Chapter 2

Basic Heuristic-Search

Procedures
AN IRREVERSIBLE STRATEGY

2.1 HILL-CLIMBING

Hill-climbing is an irreversible strategy. It makes no use of backtracking or backjumping to previous states. Once a decision is made, it cannot be undone. This makes hill-climbing vulnerable to getting stuck in local maxima. However, it is simple and effective in many cases.

Hill-climbing is generally more efficient than other methods, but it does not guarantee finding the global optimum. It is particularly useful when the search space is large and the cost of evaluating the fitness of a solution is high.

In summary, hill-climbing is a powerful tool for optimization problems, but it requires careful consideration of the search space and the fitness function to avoid getting stuck in local maxima.

SECTION 2.1

Basic Functional-Expression Procedures

Chapter 2
In Depth-first Search, as well as the popular variation called Breadth-first Search, the search tree is explored in a depth-first fashion, expanding nodes in a depth-first manner. If a goal node is encountered, the search terminates. If no goal node is encountered, the search continues, exploring deeper nodes in the search tree. In depth-first search, the first goal node encountered is explored, and if it is not the goal node, the search continues by expanding its children. If the search reaches a leaf node, it is declared a non-goal node, and the search backtracks to the previous node, exploring its remaining unexpanded children. This process continues until a goal node is found or the search tree is exhausted.

In many cases, depth-first search is used when the search space is large and the goal node is close to the root of the search tree. However, depth-first search can also lead to infinite loops if the search space is cyclic or contains loops. To prevent this, depth-first search can be augmented with heuristics or search methods that limit the depth of the search or use backtracking to avoid infinite loops.

In contrast, breadth-first search explores the search tree in a breadth-first manner, expanding nodes in a level-by-level order. This means that all nodes at a given depth are expanded before moving on to the next depth level. Breadth-first search is guaranteed to find the shortest path to a goal node, if it exists, but it can take a long time to find a goal node if the search space is large or if the goal node is far away from the root.

In summary, depth-first search is efficient when the goal node is close to the root of the search tree, but it can be prone to infinite loops if the search space is cyclic. Breadth-first search is guaranteed to find the shortest path to a goal node, but it can be inefficient if the search space is large or if the goal node is far away from the root. Both methods have their strengths and weaknesses, and the choice of which method to use depends on the specific problem at hand.

22
UNINFORMED SYSTEMATIC SEARCH
A diagram illustrating the search process for a given problem. The problem is represented by a graph, where nodes are states and edges are transitions between states. The search algorithm explores the graph by expanding nodes in a specific order, typically from left to right or top to bottom, until a solution is found or all possible paths are exhausted. The figure shows a partially ordered version of the solution path, indicating the progress of the search. The final solution is marked with a special symbol or color, indicating the goal state has been reached.
Backtracking is a version of depth-first search that applies the
Backtracking. Backtracking is accomplished with no extra
searching, whereas backtracking is accomplished with no extra
table search. When backtracking is accomplished with no extra
searching, the backtracking is accomplished with no extra
searching, whereas backtracking is accomplished with no extra
searching.
The finite depth search is the main reason that breadth-first search is used. The need for this search is often due to many steps in the game, which are needed to reach the goal. Only by maintaining a full copy of the search graph can a breadth-first search guarantee to find the shortest path to the goal. If the graph is finite, this guarantee is extended to finding a solution in a finite amount of time. However, if the graph is infinite, this guarantee is lost. The search is still guaranteed to find a solution if a solution exists, but it may not find the shortest path. This is because the search may explore paths that are longer than the shortest path to the goal.

Breadth-first search guarantees that the first time a solution is found, it is the shortest path to the goal. If the solution is not found, the search may take a long time to find it. However, if the solution exists, the search will find it eventually.

Breadth-first search is often used in games and puzzles, where the goal is to find the shortest path to the solution. It is also used in artificial intelligence, where the goal is to find the best path to the solution. In both cases, breadth-first search is a powerful tool for finding solutions to problems.
OF USING HEURISTIC INFORMATION

2.3 INFORMED: BEST-FIRST SEARCH: A WAY

The most natural place for using heuristic information is in deciding which

Let it help in backtracking search (sec. 2.8)

Figure 2.8

(e)
A General Best-First Strategy for AND/OR Graphs (GBF)

In the previous sections, we have discussed various search strategies for AND/OR graphs, including best-first search and depth-first search. In this section, we will introduce a general best-first strategy for AND/OR graphs called GBF (Gottlob, 1990).

The GBF algorithm is similar to the A* algorithm, which we discussed in the previous section. However, GBF is specifically designed for AND/OR graphs, where the state space is a graph rather than a tree.

The GBF algorithm works as follows:

1. The algorithm starts with an initial state, which is the root of the AND/OR graph.
2. The algorithm maintains two sets: an OPEN list and a CLOSED list. The OPEN list contains nodes that have not yet been expanded, while the CLOSED list contains nodes that have been expanded.
3. The algorithm selects the node with the lowest estimated cost from the OPEN list. The estimated cost is calculated as the sum of the cost to reach the node from the root and an heuristic estimate of the cost to reach a goal state from the node.
4. If the selected node is a goal state, the algorithm returns the solution path from the root to the goal state.
5. If the selected node is not a goal state, the algorithm expands the node and adds its successor nodes to the OPEN list. The successor nodes are added to the OPEN list in order of their estimated cost.
6. The algorithm repeats steps 3-5 until a goal state is found or the OPEN list becomes empty.

The GBF algorithm is guaranteed to find a solution if one exists, and it is optimal if the heuristic function is admissible. However, it may not be the most efficient algorithm for large AND/OR graphs, as it can expand many nodes that are not on the optimal path to the goal state.

In the next section, we will discuss another best-first strategy for AND/OR graphs, called DRAF (Depth-Reduced A* for AND/OR Functions), which is more efficient than GBF for large graphs.
THE GB Algorithm

1. From the expression graph of the net, connect all nodes as per (initially just 3).
2. If the main node is OPEN, go to step 3.
3. If the node is closed, go to step 4.
4. If the node is an independent node, go to step 5.
5. If the node is a dependent node, go to step 6.
6. If the node is a function node, go to step 7.
7. If the node is a function, go to step 8.

Conditions:
- Support The minimal base and the expression graph of 0, and let the node be open, if the node is not open, go to step 9.
- Support 0, let the node be open, if the node is not open, go to step 10.
- Support 0, let the node be open, if the node is not open, go to step 11.
- Support 0, let the node be open, if the node is not open, go to step 12.
- Support 0, let the node be open, if the node is not open, go to step 13.
- Support 0, let the node be open, if the node is not open, go to step 14.
- Support 0, let the node be open, if the node is not open, go to step 15.
- Support 0, let the node be open, if the node is not open, go to step 16.
- Support 0, let the node be open, if the node is not open, go to step 17.
- Support 0, let the node be open, if the node is not open, go to step 18.
- Support 0, let the node be open, if the node is not open, go to step 19.

The GB Algorithm ends when all nodes are connected.
the only solution graph possible.

The solution graph for the optimal solution may be visualized as follows:

2. Compute the most promising solution graph from the solution graph.

3. From the most promising solution graph, identify the optimal solution.

4. Solve for the optimal solution.

5. If the optimal solution is found, proceed to step 6. Otherwise, go to step 2.

6. End the simulation.

The above steps outline the process for finding the optimal solution graph.

Example:

Given a set of flow graphs, apply the above steps to determine the optimal solution graph.

1. Construct the initial solution graph.

2. Identify the most promising solution graph.

3. Solve for the optimal solution graph.

4. The optimal solution graph is found.

5. The optimal solution graph is as follows:

   - Node A
   - Node B
   - Node C
   - Node D
   - Node E

6. End the simulation.

The above example demonstrates the application of the above steps to find the optimal solution graph.
2.4.2 Specilized Best-First Algorithms:

2.4.2.1 Why Restart the Evaluation Function?
yields the expected minimum rule:

\[
\begin{array}{c}
\text{if } u \text{ is an AND node, then }

\begin{aligned}
(u)_n &= (u)_n \\
\text{if } u \text{ is an OR node, then }

\begin{aligned}
[(u)_n] + [(u)']_n &= (u)_n \\
\text{if } u \text{ is terminal, then }

\begin{aligned}
0 &= (u)_n
\end{aligned}
\end{aligned}
\end{aligned}
\]

For example, in example 2 of the Connected Containter Problem in section 2.4.2, the optimal cost of the container is 7. The optimal solution graph is shown in the figure. The node in the optimal solution graph is labeled with the cost and the optimal solution. The graph is represented as a directed graph with edges indicating the flow of material. For each node, the optimal solution is determined. The final solution is the sum of the costs of all the nodes in the graph.

Section 2.4.3: The Most Promising Solution-Node Graph

We will now demonstrate the weight-reduction procedure on a new example.

Let us demonstrate the weight-reduction procedure on a new example.
Specialized Best-First Strategies

2.4.1 For each node on OPEN

(1)

\[(u) = (u) \cdot \delta = (u) \cdot \delta \]

The evaluation function \( f \) is defined as the sum of two terms: \( g(u) \) and \( h(u) \). If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

(2)

If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

2.4.2 Specialized Best-First Strategies

2.4.2.1 The evaluation function gives rise to the following look-down definition of \( \delta \):

\[
\delta(a) = \min \{ f(u) : u \in OPEN, a \in u, u \text{ is a closed node} \}
\]

3. Definition of \( \delta \):

(3)

\[
\delta(a) = \min \{ f(u) : u \in OPEN, a \in u, u \text{ is a closed node} \}
\]

The evaluation function gives rise to the following look-down definition of \( \delta \):

For each node on OPEN

(4)

\[(u) \cdot \delta = (u) \cdot \delta \]

The evaluation function \( f \) is defined as the sum of two terms: \( g(u) \) and \( h(u) \). If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

(5)

If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

2.4.2.2 Specialized Best-First Strategies

2.4.2.2.1 The evaluation function gives rise to the following look-down definition of \( \delta \):

\[
\delta(a) = \min \{ f(u) : u \in OPEN, a \in u, u \text{ is a closed node} \}
\]

The evaluation function gives rise to the following look-down definition of \( \delta \):

For each node on OPEN

(6)

\[(u) \cdot \delta = (u) \cdot \delta \]

The evaluation function \( f \) is defined as the sum of two terms: \( g(u) \) and \( h(u) \). If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

(7)

If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

2.4.2.3 Specialized Best-First Strategies

2.4.2.3.1 The evaluation function gives rise to the following look-down definition of \( \delta \):

\[
\delta(a) = \min \{ f(u) : u \in OPEN, a \in u, u \text{ is a closed node} \}
\]

The evaluation function gives rise to the following look-down definition of \( \delta \):

For each node on OPEN

(8)

\[(u) \cdot \delta = (u) \cdot \delta \]

The evaluation function \( f \) is defined as the sum of two terms: \( g(u) \) and \( h(u) \). If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

(9)

If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

2.4.2.4 Specialized Best-First Strategies

2.4.2.4.1 The evaluation function gives rise to the following look-down definition of \( \delta \):

\[
\delta(a) = \min \{ f(u) : u \in OPEN, a \in u, u \text{ is a closed node} \}
\]

The evaluation function gives rise to the following look-down definition of \( \delta \):

For each node on OPEN

(10)

\[(u) \cdot \delta = (u) \cdot \delta \]

The evaluation function \( f \) is defined as the sum of two terms: \( g(u) \) and \( h(u) \). If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

(11)

If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

2.4.2.5 Specialized Best-First Strategies

2.4.2.5.1 The evaluation function gives rise to the following look-down definition of \( \delta \):

\[
\delta(a) = \min \{ f(u) : u \in OPEN, a \in u, u \text{ is a closed node} \}
\]

The evaluation function gives rise to the following look-down definition of \( \delta \):

For each node on OPEN

(12)

\[(u) \cdot \delta = (u) \cdot \delta \]

The evaluation function \( f \) is defined as the sum of two terms: \( g(u) \) and \( h(u) \). If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

(13)

If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

2.4.2.6 Specialized Best-First Strategies

2.4.2.6.1 The evaluation function gives rise to the following look-down definition of \( \delta \):

\[
\delta(a) = \min \{ f(u) : u \in OPEN, a \in u, u \text{ is a closed node} \}
\]

The evaluation function gives rise to the following look-down definition of \( \delta \):

For each node on OPEN

(14)

\[(u) \cdot \delta = (u) \cdot \delta \]

The evaluation function \( f \) is defined as the sum of two terms: \( g(u) \) and \( h(u) \). If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

(15)

If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

2.4.2.7 Specialized Best-First Strategies

2.4.2.7.1 The evaluation function gives rise to the following look-down definition of \( \delta \):

\[
\delta(a) = \min \{ f(u) : u \in OPEN, a \in u, u \text{ is a closed node} \}
\]

The evaluation function gives rise to the following look-down definition of \( \delta \):

For each node on OPEN

(16)

\[(u) \cdot \delta = (u) \cdot \delta \]

The evaluation function \( f \) is defined as the sum of two terms: \( g(u) \) and \( h(u) \). If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).

(17)

If \( u \) is a closed node, then \( f(u) \) is the evaluation function of \( u \).
Chapter 3

The *A*lgorythm

Let *G* be a directed graph, *v* be the start node, *z* be the goal node. The problem is to find a path from *v* to *z*.

1. Create a search graph *G*′, initially consisting of the start node v. Place an edge from v to each of v′s neighbors. The search graph consists of all nodes that can be reached from a v by a sequence of edges in *G*.

2. Trace down the marked connections of a subgraph of *G*′. (A subgraph of *G*′ is a connected component of *G*′.)

3. Repeat step 1 and 2 until *z* is found.

4. Form the tree of the marked connections of a subgraph of *G*′.

5. Remove marked edges from *G* to form a new graph *G*′′.

6. Repeat step 1 and 2 until *z* is found in the tree of marked connections of *G*′′.

The *A*lgorythm is successful if *z* is found in the tree of marked connections of *G*′. The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.

The search graph *G*′ is a directed acyclic graph (DAG), and can be used to represent the problem in a way that can be solved by a simple algorithm.
2.5 Hybrid Strategies

\[ f = \frac{1}{2}g + \frac{1}{2}h \]

Drawbacks of previous procedures: The number of evaluations required in each search can be very high, which can lead to a large number of evaluations being performed. This can be improved by using hybrid strategies that combine the advantages of different approaches.

The following hybrid strategy can be used:

1. \textbf{Preprocessing Phase:}
   - Perform a heuristic search to find initial solutions.
   - Use a local search algorithm to refine these solutions.

2. \textbf{Main Search Phase:}
   - Use a hybrid search algorithm that combines the strengths of different methods.
   - Use a combination of hill-climbing, random search, and local search.

3. \textbf{Postprocessing Phase:}
   - Improve the solutions found by the hybrid search.
   - Use postprocessing techniques to enhance the quality of the solutions.

Overall, hybrid strategies can be effective in solving complex optimization problems.
2.5.1 B-Tree Combinations

The B-tree is a self-balancing search tree data structure that supports efficient insertion, deletion, and search operations. It is widely used in databases and file systems due to its ability to minimize disk I/O operations. The B-tree can be augmented to support various operations, such as range queries, by combining it with other data structures like BSTs or heaps.

**Figure 2.11**

(a) Schema of a TF (top-first) (followed by BF ending) BF
(b) Schema of a TF (top-first) (followed by BF ending) BF
(c) Alternatives for ORB, HHH-clipping (HHH), and Partial trees (P) and (T19) are in (T19) scope of ORB.

**Figure 2.12**

(a) Represents a TF, (b) represents a TF (top-first) (followed by BF ending) BF, and (c) represents a TF (top-first) (followed by BF ending) BF.

**Diagram 2.1**

- **Scenario 1:** In scenario 1, the decision node is marked as open. At this point, instead of placing it up, we employ a dynamic decision. Then, at the decision node, the decision is made.
- **Scenario 2:** In scenario 2, the decision node is marked as open. At this point, instead of placing it up, we employ a dynamic decision. Then, at the decision node, the decision is made.
The optimal policy is derived using a backward recursive approach. The Bellman equation is solved backwards from the terminal state to the initial state. The value function is updated iteratively until convergence. The optimal policy is the one that maximizes the expected return for each state.

The book by Hooton and Shin (1974) contains a thorough discussion of decision-making processes. These are particularly relevant in decision-making contexts where multiple criteria are involved.

**Remarks**

26. Bibliographic and Historical Remarks

The qualification criteria are that no position in the decision graph is visited more than once, and that all nodes are visited. This ensures that all relevant information is considered in the decision-making process.

The decision graph is constructed based on the following criteria:

1. All nodes are visited exactly once.
2. The decision graph is acyclic.
3. The decision graph is connected.

The decision graph is represented using a decision tree. Each node in the tree represents a decision point, and each branch represents a possible decision path.

![Decision Tree Example](image)

Basic Heuristic Search Procedures

In decision-making contexts, the search for the optimal policy is often referred to as **heuristic search**. This approach involves using heuristic functions to guide the search process. The heuristic function estimates the cost of reaching the goal from a given state.”

The heuristic function is defined as:

\[ h(n) = \text{estimated cost to reach the goal from state } n \]

The goal of heuristic search is to find a path from the initial state to the goal state with the minimum cost. This is achieved by exploring the search space in a way that is guided by the heuristic function.

The basic heuristic search procedures include:

1. **Uninformed Search**:
   - **Breadth-First Search (BFS)**: Explores all nodes at the current depth before moving to the next depth level.
   - **Depth-First Search (DFS)**: Explores as far as possible along each branch before backtracking.

2. **Informed Search**:
   - **A* Search**: Combines the cost of reaching the node (g(n)) with the estimated cost to reach the goal (h(n)) to guide the search.

---

**Figure 1.3**: Decision tree example.