## 1. Problem Solving using Search

The focus of much AI research has been on solving problems. Much of the point of the AI research was to understand "how" to solve the problem, not just to get a solution. So seemingly simple problems—puzzles, games, stacking blocks—were the focus of AI programs. And one of the first areas of work, general problem-solving methods, highlighted a major barrier to artificial intelligence software. How do you represent a problem so that the computer can solve it? Even before that, how do you define the problem with enough precision so that you can figure out how to represent it?

Steps of solving the problem are:-

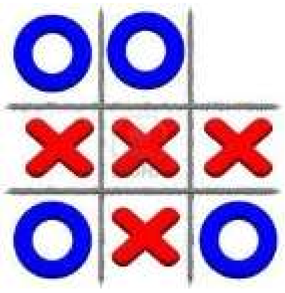
a- Clearly and succinctly define what it is we are trying to do.

b- Clarifying of how to represent our problem "representation" so that we can solve it using search techniques.

c- Which search techniques are best to solve the problem?

We explore one of the primary problem representations, ***the state-space approach.***

**Example :** Suppose that the problem we want to solve deals with playing and winning a game. This game could be a simple one such as *tic-tac-toe* or a more complex one such as checkers, chess, or backgammon



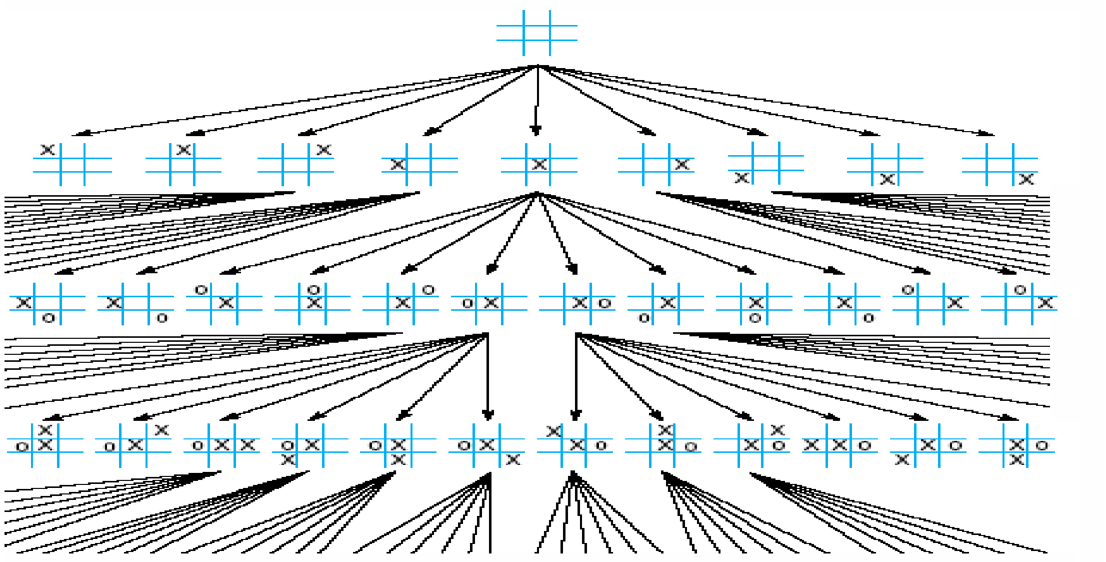
To solve each problem in AI, we need these Steps:-

- *initial state*, which in the case of tic-tac-toe could be a 3 by 3 matrix filled with spaces (or empty markers).
- *Operators (Control Strategy) :* This can be used to modify the current state, thereby creating a new state. In our case, this would be a player marking an empty space with either an X or an O.

The combination of the *initial state* and the set of *operators* make up the *state space* of the problem. The sequence of states produced by the valid application of operators from the initial state is called the *path* in the state space.

- *State space* search characterizes problem solving as the process of finding a solution path form the start state to a goal

- *goal state:* A goal may describe a state, such as winning board in tic-tac-toe. In tic-tac-toe a goal state is when any row, column, or diagonal consists of all Xs or Os. In this simple example, we can check the tic-tac-toe board to see if either player has won by explicitly testing for our goal condition. In more complicated problems, defining the *goal test* may be a substantial problem in itself.



In many search problems, we are not only interested in reaching a goal state, we would like to reach it with the lowest possible cost (or the maximum profit). Thus, we can compute a cost as we apply operators and transition from state to state. This *path cost* or cost function is usually denoted by *g*. Given a problem which can be represented by a set of states and operators and then solved using a search algorithm, we can compare the quality of the solution by measuring the path cost. In the case of tic-tac-toe we have a limited search space.

However, for many real-world problems the search space can grow very large, so we need algorithms to deal with that.

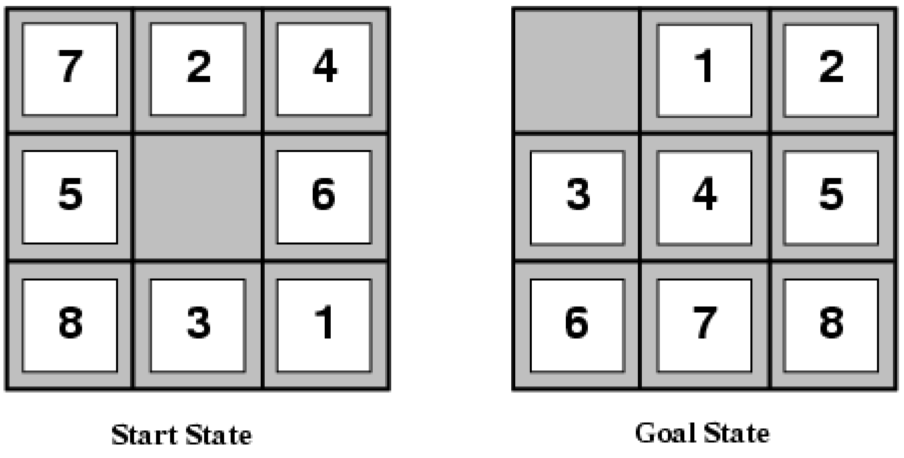
- it causes motion or traversal of the state space, and
- it does so in a controlled or systematic manner
- Random search may work in some problems, but in general, we need to search in an organized, methodical way. If we have a systematic search strategy that does not use information about the problem to help direct the search, it is called *brute-force, uninformed*, or *blind* search. The only difference between the different brute-force search techniques is the order in which nodes are expanded. But even slight changes in the order can have a significant impact on the behavior of the algorithm.
- Search algorithms which use *information about the problem*, such as the cost or distance to the goal state, are called *heuristic, informed*, or *directed* search. The primary advantage of heuristic search algorithms is that we can make better choices concerning which node to expand next. This substantially improves the efficiency of the search algorithms.

## 2. Characteristics of Search Algorithms :

- An algorithm is *optimal* if it will find the best solution from among several possible solutions.
- A strategy is *complete* if it guarantees that it will find a solution if one exists.
- The complexity of the algorithm, in terms of time complexity (how long it takes to find a solution)
- space complexity (how much memory it requires)

  **Example:** Having an optimal search algorithm that runs forever has little practical value. Nor does having a complete algorithm that is a memory hog, if it runs out of memory just before it finds the solution.

**Example : 8 puzzle game**



- states: locations of tiles
- Initial state: any state
- actions: move blank left, right, up, down
- goal test: goal state (given)
- path cost: 1 per move

## 3. Search Strategies

In this section we examine two basic search techniques used to solve problems in AI, ***breadth-first search*** and *depth-first search*. Later we will explore enhancements to these algorithms, but first we need to make sure we understand how these basic approaches work. We will work through several examples, but the steps of these two algorithms are :-

### ■ Breadth-First Search

The breadth-first search algorithm searches a state-space by constructing a hierarchical tree structure consisting of a set of nodes and links. The algorithm defines a way to move through the tree structure, examining the values at nodes in a controlled and systematic way, so that we can find a node which offers a solution to the problem we have represented using the tree-structure.

The algorithm follows:

1. Create a queue and add the first Node to it.
2. Loop :

- If the queue is empty, quit.
- Remove the first Node from the queue. If it is not a goal state
- If the Node contains the goal state, then exit with the Node as the solution.
- For each child of the current Node: Add the new state to **the back** of the queue.

## Depth-First Search

Depth-first search is another way to systematically traverse a tree structure to find a goal or solution node. Instead of completely searching each level of the tree before going deeper, the depth-first algorithm follows a single branch of the tree down as many levels as possible until we either reach a solution or a dead end. The algorithm follows:

1. Create a queue and add the first SearchNode to it.
2. Loop:
   - If the queue is empty, quit.
   - Remove the first SearchNode from the queue.
   - If the SearchNode contains the goal state, then exit with the SearchNode as the solution.
   - For each child of the current SearchNode: Add the new state to the front of the queue.