8-Puzzle Problem

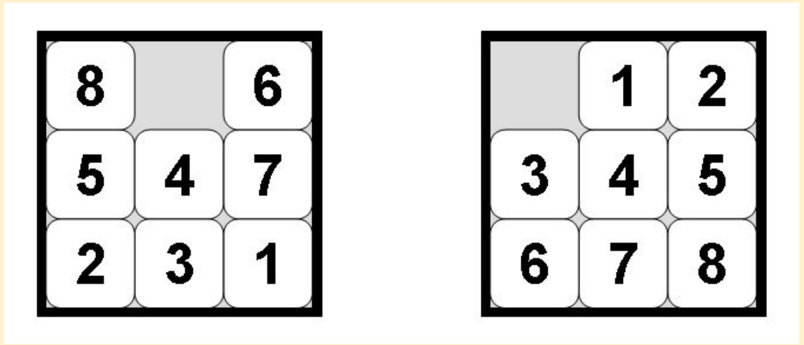
By: Jessenta Drake & Sabrina Johnson

What is the 8-Puzzle Problem?

- State: a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space.
- **Objective**: place the numbers on tiles to match the final configuration using the empty space.
- **Possible moves:** We can slide four adjacent (left, right, above, and below) tiles into the empty space.

In our implementation, the empty space corresponds to the tile labeled ‘0’. And the rules follow as described above. We are given an initial state configuration (this will typically be different from the goal/final configuration) and must find a set of moves to change that initial

Initial State: Goal State:



Formalizing the Problem/Key Words

- Consider a search tree generated from a specified
- **Stopping Conditions:** complete graph expansion for 15 layers or solution has been found

<u>**Key Words**</u>

State: The current configuration of the board

Node: An object holding the state, the parent node, the move applied to the parent to create said child, the depth of the node in a search tree

Solution path: The list of sequential actions used to transform the problems initial state to the goal state

Solution cost: The length of the solution path. Also referred to as the final path

Elapsed Time: The time it took for a function find a solution or terminate

Explored Nodes: The set of nodes evaluated during an algorithms traversal

UnInformed Searches

Depth-First Search (DFS)
Breadth-First Search (BFS)
Uniform Cost Search

UnInformed Searches

- Definition: These expand based on solely the problem definition. They have no knowledge of relations to the goal

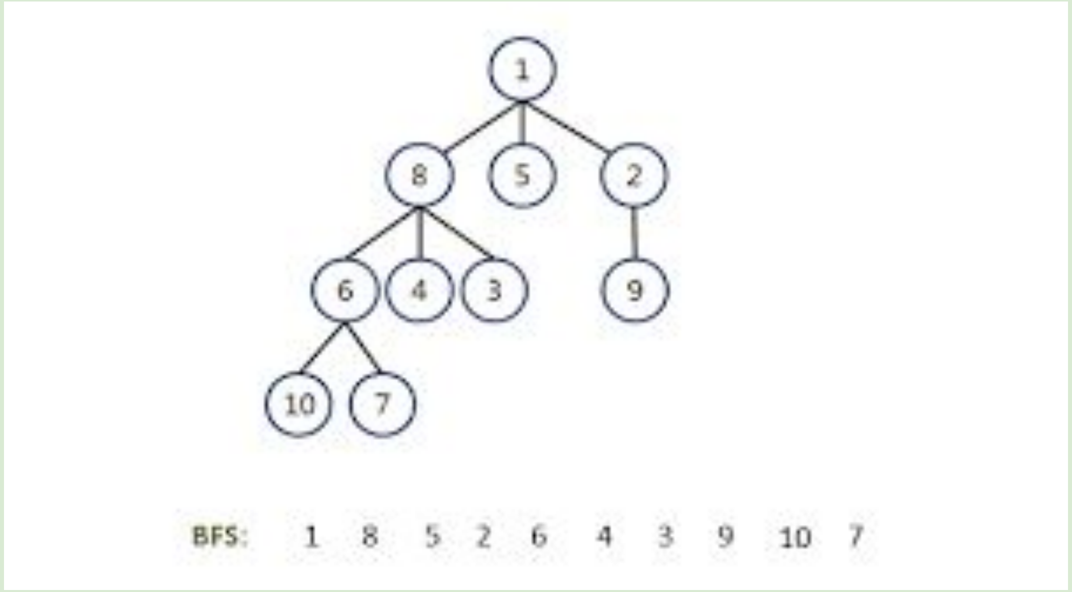
- Common examples:

  - Breadth First Search (BFS)

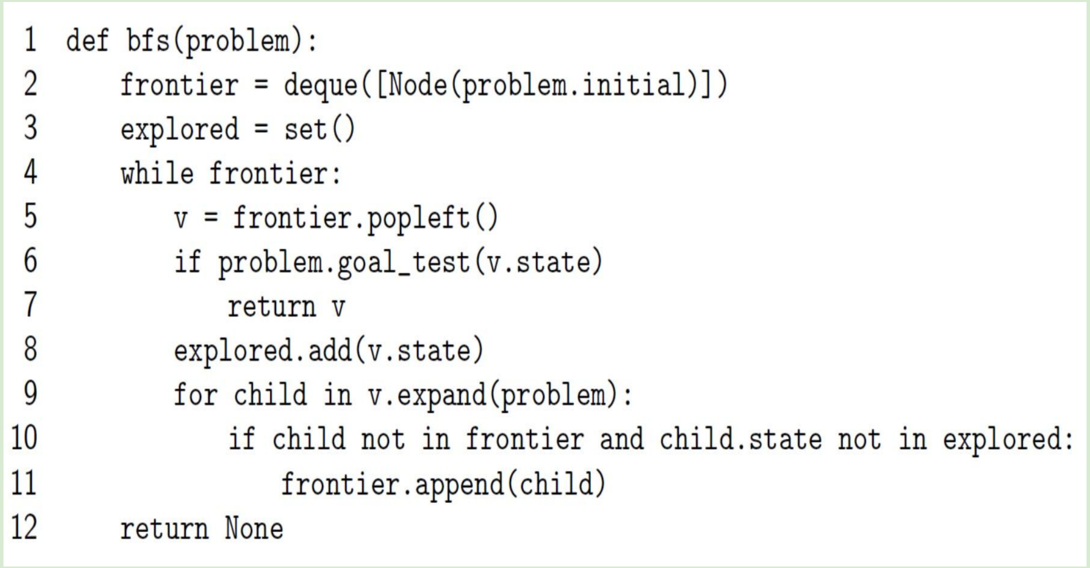
  - Depth First Search (DFS)

Breadth First Search (BFS) Walkthrough

- Implemented using a basic queue structure
- **Strategy:** We expand a nodes neighbors and add them to queue



BFS Pseudo Code

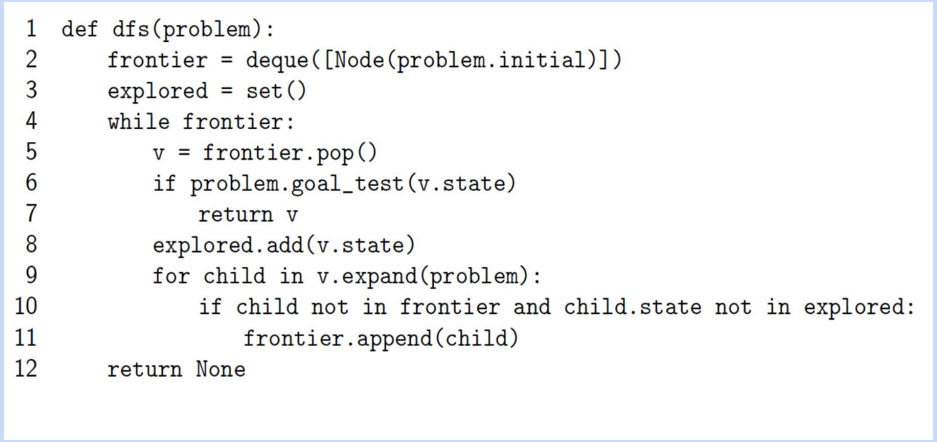


```
def bfs(problem):
    frontier = deque([Node(problem.initial)])
    explored = set()
    while frontier:
        v = frontier.popleft()
        if problem.goal_test(v.state)
            return v
        explored.add(v.state)
        for child in v.expand(problem):
            if child not in frontier and child.state not in explored:
                frontier.append(child)
    return None
```

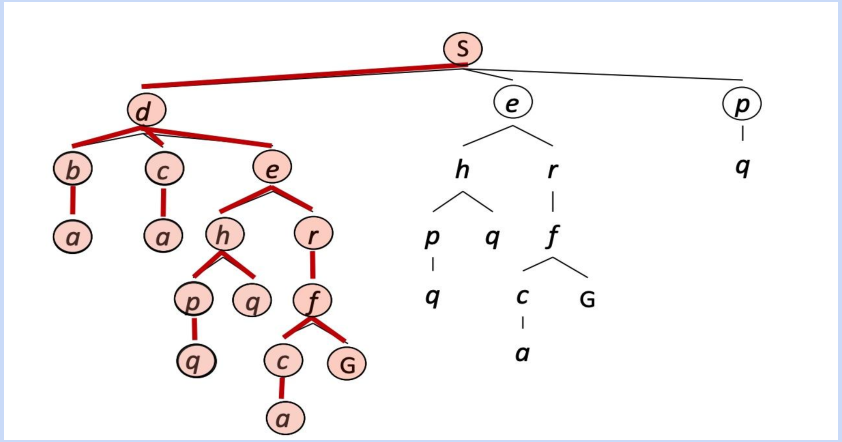
BFS Visuals

Depth First Search (DFS) Walkthrough

- **Strategy:** expand to the deepest node first
- Depth First Search uses Last In First Out implementation, because of this our main data structure will be a stack.
- Below pseudo code is include of how the implementation. Beside it is a visual representation of how DFS traverses through a graph.



```
def dfs(problem):
    frontier = deque([Node(problem.initial)])
    explored = set()
    while frontier:
        v = frontier.pop()
        if problem.goal_test(v.state)
            return v
        explored.add(v.state)
        for child in v.expand(problem):
            if child not in frontier and child.state not in explored:
                frontier.append(child)
    return None
```



DFS Visuals

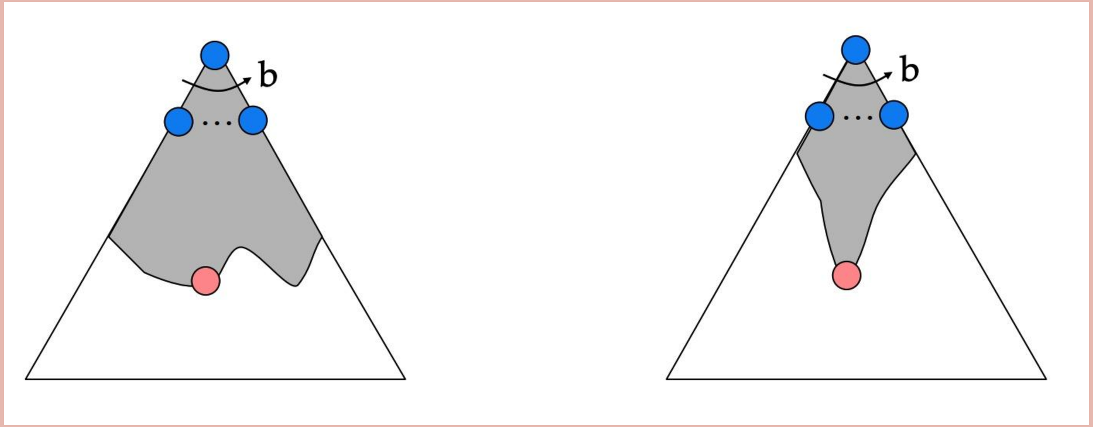
UnInformed Searches Comparison

- Depending on the problem BFS may be more efficient than DFS (or vice versa)

- Cases where BFS is more efficient:
  One possible solution path is short and close to the initial state

- Cases where DFS is more efficient:
  Most possible solution paths are longer and far from the initial state

Informed Searches



Informed Searches

- Definition: These searches expand based on the problem definition and a states relation to the goal.
- Heuristic value: **estimates the approximate cost of solving a task. Often referred to as h(n)**
- Common examples:

  - Greedy (Best First) Search

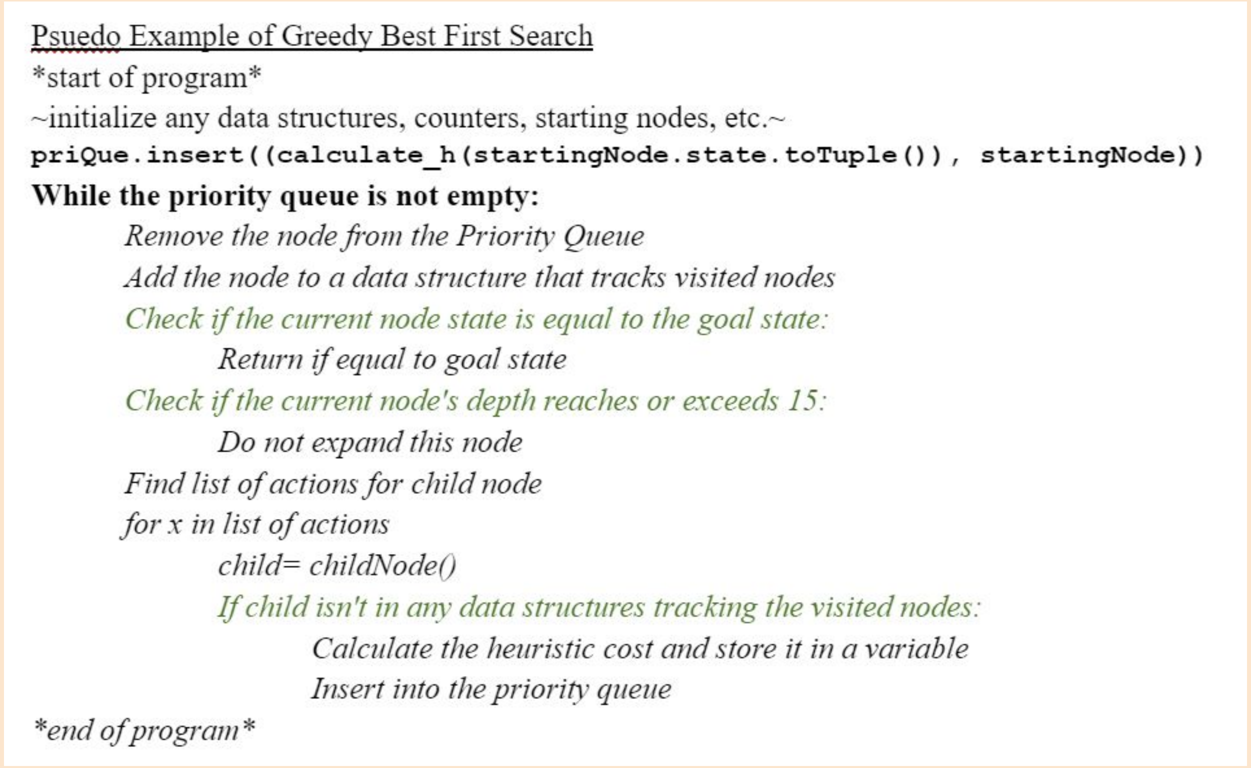
  - A* Search

Greedy Walkthrough

- Implemented using a priority queue

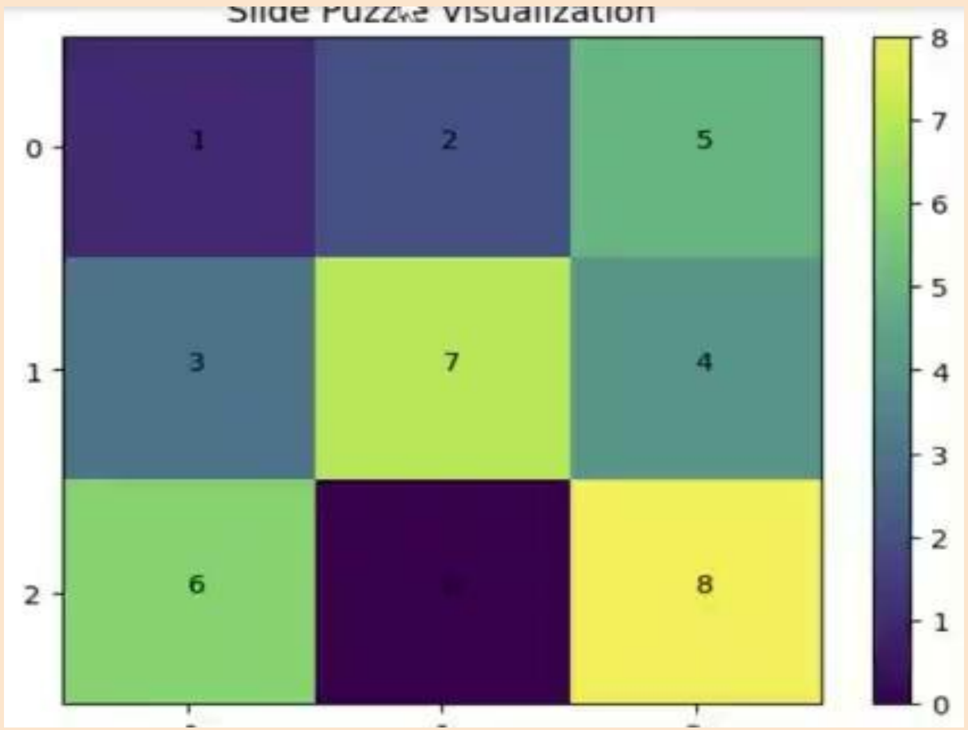
- **Strategy:** Expands based on what node has the lowest heuristic cost

Greedy Pseudo Code



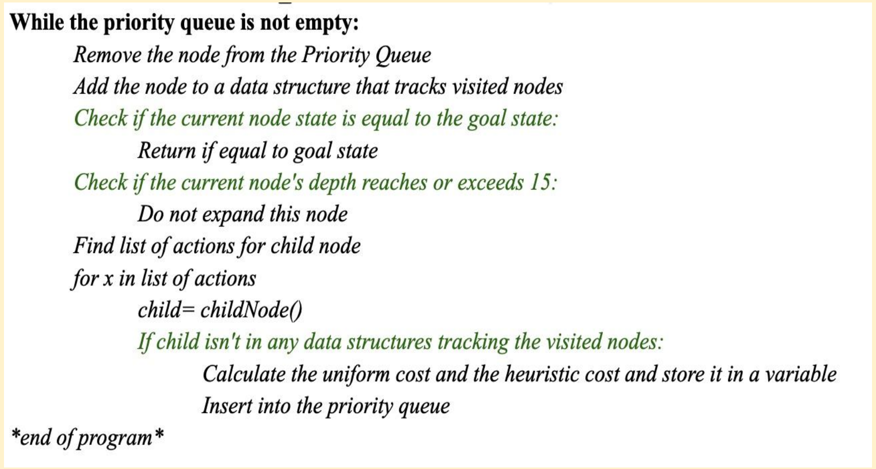
```
Psuedo Example of Greedy Best First Search
*start of program*
~initialize any data structures, counters, starting nodes, etc.~
priQue.insert((calculate_h(startingNode.state.toTuple()), startingNode))
While the priority queue is not empty:
    Remove the node from the Priority Queue
    Add the node to a data structure that tracks visited nodes
    Check if the current node state is equal to the goal state:
        Return if equal to goal state
    Check if the current node's depth reaches or exceeds 15:
        Do not expand this node
    Find list of actions for child node
    for x in list of actions
        child= childNode()
        If child isn't in any data structures tracking the visited nodes:
            Calculate the heuristic cost and store it in a variable
            Insert into the priority queue
*end of program*
```

Greedy Visuals

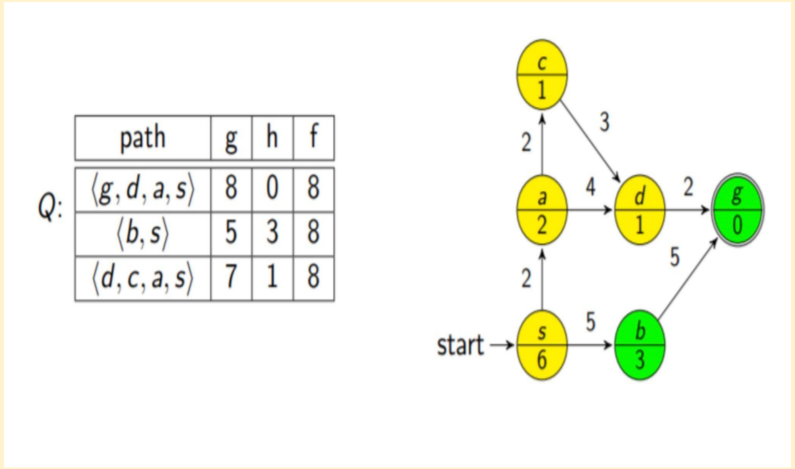


A* Walkthrough

- **Strategy:** calculate the heuristic cost while traversing from the starting node to the goal node.
- A* uses Priority Queue implementation to emphasize what path currently holds the least value of the path cost (total cost of the edges in a path) and the heuristic cost combined.
- Below pseudo code is include of how the implementation. Beside it is a visual representation of how A* traverses through a graph.



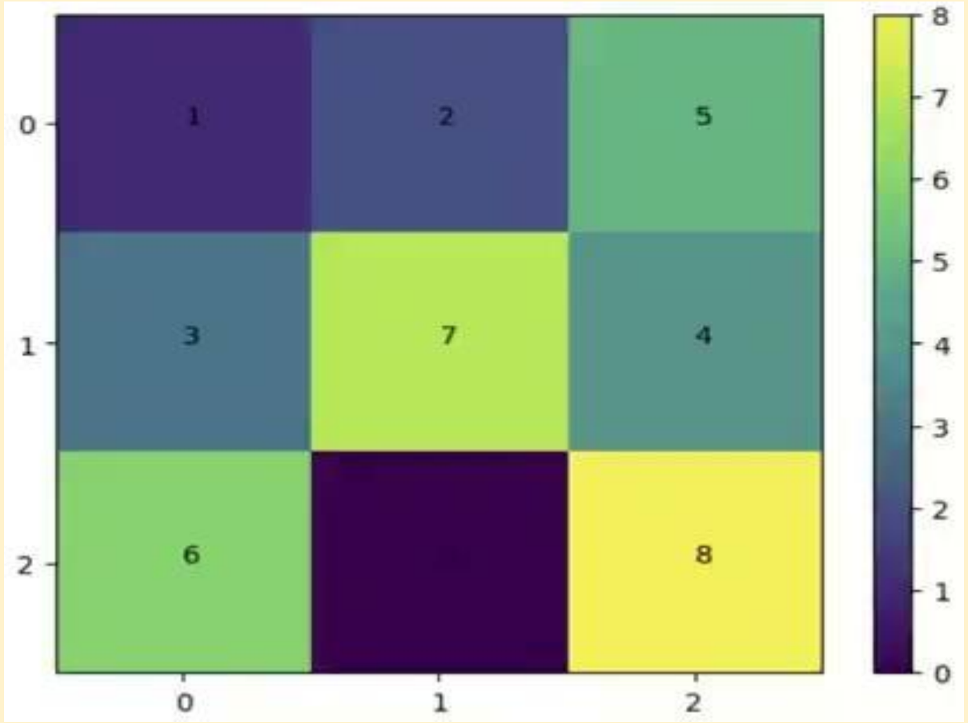
```
While the priority queue is not empty:
    Remove the node from the Priority Queue
    Add the node to a data structure that tracks visited nodes
    Check if the current node state is equal to the goal state:
        Return if equal to goal state
    Check if the current node's depth reaches or exceeds 15:
        Do not expand this node
    Find list of actions for child node
    for x in list of actions
        child= childNode()
        If child isn't in any data structures tracking the visited nodes:
            Calculate the uniform cost and the heuristic cost and store it in a variable
            Insert into the priority queue
*end of program*
```



Q:

path	g	h	f
$\langle g, d, a, s\rangle$	8	0	8
$\langle b, s\rangle$	5	3	8
$\langle d, c, a, s\rangle$	7	1	8

A* Visuals



Informed Search Comparisons

- A* is more reliable (it will always produce an optimal and complete)

- Greedy may be more efficient (lower space complexity)

- Often greedy runs more efficiently while producing the same path cost as A*. However, Greedy may not produce the optimal path for some problems.

Key Findings

- Informed is more efficient (lower explored node count, lower time elapsed)

- Informed (specifically A*) always produces an optimal path

- However , when we are not aware of how far away we are from the goal of the BFS and DFS search, BFS is most optimal.

Works Cited

https://www.geeksforgeeks.org/8-puzzle-problem-using-branch-and-bound/
Date accessed: 9/18/2023

Russell, Stuart J. (Stuart Jonathan), 1962-. Artificial Intelligence : a Modern Approach. Upper Saddle River, N.J. :Prentice Hall, 2010.