UIT2504 Artificial Intelligence

Memory-Bounded Search Strategies

C. Aravindan
<AravindanC@ssn.edu.in>

Professor of Computing
SSN College of Engineering

August 19, 2024

Evaluating a state

- Sometimes, it may be possible to design an evaluation function $f(s)$ that evaluates the "badness" (to be minimized) or "goodness" (to be maximized) of a state s
- In such cases, the most desirable state may be chosen from the working set
- Working set is maintained as a priority queue based on the evaluation function f
- Obviously, the quality of search depends on the evaluation function f

Heuristics

- Usually, such an evaluation function $f(s)$ is designed based on some heuristics $h(s)$ — estimation of cost of reaching a goal state from state s
- For example, can you think of a heuristics for the route finding problem in a map? — Straight line distance (SLD) from the current city to the destination city
- Heuristics should be an easy function to compute!
- $h(s^*)$ should be 0 for any goal state s^*

Example: Sliding puzzle

- Consider the sliding puzzle, such as

3	2	7
5	8	
1	4	6

- What may be a good heuristics for this state space?
- $h_1(s)$: Number of misplaced tiles — for the above state $h_1(s) = 7$
- $h_2(s)$: Sum of Manhattan distances of tiles from their goal positions — for the above state $h_2(s) = 2 + 0 + 2 + 2 + 1 + 1 + 4 + 1 = 13$
- Which heuristics is better? — an estimate which is closer to the actual is always better!
- We say that h_2 dominates h_1
- An admissible heuristics is one which does not overestimate — in our example, both $h_1(x)$ and $h_2(x)$ are admissible

Best-first A^* search

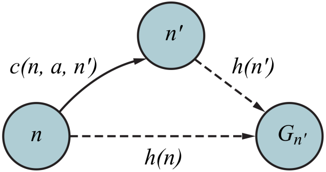
- In the greedy strategy, we have used only the "future" cost estimation to choose the next best state
- It may be prudent to consider evaluating a state s by the sum of cost of reaching that state s from the start state and the estimated cost of reaching a goal state from s
- In other words $f(s) = g(s) + h(s)$

Admissible Heuristics

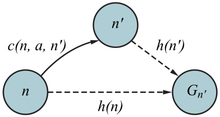
- A heuristic h is admissible if for every node n, $h(n) \leq h^*(n)$, where $h^*(n)$ is the **true** cost to reach the goal state from n.
- In other words, $f(n) = g(n) + h(n) \leq C^*$, where C^* is the optimal path cost
- An admissible heuristic **never overestimates** the cost to reach the goal, i.e., it is always **optimistic**
- The SLD heuristics and the two heuristics for sliding puzzle problem are examples of admissible heuristics

Consistent Heuristics

- A heuristics is consistent if for every node n, $h(n) \leq c(n, a, n') + h(n')$, where n' is a successor of n generated by some action a



Consistent Heuristics



- When h is consistent, we can infer the following

$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n, a, n') + h(n') \\
&\geq g(n) + h(n) \\
&\geq f(n)
\end{aligned}
$$

- That means, evaluation function f is monotonic — it is non-decreasing along any path
- Every consistent heuristics is also admissible

Properties of A^* search

- A^* is optimal with admissible heuristics
- A^* is, in general, complete — that is, it finds a solution, if exists
- A^* is optimally efficient — no other optimal algorithm is guaranteed to expand fewer nodes than A^*
- Time complexity? — $O(b^{\Delta})$ in general, where the absolute error $\Delta = h^* - h$ (where h^* is the actual cost) — it may also be expressed in terms of relative error, $\epsilon = (h^* - h)/h^*$, as $O(b^{\epsilon d})$ — however, it is fast, in practice, when good heuristics are used
- Space complexity? — same as that of time complexity (every generated node needs to be kept in memory) — worst-case space requirement is similar to that of breadth-first search — memory bounded strategies have been proposed to overcome this

Questions?

Beam Search

- Memory used by A^* search is, in some worst cases, comparable with the memory used by BFS, and could be a limiting factor

Beam Search

- Memory used by A^* search is, in some worst cases, comparable with the memory used by BFS, and could be a limiting factor
- One solution may be to limit the size of the frontier, resulting in what is known as the beam search

Beam Search

- Memory used by A^* search is, in some worst cases, comparable with the memory used by BFS, and could be a limiting factor
- One solution may be to limit the size of the frontier, resulting in what is known as the beam search
- Keep only k nodes with the best f-scores and discard other expanded nodes

Beam Search

- Memory used by A^* search is, in some worst cases, comparable with the memory used by BFS, and could be a limiting factor
- One solution may be to limit the size of the frontier, resulting in what is known as the beam search
- Keep only k nodes with the best f-scores and discard other expanded nodes
- We may compromise on the completeness and optimality, but this search runs very fast, and may find near optimal solutions for many problems

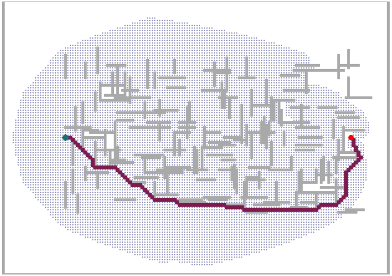
Beam Search

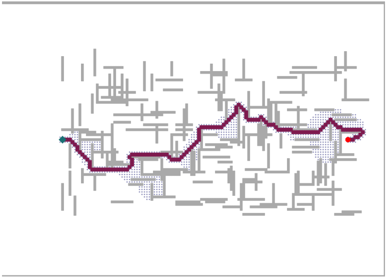
- Memory used by A^* search is, in some worst cases, comparable with the memory used by BFS, and could be a limiting factor
- One solution may be to limit the size of the frontier, resulting in what is known as the beam search
- Keep only k nodes with the best f-scores and discard other expanded nodes
- We may compromise on the completeness and optimality, but this search runs very fast, and may find near optimal solutions for many problems (Recall A^* search with inadmissible heuristics such as SLD with detour index)

Beam Search

- Memory used by A^* search is, in some worst cases, comparable with the memory used by BFS, and could be a limiting factor
- One solution may be to limit the size of the frontier, resulting in what is known as the beam search
- Keep only k nodes with the best f-scores and discard other expanded nodes
- We may compromise on the completeness and optimality, but this search runs very fast, and may find near optimal solutions for many problems (Recall A^* search with inadmissible heuristics such as SLD with detour index)
- An alternative version of beam search is to keep all the nodes with f-score within a limit of δ of the best f-score

Beam Search





Iterative Deepening A^*

- Similar to the iterative lengthening search

Iterative Deepening A^*

- Similar to the iterative lengthening search
- f-cost is used as a cut-off

Iterative Deepening A^*

- Similar to the iterative lengthening search
- f-cost is used as a cut-off
- Remember the smallest f-cost of any node that exceeded the cut-off in the previous iteration, and use it as cut-off for current iteration

Iterative Deepening A^*

- Similar to the iterative lengthening search
- f-cost is used as a cut-off
- Remember the smallest f-cost of any node that exceeded the cut-off in the previous iteration, and use it as cut-off for current iteration
- Drawback: Too many iterations required when f-cost increases very slowly

Recursive Best-First Search (RBFS)

- Dynamically adjusts the cost-limit for a particular path

Recursive Best-First Search (RBFS)

- Dynamically adjusts the cost-limit for a particular path
- Keeps track of the best alternative path available from any ancestor of the current node

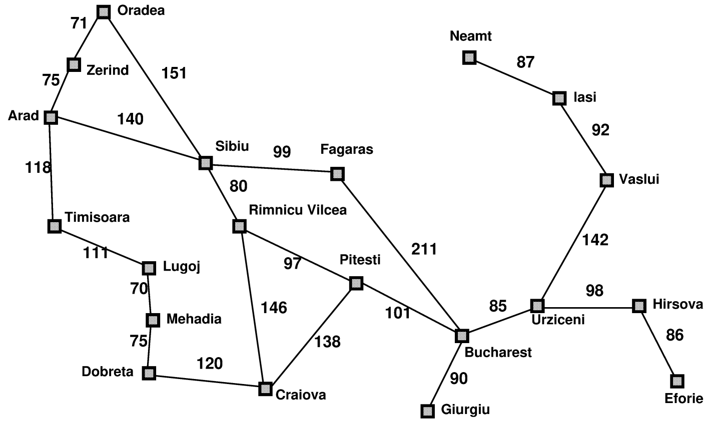
Recursive Best-First Search (RBFS)

- Dynamically adjusts the cost-limit for a particular path
- Keeps track of the best alternative path available from any ancestor of the current node
- When the best child exceeds this limit, the search unwinds

Recursive Best-First Search (RBFS)

- Dynamically adjusts the cost-limit for a particular path
- Keeps track of the best alternative path available from any ancestor of the current node
- When the best child exceeds this limit, the search unwinds
- While unwinding, cost of current node is replaced with that of best child

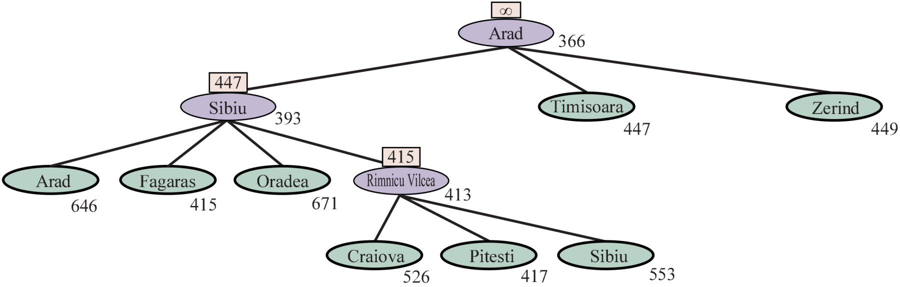
RBFS Illustration



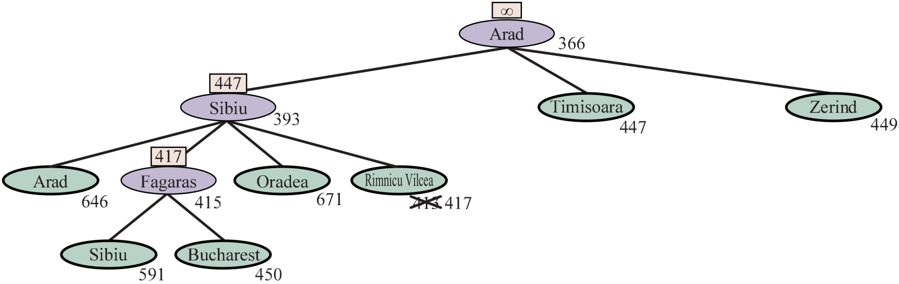
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Find the best route from Arad to Bucharest

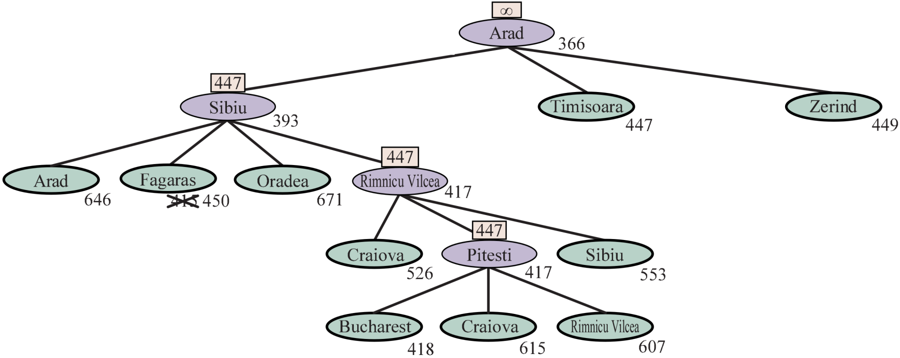
RBFS Illustration



RBFS Illustration



RBFS Illustration



RBFS

- RBFS is generally better than IDA^*, but may still suffer from excessive node regeneration

RBFS

- RBFS is generally better than IDA^*, but may still suffer from excessive node regeneration
- Every path change could require many reexpansions of forgotten nodes

RBFS

- RBFS is generally better than IDA^*, but may still suffer from excessive node regeneration
- Every path change could require many reexpansions of forgotten nodes
- Linear space complexity, but at the cost of increased time (due to several reexpansions)

RBFS

- RBFS is generally better than IDA^*, but may still suffer from excessive node regeneration
- Every path change could require many reexpansions of forgotten nodes
- Linear space complexity, but at the cost of increased time (due to several reexpansions)
- Ends up under utilizing the available memory!

Simplified Memory-Bounded A^* (SMA*)

- Motivation: Make use of as much memory as possible!

Simplified Memory-Bounded A^* (SMA*)

- Motivation: Make use of as much memory as possible!
- Proceed like normal A^* until memory permits

Simplified Memory-Bounded A^* (SMA*)

- Motivation: Make use of as much memory as possible!
- Proceed like normal A^* until memory permits
- Throw away the node with the largest f-value to get space for new ones

Simplified Memory-Bounded A^* (SMA*)

- Motivation: Make use of as much memory as possible!
- Proceed like normal A^* until memory permits
- Throw away the node with the largest f-value to get space for new ones
- Like RBFS, remember the best forgotten ones in its parent

Simplified Memory-Bounded A^* (SMA^*)

- Motivation: Make use of as much memory as possible!
- Proceed like normal A^* until memory permits
- Throw away the node with the largest f-value to get space for new ones
- Like RBFS, remember the best forgotten ones in its parent
- Drawback: How do we handle the situation where there are several nodes with same f-cost?

Simplified Memory-Bounded A^* (SMA*)

- Motivation: Make use of as much memory as possible!
- Proceed like normal A^* until memory permits
- Throw away the node with the largest f-value to get space for new ones
- Like RBFS, remember the best forgotten ones in its parent
- Drawback: How do we handle the situation where there are several nodes with same f-cost?
- Drawback: In some harder problems, situation similar to *thrashing* may occur

Bidirectional heuristic search

- Conduct bidirectional search (with two frontiers) with the hope of reducing the complexity, but optimality can not be guaranteed any more

Bidirectional heuristic search

- Conduct bidirectional search (with two frontiers) with the hope of reducing the complexity, but optimality can not be guaranteed any more
- Choosing the best f-score individually from each frontier may not be optimal

Bidirectional heuristic search

- Conduct bidirectional search (with two frontiers) with the hope of reducing the complexity, but optimality can not be guaranteed any more
- Choosing the best f-score individually from each frontier may not be optimal
- We need to consider pairs of nodes (one from each frontier) and choose the best pair

Bidirectional heuristic search

- Conduct bidirectional search (with two frontiers) with the hope of reducing the complexity, but optimality can not be guaranteed any more
- Choosing the best f-score individually from each frontier may not be optimal
- We need to consider pairs of nodes (one from each frontier) and choose the best pair
- Forward direction: $f_F(n) = g_F(n) + h_F(n)$ and backward direction: $f_B(n) = g_B(n) + h_B(n)$

Bidirectional heuristic search

- Consider a path from initial state to node m, and backward path from goal to node n. Then the pair (m, n) may be evaluated as

$$lb(m, n) = max(g_F(m) + g_B(n), f_F(m), f_B(n))$$

Bidirectional heuristic search

- Consider a path from initial state to node m, and backward path from goal to node n. Then the pair (m, n) may be evaluated as

$$lb(m, n) = max(g_F(m) + g_B(n), f_F(m), f_B(n))$$

- If $lb(m, n) < C^*$, then we must expand either m or n, but we don't know for sure which one to expand first

Bidirectional heuristic search

- Consider a path from initial state to node m, and backward path from goal to node n. Then the pair (m, n) may be evaluated as

$$lb(m, n) = max(g_F(m) + g_B(n), f_F(m), f_B(n))$$

- If $lb(m, n) < C^*$, then we must expand either m or n, but we don't know for sure which one to expand first
- Instead, we can still maintain two separate frontiers, and maintain $f_2(n) = max(2g(n), g(n) + h(n))$, and expand nodes with the best f_2 scores.

Bidirectional heuristic search

- Consider a path from initial state to node m, and backward path from goal to node n. Then the pair (m, n) may be evaluated as

$$lb(m, n) = max(g_F(m) + g_B(n), f_F(m), f_B(n))$$

- If $lb(m, n) < C^*$, then we must expand either m or n, but we don't know for sure which one to expand first
- Instead, we can still maintain two separate frontiers, and maintain $f_2(n) = max(2g(n), g(n) + h(n))$, and expand nodes with the best f_2 scores.
- This f_2 function guarantees that the algorithm never expands a node with $g(n) > \frac{C^*}{2}$

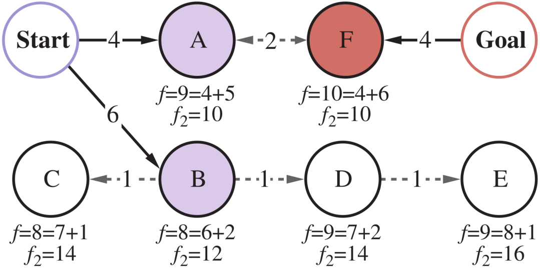
Bidirectional heuristic search

- Consider a path from initial state to node m, and backward path from goal to node n. Then the pair (m, n) may be evaluated as

$$lb(m, n) = max(g_F(m) + g_B(n), f_F(m), f_B(n))$$

- If $lb(m, n) < C^*$, then we must expand either m or n, but we don't know for sure which one to expand first
- Instead, we can still maintain two separate frontiers, and maintain $f_2(n) = max(2g(n), g(n) + h(n))$, and expand nodes with the best f_2 scores.
- This f_2 function guarantees that the algorithm never expands a node with $g(n) > \frac{C^*}{2}$
- Bidirectional search with the evaluation function f_2 and an admissible heuristic h is complete and optimal.

Bidirectional heuristic search



Questions?