**Dinic’s Algorithm**

**Iheb Gafsi**\*

INSAT Student

**Iheb.engineer@gmail.com**

**Definition:**

Dinic's Algorithm is an efficient algorithm for solving the maximum flow problem in a flow network. It is an improvement over the Ford-Fulkerson algorithm and is based on the concept of layered graphs. The algorithm uses a series of level graphs to efficiently find augmenting paths, making it highly efficient in practice.

The main idea behind Dinic's Algorithm is to build a layered graph where each layer represents the distance from the source in terms of edge count. During the algorithm's execution, it explores these layers in a Breadth-First Search (BFS)-like manner, finding augmenting paths with increasing lengths. This way, the algorithm terminates when no more augmenting paths can be found in the layered graph.

One of the key features that make Dinic's Algorithm highly efficient is that it guarantees each phase of BFS to improve the distance of at least one node from the source. This results in the algorithm converging quickly, making it much faster compared to the classical Ford-Fulkerson algorithm.

The time complexity of Dinic's Algorithm is in the worst case, where V is the number of vertices and E is the number of edges in the flow network. However, in practice, the algorithm often performs much faster, especially in sparse graphs, due to the use of layered graphs and blocking flow optimization.

Due to its efficient performance, Dinic's Algorithm is widely used in real-world applications involving network flow problems, such as transportation, telecommunications, and resource allocation. Its ability to handle large-scale flow networks with excellent time complexity makes it a popular choice for solving maximum flow problems in practical scenarios.

**Algorithm:**

To implement Dinic's Algorithm, follow these steps:

1. Build Residual Graph: Create a residual graph from the original flow network. The residual graph represents the remaining capacity for each edge after the flow has been pushed through it. Initialize the flow of all edges to zero.

2. Construct Layered Graph: Use Breadth-First Search (BFS) to construct a layered graph that indicates the shortest path from the source to each node in terms of edge count. This helps in efficiently finding augmenting paths.

3. Find Blocking Flow: While there exists a blocking flow in the layered graph (an augmenting path from the source to the sink), find the minimum capacity (bottleneck) along the path. Then, push the maximum possible flow through this path, updating the residual graph accordingly.

4. Update Layered Graph: After each blocking flow is found, update the layered graph by re-running BFS to find the shortest paths. This ensures that the algorithm converges efficiently.

5. Repeat Steps 3 and 4: Continue finding blocking flows and updating the layered graph until no more augmenting paths can be found.

6. Calculate Maximum Flow: The maximum flow in the network is the total flow outgoing from the source. Once no more augmenting paths can be found, the algorithm terminates, and the maximum flow is obtained.

1. from queue import Queue

2. # Variables

3. INF = float('inf')

4. # Edge Class

5. class Edge:

6.     def \_\_init\_\_(self, back, front, capacity):

7.         self.back = back

8.         self.front = front

9.         self.capacity = capacity

10.         self.residual = None

11.         self.flow = 0

12.     def isResidual(self):

13.         return self.capacity == 0

14.     def remaining\_capacity(self):

15.         return self.capacity - self.flow

16.     def augment(self, bottleNeck):

17.         self.flow += bottleNeck

18.         self.residual.flow -= bottleNeck

19.

20. class FlowNetwork:

21.     def \_\_init\_\_(self, n, source, sink):

22.         self.n = n

23.         self.source = source

24.         self.sink = sink

25.         self.graph = [[] for \_ in range(n)]

26.         self.visited = [0] \* n

27.         self.visitedToken = 1

28.         self.max\_flow = 0

29.         self.level = [0] \* n

30.

31.     def add\_edge(self, back, front, capacity):

32.         edge = Edge(back, front, capacity)

33.         residual = Edge(front, back, 0)

34.         edge.residual = residual

35.         residual.residual = edge

36.         self.graph[back].append(edge)

37.         self.graph[front].append(residual)

38.     # Dinic' Algorithm

39.     def bfs(self):

40.         self.level = [-1] \* self.n

41.         queue = Queue()

42.         queue.put(self.source)

43.         self.level[self.source] = 0

44.         while not queue.empty():

45.             node = queue.get()

46.             for edge in self.graph[node]:

47.                 if edge.remaining\_capacity()>0 and self.level[edge.front] == -1:

48.                     self.level[edge.front] = self.level[node] + 1

49.                     queue.put(edge.front)

50.         return self.level[self.sink]!=-1

51.

52.     def dfs(self, i, next, flow):

53.         if i == self.sink: return flow

54.         while next[i] < len(self.graph[i]):

55.             edge = self.graph[i][next[i]]

56.             if edge.remaining\_capacity()>0 and self.level[edge.front] == self.level[i]+1:

57.                 bottleneck = self.dfs(edge.front, next, min(flow, edge.remaining\_capacity()))

58.                 if bottleneck > 0:

59.                     edge.augment(bottleneck)

60.                     return bottleneck

61.             next[i] += 1

62.         return 0

63.

64.     def find\_max\_flow(self):

65.         while self.bfs():

66.             next = [0] \* self.n

67.             f = self.dfs(self.source, next, INF)

68.             while f!=0:

69.                 self.max\_flow += f

70.                 f = self.dfs(self.source, next, INF)

71.         return self.max\_flow

72.

73. #Application

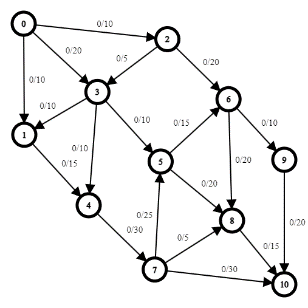
74. edmonds = FlowNetwork…

75.

76. print(edmonds.find\_max\_flow())

**Example:**

Here’s a small example illustrating an example of input outputs for the Cs-Karp Algorithm:



We will use the Python code down below to outline the output of the algorithm on this graph:

1. from queue import Queue

2. # Variables

3. INF = float('inf')

4. # Edge Class

5. class Edge:

6.     def \_\_init\_\_(self, back, front, capacity):

7.         self.back = back

8.         self.front = front

9.         self.capacity = capacity

10.         self.residual = None

11.         self.flow = 0

12.     def isResidual(self):

13.         return self.capacity == 0

14.     def remaining\_capacity(self):

15.         return self.capacity - self.flow

16.     def augment(self, bottleNeck):

17.         self.flow += bottleNeck

18.         self.residual.flow -= bottleNeck

19.

20. class FlowNetwork:

21.     def \_\_init\_\_(self, n, source, sink):

22.         self.n = n

23.         self.source = source

24.         self.sink = sink

25.         self.graph = [[] for \_ in range(n)]

26.         self.visited = [0] \* n

27.         self.visitedToken = 1

28.         self.max\_flow = 0

29.         self.level = [0] \* n

30.

31.     def add\_edge(self, back, front, capacity):

32.         edge = Edge(back, front, capacity)

33.         residual = Edge(front, back, 0)

34.         edge.residual = residual

35.         residual.residual = edge

36.         self.graph[back].append(edge)

37.         self.graph[front].append(residual)

38.     # Dinic' Algorithm

39.     def bfs(self):

40.         self.level = [-1] \* self.n

41.         queue = Queue()

42.         queue.put(self.source)

43.         self.level[self.source] = 0

44.         while not queue.empty():

45.             node = queue.get()

46.             for edge in self.graph[node]:

47.                 if edge.remaining\_capacity()>0 and self.level[edge.front] == -1:

48.                     self.level[edge.front] = self.level[node] + 1

49.                     queue.put(edge.front)

50.         return self.level[self.sink]!=-1

51.

52.     def dfs(self, i, next, flow):

53.         if i == self.sink: return flow

54.         while next[i] < len(self.graph[i]):

55.             edge = self.graph[i][next[i]]

56.             if edge.remaining\_capacity()>0 and self.level[edge.front] == self.level[i]+1:

57.                 bottleneck = self.dfs(edge.front, next, min(flow, edge.remaining\_capacity()))

58.                 if bottleneck > 0:

59.                     edge.augment(bottleneck)

60.                     return bottleneck

61.             next[i] += 1

62.         return 0

63.

64.     def find\_max\_flow(self):

65.         while self.bfs():

66.             next = [0] \* self.n

67.             f = self.dfs(self.source, next, INF)

68.             while f!=0:

69.                 self.max\_flow += f

70.                 f = self.dfs(self.source, next, INF)

71.         return self.max\_flow

72.

73. #Application

74. edmonds = FlowNetwork(11, 0, 10)

75. edmonds.add\_edge(0, 2, 10)

76. edmonds.add\_edge(0, 3, 20)

77. edmonds.add\_edge(0, 1, 10)

78. edmonds.add\_edge(1, 4, 15)

79. edmonds.add\_edge(2, 6, 20)

80. edmonds.add\_edge(2, 3, 5)

81. edmonds.add\_edge(3, 5, 10)

82. edmonds.add\_edge(3, 4, 10)

83. edmonds.add\_edge(3, 1, 10)

84. edmonds.add\_edge(4, 7, 30)

85. edmonds.add\_edge(5, 6, 15)

86. edmonds.add\_edge(5, 8, 20)

87. edmonds.add\_edge(6, 9, 10)

88. edmonds.add\_edge(6, 8, 20)

89. edmonds.add\_edge(7, 5, 25)

90. edmonds.add\_edge(7, 8, 5)

91. edmonds.add\_edge(7, 10, 30)

92. edmonds.add\_edge(8, 10, 15)

93. edmonds.add\_edge(9, 10, 20)

94.

95. print(edmonds.find\_max\_flow())

The corresponding output is:

Python>> 40

