Search
Professor Marie Roch
Chapter 3, Russell & Norvig

Solving problems through search

• State – atomic representation of world
• Goal formulation
  • What objective(s) are we trying to meet?
  • Can be represented as a set of states that meet objectives: goal states
• Problem formulation
  • Decide actions and states to reach a goal
Search

- Assume environment is
  - observable
  - discrete (finite # of actions)
  - deterministic actions

- Search process returns a plan:
  set of states & actions to reach a goal state
- Plan can be executed

Search problem components

- Initial state

map of Romania (Google maps)
Search problem components

• Initial state
• Actions
  • function that returns set of possible decisions from a given state
  • actions(in(arad)) \rightarrow \{go(sibiu), go(Timisoara), go(zerind)\}

Note: Abstractions are valid when we can map them onto a more detailed world

Search problem components

• Initial state
• Cost
  • Each action has a step cost:
    cost(in(arad), go(zerind), in(zerind)) = 75
  • A path has a cost which is the sum of its step costs:
    path: in(arad), in(zerind), in(Oradea) has cost: 146 (75+71)
Search problem components

• Initial state
• Actions
• Cost
• Transition model
  Function that reports the result of an action applied to a state:
  \[ \text{result}(\text{in}(\text{arad}), \text{go}(\text{zerind})) \rightarrow \text{in}(\text{zerind}) \]

• Goal predicate
  Is the new state a member of the goal set?
  \[ \text{goal: } \{\text{in}(\text{bucharest})\} \]

Any path that reaches a goal is a \textit{solution}, the lowest cost path is an \textit{optimal solution}. 

Bucharest or bust!
Sample toy problem

- n-puzzle

- n-queens

Constructing a problem: n-queens

- States
  1. complete-state:
     - n-queens on board
     - move until no queen can capture another.
  2. Incrementally place queens
     - initial empty board
     - add one queen at a time
Incremental n-queens

- state: Any arrangement of [0,n] queens
- initial state: empty board
- actions: add queen to empty square
- transition model: new state with additional queen
- goal test: n queens on board, none can attack one another

Incremental n-queens

- A well-designed problem restricts the state space
  - Naïve 8 queens
    1\textsuperscript{st} queen has 64 possibilities
    2\textsuperscript{nd} queen has 63 possibilities...
    \[ 64 \times 63 \times 62 \ldots \approx 1.8 \times 10^{14} \]
  - Smarter:
    - Actions only returns positions that would not result in capture
    - State space reduced to 2057 states.
Classic real-world problems

- route-finding problem
  - transportation (car, air, train, boat, ...)
  - networks
  - operations planning
- touring problem
  - Visit a set of states ≥1 time
- traveling salesperson
  - Visit a set of state 1 time

- Others: VLSI layout, autonomous vehicle navigation & planning, assembly sequencing, pharmaceutical discovery

Search trees

(a) The initial state

(b) After expanding Arad

(frontier set also known as an open list)
Search trees

- Frontier set consists of leaf nodes
- Redundant paths occur when
  - $\exists$ more than 1 path between a pair of states
  - cycles in the search tree (loops) are a special case
Redundant paths

"Those who cannot remember the past are condemned to repeat it"

George Santayana,
Spanish-American philosopher 1863-1952

• Sometimes, we can define are our problem to avoid cycles
e.g. n-queens: queen must be placed in the leftmost empty column
• Otherwise: Explored set
  • Track states that have been investigated
  • Don’t add any actions that have already occurred

Tree Search

function tree-search(problem)
  frontier = problem.initial_state()
  done = found = False
  while not done
    node = frontier.get_node() # remove state
    if node in problem.goals()
      found = done = True
    else
      frontier.add_nodes(results from actions(node))
      done = frontier.is_empty()
  return solution if found else return failure
Graph Search

function graph-search(problem)
  frontier = problem.initial_state()
  done = found = False
  explored = () # keep track of nodes we have checked
  while not done
    node = frontier.get_node() # remove state
    explored = union(explored, node)
    if node in problem.goals():
      found = done = True
    else
      # only add novel results from the current node
      nodes = setdiff(results from actions(node), union(frontier, explored))
      frontier.add_nodes(nodes)
    done = frontier.is_empty()
  return solution if found else return failure

Search architecture

• Node representation
  • state
  • parent – ancestor in tree
    allows us to find the solution from a goal node by
    chasing pointers and reversing the path
  • action – Which action was used on parent to generate
    this node
  • path-cost – What is the cost to reach this node from the
    tree’s root. Usually denoted g(n).
Search architecture

function child-node(problem, node, action)
    child.state = problem.result(node.state, action)
    child.parent = node
    child.path_cost = node.path_cost +
        problem.cost(node.state, action, child.state)
return child

Search architecture

• frontier set is usually implemented as a queue
  • FIFO – traditional queue
  • LIFO – stack
  • priority
    We will develop a way such that it can always be a priority queue.

• Explored set – Need to make states easily comparable
  • hash the state or
  • store in canonical form (e.g. sort visited cities for traveling salesperson problem)
**Search architecture**

- \( g(n) \) – cost from initial state to \( n \)
- \( h(n) \) – cost from \( n \) to least expensive goal

\( g(n) \) and \( h(n) \) are frequently not known precisely.
Estimates are denoted or \( g'(n) \) & \( h'(n) \) or \( \tilde{g}(n) \) & \( \tilde{h}(n) \)

---

**A generic graph search algorithm**

```python
function tree-search(problem):
    frontier = problem.initial_state()  # priority queue based on lowest cost
    done = found = False
    explored = {}  # keep track of nodes we have checked
    while not done:
        node = frontier.get_node()  # remove state
        explored = union(explored, node)
        if node in problem.goals():
            found = done = True
        else:
            # only add novel results from the current node
            nodes = setdiff(results from actions(node), union(frontier, explored))
            for n in nodes:
                estimate a cost \( g'(n) + h'(n) \)
                frontier.add_nodes(n)  # merge new nodes in by estimated cost
            done = frontier.is_empty()
    return solution if found else return failure
```
Uninformed (blind) search

• No awareness of whether or not a state is promising
• Strategies depend on order of node expansion
  • breadth-first
  • uniform-cost
  • depth-first
  • variants: depth-limited, iterative deepening, bidirectional

• Note: Text uses different queue types for frontier, with generic search algorithm everything is a priority queue, smallest values first.

Breadth-first search

• $\forall n. g'(n) = \text{depth}(n)$ and $h'(n) = k$ (e.g. $k=0$)

Abstract view of Romanian roads (Russell and Norvig 2010, Fig 3.2)
Breadth-first search

• Guarantees
  • completeness – will find a solution
  • best path if cost is a nondecreasing \( f(\text{depth}) \)

• How can we measure performance?
  • Time complexity
  • Space complexity

Complexity (CS 310 material)

• Measure of the number of operations (time) or memory (space) required
• Analysis of performance as the number of items \( n \) grows:
  • worst case
  • average case

• Example:
  ```python
def foobar(n):
    x = 0
    for i in xrange(n):
      for j in xrange(n):
        x = x + i*i + j*j
    return x * x
  ```

There are \( T(n) = 4n^2 + 1 \) arithmetic operations
Complexity

• We define “big oh” of n as follows:
  \[ T(n) \text{ is } O\left(f(n)\right) \text{ if } T(n) \leq kf(n) \]
  for some \( k \) & \( \forall n > n_0 \)

• Role of \( k \) and \( n_0 \)
  Coefficients of highest order polynomial aren’t relevant.

• Implications:
  • \( T(n) = 4n^2 + 1 \rightarrow O(n^2) \)
  • \( T'(n) = 500n + 8 \rightarrow O(n) \)
  For some small values of n, \( T(n) \) is better, but as \( n \) increases \( T(n) \) will be worse. Using the big-oh notation abstracts this away and we know in general that the second algorithm is better.

Search complexity

Measured with respect to search tree:

• Complexity is a function of
  • Branch factor – max # of successors
  • Depth of the shallowest goal node
  • Maximum length of a state-space path

• Time measurement: # nodes expanded
• Space measurement: maximum # nodes in memory
Search complexity

• “Search cost” – time complexity
• “Total cost” – time and space complexity
  Problematic to fuse the metrics...

Breadth-first search performance

• Assume branch factor $b$
• Time complexity:
  $$b + b^2 + b^3 + \cdots + b^d = \mathcal{O}(b^d)$$
• Space complexity
  • Every generated node remains in memory, $\mathcal{O}(b^{d-1})$ in explored and $\mathcal{O}(b^d)$ in frontier.

* See text for discussion of $\mathcal{O}(b^d)$ vs. $\mathcal{O}(b^{d+1})$
Uniform-cost search

- Similar to breadth-first, $g'(n)$ uses edge costs
- $\forall n g'(n) = g(n)$ and $h'(n) = k$
- Nodes are expanded in order of optimal cost $\rightarrow$ optimal solution
- Complexity function of minimum cost for all actions

Depth-first search

- Deepest node is expanded first
- $\forall n g'(n) = k$ and $h'(n) = -\text{depth}(k)$
- Non-optimal
- Incomplete search
- Why bother?
Depth-first search (DFS)

• DFS will explore other paths when there are no successors.
• Fast! If you hit the right path... but the average case analysis is $O(b^m)$ where $m$ is maximum depth.
• Space complexity is better: $O(bm)$

Iterative deepening

• Prevents infinite loops of depth-first search
• Basic idea
  • Depth-first search with a maximum depth
  • If the search fails, repeat with a deeper depth
Uninformed search

• Other variants exist

• For large search spaces, it is generally a bad idea

Informed, or heuristic, search

• General idea: Can we guess the cost to a goal based on the current state?
Heuristic

• h(n) – Actual cost from a search graph node to a goal state along the cheapest path.

• h’(n) – An estimate of h(n), known as a heuristic.

Note that your text does not make a notational distinction between the actual cost and the estimated one and always uses h(n), so we will frequently follow suit.

Heuristic

• h(n) is always ≥ 0
• h(n) is problem specific
• Estimators of h(n) are similar.

• One can think of a heuristic as an educated guess. We will look at how to construct these later...
Greedy best-first search

- $g(n) = 0$, $h(n)$ is heuristic value
- Example $h(n)$ for Romania example: as the bird flies distance

A* Search

- “A-star” search uses:
  - $g(n) =$ cost incurred to $n$
  - $h(n) =$ estimate to goal
- A* is the estimated cost form start to goal through state $n$
Heuristic properties

• admissible – \( h'(n) \) is admissible if it *never overestimates* the cost to goal
  One can think of it as optimistic: \( h'(n) \)

• consistency (aka monotonicity) – \( h'(n) \) is consistent if
  \[
  h'(n) \leq \text{cost}(n, \text{action}, n') + h'(n')
  \]

Note: We are being careful about distinguishing the heuristic estimator \( h'(n) \) from the actual distance \( h(n) \)

Heuristic properties

• Every consistent heuristic is also admissible.

• A* is guaranteed to be:
  • for trees
    A* optimal if \( h'(n) \) is admissible
  • for graphs
    A* optimal if \( h'(n) \) is consistent
Understanding A*

Figure 3.2.3, R&N
Understanding A* optimality

Consistency revisited:
the ▲ inequality – the sum of any two sides ≥ third side

\[ h'(n) \leq c(n, action, n') + h'(n') \]

If \( h' \) consistent and costs are nonnegative, values of \( f(n) \) along any path are nondecreasing.

Understanding A* optimality

• Suppose we pick node \( n \)

• Is the path to node \( n \)'s state optimal?

  Proof by contradiction
  Suppose a better path to the same state is in node \( b \).
  As \( b \) and \( n \) have the same state, so \( h(n) = h(b) \).
  Relative position in queue will be driven by \( g(n) \) and \( g(b) \).
  If \( b \) is better, \( g(b) < g(n) \) and we would have picked \( b \) first
Understanding A* optimality

- When \( h(n) \) is consistent, the properties of:
  - nondecreasing values of \( f(n) \)
  - guarantee that we pick the best path to \( n \)

  ensure that the first goal node we find is optimal.

- Completeness holds when there are a finite number of nodes with \( f(n) < \) the optimal cost

Limitations of A*

- Need to find a heuristic
- Show it is consistent (for graph search) if optimal goal is required.
- Show the graph is finite for nodes with cost lower than the optimal one if completeness is required

- Note: expanded set requires nodes in memory and is a frequent limitation of A*
A* variants

• iterative deepening A*
  Same idea as iterative depth-first search, but we limit on f(n)

• SMA* - simplified memory A*
  • When memory is full
    • drops worst frontier node (highest f(n))
    • stores that value in parent, and will only reconsider branch when everything looks worse than the stored value
  • Details beyond our scope

Heuristic search summary

• A* can still have problems with space complexity
  • iterative deepening A*
  • other alternatives listed in text

• Complexity of A* search is tricky, but is related to
  • the error in the heuristic, h(n)-h'(n)
  • and solution depth
Developing heuristics

- Requires
  - knowledge of problem domain
  - thinking a bit (usually)

- Effort to show that heuristic is
  - admissible
  - consistent

- What heuristics could we use for the N-puzzle

N-puzzle heuristics

- Common heuristics
  - $h_1(n)$ – Number of misplaced tiles
  - $h_2(n)$ – Sum of Manhattan\(^1\) distance of tiles to solution

- Are these
  - admissible? (never overestimates)
  - consistent? (non-decreasing path cost)

\(^1\) Also known as city-block distance, the sum of vertical and horizontal displacement.
Heuristics and performance

- Branching factor
  - Measured against a complete tree of solution depth \( d \)
  - Suppose A* finds a solution at
    - depth 5
    - 52 nodes expanded (53 with root)
  - A complete tree of depth 5 would have
    \[
    52 + 1 = b^* + (b^*)^2 + (b^*)^3 + (b^*)^4 + (b^*)^5
    \]
    where \( b^* \) is the branch factor
  - Using a root finder for
    \[
    53(b^*)^5 - 51(b^*)^4 + 17(b^*)^3 - 17(b^*)^2 + 11(b^*)^1 - 11 = 0
    \]
    we see \( b^* \approx 1.92 \)

- 8-puzzle example averaged over 100 instances

<table>
<thead>
<tr>
<th>( d )</th>
<th>IDS</th>
<th>( N(h_1) )</th>
<th>( N(h_2) )</th>
<th>IDS</th>
<th>( N(h_1) )</th>
<th>( N(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>1.48</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>123</td>
<td>25</td>
<td>24</td>
<td>2.80</td>
<td>1.33</td>
<td>1.29</td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
<td>91</td>
<td>2.79</td>
<td>1.38</td>
<td>1.22</td>
</tr>
<tr>
<td>12</td>
<td>3644035</td>
<td>227</td>
<td>73</td>
<td>2.78</td>
<td>1.42</td>
<td>1.28</td>
</tr>
<tr>
<td>14</td>
<td>539</td>
<td>113</td>
<td>112</td>
<td>2.78</td>
<td>1.44</td>
<td>1.23</td>
</tr>
<tr>
<td>16</td>
<td>3895</td>
<td>211</td>
<td>209</td>
<td>2.78</td>
<td>1.45</td>
<td>1.26</td>
</tr>
<tr>
<td>18</td>
<td>3056</td>
<td>363</td>
<td>361</td>
<td>2.78</td>
<td>1.46</td>
<td>1.26</td>
</tr>
<tr>
<td>20</td>
<td>7276</td>
<td>676</td>
<td>674</td>
<td>2.78</td>
<td>1.48</td>
<td>1.28</td>
</tr>
<tr>
<td>22</td>
<td>16094</td>
<td>1219</td>
<td>1217</td>
<td>2.78</td>
<td>1.48</td>
<td>1.28</td>
</tr>
<tr>
<td>24</td>
<td>39135</td>
<td>1641</td>
<td>1639</td>
<td>2.78</td>
<td>1.48</td>
<td>1.28</td>
</tr>
</tbody>
</table>
Finding heuristics

• Okay, developing a heuristic is hard

• Can we make it easier?

Relaxed problem heuristics

• Let’s return to the N-puzzle
• Suppose we allowed

  • A tile to move onto the next square regardless of whether or not it was empty.

  • A tile to move anywhere.

• These are relaxations of the rules
Relaxed problems

We can think of these as expanding the state space graph.

Relaxed problem heuristics

- The original state space is a subgraph of the new one.
- Heuristics on relaxed state space
  - Frequently easier to develop
  - If admissible/consistent properties hold in relaxed space, they also hold in the problem state space.
Relaxation

- Can specify problem in a formal language, e.g.
  - move(A,B) if
    - verticalAdjacent(A,B) and
    - horizontalAdjacent(A,B) and
    - isempty(B)

- Possible relaxations
  - move(A,B) if adjacent(A,B)
  - move(A,B) if isempty(B)
  - move(A,B)

Automatically generated heuristics

With a formal specification of the problem there exist algorithms to find heuristics (beyond our scope, e.g. ABSOLVER)
Multiple heuristics

• Regardless of how generated, one may develop multiple heuristics for a problem

• We can merge them

\[ h'(n) = \max \left( h'_1(n), h'_2(n), \ldots, h'_i(n) \right) \]

why maximum?

Pattern database heuristics

• Can we solve a subproblem?

• If we can, we can store its h(n)
Pattern database heuristics

- Cost usually found by searching back from goal nodes.
- Worth it if the search will be executed many times.
- Sometimes patterns are disjoint. If so, the heuristic costs may be added (doesn’t work for 8 puzzle)

Learning heuristics

- Use experience to learn heuristics
- Beyond our reach for now... (machine learning)
Heuristic summary
(rough outline, no substitute for a little thought)

1. Can I relax the problem?
   - Yes: Think and come up with heuristic
   - No: Reasonable?
2. Reasonable?
   - Yes: Learn heuristic vs. machine-learning
   - No: Automated heuristic generation
3. Automated heuristic generation:
   - Yes: Formal specification
   - No: Learn heuristic vs. machine-learning