# PATAN MULTIPLE CAMPUS

## PATAN DHOKA, LALITPUR

## ARTIFICIAL INTELLIGENCE:

## PRACTICALS - I

**Submitted By:**
Name: Arun Dhungana
Roll: 13/078
B.Sc.CSIT 2nd Year 4th Sem
Symbol No:28023/078

**Submitted To:**
Bipin Timilsina
Department of CSIT
Patan Multiple Campus

# List Of Practicals

| Lab No. | Title/Question | Submission Date | Signature | Remarks |
|---|---|---|---|---|
| 1 | Program to simulate Simple Reflex Agent for detecting source of water leakage in a house. | 2080/9/19 | | |
| 2 | WAP to implement BFS for a graph. | 2080/9/19 | | |
| 3 | WAP to implement Uniform-Cost search. | 2080/9/19 | | |
| 4 | WAP to implement DFS for a graph. | 2080/9/19 | | |
| 5 | WAP to implement Depth Limited search for a graph. | 2080/9/19 | | |
| 6 | WAP to implement Greedy Best First search. | 2080/9/19 | | |
| 7 | WAP to implement A* search. | 2080/9/19 | | |
| 8 | WAP to implement Hill Climbing (Steepest Ascent) search. | 2080/9/19 | | |
| 9 | WAP to solve any one Cryptarithmetic Problem (like TWO+TWO = FOUR or SEND+MORE= MONEY). | 2080/9/19 | | |

## 1. Program to simulate Simple Reflex Agent for detecting source of water leakage in a house

### Theory:

The simplest kind of agent is the simple reflex agent.These agents select actionson the basis of the current percept, ignoring the rest of the percept history.
For example: a water leakage detecting agent : its decision is based only on theinformation of the present environment.

It is based on on condition–action rule (if-then rule / situation–action rule)

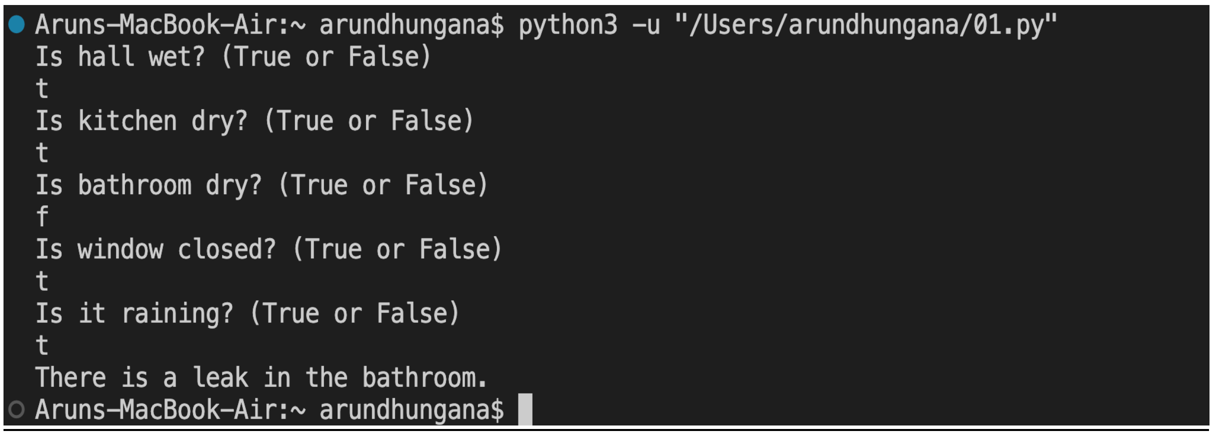
- If hall is wet and kitchen is dry then there is leak in bathroom
- If hall is wet and bathroom is dry then there is problem in kitchen
- If window is closed or it is not raining then it is confirmed that
- water is not from outside.
- If no water from outside and problem is in kitchen then leak is in
- kitchen.

Simple reflex agents have the admirable property of being simple, but they turnout to be of limited intelligence
They will work only if the correct decision can be made on the basis of only thecurrent percept—that is, only if the environment is fully observable

### Code:

```
def get_user_input(Q, i):
    response = input(f"{Q[i]}? (True or False)\n")
    if (response.lower().startswith('t')):
        return True
    else:
        return False
def agent():
    p, q, r, s, t, u, v, w, x = range(9)

    Q = [
        "Is hall wet",
        "Is kitchen dry",
        "Is bathroom dry",
        "Is window closed",
        "Is it raining" ]
    truth_values = {
            u: False, v: False,
            w: False, x: False }

    for i in range(len(Q)):
        truth_values[i] = (get_user_input(Q, i))

     if truth_values[p] and truth_values[q]:
        truth_values[u] = True
    if truth_values[p] and truth_values[r]:
        truth_values[v] = True
    if truth_values[s] or (not truth_values[t]):
        truth_values[w] = False
    if (not truth_values[w]) and truth_values[v]:
        truth_values[x] = True
    if truth_values[u] == True:
        print("There is leak in bathroom.")
    elif truth_values[x] == True:
        print("There is leak in kitchen.")
    elif truth_values[w] == True:
        print("Water is from outside.")
    else:
        print("The cause of leak is inconclusive.")
if __name__ == "__main__":
    agent()
```

Output:



```
● Aruns-MacBook-Air:~ arundhungana$ python3 -u "/Users/arundhungana/01.py"
  Is hall wet? (True or False)
  t
  Is kitchen dry? (True or False)
  t
  Is bathroom dry? (True or False)
  f
  Is window closed? (True or False)
  t
  Is it raining? (True or False)
  t
  There is a leak in the bathroom.
○ Aruns-MacBook-Air:~ arundhungana$
```

## 2. WAP to implement BFS for a graph.

Theory:

Breadth-first search (BFS) is a simple uninformed strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

The shallowest child node is expanded first. This is achieved very simply by using a FIFO queue for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first.

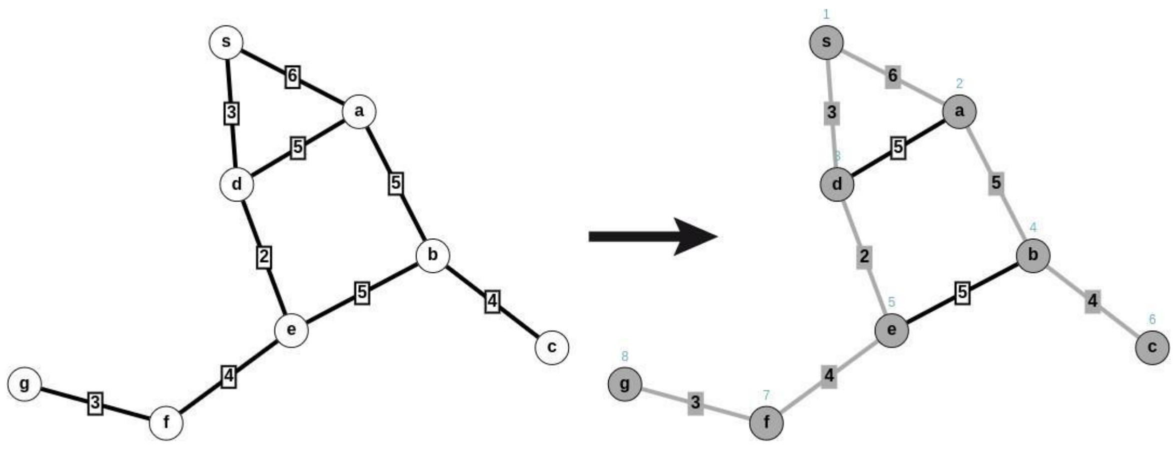


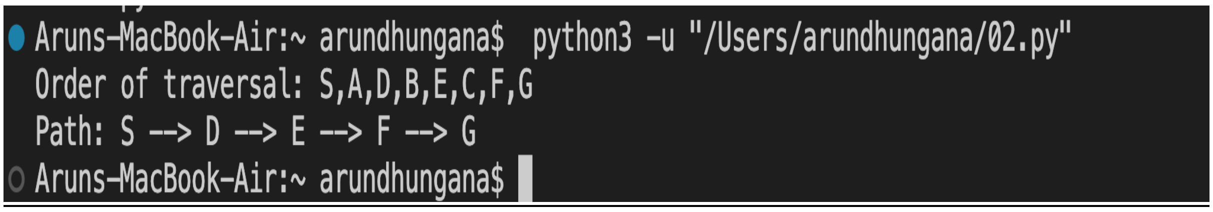
Fig: BFS on a Graph

Code:

```
from queue import Queue

graph = {
    "S": ("A", "D"),
    "A": ("B", "D", "S"),
    "B": ("A", "C", "E"),
    "C": ("B"),
    "D": ("A", "S", "E"),
    "E": ("D", "B", "F"),
    "F": ("E", "G"),
    "G": ("F"),
}
```

```
def bfs(source, destination):
    search_queue = Queue()
    search_queue.put([source])
    visited = []

    while search_queue:
        path = search_queue.get()
        node = path[-1]
        if not node in visited:
            visited.append(node)
            if node == destination:
                print(f"Order of traversal:
{', '.join(visited)}")
                print(f"Path: {' --> '.join(path)}")
                return True
            for child in graph[node]:
                search_queue.put(path + [child])
    return False


if___name___== "___main__":
    bfs("S", "G")
```

<u>Output:</u>



```
Aruns-MacBook-Air:~ arundhungana$  python3 -u "/Users/arundhungana/02.py"
Order of traversal: S,A,D,B,E,C,F,G
Path: S --> D --> E --> F --> G
Aruns-MacBook-Air:~ arundhungana$
```

## 3. WAP to implement Uniform-Cost search.

Theory:

It is an extension to the Bread First Search approach that is optimal for any step-cost function. It is also an uninformed search. Instead of expanding the shallowest node, uniform-cost search expands the node n with the lowest path cost g(n). This is done by storing the frontier as a priority queue ordered by g. The goal test is applied when a node is selected for expansion. It examines all the nodes at the goal's depth to see if one has a lower cost; thus uniform-cost search does strictly more work by expanding nodes at depth d unnecessarily.

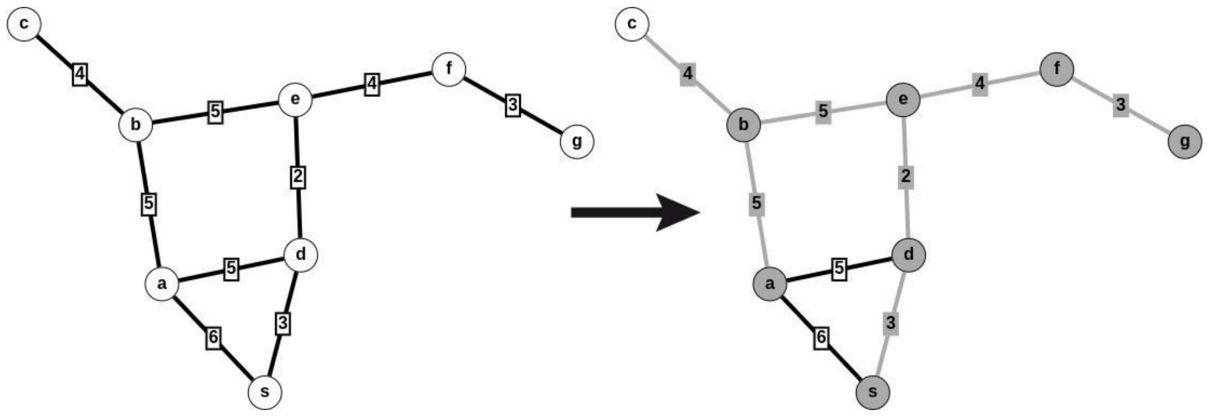


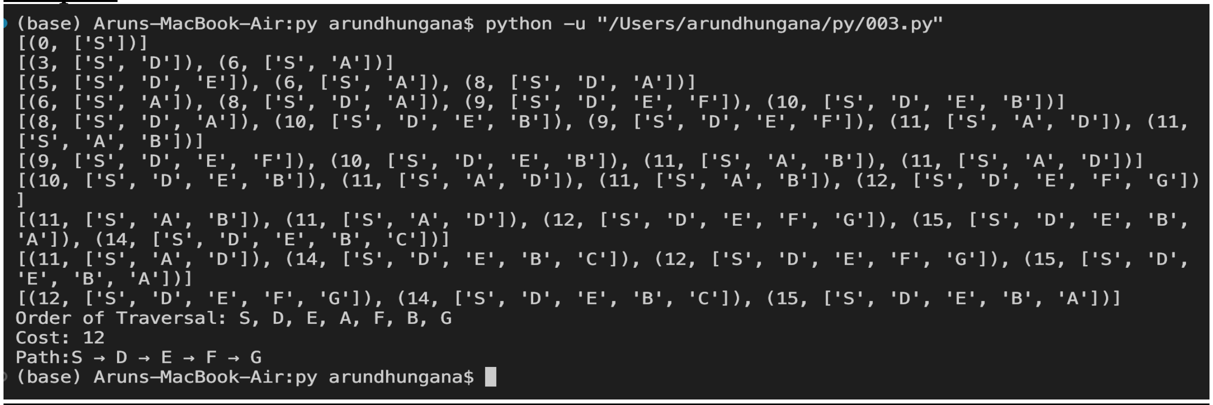
Fig: UCS on a Graph

Code:

```
from queue import PriorityQueue

graph = {
    "S": {(6, "A"), (3, "D")},
    "A": {(5, "B"), (5, "D"), (6, "S")},
    "B": {(5, "A"), (4, "C"), (5, "E")},
    "C": {(4, "B")},
    "D": {(5, "A"), (3, "S"), (2, "E")},
    "E": {(2, "D"), (5, "B"), (4, "F")},
    "F": {(4, "E"), (3, "G")},
    "G": {(3, "F")},
}
```

```
def show_traversed_nodes(visited_arr):
    order = "Order of Traversal: "
    order += ", ".join(visited_arr)
    print(order)
def uniform_cost(source, destination):
    search_queue = PriorityQueue()
    search_queue.put((0, [source]))
    visited = []
    while search_queue:
        print(search_queue.queue)
        total_cost, current_path = search_queue.get()
        current_node = current_path[-1] # last node in the
current_path
        if not current_node in visited:
            visited.append(current_node) if
            current_node == destination:
                show_traversed_nodes(visited)
                print(f"Cost: {total_cost}\nPath:
              {' → '.join(current_path)}")
                return True
            for cost, child in graph[current_node]:
                if not child in current_path:
                    search_queue.put((total_cost + cost,
current_path +
                [child]))
    return False

if___name___== "___main__":
    uniform_cost("S", "G")
```

Output:



```
(base) Aruns-MacBook-Air:py arundhungana$ python -u "/Users/arundhungana/py/003.py"
[(0, ['S'])]
[(3, ['S', 'D']), (6, ['S', 'A'])]
[(5, ['S', 'D', 'E']), (6, ['S', 'A']), (8, ['S', 'D', 'A'])]
[(6, ['S', 'A']), (8, ['S', 'D', 'A']), (9, ['S', 'D', 'E', 'F']), (10, ['S', 'D', 'E', 'B'])]
[(8, ['S', 'D', 'A']), (10, ['S', 'D', 'E', 'B']), (9, ['S', 'D', 'E', 'F']), (11, ['S', 'A', 'D']), (11,
['S', 'A', 'B'])]
[(9, ['S', 'D', 'E', 'F']), (10, ['S', 'D', 'E', 'B']), (11, ['S', 'A', 'B']), (11, ['S', 'A', 'D'])]
[(10, ['S', 'D', 'E', 'B']), (11, ['S', 'A', 'D']), (11, ['S', 'A', 'B']), (12, ['S', 'D', 'E', 'F', 'G'])
]
[(11, ['S', 'A', 'B']), (11, ['S', 'A', 'D']), (12, ['S', 'D', 'E', 'F', 'G']), (15, ['S', 'D', 'E', 'B',
'A']), (14, ['S', 'D', 'E', 'B', 'C'])]
[(11, ['S', 'A', 'D']), (14, ['S', 'D', 'E', 'B', 'C']), (12, ['S', 'D', 'E', 'F', 'G']), (15, ['S', 'D',
'E', 'B', 'A'])]
[(12, ['S', 'D', 'E', 'F', 'G']), (14, ['S', 'D', 'E', 'B', 'C']), (15, ['S', 'D', 'E', 'B', 'A'])]
Order of Traversal: S, D, E, A, F, B, G
Cost: 12
Path:S → D → E → F → G
(base) Aruns-MacBook-Air:py arundhungana$
```

## 4. WAP to implement DFS for a graph

Theory:

It always expands the deepest node in the current frontier of the search tree. The search proceeds immediately to the deepest level of the search tree, where the nodes have no successors. As those nodes are expanded, they are dropped from the frontier, so then the search "backs up" to the next deepest node that still has unexplored successors. Depth-first search uses a LIFO queue (i.e. a Stack). A LIFO queue means that the most recently generated node is chosen for expansion. It is an uninformed search strategy.

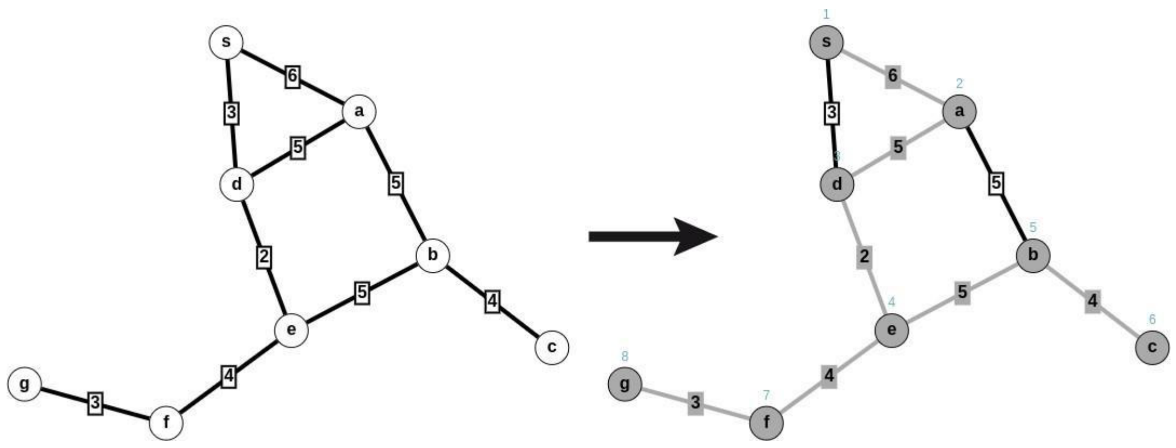
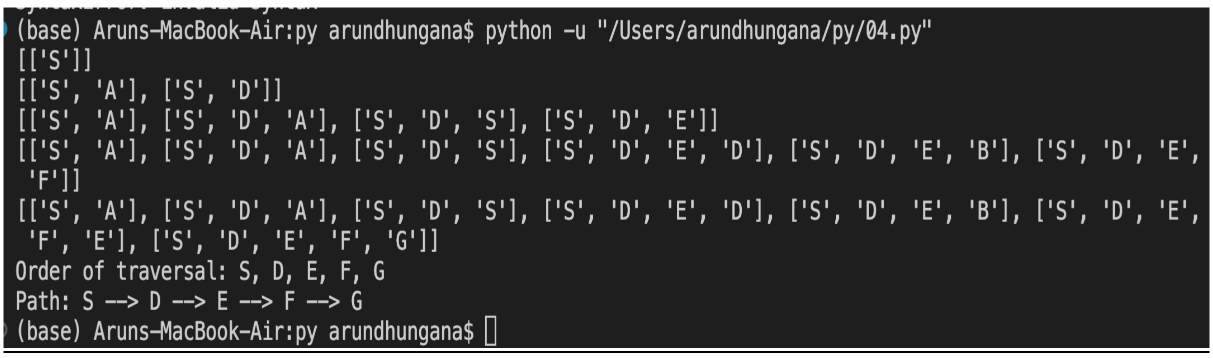


Fig: DFS on a Graph

Code:

```
from queue import LifoQueue

graph = {
    "S": ("A", "D"),
    "A": ("B", "D", "S"),
    "B": ("A", "C", "E"),
    "C": ("B"),
    "D": ("A", "S", "E"),
    "E": ("D", "B", "F"),
    "F": ("E", "G"),
    "G": ("F"),
}

def dfs(source, destination):
    search_queue = LifoQueue()
    search_queue.put([source])
    visited = []

    while search_queue:
        print(list(search_queue.queue))
        path = search_queue.get() node =
        path[-1]
        if not node in visited:
            visited.append(node)
            if node == destination:
                print(f"Order of traversal:
{', '.join(visited)}")
                print(f"Path: {' --> '.join(path)}")
                return True
            for child in graph[node]:
                search_queue.put(path + [child])
    return False


if___name___== "___main__":
    dfs("S", "G")
```

Output:



```
(base) Aruns-MacBook-Air:py arundhungana$ python -u "/Users/arundhungana/py/04.py"
[['S']]
[['S', 'A'], ['S', 'D']]
[['S', 'A'], ['S', 'D', 'A'], ['S', 'D', 'S'], ['S', 'D', 'E']]
[['S', 'A'], ['S', 'D', 'A'], ['S', 'D', 'S'], ['S', 'D', 'E', 'D'], ['S', 'D', 'E', 'B'], ['S', 'D', 'E',
 'F']]
[['S', 'A'], ['S', 'D', 'A'], ['S', 'D', 'S'], ['S', 'D', 'E', 'D'], ['S', 'D', 'E', 'B'], ['S', 'D', 'E',
 'F', 'E'], ['S', 'D', 'E', 'F', 'G']]
Order of traversal: S, D, E, F, G
Path: S --> D --> E --> F --> G
(base) Aruns-MacBook-Air:py arundhungana$ 
```

## 5. WAP to implement Depth Limited search for a graph.

Theory:

The failure of depth-first search in infinite state spaces can be alleviated by supplying depth-first search with a predetermined depth limit l. That is, nodes at depth l are treated as if they have no successors.(i.e. they are not expanded) This approach is called depth-limited search. The depth limit solves the infinite-path problem. Even if it could expand a vertex beyond some depth, it will not do so. That is why it cannot get stuck in cycles. It finds a solution only if it is within the depth limit.

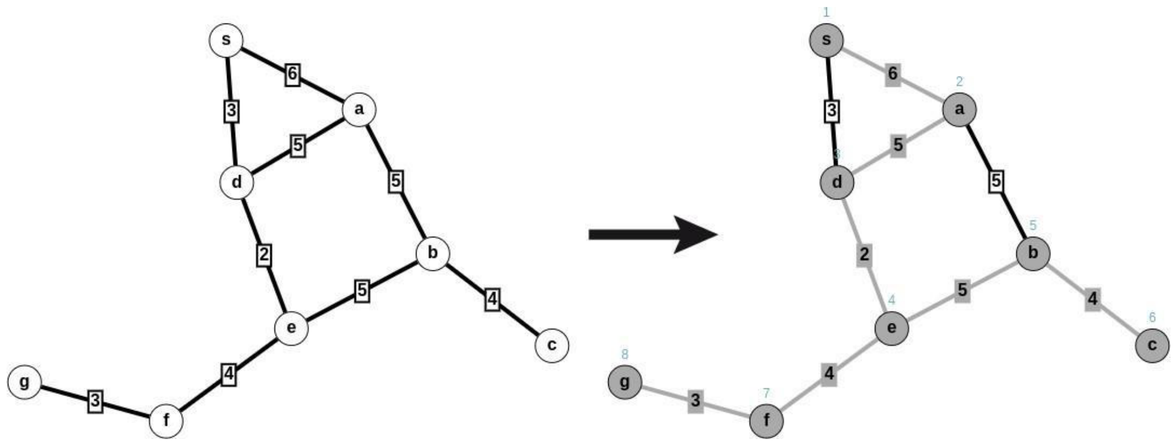
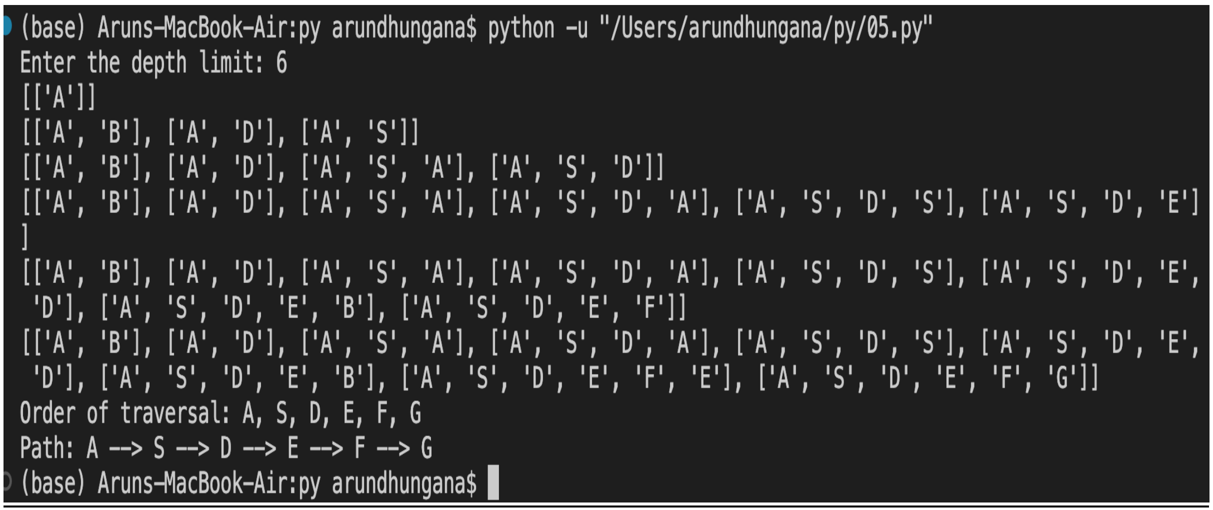


Fig: Limited DFS on a Graph

Code:

```
from queue import LifoQueue
graph = {
    "S": ("A", "D"),
    "A": ("B", "D", "S"),
    "B": ("A", "C", "E"),
    "C": ("B"),
    "D": ("A", "S", "E"),
    "E": ("D", "B", "F"),
    "F": ("E", "G"),
    "G": ("F"),
}
def dfs(source, destination, depth_limit):
    search_queue = LifoQueue()
    search_queue.put([source])
    visited = []
    depth = 0
    while search_queue and depth <= depth_limit:
        print(list(search_queue.queue))
        path = search_queue.get()
        node = path[-1]
        depth += 1
        if not node in visited:
            visited.append(node)
            if node == destination:
                print(f"Order of traversal:
{', '.join(visited)}")
                print(f"Path: {' --> '.join(path)}")
                return True
            for child in graph[node]:
                search_queue.put(path + [child])
    return False

if __name__ == "__main__":
    d_limit = input("Enter the depth limit: ")
    if(not dfs("A", "G", int(d_limit))):
        print(f"Destination not found with within depth
limit {d_limit}")
```

<u>Output:</u>



```
(base) Aruns-MacBook-Air:py arundhungana$ python -u "/Users/arundhungana/py/05.py"
Enter the depth limit: 6
[['A']]
[['A', 'B'], ['A', 'D'], ['A', 'S']]
[['A', 'B'], ['A', 'D'], ['A', 'S', 'A'], ['A', 'S', 'D']]
[['A', 'B'], ['A', 'D'], ['A', 'S', 'A'], ['A', 'S', 'D', 'A'], ['A', 'S', 'D', 'S'], ['A', 'S', 'D', 'E']
]
[['A', 'B'], ['A', 'D'], ['A', 'S', 'A'], ['A', 'S', 'D', 'A'], ['A', 'S', 'D', 'S'], ['A', 'S', 'D', 'E',
 'D'], ['A', 'S', 'D', 'E', 'B'], ['A', 'S', 'D', 'E', 'F']]
[['A', 'B'], ['A', 'D'], ['A', 'S', 'A'], ['A', 'S', 'D', 'A'], ['A', 'S', 'D', 'S'], ['A', 'S', 'D', 'E',
 'D'], ['A', 'S', 'D', 'E', 'B'], ['A', 'S', 'D', 'E', 'F', 'E'], ['A', 'S', 'D', 'E', 'F', 'G']]
Order of traversal: A, S, D, E, F, G
Path: A --> S --> D --> E --> F --> G
(base) Aruns-MacBook-Air:py arundhungana$
```

## 6. WAP to implement Greedy Best First search.

Theory:

Greedy Best-First Search is an informed search algorithm. That means, it utilizes additional information about the state space to find the path between source and destination nodes. It tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly.

Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.

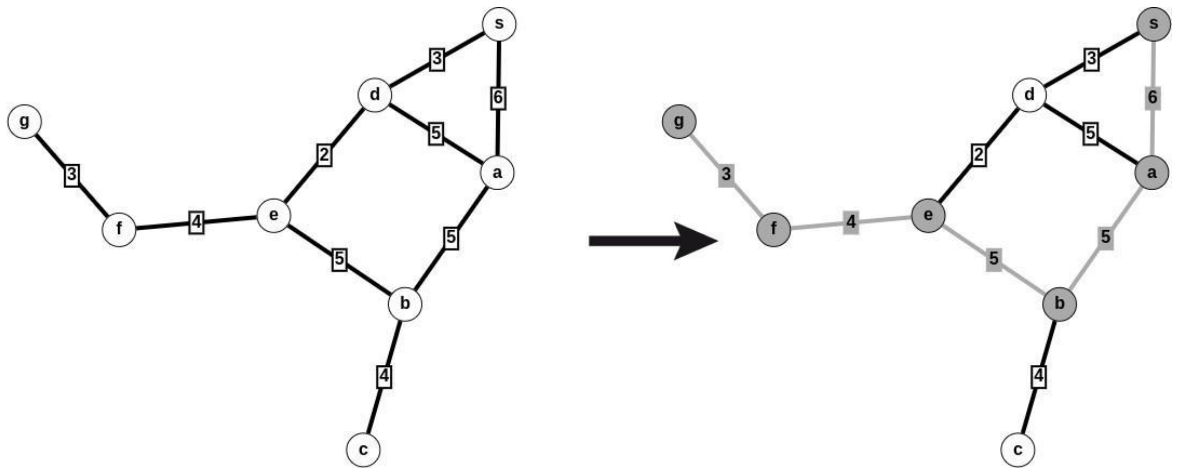


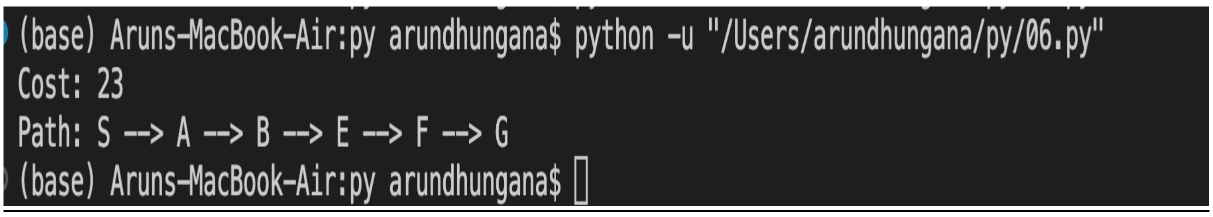
Fig: GBFS on a Graph

Code:

```
from queue import PriorityQueue

graph = {
    "S": {(6, "A"), (3, "D")},
    "A": {(5, "B"), (5, "D"), (6, "S")},
    "B": {(5, "A"), (4, "C"), (5, "E")},
    "C": {(4, "B")},
    "D": {(5, "A"), (3, "S"), (2, "E")},
    "E": {(2, "D"), (5, "B"), (4, "F")},
    "F": {(4, "E"), (3, "G")},
    "G": {(3, "F")},
}
heuristics = {
    "S": 12, "C": 5, "F": 2,
    "A": 8,  "D": 9, "G": 0
    "B": 7,  "E": 4, }
```

```
def add_children_of(node, q):
    for cost, child in graph[node]:
        heuristic = heuristics[child]
        q.put((heuristic, cost, child))

def gbfs(source, destination):
    search_queue = PriorityQueue()
    search_queue.put((heuristics[source], 0, source))
    add_children_of(source, search_queue)
    visited = []
    visited.append((0, source))
    total_cost = 0
    while search_queue:
        _, cost, node = search_queue.get()
        if not node in visited:
            visited.append((cost, node))
            if node == destination:
                path = ' --> '.join([node for _, node in
visited])
                for cost, _ in visited:
                    total_cost += cost
                print(f"Cost: {total_cost}\nPath: {path}")
                return True
            add_children_of(node, search_queue)
    return False

if___name___== "___main___":
    gbfs("S", "G")
```

<u>Output:</u>



```
(base) Aruns-MacBook-Air:py arundhungana$ python -u "/Users/arundhungana/py/06.py"
Cost: 23
Path: S --> A --> B --> E --> F --> G
(base) Aruns-MacBook-Air:py arundhungana$
```

## 7. WAP to implement A* search.

### Theory:

The most widely known form of best-first search is called A* search (pronounced “A-star search”). It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal:

*f(n) = g(n) + h(n)*

Since g(n) gives the path cost from the start node to node n, and h(n) is the estimated cost of the cheapest path from n to the goal, we have f(n) = estimated cost of the cheapest solution through n. The algorithm is identical to Uniform Cost Search except that A* uses *g* + *h* instead of *g*.

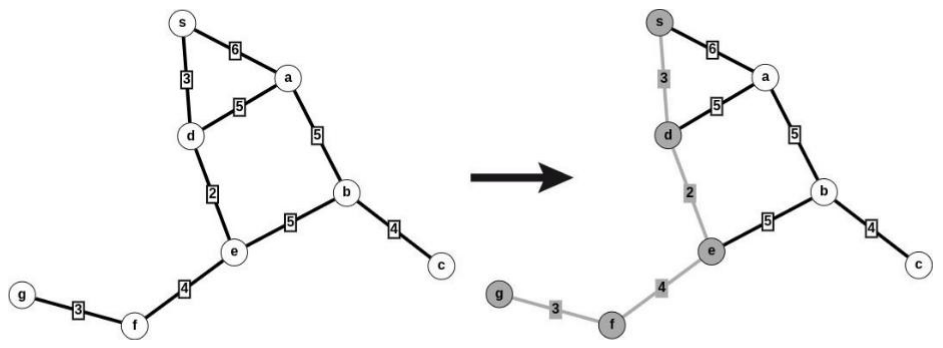


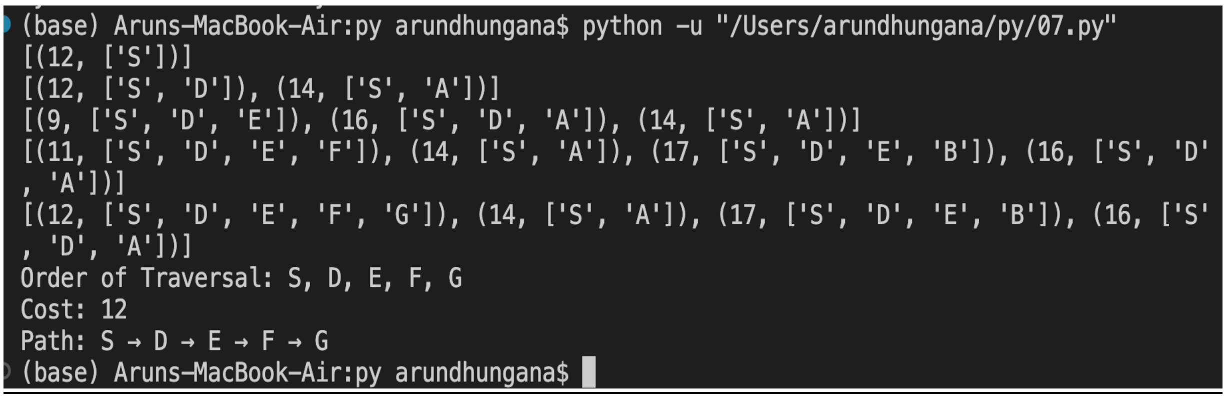
Fig: A* on a Graph

### Code:

```
from queue import PriorityQueuegraph

= {

     "S": {(6, "A"), (3, "D")},
     "A": {(5, "B"), (5, "D"), (6,  "S")},
     "B": {(5, "A"), (4, "C"), (5,  "E")},
     "C": {(4, "B")},
     "D": {(5, "A"), (3, "S"), (2,  "E")},
     "E": {(2, "D"), (5, "B"), (4,  "F")},
     "F": {(4, "E"), (3, "G")},
     "G": {(3, "F")},
}
heuristics = {
     "S": 12, “C”: 5, “F”: 2,
     "A":  8,   “D”:  9, “G”: 0
     "B":  7,   “E”:  4, }
```

```
def show_traversed_nodes(visited_arr):order
     = "Order of Traversal: " order += ",
     ".join(visited_arr) print(order)

def a_star(source, destination):
     search_queue = PriorityQueue()
     search_queue.put((heuristics[source] + 0, [source]))visited
     = []
     while search_queue:
          print(search_queue.queue)
          total_cost, current_path = search_queue.get()
          current_node = current_path[-1]
          if not current_node in visited:
               visited.append(current_node)
               if current_node == destination:
                    show_traversed_nodes(visited)
                    print(f"Cost: {total_cost}\nPath: {' →
                    '.join(current_path)}")
                    return True
               for cost, child in graph[current_node]:if
                    not child in current_path:
                         parent_path_cost = total_cost -
                          heuristics[current_node]
                         child_path_cost = parent_path_cost + cost
                         search_queue.put((heuristics[child] +
                          child_path_cost, current_path + [child]))
     return False

if __name__ == " __main__ ":
     a_star("S", "G")
```

Output:



```
(base) Aruns-MacBook-Air:py arundhungana$ python -u "/Users/arundhungana/py/07.py"
[(12, ['S'])]
[(12, ['S', 'D']), (14, ['S', 'A'])]
[(9, ['S', 'D', 'E']), (16, ['S', 'D', 'A']), (14, ['S', 'A'])]
[(11, ['S', 'D', 'E', 'F']), (14, ['S', 'A']), (17, ['S', 'D', 'E', 'B']), (16, ['S', 'D'
, 'A'])]
[(12, ['S', 'D', 'E', 'F', 'G']), (14, ['S', 'A']), (17, ['S', 'D', 'E', 'B']), (16, ['S'
, 'D', 'A'])]
Order of Traversal: S, D, E, F, G
Cost: 12
Path: S → D → E → F → G
(base) Aruns-MacBook-Air:py arundhungana$
```

## 8. WAP to implement Hill Climbing (Steepest Ascent) search.

### Theory:

It is a local search algorithm. It can be used to solve problems that have many solutions, some of which are better than others. It starts with a random (potentially poor) solution, and iteratively makes small changes to the solution, each time improving it a little. When the algorithm cannot see any improvement anymore, it terminates. Ideally, at that point the current solution is close to optimal, but it is not guaranteed that hill climbing will ever come close to the optimal solution.

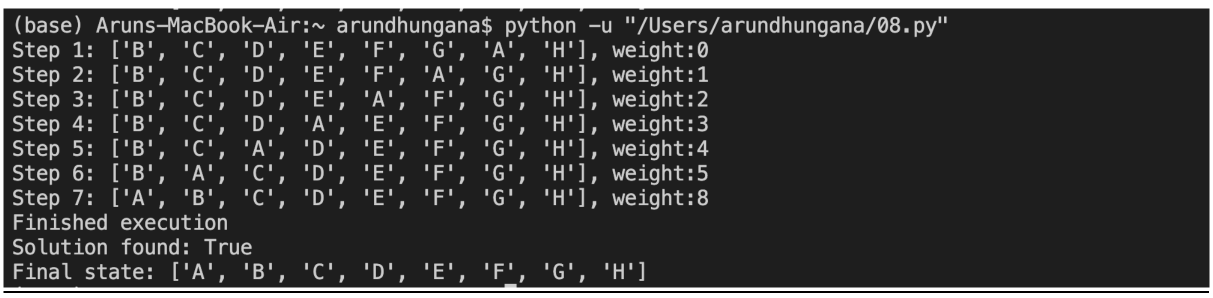
In hill climbing the basic idea is to always head towards a state which is better than the current one. So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move IF town B or C appear nearer to town D than town A does. The hill-climbing search algorithm (steepest-ascent version) is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a "peak" where no neighbor has a higher value.

### Code:

```
def perform_step(current, goal):
    successor = current
    successor_weight = -1
    for i in range(0, len(current) - 1): candidate
        = swapped(current, i, i + 1)
        candidate_weight = calculate_weight(candidate,
        goal)if candidate_weight > successor_weight:
            successor = candidate successor_weight
            = candidate_weight
    return successor, successor_weight

def calculate_weight(state, goal):
    weight = 0
    for i in range(len(state)):
        block = state[i]
        goal_index = goal.index(block)
        assert goal_index >= 0
        should_be_above = None if goal_index == 0
else goal[goal_index - 1]


        is_above_correct = i >= 0 if should_be_above is
Noneelse state.index(should_be_above) <= i
        is_incorrect_position = i !=
        goal_indexif is_above_correct:
            weight += 1
        if is_incorrect_position:
            weight -= 1
    return weight
```

```
def swapped(state, a, b):
    new_list = []
    for i in range(len(state)):if
        i == a:
            new_list.insert(i, state[b])
        elif i == b:
            new_list.insert(i, state[a])
        else:
            new_list.insert(i, state[i])
    return new_list

if___name___== '___main___':
    initial_state = ['B', 'C', 'D', 'E', 'F', 'G', 'H',
    'A'] goal_state = ['A', 'B', 'C', 'D', 'E', 'F', 'G',
    'H'] current_state = initial_state
    current_step = 1
    while current_state != goal_state:
        next_state, step_weight =
perform_step(current_state, goal_state)
        if next_state == current_state:
            print("Infinite loop detected (might be local
maximum or plateau)")
            break
        current_state = next_state
        print(f"Step {current_step}: {current_state}, weight:
{step_weight}")
        current_step += 1
    print("Finished execution")
    print(f"Solution found: {current_state == goal_state}")
    print(f"Final state: {current_state}")
```

Output:



```
(base) Aruns-MacBook-Air:~ arundhungana$ python -u "/Users/arundhungana/08.py"
Step 1: ['B', 'C', 'D', 'E', 'F', 'G', 'A', 'H'], weight:0
Step 2: ['B', 'C', 'D', 'E', 'F', 'A', 'G', 'H'], weight:1
Step 3: ['B', 'C', 'D', 'E', 'A', 'F', 'G', 'H'], weight:2
Step 4: ['B', 'C', 'D', 'A', 'E', 'F', 'G', 'H'], weight:3
Step 5: ['B', 'C', 'A', 'D', 'E', 'F', 'G', 'H'], weight:4
Step 6: ['B', 'A', 'C', 'D', 'E', 'F', 'G', 'H'], weight:5
Step 7: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H'], weight:8
Finished execution
Solution found: True
Final state: ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

## 9.WAP to solve any one Cryptarithmetic Problem (like TWO+TWO = FOUR or SEND+MORE= MONEY ).

Theory:

Many problems in AI can be considered as problems of constraint satisfaction, in which the goal state satisfies a given set of constraints. An example of such a problem is the Crypt-Arithmetic Problem. It is a class of mathematical puzzles in which each letter/ alphabet is replaced by a unique digit.

Cryptarithmetic puzzles involve an arbitrary number of variables. In such problems, all the variables involved must have different values.

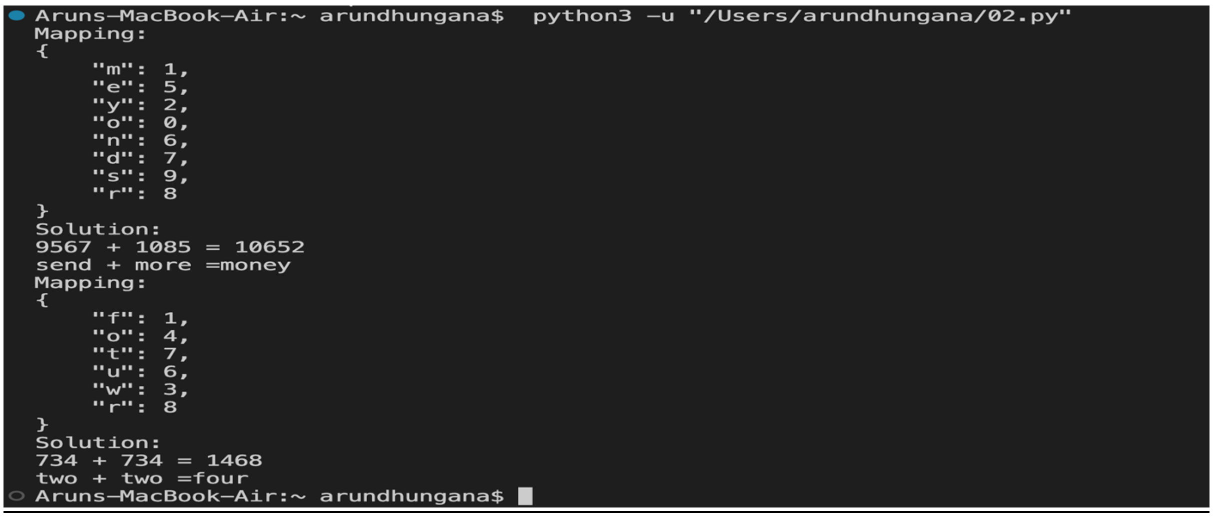
The constraints the must be satisfied are as follows:

- Values are to be assigned to letters from 0 to 9 only/
- No two letters should have the same value.
- Each letter/symbol represents only one digit throughout the problem.
- Numbers must not begin with zero i.e. 0567 (wrong), 567 (correct).
- After replacing letters by their digits, the resulting arithmetic operations must be correct.

Code:

```
from itertools import permutations
from json import dumps

def get_value(word, substitution):
    s = 0
    factor = 1
    for letter in reversed(word):
        s += factor *
        substitution[letter] factor *= 10
    return s

def solve(equation):
    left, right = equation.lower().replace(' ', '').split('=')
    left = left.split('+')
    letters = set(right)
    for word in left:
        for letter in word:
            letters.add(letter)
    letters = list(letters)
    first_letters = set(word[0] for word in left)
    first_letters.add(right[0])

    digits = range(10)
    for perm in permutations(digits,
        len(letters)): sol = dict(zip(letters,
        perm)) first_letter_is_zero = False
        for letter in first_letters:
            if sol[letter] == 0:
                first_letter_is_zero = True
                break
        if first_letter_is_zero:
            continue
        if sum(get_value(word, sol) for word in left)
== get_value(right, sol):
            print(f"Mapping:\n{dumps(sol, indent=4)}")
            print(f"Solution:\n{' + '.join(str(get_value(word,
sol)) for word in left)} = {get_value(right, sol)}")
            print(f"{' + '.join(word for word in left)} =
{right}")
            break

if__name__ == '__main__': solve('SEND
    + MORE = MONEY') solve('TWO +
    TWO = FOUR')
```

Output:



```
Aruns-MacBook-Air:~ arundhungana$  python3 -u "/Users/arundhungana/02.py"
Mapping:
{
    "m": 1,
    "e": 5,
    "y": 2,
    "o": 0,
    "n": 6,
    "d": 7,
    "s": 9,
    "r": 8
}
Solution:
9567 + 1085 = 10652
send + more =money
Mapping:
{
    "f": 1,
    "o": 4,
    "t": 7,
    "u": 6,
    "w": 3,
    "r": 8
}
Solution:
734 + 734 = 1468
two + two =four
Aruns-MacBook-Air:~ arundhungana$ 
```