Search Techniques for Artificial Intelligence
Pathfinding in maze – simulation in Greenfoot

Search plays a major role in solving many Artificial Intelligence (AI) problems. Search is a universal problem-solving mechanism in AI. In many problems, sequence of steps required to solve is not known in advance but must be determined by systematic trial-and-error exploration of alternatives.

The problems that are addressed by AI search algorithms fall into three general classes:

- single-agent path-finding problems,
- two-players games,
- constraint-satisfaction problems

Single-agent path-finding problems

Classic examples in the AI literature of path-finding problems are sliding-title puzzles, Rubik's Cube and theorem proving. The single-title puzzles are common test beds for research in AI search algorithms as they are very simple to represent and manipulate. Real-world problems include the traveling salesman problem, vehicle navigation, and the wiring of VLSI circuits. In each case, the task is to find a sequence of operations that map an initial state to a goal state.

Two-players games

Two-players games are two-player perfect information games. Chess, checkers, and othello are some of the two-player games.

Constraint Satisfaction Problems

Eight Queens problem is the best example. The task is to place eight queens on an 8*8 chessboard such that no two queens are on the same row, column or diagonal. Real-world examples of constraint satisfaction problems are planning and scheduling applications.

Problem Space

Problem space is a set of states and the connections between to represent the problem. Problem space graph is used to represent a problem space. In the graph, the states are represented by nodes of the graph, and the operators by edges between nodes. Although most problem spaces correspond to graphs with more than one path between a pair of nodes, for simplicity they are often represented as trees, where the initial state is the root of the tree. The cost of the simplification is that any state that can be reached by two different paths will be represented by duplicate nodes thereby increasing the tree size. The benefit of using tree is that the absence of cycles greatly simplifies many search algorithms.

Uninformed (blind) search algorithms

- Breadth-first search (BFS)
- Depth-First Search
- Bidirectional Search
- Uniform Cost Search

Informed (Heuristic) Search Strategies

- A * Search (Best First search)
- Greedy Best First Search

Local Search Algorithms

- Hill-Climbing Search
- Local Beam Search

Implement algorithms was made in Greenfoot application. Searching is done in a maze randomly generated. The maze is generated by Hunt and Kill algorithm.

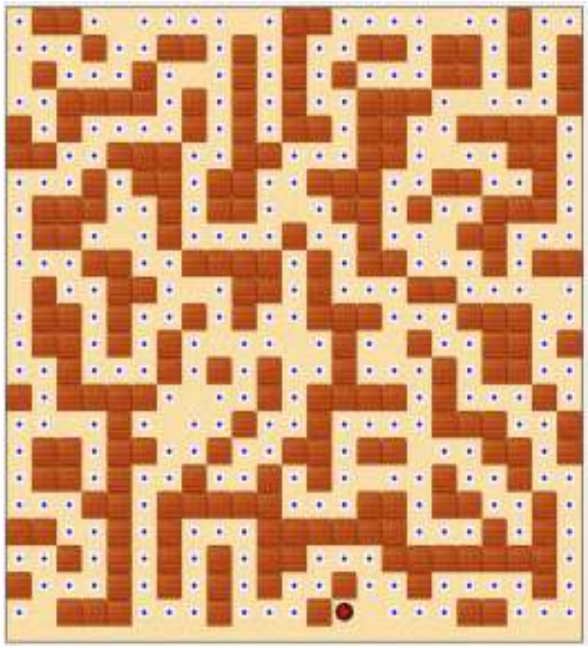

Illustration 1: Maze

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts from the root node, explores the neighboring nodes first and moves towards the next level neighbors. It generates one tree at a time until the solution is found. It can be implemented using FIFO queue data structure. This method provides shortest path to the solution.

BFS was invented in the late 1950s by E. F. Moore, who used it to find the shortest path out of a maze,[2] and discovered independently by C. Y. Lee as a wire routing algorithm (published 1961).

BFS explores the tree uniformly checks all paths one step away from the start, then two steps, then three, and so on.

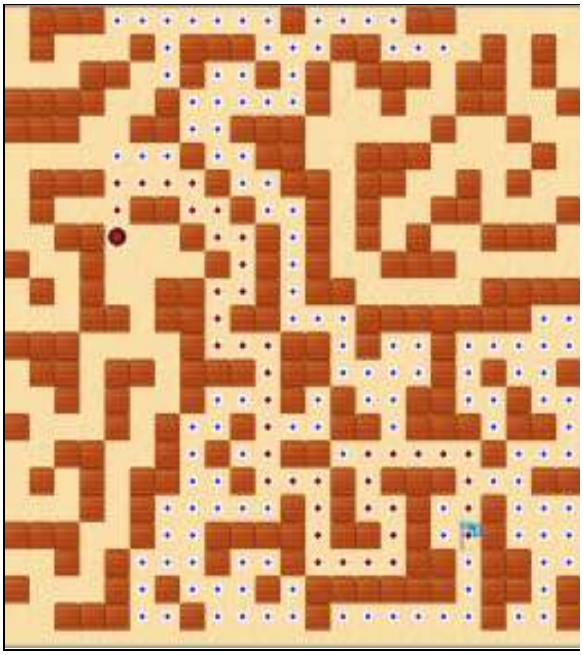

Illustration 2: Breadth-first search

Breadth-first search is useful when

- space is not a problem;
- you want to find the solution containing the fewest arcs;
- few solutions may exist, and at least one has a short path length; and
- infinite paths may exist, because it explores all of the search space, even with infinite paths.

It is a poor method when all solutions have a long path length or there is some heuristic knowledge available. It is not used very often because of its space complexity.

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Trémaux as a strategy for solving mazes.

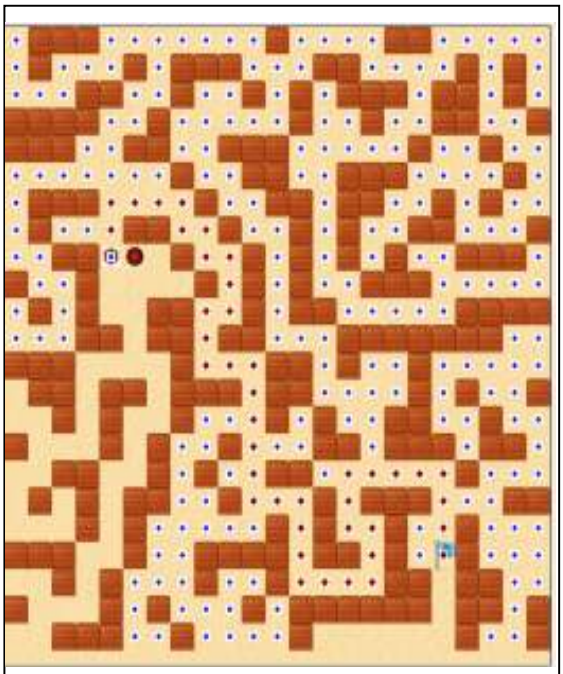

Illustration 3 Depth-first search

Depth-first search is appropriate when either

- space is restricted;
- many solutions exist, perhaps with long path lengths, particularly for the case where nearly all paths lead to a solution; or
- the order of the neighbors of a node are added to the stack can be tuned so that solutions are found on the first try.

It is a poor method when

- it is possible to get caught in infinite paths; this occurs when the graph is infinite or when there are cycles in the graph; or

- solutions exist at shallow depth, because in this case the search may look at many long paths before finding the short solutions.

Bidirectional Search

It searches forward from initial state and backward from goal state till both meet to identify a common state.

The path from initial state is concatenated with the inverse path from the goal state. Each search is done only up to half of the total path.

The idea of a bidirectional search is to reduce the search time by searching forward from the start and backward from the goal simultaneously. When the two search frontiers intersect, the algorithm can reconstruct a single path that extends from the start state through the frontier intersection to the goal.

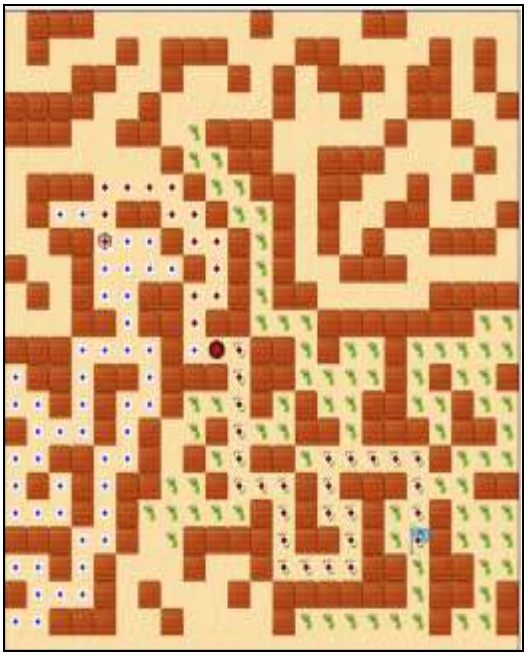

Illustration 4: Bidirectional Search

Uniform Cost Search

Sorting is done in increasing cost of the path to a node. It always expands the least cost node. It is identical to Breadth First search if each transition has the same cost.

It explores paths in the increasing order of cost.

***A** search** is a combination of lowest-cost-first and best-first searches that considers both path cost and heuristic information in its selection of which path to expand. For each path on the frontier, *A** uses an estimate of the total path cost from a start node to a goal node constrained to start along that path. It uses *cost(p)*, the cost of the path found, as well as the heuristic function *h(p)*, the estimated path cost from the end of *p* to the goal.

Greedy Best First Search

It expands the node that is estimated to be closest to goal. It expands nodes based on f(n) = h(n). It is implemented using priority queue.

Disadvantage − It can get stuck in loops. It is not optimal.

Hill Climbing is a variety of depth-first (generate - and - test) search. A feedback is used here to decide on the direction of motion in the search space. In the depth-first search, the test function will merely accept or reject a solution. But in hill climbing the test function is provided with a heuristic function which provides an estimate of how close a given state is to goal state

Hill climbing becomes inefficient in large problem spaces, and when combinatorial explosion occurs. But it is a useful when combined with other methods.

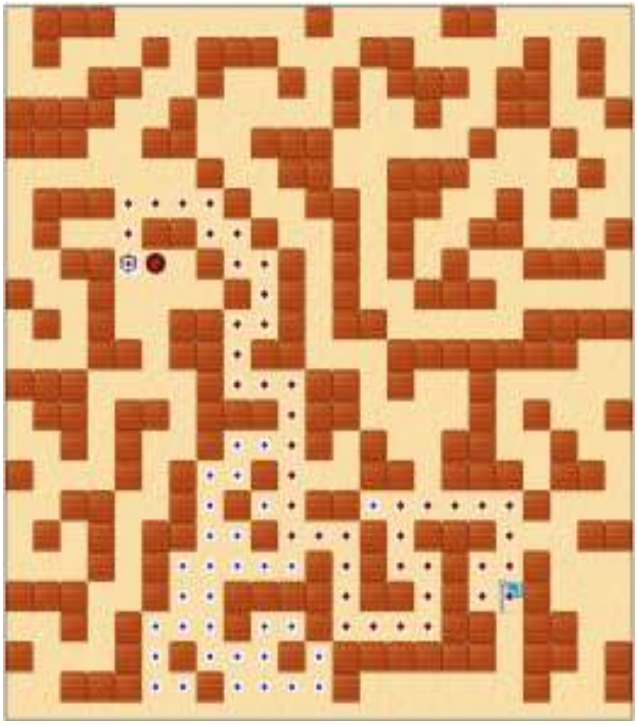

Illustration 5: Hill Climbing

Local Beam Search

In this algorithm, it holds k number of states at any given time. At the start, these states are generated randomly. The successors of these k states are computed with the help of objective function. If any of these successors is the maximum value of the objective function, then the algorithm stops.