# Table of Contents

# 01. A* Search

## Source Code:

```
from queue import heappop, heappush
from math import inf

class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.huristics = {}
        self.directed = directed

    def add_edge(self, node1, node2, cost = 1, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = {}
        neighbors[node2] = cost
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, cost, True)

    def set_huristics(self, huristics={}):
        self.huristics = huristics

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def cost(self, node1, node2):
        try: return self.edges[node1][node2]
        except: return inf


    def a_star_search(self, start, goal):
        found, fringe, visited, came_from, cost_so_far = False, [(self.huristics[start], start)],
set([start]), {start: None}, {start: 0}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('---------------------')
        print('{:11s} | {}'.format('-', str(fringe[0])))
        while not found and len(fringe):
            _, current = heappop(fringe)
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            for node in self.neighbors(current):
                new_cost = cost_so_far[current] + self.cost(current, node)
                if node not in visited or cost_so_far[node] > new_cost:
                    visited.add(node); came_from[node] = current; cost_so_far[node] = new_cost
                    heappush(fringe, (new_cost + self.huristics[node], node))
            print(', '.join([str(n) for n in fringe]))
        if found: print(); return came_from, cost_so_far[goal]
        else: print('No path from {} to {}'.format(start, goal)); return None, inf

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')


    def __str__(self):
        return str(self.edges)
```

```python
graph = Graph(directed=True)
graph.add_edge('A', 'B', 4)
graph.add_edge('A', 'C', 1)
graph.add_edge('B', 'D', 3)
graph.add_edge('B', 'E', 8)
graph.add_edge('C', 'C', 0)
graph.add_edge('C', 'D', 7)
graph.add_edge('C', 'F', 6)
graph.add_edge('D', 'C', 2)
graph.add_edge('D', 'E', 4)
graph.add_edge('E', 'G', 2)
graph.add_edge('F', 'G', 8)
graph.set_huristics({'A': 8, 'B': 8, 'C': 6, 'D': 5, 'E': 1, 'F': 4, 'G': 0})
start, goal = 'A', 'G'
traced_path, cost = graph.a_star_search(start, goal)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal); print('\nCost:', cost)
```

## Output:

```
Expand Node | Fringe

--------------------

-           | (8, 'A')

A           | (7, 'C'), (12, 'B')

C           | (11, 'F'), (13, 'D'), (12, 'B')

F           | (12, 'B'), (13, 'D'), (15, 'G')

B           | (12, 'D'), (13, 'E'), (13, 'D'), (15, 'G')

D           | (12, 'E'), (13, 'D'), (15, 'G'), (13, 'E')

E           | (13, 'D'), (13, 'E'), (15, 'G'), (13, 'G')

D           | (13, 'E'), (13, 'G'), (15, 'G')

E           | (13, 'G'), (15, 'G')

G           |

Path: A => B => D => E => G

Cost: 13
```

# 02. Bi-directional Search

## Source Code:

```
from collections import deque

class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed

    def add_edge(self, node1, node2, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = set()
        neighbors.add(node2)
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, True)

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def bi_directional_search(self, start, goal):
        found, fringe1, visited1, came_from1 = False, deque([start]), set([start]), {start: None}
        meet, fringe2, visited2, came_from2 = None, deque([goal]), set([goal]), {goal: None}
        while not found and (len(fringe1) or len(fringe2)):
            print('FringeStart: {:30s} | FringeGoal: {}'.format(str(fringe1), str(fringe2)))
            if len(fringe1):
                current1 = fringe1.pop()
                if current1 in visited2: meet = current1; found = True; break
                for node in self.neighbors(current1):
                    if node not in visited1: visited1.add(node); fringe1.appendleft(node);
came_from1[node] = current1
            if len(fringe2):
                current2 = fringe2.pop()
                if current2 in visited1: meet = current2; found = True; break
                for node in self.neighbors(current2):
                    if node not in visited2: visited2.add(node); fringe2.appendleft(node);
came_from2[node] = current2
        if found: print(); return came_from1, came_from2, meet
        else: print('No path between {} and {}'.format(start, goal)); return None, None, None

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)


graph = Graph(directed=False)
graph.add_edge('A', 'B'); graph.add_edge('A', 'S'); graph.add_edge('S', 'G')
graph.add_edge('S', 'C'); graph.add_edge('C', 'F'); graph.add_edge('G', 'F')
```

```python
graph.add_edge('C', 'D'); graph.add_edge('C', 'E'); graph.add_edge('E', 'H')
graph.add_edge('G', 'H')
start, goal = 'A', 'H'
traced_path1, traced_path2, meet = graph.bi_directional_search(start, goal)
if meet:
    print('Meeting Node:', meet)
    print('Path From Start:', end=' '); Graph.print_path(traced_path1, meet); print()
    print('Path From Goal:', end=' '); Graph.print_path(traced_path2, meet); print()
```

## Output:

```
FringeStart: deque(['A'])                | FringeGoal: deque(['H'])

FringeStart: deque(['S', 'B'])           | FringeGoal: deque(['G', 'E'])

FringeStart: deque(['S'])                | FringeGoal: deque(['C', 'G'])


Meeting Node: G

Path From Start: A => S => G

Path From Goal: H => G
```

# 03. Breadth First Search

## Source Code:

```
from collections import deque
class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed

    def add_edge(self, node1, node2, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = set()
        neighbors.add(node2)
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, True)

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def breadth_first_search(self, start, goal):
        found, fringe, visited, came_from = False, deque([start]), set([start]), {start: None}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', start))
        while not found and len(fringe):
            current = fringe.pop()
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            for node in self.neighbors(current):
                if node not in visited: visited.add(node); fringe.appendleft(node); came_from[node]
= current
            print(', '.join(fringe))
        if found: print(); return came_from
        else: print('No path from {} to {}'.format(start, goal))

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)

graph = Graph(directed=False)
graph.add_edge('A', 'B')
graph.add_edge('A', 'S')
graph.add_edge('S', 'G')
graph.add_edge('S', 'C')
graph.add_edge('C', 'F')
graph.add_edge('G', 'F')
graph.add_edge('C', 'D')
graph.add_edge('C', 'E')
graph.add_edge('E', 'H')
graph.add_edge('G', 'H')
start, goal = 'A', 'H'
traced_path = graph.breadth_first_search(start, goal)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal);print()
```

**Output:**

```
Expand Node | Fringe
--------------------
-           | A
A           | S, B
B           | S
S           | G, C
C           | D, F, E, G
G           | H, D, F, E
E           | H, D, F
F           | H, D
D           | H
H           |
Path: A => S => G => H
```

# 04. Depth First Search

## Source Code:

```
from collections import deque

class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed

    def add_edge(self, node1, node2, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = set()
        neighbors.add(node2)
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, True)

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def breadth_first_search(self, start, goal):
        found, fringe, visited, came_from = False, deque([start]), set([start]), {start: None}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', start))
        while not found and len(fringe):
            current = fringe.pop()
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            for node in self.neighbors(current):
                if node not in visited: visited.add(node); fringe.append(node); came_from[node] =
current
            print(', '.join(fringe))
        if found: print(); return came_from
        else: print('No path from {} to {}'.format(start, goal))

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')


    def __str__(self):
        return str(self.edges)


graph = Graph(directed=False)
graph.add_edge('A', 'B')
graph.add_edge('A', 'S')
graph.add_edge('S', 'G')
graph.add_edge('S', 'C')
graph.add_edge('C', 'F')
graph.add_edge('G', 'F')
graph.add_edge('C', 'D')
graph.add_edge('C', 'E')
graph.add_edge('E', 'H')
graph.add_edge('G', 'H')
start, goal = 'A', 'H'
```

```
traced_path = graph.breadth_first_search(start, goal)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal);print()
```

## Output:

```
Expand Node | Fringe
--------------------
-           | A
A           | B, S
S           | B, G, C
C           | B, G, F, E, D
D           | B, G, F, E
E           | B, G, F, H
H           |
Path: A => S => C => E => H
```

# 05. Depth Limited Search

## Source Code:

```python
from collections import deque
class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed

    def add_edge(self, node1, node2, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = set()
        neighbors.add(node2)
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, True)

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def depth_limited_search(self, start, goal, limit=-1):
        print('Depth limit =', limit)
        found, fringe, visited, came_from = False, deque([(0, start)]), set([start]), {start: None}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', start))
        while not found and len(fringe):
            depth, current = fringe.pop()
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            if limit == -1 or depth < limit:
                for node in self.neighbors(current):
                    if node not in visited:
                        visited.add(node); fringe.append((depth + 1, node))
                        came_from[node] = current
            print(', '.join([n for _, n in fringe]))
        if found: print(); return came_from
        else: print('No path from {} to {}'.format(start, goal))

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)

graph = Graph(directed=False)
graph.add_edge('A', 'B')
graph.add_edge('A', 'S')
graph.add_edge('S', 'G')
graph.add_edge('S', 'C')
graph.add_edge('C', 'F')
graph.add_edge('G', 'F')
graph.add_edge('C', 'D')
graph.add_edge('C', 'E')
graph.add_edge('E', 'H')
graph.add_edge('G', 'H')
start, goal, l = 'A', 'H', 3
```

```python
traced_path = graph.depth_limited_search(start, goal, l)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal);print()
```

## Output:

```
Depth limit = 3

Expand Node | Fringe

--------------------

-           | A

A           | S, B

B           | S

S           | C, G

G           | C, H, F

F           | C, H

H           |

Path: A => S => G => H
```

# 06. Best First Search

## Source Code:

```
from queue import heappop, heappush
from math import inf

class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.huristics = {}
        self.directed = directed

    def add_edge(self, node1, node2, cost = 1, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = {}
        neighbors[node2] = cost
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, cost, True)

    def set_huristics(self, huristics={}):
        self.huristics = huristics

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def cost(self, node1, node2):
        try: return self.edges[node1][node2]
        except: return inf


    def best_first_search(self, start, goal):
        found, fringe, visited, came_from, cost_so_far = False, [(self.huristics[start], start)],
set([start]), {start: None}, {start: 0}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', str(fringe[0])))
        while not found and len(fringe):
            _, current = heappop(fringe)
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            for node in self.neighbors(current):
                new_cost = cost_so_far[current] + self.cost(current, node)
                if node not in visited or cost_so_far[node] > new_cost:
                    visited.add(node); came_from[node] = current; cost_so_far[node] = new_cost
                    heappush(fringe, (new_cost + self.huristics[node], node))
            print(', '.join([str(n) for n in fringe]))
        if found: print(); return came_from, cost_so_far[goal]
        else: print('No path from {} to {}'.format(start, goal)); return None, inf

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')


    def __str__(self):
        return str(self.edges)
```

```python
graph = Graph(directed=True)
graph.add_edge('A', 'B', 4)
graph.add_edge('A', 'C', 1)
graph.add_edge('B', 'D', 3)
graph.add_edge('B', 'E', 8)
graph.add_edge('C', 'C', 0)
graph.add_edge('C', 'D', 7)
graph.add_edge('C', 'F', 6)
graph.add_edge('D', 'C', 2)
graph.add_edge('D', 'E', 4)
graph.add_edge('E', 'G', 2)
graph.add_edge('F', 'G', 8)
graph.set_huristics({'A': 8, 'B': 8, 'C': 6, 'D': 5, 'E': 1, 'F': 4, 'G': 0})
start, goal = 'A', 'G'
traced_path, cost = graph.best_first_search(start, goal)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal); print('\nCost:',
cost)
```

## Output:

```
Expand Node | Fringe

--------------------

-           | (8, 'A')

A           | (7, 'C'), (12, 'B')

C           | (11, 'F'), (13, 'D'), (12, 'B')

F           | (12, 'B'), (13, 'D'), (15, 'G')

B           | (12, 'D'), (13, 'E'), (13, 'D'), (15, 'G')

D           | (12, 'E'), (13, 'D'), (15, 'G'), (13, 'E')

E           | (13, 'D'), (13, 'E'), (15, 'G'), (13, 'G')

D           | (13, 'E'), (13, 'G'), (15, 'G')

E           | (13, 'G'), (15, 'G')

G           |

Path: A => B => D => E => G

Cost: 13
```

# 07. Greedy Search

## Source Code:

```
from queue import heappop, heappush
from math import inf

class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.huristics = {}
        self.directed = directed

    def add_edge(self, node1, node2, cost = 1, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = {}
        neighbors[node2] = cost
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, cost, True)

    def set_huristics(self, huristics={}):
        self.huristics = huristics

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def cost(self, node1, node2):
        try: return self.edges[node1][node2]
        except: return inf


    def greedy_search(self, start, goal):
        found, fringe, visited, came_from, cost_so_far = False, [(self.huristics[start], start)],
set([start]), {start: None}, {start: 0}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', str(fringe[0])))
        while not found and len(fringe):
            _, current = heappop(fringe)
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            for node in self.neighbors(current):
                new_cost = cost_so_far[current] + self.cost(current, node)
                if node not in visited or cost_so_far[node] > new_cost:
                    visited.add(node); came_from[node] = current; cost_so_far[node] = new_cost
                    heappush(fringe, (self.huristics[node], node))
            print(', '.join([str(n) for n in fringe]))
        if found: print(); return came_from, cost_so_far[goal]
        else: print('No path from {} to {}'.format(start, goal)); return None, inf

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')


    def __str__(self):
        return str(self.edges)
```

```python
graph = Graph(directed=True)
graph.add_edge('A', 'B', 4)
graph.add_edge('A', 'C', 1)
graph.add_edge('B', 'D', 3)
graph.add_edge('B', 'E', 8)
graph.add_edge('C', 'C', 0)
graph.add_edge('C', 'D', 7)
graph.add_edge('C', 'F', 6)
graph.add_edge('D', 'C', 2)
graph.add_edge('D', 'E', 4)
graph.add_edge('E', 'G', 2)
graph.add_edge('F', 'G', 8)
graph.set_huristics({'A': 8, 'B': 8, 'C': 6, 'D': 5, 'E': 1, 'F': 4, 'G': 0})
start, goal = 'A', 'G'
traced_path, cost = graph.greedy_search(start, goal)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal); print('\nCost:',
cost)
```

## Output:

```
Expand Node | Fringe

--------------------

-           | (8, 'A')

A           | (6, 'C'), (8, 'B')

C           | (4, 'F'), (8, 'B'), (5, 'D')

F           | (0, 'G'), (8, 'B'), (5, 'D')

G           |

Path: A => C => F => G

Cost: 15
```

# 08. Iterative Deepening A* Search

## Source Code:

```
from queue import heappop, heappush
from math import inf

class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.huristics = {}
        self.directed = directed

    def add_edge(self, node1, node2, cost = 1, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = {}
        neighbors[node2] = cost
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, cost, True)

    def set_huristics(self, huristics={}):
        self.huristics = huristics

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def cost(self, node1, node2):
        try: return self.edges[node1][node2]
        except: return inf

    def iterative_deepening_astar_search(self, start, goal):
        prev_visited, depth = 0, 0
        while True:
            trace, cost, visited = self.dept_limited_astar_search(start, goal, depth)
            if trace or visited == prev_visited: return trace, cost
            prev_visited = visited
            depth += 1


    def dept_limited_astar_search(self, start, goal, limit=-1):
        print('Depth Limit =', limit)
        found, fringe, visited = False, [(self.huristics[start], start, 0)], set([start])
        came_from, cost_so_far = {start: None}, {start: 0}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', str(fringe[0][:-1])))
        while not found and len(fringe):
            _, current, depth = heappop(fringe)
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            if limit == -1 or depth < limit:
                for node in self.neighbors(current):
                    new_cost = cost_so_far[current] + self.cost(current, node)
                    if node not in visited or cost_so_far[node] > new_cost:
                        visited.add(node); came_from[node] = current; cost_so_far[node] = new_cost
                        heappush(fringe, (new_cost + self.huristics[node], node, depth + 1))
            print(', '.join([str(n[:-1]) for n in fringe]))
        if found: print(); return came_from, cost_so_far[goal], len(visited)
        else: print('No path from {} to {}'.format(start, goal)); return None, inf, len(visited)

    @staticmethod
    def print_path(came_from, goal):
```

```
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')


    def __str__(self):
        return str(self.edges)

graph = Graph(directed=True)
graph.add_edge('A', 'B', 4)
graph.add_edge('A', 'C', 1)
graph.add_edge('B', 'D', 3)
graph.add_edge('B', 'E', 8)
graph.add_edge('C', 'C', 0)
graph.add_edge('C', 'D', 7)
graph.add_edge('C', 'F', 6)
graph.add_edge('D', 'C', 2)
graph.add_edge('D', 'E', 4)
graph.add_edge('E', 'G', 2)
graph.add_edge('F', 'G', 8)
graph.set_huristics({'A': 8, 'B': 8, 'C': 6, 'D': 5, 'E': 1, 'F': 4, 'G': 0})
start, goal, limit = 'A', 'G', 3
traced_path, cost = graph.iterative_deepening_astar_search(start, goal)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal); print('\nCost:',
cost)
```

## Output:

```
Depth Limit = 0

Expand Node | Fringe

--------------------

-           | (8, 'A')

A           |

No path from A to G

Depth Limit = 1

Expand Node | Fringe

--------------------

-           | (8, 'A')

A           | (7, 'C'), (12, 'B')

C           | (12, 'B')

B           |

No path from A to G

Depth Limit = 2

Expand Node | Fringe

--------------------

-           | (8, 'A')

A           | (7, 'C'), (12, 'B')

C           | (11, 'F'), (13, 'D'), (12, 'B')
```

```
F           | (12, 'B'), (13, 'D')
B           | (12, 'D'), (13, 'D'), (13, 'E')
D           | (13, 'D'), (13, 'E')
D           | (13, 'E')
E           |
No path from A to G
Depth Limit = 3
Expand Node | Fringe
--------------------
-           | (8, 'A')
A           | (7, 'C'), (12, 'B')
C           | (11, 'F'), (13, 'D'), (12, 'B')
F           | (12, 'B'), (13, 'D'), (15, 'G')
B           | (12, 'D'), (13, 'E'), (13, 'D'), (15, 'G')
D           | (12, 'E'), (13, 'D'), (15, 'G'), (13, 'E')
E           | (13, 'D'), (13, 'E'), (15, 'G')
D           | (13, 'E'), (15, 'G')
E           | (13, 'G'), (15, 'G')
G           |
Path: A => B => D => E => G
Cost: 13
```

# 09. Iterative Deepening Depth First Search

## Source Code:

```
from collections import deque

class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed

    def add_edge(self, node1, node2, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = set()
        neighbors.add(node2)
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, True)

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def iterative_deepening_dfs(self, start, goal):
        prev_iter_visited, depth = [], 0
        while True:
            traced_path, visited = self.depth_limited_search(start, goal, depth)
            if traced_path or len(visited) == len(prev_iter_visited): return traced_path
            else: prev_iter_visited = visited; depth += 1

    def depth_limited_search(self, start, goal, limit=-1):
        print('Depth limit =', limit)
        found, fringe, visited, came_from = False, deque([(0, start)]), set([start]), {start: None}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', start))
        while not found and len(fringe):
            depth, current = fringe.pop()
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            if limit == -1 or depth < limit:
                for node in self.neighbors(current):
                    if node not in visited:
                        visited.add(node); fringe.append((depth + 1, node))
                        came_from[node] = current
            print(', '.join([n for _, n in fringe]))
        if found: print(); return came_from, visited
        else: print('No path from {} to {}'.format(start, goal)); return None, visited

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')

    def __str__(self):
        return str(self.edges)


graph = Graph(directed=False)
graph.add_edge('A', 'B')
graph.add_edge('A', 'S')
```

```
graph.add_edge('S', 'G')
graph.add_edge('S', 'C')
graph.add_edge('C', 'F')
graph.add_edge('G', 'F')
graph.add_edge('C', 'D')
graph.add_edge('C', 'E')
graph.add_edge('E', 'H')
graph.add_edge('G', 'H')
start, goal = 'A', 'H'
traced_path = graph.iterative_deepening_dfs(start, goal)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal);print()
```

## Output:

```
Depth limit = 0

Expand Node | Fringe

--------------------

-           | A

A           |

No path from A to H

Depth limit = 1

Expand Node | Fringe

--------------------

-           | A

A           | S, B

B           | S

S           |

No path from A to H

Depth limit = 2

Expand Node | Fringe

--------------------

-           | A

A           | S, B

B           | S

S           | G, C

C           | G

G           |

No path from A to H

Depth limit = 3

Expand Node | Fringe

--------------------

-           | A
```

```
A           | S, B
B           | S
S           | G, C
C           | G, D, E, F
F           | G, D, E
E           | G, D
D           | G
G           | H
H           |
Path: A => S => G => H
```

# 10. Uninformed Cost Search

## Source Code:

```
class Graph:
    def __init__(self, directed=True):
        self.edges = {}
        self.directed = directed

    def add_edge(self, node1, node2, cost = 1, __reversed=False):
        try: neighbors = self.edges[node1]
        except KeyError: neighbors = {}
        neighbors[node2] = cost
        self.edges[node1] = neighbors
        if not self.directed and not __reversed: self.add_edge(node2, node1, cost, True)

    def neighbors(self, node):
        try: return self.edges[node]
        except KeyError: return []

    def cost(self, node1, node2):
        try: return self.edges[node1][node2]
        except: return inf


    def uniform_cost_search(self, start, goal):
        found, fringe, visited, came_from, cost_so_far = False, [(0, start)], set([start]), {start: None}, {start: 0}
        print('{:11s} | {}'.format('Expand Node', 'Fringe'))
        print('--------------------')
        print('{:11s} | {}'.format('-', str((0, start))))
        while not found and len(fringe):
            _, current = heappop(fringe)
            print('{:11s}'.format(current), end=' | ')
            if current == goal: found = True; break
            for node in self.neighbors(current):
                new_cost = cost_so_far[current] + self.cost(current, node)
                if node not in visited or cost_so_far[node] > new_cost:
                    visited.add(node); came_from[node] = current; cost_so_far[node] = new_cost
                    heappush(fringe, (new_cost, node))
            print(', '.join([str(n) for n in fringe]))
        if found: print(); return came_from, cost_so_far[goal]
        else: print('No path from {} to {}'.format(start, goal)); return None, inf

    @staticmethod
    def print_path(came_from, goal):
        parent = came_from[goal]
        if parent:
            Graph.print_path(came_from, parent)
        else: print(goal, end='');return
        print(' =>', goal, end='')


    def __str__(self):
        return str(self.edges)


graph = Graph(directed=True)
graph.add_edge('A', 'B', 4)
graph.add_edge('A', 'C', 1)
graph.add_edge('B', 'D', 3)
graph.add_edge('B', 'E', 8)
graph.add_edge('C', 'C', 0)
```

```python
graph.add_edge('C', 'D', 7)
graph.add_edge('C', 'F', 6)
graph.add_edge('D', 'C', 2)
graph.add_edge('D', 'E', 4)
graph.add_edge('E', 'G', 2)
graph.add_edge('F', 'G', 8)
start, goal = 'A', 'G'
traced_path, cost = graph.uniform_cost_search(start, goal)
if (traced_path): print('Path:', end=' '); Graph.print_path(traced_path, goal); print('\nCost:', 
cost)
```

## Output:

```
Expand Node | Fringe

--------------------

-           | (0, 'A')

A           | (1, 'C'), (4, 'B')

C           | (4, 'B'), (8, 'D'), (7, 'F')

B           | (7, 'D'), (8, 'D'), (7, 'F'), (12, 'E')

D           | (7, 'F'), (8, 'D'), (12, 'E'), (11, 'E')

F           | (8, 'D'), (11, 'E'), (12, 'E'), (15, 'G')

D           | (11, 'E'), (15, 'G'), (12, 'E')

E           | (12, 'E'), (15, 'G'), (13, 'G')

E           | (13, 'G'), (15, 'G')

G           |

Path: A => B => D => E => G

Cost: 13
```