HEURISTIC SEARCH

Dr. Hrishikesh Bhaumik
Associate Professor
Department of Information Technology
RCC Institute of Information Technology

Heuristics

Heuristic means the study of the **methods** and **rules** for discovery and invention.

In state space search, **heuristics** are **formalized** as **rules for choosing** those branches in a **state space** that are most likely to lead to an **acceptable problem solution.**

Heuristics

Heuristics may be applied in two basic situations:

1. A problem may **not have an exact solution** because of inherit ambiguities in the problem statement or available data. Ex: Medical Diagnosis.
2. A problem may have an **exact solution**, but the **computational cost** of finding it may be **very large** Ex: Solution of a chess problem.

Heuristics

In these cases, **brute force search** techniques **cannot** be applied.

Heuristics attack the **complexity** by guiding the search along the **most promising path**.

Heuristics provide **no guarantee of success** but it is an important search strategy because the **exponential nature** of the problems are **reduced to polynomial number**, thereby obtaining a **solution in a tolerable amount of time**.

Heuristic Information

F*(n) = g*(n) + h* (n)
F*(n) = Total function

Two components g*(n) and h*(n) are estimates of distance from the start node to node n and node n to goal node respectively.

Nodes in the open list are nodes that have been generated but not expanded while nodes on the closed list are nodes that have been expanded and who have children.

Desirable properties of Heuristic Search

- Admissibility condition- algo A is admissible if it is guaranteed to return an optimal solution when one exist.
- Completeness property- algo A is complete if it always terminates with a solution when one exist.
- Dominance property – A1 ,A2 be admissible algorithm with heuristic estimation function h1* and h2* respectively . A1 is said to be more informed than A2 if h1*(n)>h2*(n). Here A1 is said to dominate A2
- Optimality property – algo A is optimal over a class of algorithms if A dominate all member of the class.

Hill Climbing Method

At each point in the search path, a successor node that appears to lead most quickly to the goal (hill top) is selected for exploration.

When the children have been generated, alternative choices are evaluated using some type of heuristic function and no reference to the parent or other children is retained.

This process continues from node to node with previously expanded nodes being discarded. Hill climbing produces substantial savings over blind searches when an informative, reliable function is available to guide the search to a global goal.

Problems in Hill Climbing

1. Foothill trap: This results when local maxima or peaks are found. The children have less promising goal distance than the parent node. The search is trapped at the local node with no indication of goal direction. The remedy to this problem is to try moving in some arbitrary direction a few generations in the hope that the real goal is evident. Back tracking to an ancestor node and trying a secondary path choice or altering the computation procedure to expand a head a few generations each time before choosing a path can be some remedies.
2. Ridge: When several adjoining nodes have higher values than surrounding nodes.
3. Plateau: When all neighbouring nodes have the same values.

Best-First Search

In this algorithm, **all estimates computed** from **previously generated nodes** are **retained** and makes its **selection based on the best** among them all.

Best-First searches requires that a **good measure of goal distance** be used.

Best-First Search

Algorithm:

1. Place the **starting node** S on the queue.
2. If the **queue is empty**, return **failure** and **stop**.
3. If the **first element** on the queue is a **goal node g**, return **success** and **stop**. Otherwise goto step 4.
4. **Remove the first element** from the queue, expand it and **compute the estimated goal distance** for **each child**. Place the children on the queue (at either end) and **arrange** all queue elements in **ascending order** corresponding to **goal distance** from the **front of the queue**.
5. Return to step 2.

Branch and Bound Search

This can be applied to problems having a **graph search space** where **more than one alternative path** may exists between two nodes.

It **saves all path lengths** (costs) from a node to all generated nodes and **chooses the shortest path** for further expansion.

It then **compares the new path lengths** with all old ones and again chooses **the shortest path for expansion**. Thus a path to a goal node is certain to be a **minimal length path**, but it is at the expense of **computing and remembering** all competing paths.

Branch and Bound Search Algorithm

1. Place the **start node** of zero path length on the queue.
2. Until the **queue is empty** or a goal node has been found:
   a. **Determine** if the **first path** in the queue contains a **goal node**.
   b. If the first path contains a **goal node exit with success**.
   c. If the first path **does not contain** a goal node, **remove the path** from the queue and **form new paths** by extending the removed path by one step.
   d. Compute the **cost of the new paths** and add them to the queue.
   e. **Sort the paths** on the queue with lowest cost paths in front.
3. Otherwise, **exit with failure**.

A* Algorithm

This is a specialization of the best-first search. It provides general guidelines which help to estimate goal distances for general search graphs.

At each node along a path to the goal, the A* algorithm generates all successor nodes and computes an estimate of the distance(cost) from the start node to a goal node through each of the successors. It then chooses the successor with the shortest estimated distance for expansion. The successors for this node are then generated, their distances estimated, and the process continues until a goal is found or search ends in failure. The heuristic estimation function for A* is $f^*(n)=g^*(n)+h^*(n)$ where $g^*(n)$ is the estimate of the cost (or distance) from the start node to node n and $h^*(n)$ is the cost form node n to a goal node.

$f^*(n)$= Total function.

Nodes in the open list are nodes that have been generated but not expanded while nodes on the closed list are nodes that have been expanded and who have children.

A* Algorithm

1. Place the starting node S in open list.
2. If open is empty, stop and return failure
3. Remove from the open node n that has the smallest value of $f^*(n)$, if the node is a goal node return success and stop. Otherwise go to step 4.
4. Expand n, generating all of its successors n' and place n in closed list. For every successor n', if n' is not already on open or closed attach a back-pointer to n, compute $f^*(n')$ and place it in open list.
5. Each n' that is already in open or closed should be attached to back-pointers, which reflect the lowest $g^*(n')$ path. If n' was on closed and its pointer was changed, remove it and place if in open list.
6. Return on step 2.

AO* Algorithm

This algorithm requires that nodes traversed in a tree be labeled as solved or unsolved in the solution process to account for AND node solutions which require solutions to all successor nodes. A solution is found when the start node is labeled as solved.

Algorithm

1. Place the start node s on open.
2. Using the search tree constructed thus for, compute the most promising solution tree T0.
3. Select a node n that is both on open and a part of To. Remove n from open and place it on closed.

AO* Algorithm contd..

4. If n is a terminal goal node, labeled n as solved. If the solution of n results in any of n's ancestors being solved, label all the ancestors as solved. If the start node s is solved, exit with success where To is the solution tree. Remove from open all node with a solved ancestor.
5. If n is not a solvable node (operations cannot be applied), label n as unsolvable. If the start node is labeled unsolvable, exit with failure. If any of n's ancestors become unsolvable because n is, label them unsolvable as well. Remove from open all nodes with unsolvable ancestors.
6. Otherwise, expand node n generating all of its successors. For each such successor node that contains more than one sub-problem, generate their successors to give individual sub-problems. Attach to each newly generated node a back pointer to its predecessor. Compute the cost estimate h* for each newly generated node and place all such nodes that do not yet have descendents on open. Next, recomputed the values of h* at n and each ancestor of n.
7. Return to step 2.