<td>36</td>
<td>1.09</td>
</tr>
<tr>
<td>11</td>
<td>8290</td>
<td>36</td>
<td>36</td>
<td>1.04</td>
</tr>
<tr>
<td>12</td>
<td>10200</td>
<td>102</td>
<td>102</td>
<td>1.00</td>
</tr>
<tr>
<td>13</td>
<td>12800</td>
<td>102</td>
<td>102</td>
<td>1.00</td>
</tr>
</tbody>
</table>

\(d\) - depth of goal node

\(h_2\) indeed significantly better than \(h_1\)
Examples of relaxed problems

Original: A tile can move from square A to square B iff
(1) A is horizontally or vertically adjacent to B and (2) B is blank.

Relaxed versions:
– A tile can move from A to B if A is adjacent to B (“overlap”; Manhattan distance)
– A tile can move from A to B if B is blank (“teleport”)
– A tile can move from A to B (“teleport and overlap”)

Key: Solutions to these relaxed problems can be computed without search
and therefore provide a heuristic that is easy/fast to compute.

This technique was used by ABSOLVER (1993) to invent heuristics
for the 8-puzzle better than existing ones and it also found a useful
heuristic for famous Rubik’s cube puzzle.

Inventing admissible heuristics: Relaxed Problems

The cost of an optimal solution to a relaxed problem is an admissible heuristic
for the original problem. Why?

1) The optimal solution in the original problem is also a solution to
the relaxed problem (satisfying in addition all the relaxed
constraints). So, the solution cost matches at most the original
optimal solution.

2) The relaxed problem has fewer constraints. So, there may be
other, less expensive solutions, given a lower cost (admissible)
relaxed solution.

What if we have multiple heuristics available? i.e., \( h_1(n) \), \( h_2(n) \), ...

\[ h(n) = \max \{ h_1(n), h_2(n), ..., h_m(n) \} \]

If component heuristics are admissible so is the
composite.

Inventing admissible heuristics: Sub-problem solutions as heuristic

What is the optimal cost of solving some portion of original problem?
– subproblem solution is heuristic of original problem

Start State: 

```
1 2 3
4 5 6
```

Goal State: 

```
1 2 3
4 5 6
```

Inventing admissible heuristics: Learning

Also automatically learning admissible heuristics
using machine learning techniques, e.g.,
inductive learning and reinforcement learning.

Generally, you try to learn a “state-evaluation”
function or “board evaluation” function. (How
desirable is state in terms of getting to the
goal?) Key: What “features / properties” of state
are most useful?

More later…

Pattern Databases

Store optimal solutions to subproblems in database
– We use an exhaustive search to solve every permutation
of the 1,2,3,4-piece subproblem of the 8-puzzle
– During solution of 8-puzzle, look up optimal cost to solve
the 1,2,3,4-piece sub-problem and use as heuristic
– Other configurations can be considered

Memory problems with A*

\( A^* \) is similar to breadth-first:

Expand by depth-layers

Expands by f-contours
Memory Bounded Search

- Can A* be improved to use less memory?
- Iterative deepening A* search (IDA*)
  - Each iteration is a depth-first search, just like regular iterative deepening
  - Each iteration is not an A* iteration: otherwise, still O(b^m) memory
  - Use limit on cost (f), instead of depth limit as in regular iterative deepening

Iterative deepening A* (IDA*)

- Perform depth-first search LIMITED to some f-bound.
- If goal found: ok.
- Else: increase f-bound and restart.
- Note: DFS does not need to “sort” the frontier nodes. All at f-bound value.

How to establish the f-bounds?
- Initially: f(S) (S – start node)
- generate all successors
- record the minimal f(succ) > f(S)
- Continue with minimal f(succ) instead of f(S)

Example

[check at home]

Properties: practical

- If there are only a reduced number of different contours:
  - IDA* is one of the very best optimal state-space search techniques!
  - Example: the 15-puzzle
  - Also for many other practical problems
- Else, the gain of the extended f-contour is not sufficient to compensate recalculating the previous parts. Do:
  - increase f-bound by a fixed number g at each iteration:
    - effects: less re-computations, BUT: optimality is lost: obtained solution can deviate up to g
    - Can be remedied by completing the search at this layer Issue: finding goal state potentially “too early” in final layer, without having the optimal path. (Consider “Bucharest” twice on frontier; but we don’t keep full frontier in memory, with DFS.)
IDA* Analysis

- **Time complexity**
  - If cost value for each node is distinct, only adds one state per iteration
  - BAD!
  - Can improve by increasing cost limit by a fixed amount each time
  - If only a few choices (like 8-puzzle) for cost, works really well

- **Memory complexity**
  - Approximately $O(bd)$ (like depth-first)
  - Completeness and optimality same as A*

Simplified Memory-Bounded A* (SMA*)

- Uses all available memory
- Basic idea:
  - Do A* until you run out of memory
  - Throw away node with highest f cost
  - Store f-cost in ancestor node
  - Expand node again if all other nodes in memory are worse

---

**SMA* Example:** G is the only goal

```
A
  B
  F
  G
  C
  D
  E
```

**SMA* Example:** Memory of size 3

```
A
  B
  C
  D
  E
```

A is the only node; expand it.

For the completeness, the children of a node are sorted and high-value children may be discarded but that value is kept by its parent, as forgotten value.

---

**SMA* Example:** Memory of size 3

```
A
  B
  C
```

A has two children; next expand C

---

**SMA* Example:** Memory of size 3

```
A
  B
  C
```

C has two children: D ($f=18$) and E ($f=24$). D is better than B
SMA* Example: Memory of size 3

A  f = 12, expanded

B  f = 19

C  f = 13

D  f = 18, expanded

D has no children; next expand B

B has two children F(f=25) and G(f=27)

F  f = 25

E

SMA* Example: Memory of size 3

A  f = 12, expanded

B  f = 19

forgotten f = 27

C  f = 13

forgotten f = 24

D expanded

B f = 19

forgotten f = 27

F f = 25

E expanded

SMA* Example: Memory of size 3

A  f = 12, expanded

B  f = 19

forgotten f = 27

C  f = 13

forgotten f = 24

D expanded

B f = 19

forgotten f = 27

F f = 25

E expanded

next expand E, which has no children

SMA* Example: Memory of size 3

A  f = 12, expanded

B  f = 19

forgotten f = 27

C  f = 13

forgotten f = 24

D expanded

B f = 19

forgotten f = 27

F f = 25

E expanded

next expand F, which has no children

SMA* Example: Memory of size 3

A  f = 12, expanded

B  f = 19

forgotten f = 27

C  f = 13

forgotten f = 24

D expanded

B f = 19

forgotten f = 27

F f = 25

E expanded

next expand B again
SMA* Example: Memory of size 3

A  \( f = 12 \), expanded

B  expanded

C  \( f = 13 \), expanded

D  expanded

E  expanded

F  expanded

next expand G, which is the goal.

SMA* Properties

- Complete if can store all nodes whose \( f \)-value are smaller than that of goal in memory
- Finds best solution (and recognizes it), under the same condition.

Summary

Uninformed search:
(1) Breadth-first search (2) Uniform-cost search
(3) Depth-first search (4) Depth-limited search
(5) Iterative deepening search (6) Bidirectional search

Informed search:
(1) Greedy Best-First
(2) A∗
(3) IDA∗
(4) SMA∗