Overview

- Heuristic Search - exploiting knowledge about the problem
- Heuristic Search Algorithms
  - “Best-first” search
  - Greedy Search
  - A* Search
  - Extensions to A*
  - Constructing Heuristics

Informing Search

- Previous algorithms were able to find solutions, but were very inefficient.
- Exponential number of nodes expanded.
- By taking advantage of knowledge about the problem structure, we can improve performance.
- Two caveats:
  - We have to get knowledge about the problem from somewhere.
  - This knowledge has to be correct.

Best-first Search

- Recall Uniform-cost search
- Nodes were expanded based on their total path cost.
- A priority queue was used to implement this.
- Path cost is an example of an evaluation function.
- We’ll use the notation $f(n)$ to refer to an evaluation function.
- An evaluation function tells us how promising a node is.
- Indicates the quality of the solution that node leads to.

Best-first Pseudocode

```c
queue = []
queue.append(initialState)
while len(queue) > 0:
    node = queue.pop(0)
    if goalTest(node):
        return node
    children = successorFn(node)
    for child in children:
        queue.append(child)
```

Best-first Search

- By ordering and expanding nodes according to their $f$ value, we search the “best” nodes “first”.
- If $f$ was perfect, we would expand a straight path from the initial state to the goal state.
- Arad, Sibiu, Rimnicu Vîlcea, Pitesti, Bucharest.
- Of course, if we had a perfect $f$, we wouldn’t need to solve the problem in the first place.
- Instead, we’ll try to develop heuristic functions $h(n)$ that help us estimate $f(n)$.

Best-first Search

- Best-first Pseudocode
  ```c
  queue = []
  queue.append(initialState)
  while len(queue) > 0:
      node = queue.pop(0)
      if goalTest(node):
          return node
      children = successorFn(node)
      for child in children:
          queue.append(child)
  ```
  - where insert-with orders our priority queue accordingly.
Greedy Search

- Let's start with the opposite of uniform-cost search
- UCS used the solution cost to date as an estimate of \( f \)
- Greedy search uses an estimate of distance to the goal for \( f \).
- Rationale: Always pick the node that looks like it will get you closest to the solution.
- Let's start with a simple estimate of \( f \) for the Romania domain.
  - \( h(city) = \) Straight-line distance between that city and Bucharest.

Issues with Greedy Search

- Sometimes the optimal solution to a problem involves moving 'away' from the goal.
- For example, to solve 8-puzzle, you often need to 'undo' a partial solution.
- Greedy search has many of the same appeals and weaknesses as DFS.
- Expands a linear number of nodes
- Is not complete or optimal
- Its ability to cut toward a goal is appealing - can we salvage this?

A* search

- A* search is a combination of uniform cost search and greedy search.
- A node's \( f(n) = g(n) + h(n) \)
  - \( g(n) = \) current path cost
  - \( h(n) = \) heuristic estimate of distance to goal.
  - Favors nodes that have a cheap solution to date and also look like they’ll get close to the goal.
- If \( h(n) \) satisfies certain conditions, A* is both complete (always finds a solution) and optimal (always finds the best solution).

A* example - Romania

\[
\begin{align*}
A &= 336 \\
\text{(enqueue A)} & S = 359, Z = 374 \\
\text{(enqueue B)} & F = 176, RV = 193, T = 329, A = 336, Z = 374, O = 380 \\
\text{(enqueue F)} & B = 8, RV = 193, S = 253, T = 329, A = 336, Z = 374, O = 380 \\
\text{(enqueue B)} & A = S = F = B
\end{align*}
\]

We found a solution quickly, but it was not optimal.

What was the problem with our approach?
A* example - Romania

( dequeue P: g = 317) T = 118 + 329 = 447, Z = 374 + 75 = 449, B = 518 + 0 = 518, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, C = 345 + 160 = 505, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, O = 291 + 380 = 671

( dequeue T: g = 118) Z = 374 + 75 = 449, L = 229 + 244 = 473, B = 518 + 0 = 518, C = 366 + 160 = 526, B = 550 + 0 = 550, S = 300 + 253 = 553, A = 236 + 336 = 572, S = 300 + 253 = 553, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, O = 291 + 380 = 671

( dequeue Z: g = 75) L = 229 + 244 = 473, A = 150 + 336 = 486, B = 518 + 0 = 518, O = 146 + 380 = 526, C = 366 + 160 = 526, M = 299 + 241 = 540, B = 550 + 0 = 550, S = 300 + 253 = 553, A = 236 + 336 = 572, S = 300 + 253 = 553, RV = 414 + 193 = 607, C = 455 + 160 = 615, A = 280 + 336 = 616, O = 291 + 380 = 671

Optimality of A*

- Notice that we can’t discard repeated states.
- We could always keep the version of the state with the lowest g.
- More simply, we can also ensure that we always traverse the best path to a node first.
  - A monotonic heuristic guarantees this.
- A heuristic is monotonic if, for every node n and each of its successors (n’), h(n) is less than or equal to stepCost(n, n’) + h(n’).
- In geometry, this is called the triangle inequality.

Optimality of A*

- SLD is monotonic. (In general, it’s hard to find realistic heuristics that are admissible but not monotonic).
- Corollary: If h is monotonic, then f is nondecreasing as we expand the search tree.
- Alternative proof of optimality.
- Notice also that UCS is A* with h(n) = 0.
- A* is also optimally efficient.
- No other complete and optimal algorithm is guaranteed to expand fewer nodes.

Another A* example

Node: Queue : 
-- 
[Node f = 17, g = 6, h = 117]
Another A* example

Node: Queue : A
[(C f = 22, g = 7, h = 15), (B f = 28, g = 8, h = 20)]

Another A* example

Node: Queue : C
[(D f = 23, g = 15, h = 8), (B f = 28, g = 8, h = 20)]

Another A* example

Node: Queue : D
[(I f = 26, g = 20, h = 6), (F f = 27, g = 21, h = 6),
(B f = 28, g = 8, h = 20), (E f = 28, g = 20, h = 8)]

Another A* example

Node: Queue : I
[(F f = 27, g = 21, h = 6), (B f = 28, g = 8, h = 20),
(E f = 28, g = 20, h = 8), (G f = 30, g = 26, h = 4)]

Another A* example

Node: Queue : F
[(B f = 28, g = 8, h = 20), (E f = 28, g = 20, h = 8),
(G f = 30, g = 26, h = 4)]

Another A* example

Node: Queue : B
[(E f = 28, g = 20, h = 8), (E f = 29, g = 21, h = 8),
(G f = 30, g = 26, h = 4), (G f = 30, g = 26, h = 4)]
Pruning and Contours

- Topologically, we can imagine A* creating a set of contours corresponding to f values over the search space.
- A* will search all nodes within a contour before expanding.
- This allows us to prune the search space.
- We can chop off the portion of the search tree corresponding to Zerind without searching it.

Building Effective Heuristics

- While A* is optimally efficient, actual performance depends on developing accurate heuristics.
- Ideally, h is as close to the actual cost to the goal (h*) as possible while remaining admissible.
- Developing an effective heuristic requires some understanding of the problem domain.

Effective Heuristics - 8-puzzle

- h1 - number of misplaced tiles.
- This is clearly admissible, since each tile will have to be moved at least once.
- h2 - Manhattan distance between each tile's current position and goal position.
- Also admissible - best case, we'll move each tile directly to where it should go.
- Which heuristic is better?
Effective Heuristics - 8-puzzle

- $h_2$ is better.
- We want $h$ to be as close to $h^*$ as possible.
- If $h_2(n) > h_1(n)$ for all $n$, we say that $h_2$ dominates $h_1$.
- We would prefer a heuristic that dominates other known heuristics.

Finding a heuristic

- So how do we find a good heuristic?
- Solve a relaxed version of the problem.
- Cost of an optimal solution to the relaxed version is an admissible heuristic for the original problem.
- 8-puzzle - allow tiles to move over each other.
- Romania - assume that there is a road from every city to the goal.
- Solve subproblems
- Cost of moving one tile, traversing a path
- Often, these subproblems (and their solutions) are then cached.
- Learning from experience

Improving A*

- A* has one big weakness - Like BFS, it potentially keeps an exponential number of nodes in memory at once.
- Iterative deepening A* is a workaround.
- Rather than searching to a fixed depth, we search to a fixed f-cost.
- If the solution is not found, we increase $f$ and start again.
- Works in worlds with uniform, discrete-valued step costs

Improving A*

- Recursive best-first search
- Combination of DFS and A*.
- Do DFS, but keep the f-cost of all fringe nodes.
- If expansion leads to a node worse than something in the fringe, backtrack.
- Improvement over A*, but not spectacular.
- Both IDA* and RBFS throw away too much.

Improving A*

- SMA*
- Regular A*, plus a fixed limit on memory used.
- When memory is full, discard the node with the highest f.
- Value of discarded node is assigned to the parent.
- This allows SMA* to ‘remember’ the value of that branch.
- If all other branches get a higher f value, this child will be regenerated.
- SMA* is complete and optimal.
- On very hard problems, SMA* can wind up repeatedly deleting and regenerating branches.
- Moral: Often, memory requirements make our problem intractable before time requirements.

Summary

- Problem-specific heuristics can improve search.
- Greedy search
- A*
- Developing heuristics
- Admissibility, monotonicity, dominance
- Memory issues