DESIGN & ANALYSIS OF ALGORITHMS PROJECT

TOPIC:

APPLICATIONS OF BREADTH FIRST SEARCH

GROUP MEMBERS:

ADRIJ SHARMA – IMT2019004

AGRIM JAIN – IMT2019005

ANKIT AGRAWAL – IMT2019010

INTRODUCTION

To

BREADTH

FIRST

SEARCH

ALGORITHM

Introduction to the BFS algorithm

The BFS algorithm, also known as the Breadth-first search algorithm, is an algorithm that is used for traversing (or searching) graph data structures. It is a very important graph algorithm, and has found use in a lot of real world problems, such as finding connected components of an undirected graph and finding the shortest path out of a maze.

The BFS algorithm explores the edges G to "discover" every vertex that is reachable from s, and in this process, also computes the "distance" (smallest number of edges) from s to each reachable vertex.

The BFS algorithm

The BFS algorithm inputs a graph G and a starting vertex s. The pseudocode for the BFS algorithm is as follows:

Algorithm 1: The Breadth-First Search algorithm

Data: Graph G, source s
Result: Performs Breadth-first Search on the graph

```
1  begin
2      for each vertex u ∈ G.V − {s} do
3          u.color = WHITE, u.d = ∞, u.π = NIL
4      s.color = GRAY, s.d = 0, s.π = NIL
5      Q = ∅
6      ENQUEUE(Q, s)
7      while Q ≠ ∅ do
8          u = DEQUEUE(Q)
9          for each v ∈ G.Adj[u] do
10             if v.color == WHITE then
11                 v.color = GRAY
12                 v.d = u.d + 1
13                 v.π = u
14                 ENQUEUE(Q, v)
15         u.color = BLACK
```

In the above algorithm, $u.\pi$ denotes the parent of the vertex u, and $u.d$

denotes the minimum distance of u from the source vertex s.

Working of the BFS algorithm

The steps below demonstrate the working of the BFS algorithm:

1. The algorithm begins by coloring each vertex to WHITE. It sets the distance of each vertex from the source s to ∞, and also sets the parent of each vertex to NIL.

2. The algorithm now sets the color of the source vertex s to GRAY. Since it is the source vertex, its distance is set as 0, and its parent is set as NIL.

3. The algorithm now initializes an empty queue Q, which is a FIFO (First-In, First Out) data structure, and enqueues the source vertex into the queue.

4. The algorithm now executes a **while** loop till Q becomes empty. It dequeues a vertex u from the queue, and then begins exploring each of its adjacent vertices.

5. The algorithm checks if the adjacent vertices (of vertex u) have been visited by it before (by checking the color of the vertex). If it is WHITE, then the algorithm enqueues the vertex and sets its color to GRAY. It also sets the parent of those vertices as u, and the distance of those vertices from the source vertex s to $u.d + 1$.

6. The algorithm now colors the vertex u to BLACK, as all of its adjacent vertices have been visited by the algorithm.

7. The algorithm terminates when the queue Q becomes empty.

We now explore some applications of the **Breadth-First Search algorithm**.

APPLICATION - 1

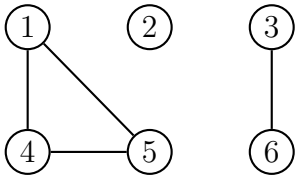
BREADTH FIRST SEARCH IN MICROSOFT PAINT

1 Application of BFS in Microsoft Paint

1.1 Connected components of an undirected graph

A connected component of an undirected graph is a maximal set of nodes such that each pair of nodes is connected by a path. Maximal means that it is the largest possible subgraph: it is not possible to find a node in the graph, which could be added to the subgraph and all the nodes in the subgraph would still be connected.

For example, consider the below graph:



The above graph has 3 connected components, which are {1,4,5}, {2} and {3,6}. It is important to note that {1,4} is *not* a connected component, because it is a subset of a connected component, namely {1,4,5}.

It is possible to find connected components of a given graph using the Breadth-first search (BFS) algorithm. We will examine how one of the tools in Microsoft Paint uses the BFS algorithm to find the connected component corresponding to a pixel in an image to perform its function.

1.2 Introduction to Microsoft Paint

Microsoft Paint is a simple graphics editor, which can be used to create, edit and save images in different formats such as JPEG, GIF and PNG. It comes bundled with the Windows operating system, and provides a simple yet powerful interface to perform image manipulations.

1.3 The flood fill tool

The flood fill tool (also known as the bucket fill tool) is used to color similarly colored pixels (or cells) with another color. The symbol for this tool in Paint

is shown in Figure (1).

Figure 1: Symbol of the flood fill tool in Microsoft Paint

1.4 Working of the flood fill tool

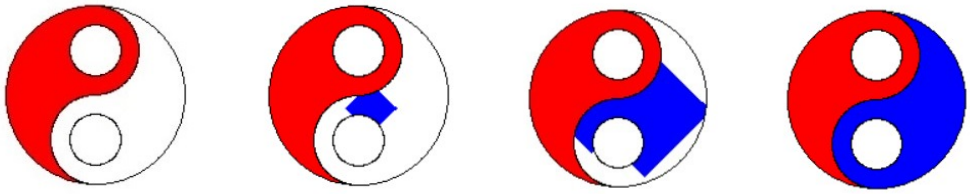
Images are made of pixels, which are usually organized in the form of an ordered rectangular array. The image width is the number of columns, and the image height is the number of rows in the array. Therefore, the pixel array can be considered to be a matrix of M rows $\times$ N columns.

The color of each pixel is usually specified in the RGB format, in which the color of each pixel is encoded in 24 bits.

The steps given below briefly explain how the flood fill tool works in Microsoft Paint:

1. When the user clicks on some part of the image, the color of the pixel on which the user has clicked is recorded.
2. The tool colors this pixel with a new color specified by the user, by changing the RGB value of the pixel.
3. The tool now searches the pixels which are adjacent to the pixel clicked by the user, and have the same color as the original color of the pixel clicked by the user.
4. The tool now colors those pixels which satisfy the conditions given in step 3.
5. The tool now repeats steps 3 and 4 for the adjacent pixels satisfying the conditions of step 3.
6. The tool stops coloring pixels when it cannot find any pixels which satisfy the conditions of step 3.

Figure (2) shows an example of how the flood fill tool fills blue color in an image.



(a) (b) (c) (d)

Figure 2: Working of the flood fill tool for filling blue color in an image.

1.5 Connected components in an image

As described earlier, an image can be considered as a matrix having M rows and N columns. An image can be thought of as an undirected graph, where we have:

1. Node: Each pixel can be considered as a node in the graph.
2. Edge: An edge exists between two neighboring nodes if their corresponding pixels have the same color. Two nodes having coordinates (x_1, y_1) and (x_2, y_2) are considered neighbors if and only if:

$$|x_2 - x_1| + |y_2 - y_1| = 1$$

Now, we can apply the definition of connected components in this graph to find the connected components in an image. For example, consider the below image:

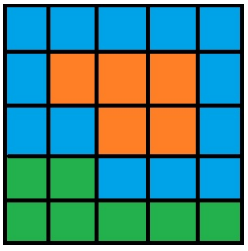
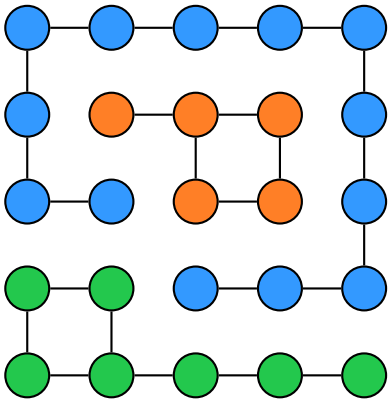


Figure 3: In this image, the pixels are separated by black lines for better visualization.

We can convert the above image to a graph, as shown below:



The above graph (and hence the image) has 3 connected components, which are:

1. Subgraph having blue ( ) colored nodes
2. Subgraph having orange ( ) colored nodes
3. Subgraph having green ( ) colored nodes

We can now see a relation between the connected components of an image and the working of the flood fill tool. The flood fill tool finds the connected component containing the pixel which was clicked by the user, and fills the pixels in the connected component with a new color chosen by the user.

We now show how the BFS algorithm is used for implementing the flood fill tool. Rather than finding all the connected components of the image, we will emphasize on how the BFS algorithm finds the connected component which contains the pixel that was clicked by the user.

1.6 Implementing flood fill tool using BFS

1.6.1 The algorithm

Consider an image matrix $image[1 \ldots m][1 \ldots n]$, having m rows and n columns. Also, let *start* be the pixel which was clicked by the user when using the flood

fill tool. The BFS algorithm for flood fill is as follows:

Algorithm 1: Flood fill tool using BFS

Data: $image[1 \ldots m][1 \ldots n], newcolor, start$
Result: Fills connected component containing *start* with *newcolor*

```
1  begin
2      Queue Q ⟵ ∅
3      visited[1...m][1...n] ⟵ 0      /* Tracks visited pixels */
4      Enqueue(Q, start)
5      visited[start.y][start.x] ⟵ 1
6      while Q ≠ ∅ do
7          front ⟵ Q.front
8          deque(Q)                   /* Remove front pixel from Q */
9          image[front.y][front.x] ⟵ newcolor
10         for all n ∈ front.neighbors do
11             if n.color == front.color and !visited[n.y][n.x] then
12                 Enqueue(Q, n)
13                 visited[n.y][n.x] ⟵ 1
14     return
```

1.6.2 Working of the algorithm

The algorithm begins by initializing an empty queue Q. It also initializes a two-dimensional array $visited[1 \ldots m][1 \ldots n]$, which keeps tracks of whether a pixel having coordinates (x, y) has been visited or not. If the pixel at (x, y) has not been visited, then $visited[y][x] = 0$, otherwise it is 1. It also enqueues the pixel (called *start*) which was clicked by the user, and marks the pixel as visited.

From lines 7-9, the algorithm takes out a pixel (called $front$) from the queue and fills this pixel in the image with *newcolor*.

From lines 10-13, the algorithm searches the neighbors of the pixel $front$ which have the same color as that of $front$ and have not been visited by the algorithm earlier. The algorithm enqueues such pixels and marks them as visited.

Finally, the algorithm terminates when it has colored all the pixels of the connected component containing the pixel *start* with *newcolor*.

1.6.3 Running time analysis

In the worst case, the above algorithm might need to go through all the pixels of the image. Since the total number of pixels in the image is $m \times n$ and each pixel is enqueued and dequeued atmost once, the worst case time complexity of the algorithm is $\mathcal{O}(m \times n)$.

As a small note, each pixel can have atmost 4 neighbors, and hence the **for** loop starting at line 10 runs for no more than 4 times for each pixel. Thus, it does not increase the asymptotic time complexity of the algorithm.

1.6.4 Space complexity analysis

The space complexity of the algorithm is $\mathcal{O}(m \times n)$. This is because the algorithm uses the matrix *visited*, which has m rows and n columns. The size of the queue Q cannot exceed the the number of pixels in the image (which is $m \times n$), and hence, it does increase the asymptotic space complexity of the algorithm.

1.6.5 Proof of correctness

Claim: The algorithm correctly finds and fills the connected component containing the pixel *start* (which was clicked by the user).

Proof: When we begin searching from the pixel *start*, we will clearly never reach any pixel outside the connected component with BFS. So, now we prove that we will reach all the pixels (and hence fill them with *newcolor*) in the connected component containing the pixel *start* using induction on the level of each pixel, which we denote by i.

Base Case: When $i = 0$, it means that we are referring to the *start* pixel, which was clicked by the user. When the algorithm executes the **while** loop (on line 6) for the first time, it fills the *start* pixel with *newcolor*, as it was the first pixel to be pushed in the queue Q. Hence, the algorithm correctly finds (and colors) the *start* pixel.

Induction hypothesis: The algorithm correctly reaches and fill all pixels (in the connected component) upto level i with *newcolor*.

Induction step: We now need to prove that all the pixels at the $(i+1)^{th}$

level and in the connected component are also reached and colored by the algorithm.

Since we consider only the pixels in the $(i+1)^{th}$ level which belongs to the same connected component containing the *start* pixel, it must be the case that these pixels have the same color as the original color of the *start* pixel.

Clearly, the pixels in the $(i+1)^{th}$ level have not been visited. This is because we are using a queue (and not a stack), which is a FIFO (First-In, First-Out) data structure. Therefore, it is not possible to visit the pixels at the $(i+1)^{th}$ level before the pixels at the i^{th} level have been visited. Therefore, the pixels in the $(i+1)^{th}$ level would not have been visited before by the algorithm.

Since we have reached and colored all the pixels in the connected component till the i^{th} level, we must be able to reach the pixels in the connected component at the $(i+1)^{th}$ level as the pixels at the $(i+1)^{th}$ level must be connected to the *start* pixel by some pixel at the i^{th} level. Hence, these pixels would be the neighbors of the pixels present at the i^{th} level.

Therefore, the pixels present in the $(i+1)^{th}$ level satisfy all the conditions in lines 11 and 12 of the algorithm, they will be visited by the algorithm, and hence will be eventually filled with *newcolor*.

Hence, we have proved the induction step and therefore the algorithm correctly reaches and fills all the pixels in the connected component containing the *start* pixel. ■

1.7 Example of working of algorithm

We now show the working of the algorithm on a simple 3×3 image, which is shown in Figure (4).

Suppose the user clicks on the pixel at location $(1, 2)$, i.e. the cell which is at the 2^{nd} row and the 1^{st} column, and wishes to fill it with a new color, pink. The expected result of the flood fill tool, after filling all the necessary pixels is shown in Figure (5).

Therefore, the input parameters for our algorithm are:

- *newcolor* = pink
- *start* = Pixel at $(1, 2)$

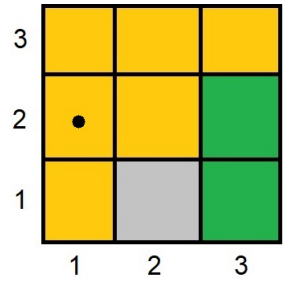


Figure 4: Example image. The black dot indicates the pixel on which the user has clicked.

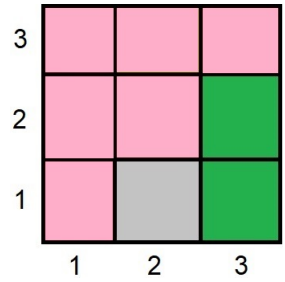


Figure 5: The expected image, after the algorithm has terminated.

- $image[1 \ldots 3][1 \ldots 3]$

1.7.1 Step 1

The algorithm begins by initializing an empty queue called Q, and creates a 3×3 matrix called *visited*, whose all entries all initially 0, as shown below:

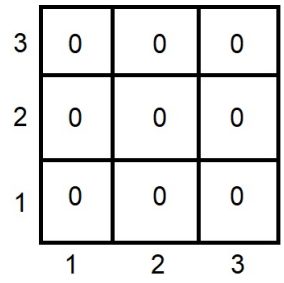


Figure 6: The matrix *visited*. Initially, all of its entries are 0, as none of the pixels have been visited by the algorithm.

1.7.2 Step 2

Now, the algorithm enqueues the pixel at $(1, 2)$ (which is the *start* pixel) into the queue Q, and marks it visited. For simplicity, we will denote the pixels present in Q by their coordinates in the form of (x, y), which indicates that the pixel is at the x^{th} column and at the y^{th} row.

Therefore, the matrix *visited* and the queue Q now look as shown below:

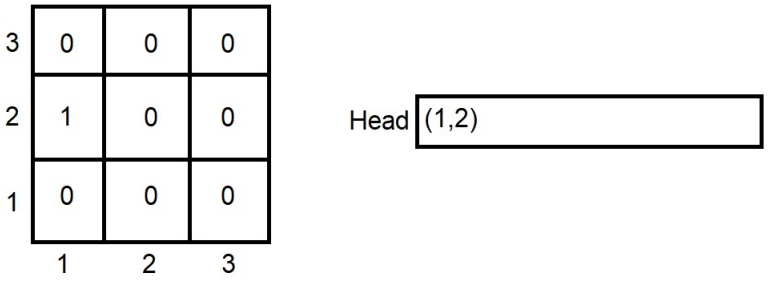


Figure 7: The *visited* matrix and the queue Q.

1.7.3 Step 3

Now, we enter the **while** loop of the algorithm on line 6. At this moment, our queue is not empty, and therefore, the algorithm takes out the front pixel (which is $(1, 2)$) from the queue, and colors it with *newcolor*, which is pink. At this stage, the image, the matrix *visited*, and the queue Q look as shown below:

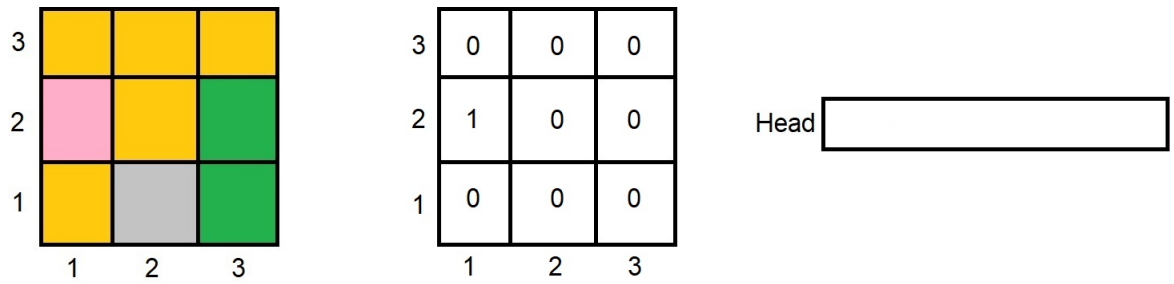


Figure 8: The image, the *visited* matrix and the queue Q.

1.7.4 Step 4

Now, we move on to the **for** loop at line 10 of the algorithm. Inside this **for** loop, the algorithm checks all the neighbors of the pixel $(1, 2)$ which:

- have the same color as that of the pixel at $(1, 2)$.
- have not been visited by the algorithm before.

In our case, there are 3 such pixels at this stage which satisfy the above conditions. They are shown below:

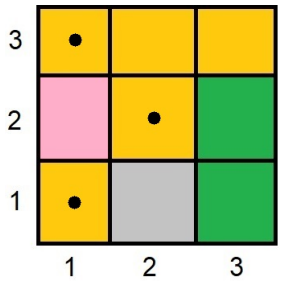


Figure 9: The three neighbors satisfying the above listed conditions, which have been been indicated by black dots.

The coordinates of these three pixels are: $(1, 1), (2, 2), (1, 3)$.

The algorithm now enqueues all of the above pixels into the queue Q, and marks them as visited. The algorithm can enqueue the pixels in any order as it wishes, and so we will assume that firstly $(1, 1)$ has been enqueued, which is followed by $(2, 2)$ and $(1, 3)$. At this stage, the matrix *visited* and the queue Q look as shown below:

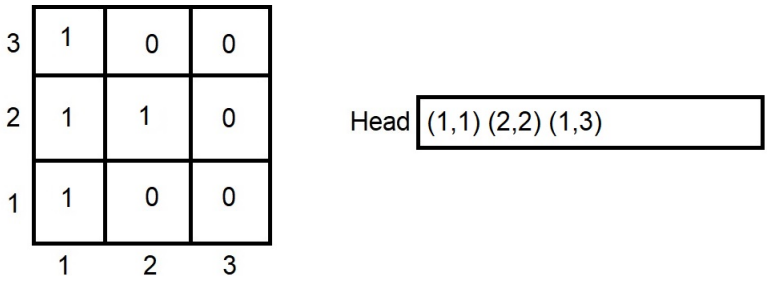


Figure 10: The matrix *visited* and the queue Q.

1.7.5 Step 5

Now, as the queue is not empty, the algorithm starts the **while** loop again from line 6. It dequeues the pixel at $(1, 1)$ from the queue, and colors it with *newcolor*, which is pink. At this stage, the image, the matrix *visited* and the queue Q are shown in Figure (11).

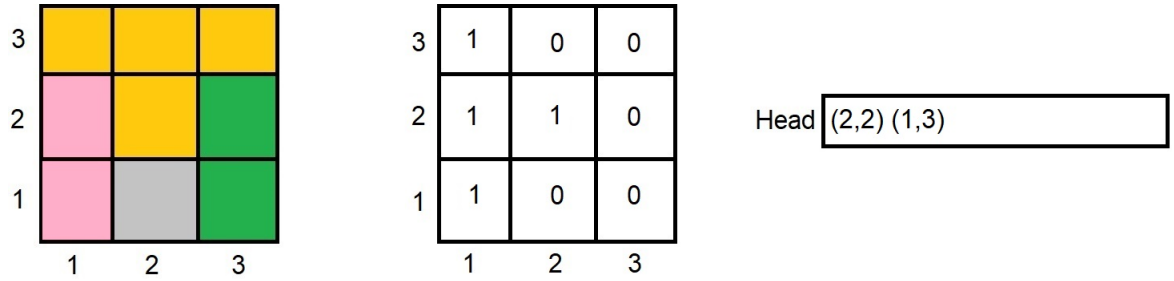


Figure 11: The image, the matrix *visited* and the queue Q.

Now, we enter the **for** loop at line 10 of the algorithm. There are 2 neighbors of this pixel, which are $(2, 1)$ and $(1, 2)$. However, none of them will get enqueued, this is because:

- The pixel $(1, 2)$ has already been marked visited by the algorithm, as its entry in the *visited* matrix is 1. Hence it will not get enqueued.
- The pixel $(2, 1)$ has a different color (grey) than the original color of the pixel $(1, 1)$ (orange). Hence, it will not get enqueued.

Hence, in this step, we do not add any extra pixel to the queue Q.

1.7.6 Step 6

Now, the queue is not empty, the algorithm again executes the **while** loop at line 6. It dequeues the pixel $(2, 2)$ from the queue, and colors it with *newcolor*, which is pink. At this stage, the image, the matrix *visited* and the queue Q are shown below:

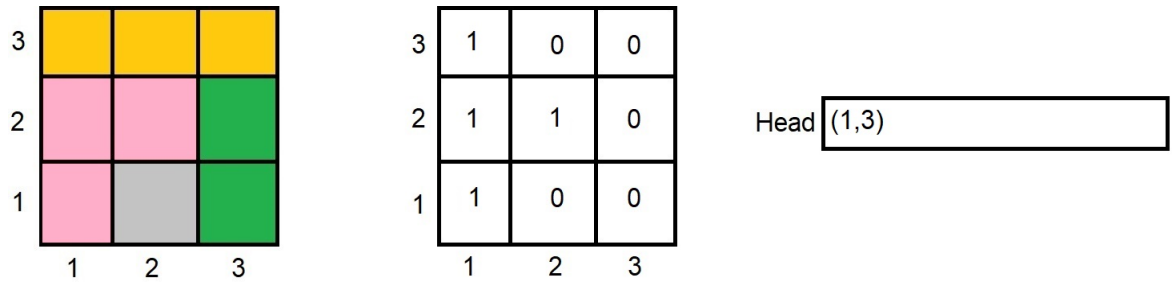


Figure 12: The image, the matrix *visited* and the queue Q.

1.7.7 Step 7

We now enter the **for** loop at line 10 of the algorithm. At this step, we explore all the neighbors of the pixel $(2, 2)$, which are $(1, 2), (2, 1), (3, 2)$ and $(2, 3)$. However, not all of these pixels get enqueued by the algorithm.

- The pixel $(1, 2)$ has already been visited by the algorithm before, as the corresponding entry in the *visited* matrix is 1. So it won't get enqueued.
- The pixel $(2, 1)$ is grey colored, which is different from the original color of the pixel at $(2, 2)$ (which was orange), and hence it won't get enqueued.
- The pixel $(3, 2)$ is green colored, which is different from the original color of the pixel at $(2, 2)$ (which was orange), and hence it won't get enqueued.
- The pixel at $(2, 3)$ has not been visited before, and has the same color as that of the original color of the pixel at $(2, 2)$, and hence it gets enqueued.

Therefore, the only pixel to get enqueued is $(2, 3)$. The algorithm enqueues this pixel, and marks this pixel as visited. Therefore, at the end of this step, the matrix *visited* and the queue Q look as shown in Figure (13).

1.7.8 Step 8

The queue is again not empty, and therefore, the algorithm executes the **while** loop at line 6. It dequeues the pixel $(1, 3)$ from the queue, and colors

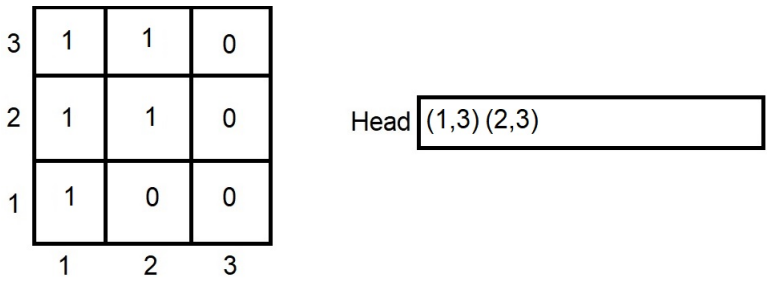


Figure 13: The matrix *visited* and the queue Q.

it with *newcolor*, which is pink. At this stage, the image, the matrix *visited* and the queue Q are shown below:

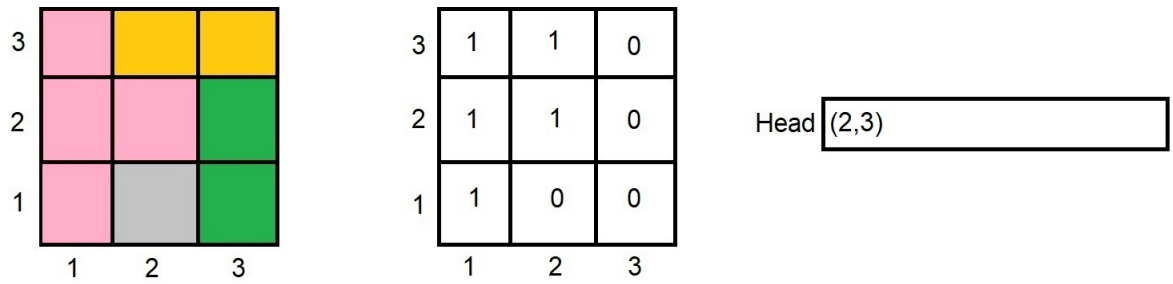


Figure 14: The image, the matrix *visited* and the queue Q.

Now, we enter the **for** loop at line 10 of the algorithm. There are 2 neighbors of this pixel, which are $(1, 2)$ and $(2, 3)$. However, none of them will get enqueued, this is because:

- The pixel at $(1, 2)$ has already been visited by the algorithm, as its corresponding entry in the *visited* matrix is 1.
- The pixel at $(2, 3)$ has already been visited by the algorithm, as its corresponding entry in the *visited* matrix is 1.

Hence, in this step, we do not add any extra pixel to the queue Q.

1.7.9 Step 9

At this step, the queue is not empty. Therefore, the algorithm executes the **while** loop at line 6 again. It dequeues the pixel $(2, 3)$ from the queue, and

colors it with *newcolor*, which is pink. At this stage, the image, the matrix *visited* and the queue Q are shown below:

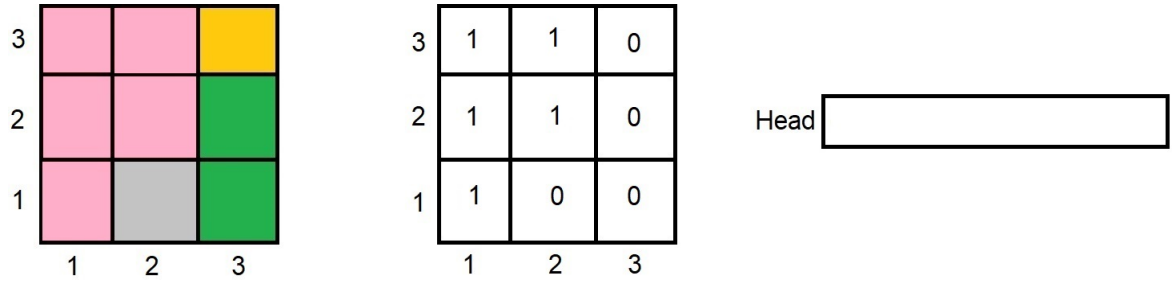


Figure 15: The image, the matrix *visited* and the queue Q.

1.7.10 Step 10

Now, we enter the **for** loop at line 10 of the algorithm, where the algorithm searches for the neighbors satisfying the **if** conditions which have been put in line 11 of the algorithm. There are 3 neighbors of the pixel $(2, 3)$, which are $(1, 3), (2, 2)$ and $(3, 3)$. However, not all of these pixels get enqueued by the algorithm.

- The pixel $(1, 3)$ has already been visited by the algorithm, as its corresponding entry in the *visited* matrix is 1.
- The pixel $(2, 2)$ has already been visited by the algorithm, as its corresponding entry in the *visited* matrix is 1.
- The pixel $(3, 3)$ has not been visited by the algorithm before, and has the same color as that of the original color of the pixel $(2, 3)$ (which is orange). Hence, it gets enqueued by the algorithm.

Therefore, the only pixel which gets enqueued by the algorithm in this step is $(3, 3)$. The algorithm also marks this pixel as visited. At the end of this step, the matrix *visited* and the queue Q look as shown in Figure (16):

1.7.11 Step 11

At this stage, the queue is not empty, and therefore, the algorithm again executes the **while** loop at line 6 of the algorithm. It dequeues the pixel

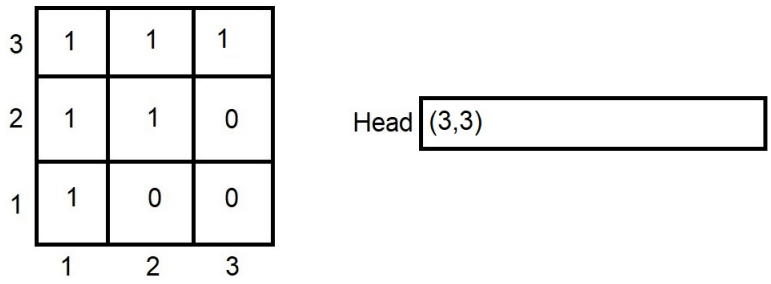


Figure 16: The matrix *visited* and the queue Q.

$(3, 3)$ from the queue, and colors it with *newcolor*, which is pink. At this stage, the image, the matrix *visited* and the queue Q are shown below:

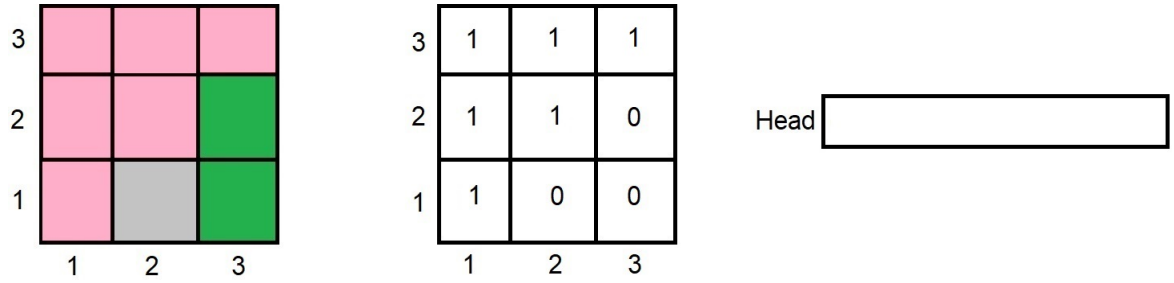


Figure 17: The image, the matrix *visited* and the queue Q.

Now, there are 2 neighbors of the pixel $(3, 3)$, which are $(2, 3)$ and $(3, 2)$. However, none of them will get enqueued because:

- The pixel $(2, 3)$ has already been visited by the algorithm, as its corresponding entry in the *visited* matrix is 1.
- The pixel $(3, 2)$ has a different color (green) as compared to the original color of the pixel $(3, 3)$, which was orange.

Hence, in this step, we do not add any extra pixel to the queue Q.

1.7.12 Step 12

Now, the queue Q is empty. Therefore, we break out of the **while** loop, and the algorithm terminates. We notice that the algorithm has correctly filled the required pixels with pink color, as the final image generated after executing the flood fill algorithm is:

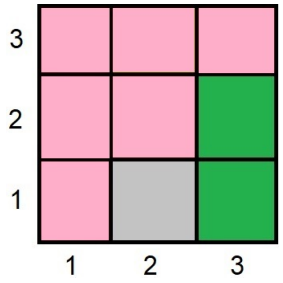


Figure 18: The image after execution of the flood fill algorithm.

Hence, we have demonstrated the working of the BFS algorithm for this example.

References

[1] Stephen P. Borgatti. *Intro to Graph Theory*. 1997. URL: http://www.analytictech.com/mb021/graphtheory.htm.

[2] Massimo Franceschet. *Connected Components*. 2014. URL: https://www.sci.unich.it/~francesc/teaching/network/components.html.

[3] GeeksforGeeks. *Connected Components in an undirected graph*. 2020. URL: https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/.

[4] GeeksforGeeks. *Flood fill Algorithm – how to implement fill() in paint?* 2020. URL: https://www.geeksforgeeks.org/flood-fill-algorithm-implement-fill-paint/.

[5] Luke Johnsto Mary Wootters. *Connected components of undirected and directed graphs*. 2017. URL: https://web.stanford.edu/class/archive/cs/cs161/cs161.1176/Lectures/CS161Lecture10.pdf.

[6] Finlay McWalter. *Wfm_floodfill_animation_queue.gif*. 2008. URL: https://commons.wikimedia.org/wiki/File:Wfm_floodfill_animation_queue.gif.

[7] Tyler Moore. *BFS and DFS applications*. 2017. URL: https://tylermoore.ens.utulsa.edu/courses/cse3353/slides/l07-handout.pdf.

[8] Kristopher Sheets. *What is a digital Image?* May 1, 2013. URL: https://sites.google.com/site/learnimagej/image-processing/what-is-a-digital-image.

[9] Edpresso Team. *What is the Flood fill algorithm?* 2021. URL: https://www.educative.io/edpresso/what-is-the-flood-fill-algorithm.

[10] Wikipedia contributors. *Microsoft Paint — Wikipedia, The Free Encyclopedia*. [Online; accessed 12-March-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Microsoft_Paint&oldid=1009774540.

APPLICATION - 2

FORD – FULKERSON ALGRORITHM FOR THE MAXIMUM FLOW PROBLEM

What is the maximum flow problem?

Basic Idea:

Maximum flow problem involves finding a feasible flow through a single-source, single-sink flow network such that the flow is maximized.

Problem Statement:

We are given a directed graph which represents a flow network where every edge has a non-negative capacity. There are no self-loops in the flow network [It does not make sense that something is flowing out from a vertex and the very next momemt flowing into the same vertex]. The flow running through an edge must be less than or equal to its capacity. We are also given the two vertices, namely, *source* **'s'** and *sink* **'t'** in the flow graph. The motive of the maximum flow problem is to find the maximum possible flow from the source vertex s to the sink vertex t, with an infinite input of the flowing material, in the given flow network (represented through a directed graph) with the following constraints:

- Flow through an edge should not exceed the given capacity of that edge.
- Incoming flow is equal to outgoing flow for every vertex in the flow graph (except the source vertex s and the sink vertex t).
  - For the source vertex s, the flow is only outgoing.
  - For the sink vertex t, the flow is only incoming.

Naive Greedy Algorithm Approach

This Greedy approach does NOT produce the optimal result or correct output at all instances. The idea is to start with all-zero flow and greedily produce flows with higher values. The way to proceed from one to the next is to send more flow on some path from the source vertex s to the sink vertex t.

Working of the Greedy approach to find the maximum flow:

Let 'E' be the number of edge in the flow graph.
Flow(e) = flow through the edge e.
Cap(e) = maximum flow carrying capacity of edge e.

1. Initialization Step: max_flow = 0
   Flow(e) = 0 for every edge 'e' in E.

2. While there exists a path 'P' from source s to sink t such that
   Flow(e) < Cap(e) for every edge 'e' on the path P.
   {
   - Find minimum edge value for path P.
     Our flow is limited by the edge on path P which has the least remaining capacity.
     flow = min{Cap(e) - Flow(e)}
     for all the edges e on path P.

   - max_flow += flow

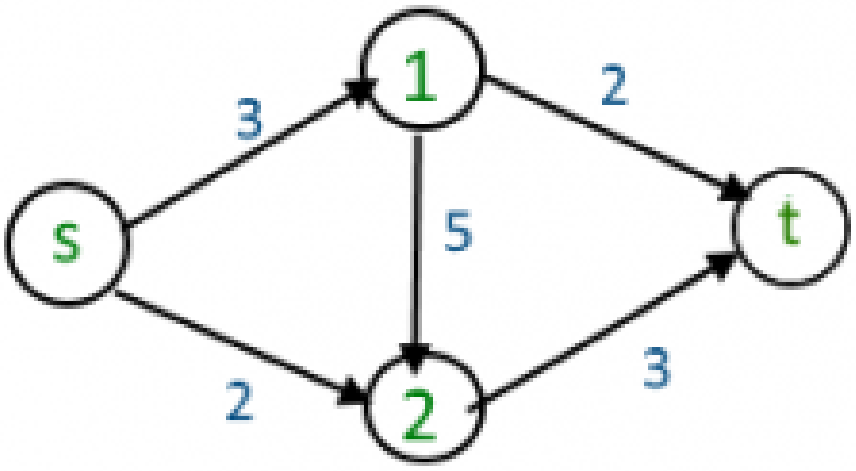
   - Increment the flow through all the edges on path P.
     Flow(e) += flow; for all edges e on path P.

   }

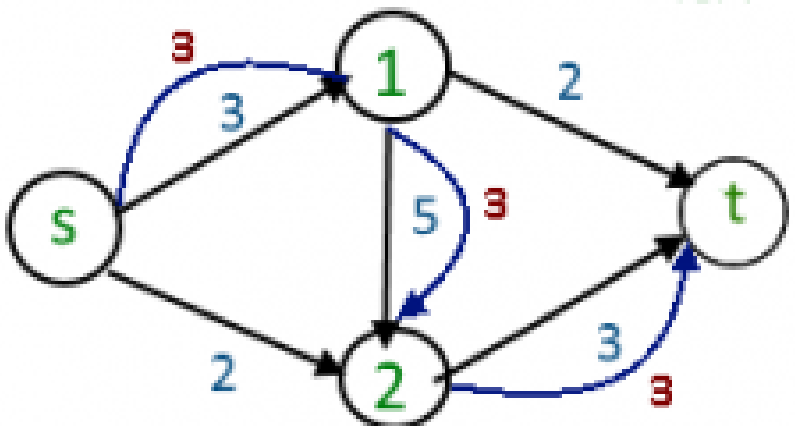
3. Return max_flow

Breadth First Search (BFS) will help us in searching a path from the source vertex s to the sink vertex t in the subgraph of edges 'e' with

Flow(e) < Cap(e) in linear time.



Let us take the example of the above shown graph to prove that this greedy approach won't give the correct output in all the cases.

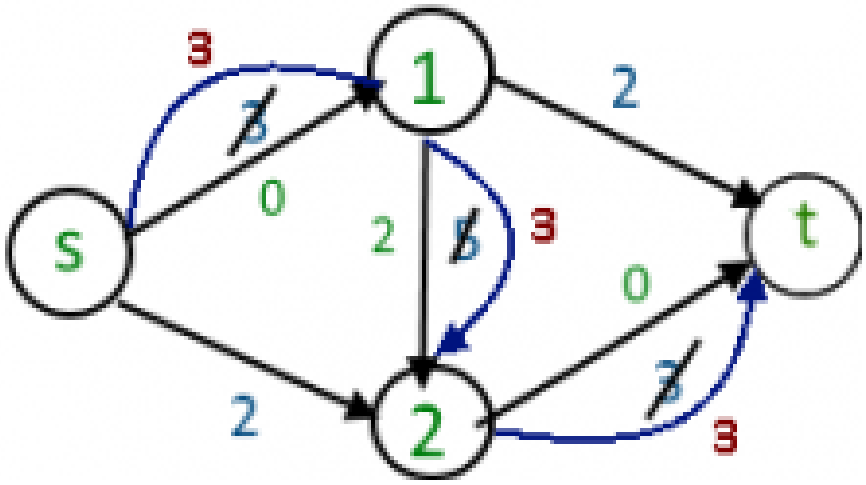


There is a path from source (s) to sink (t) shown with blue colour in the above diagram

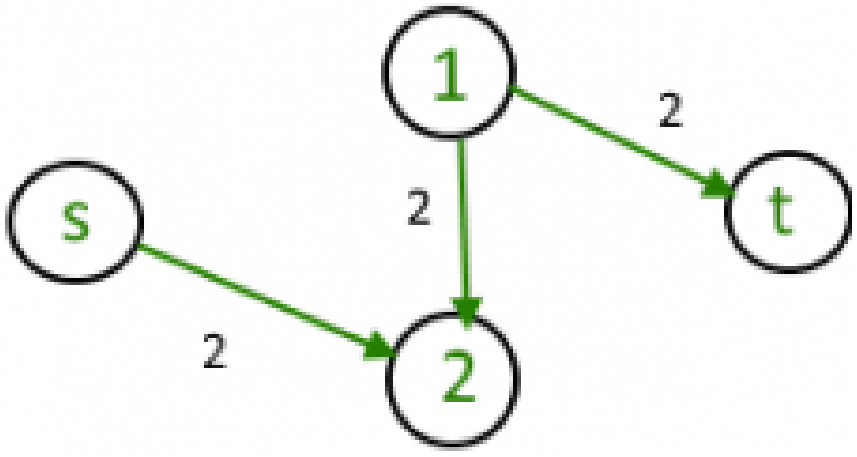
s → 1 → 2 → t

The maximum flow through the path is **3 units**.

The remaining capacities of each edge are shown in the diagram below



For further clarification we remove all the edges from the graph which have lost the capacity to carry any flow i.e. **Cap(e) – Flow(e) = 0**. This is how the graph looks after that:

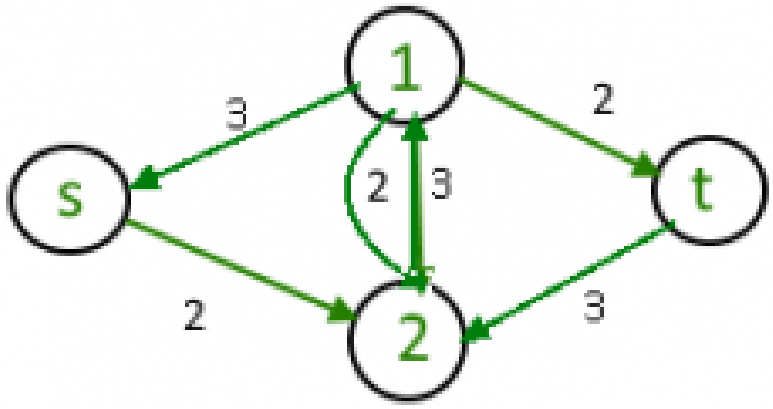


As per our greedy algorithm there is no path from source s to sink t in the above graph. Therefore, our algorithm should return the maximum flow from s to t as 3. But this is incorrect as the maximum possible flow from s to t is 5. For solving this issue we will introduce a new concept, namely, residual graphs.

What are residual graphs?

The motive behind the introduction of the new concept of residual graphs is to extent the greedy algorithm by allowing us to "undo" operations. The term "undo" operations refers to sending material in a direction opposite to the direction of present flow in the edge.

From the point where the greedy algorithm gets stuck in above image, we'd like to route two more units of flow along the edge s→2, then backward along the edge 1→2, "undoing" 2 of the 3 units of flowwe routed the previous iteration, and finally send those 2 units of flow along the edge 1→t to the sink vertex t. Below is an image illustrating the "undoing" more clearly.



So the conditions that we derive are:

- For undoing the flow through an edge 'e' (in other words, making material flow in a direction opposite to the direction of the edge), Flow(e) > 0.

- For material to flow in a direction same as the direction of the directed edge ‘e’, Cap(e) – Flow(e) > 0.

Given a graph G and a flow f in it, we form a new flow network G_f that has the same vertex set of G and that has two edges for each edge of G.

An edge ‘e’ of G that carries flow = Flow(e) and has an initial capacity = Cap(e) is considered a “forward edge” of G_f with residual capacity = Cap(e) - Flow(e) (the remaining capacity for allowing flow to occur) and a “backward edge” of G_f with residual capacity = Flow(e) (the amount of previously routed flow that can be undone).

Pre-requisite Definitions:

1. ***Residual Graph:*** It is a graph which indicates additional possible flow. If there is a path from source to sink in residual graph, then it is possible to add flow.

2. ***Residual Capacity:*** Every edge of a residual graph has this value called the residual capacity which is equal to original capacity of the edge minus current flow in case of forward edges and equal to the cyurrent flow in case of backward edges. Residual capacity is basically the present flow carrying capacity of the edge.

3. ***Residual Edges:*** These are the directed edges going in the backwards direction in the reverse order augmenting path. These exist so that we do not end up considering the augmenting paths which do not lead to maximum flow. We can think of every edge in the original graph as having a residual edge with a flow/capacity of 0/0 which is not usually shown in the diagrams used in the example.

4. ***Minimal Cut:*** This is the maximum possible flow from the source vertex to the sink vertex. This is also called bottle-neck capacity.

5. ***Augmenting Path:*** It is a path of edges in the residual flow graph with unused capacity greater than 0 from the source vertex s to the sink vertex t. Augmenting paths can be done in 2 ways –
   - Non-full forward edges.
   - Non-empty backward edges.

6. ***Augmenting the flow:*** This is the process of updating the flow values of the edges along the augmenting path.
   - For forward edges, this refers to increasing the flow in that edge by the bottleneck value.
   - When augmenting the flow along the augmenting path we also need to decrease the flow along each backward edge, which is called residual edge, by the bottleneck value.

Ford-Fulkerson Algorithm

The following is the idea of Ford-Fulkerson algorithm:

1. Start with initial maximum flow as 0. *max_flow* = 0;
   Initially, the flow through each edge 'e' = Flow(e) = 0.
   Initialize, the residual network graph G_f equal to graph G.

2. While (there exists an augmenting path from the source vertex s to the sink vertex t in the residual graph G_f)
   {
   - Compute the bottle-neck capacity of the path presently being considered by backtracking from sink vertex t to the source vertex s using the **Parent[]** array given by Breadth First Search algorithm.
   - Update the residual graph G_f by augmenting each edge of the path presently being considered. For doing so you need to backtrack from the sink vertex t to the source vertex s using the **Parent[]** array.
     - For all forward edges 'e' => Flow(e) -= bottle neck

- For all backward/reverse edges ‘e’ => Flow(e) += bottle neck

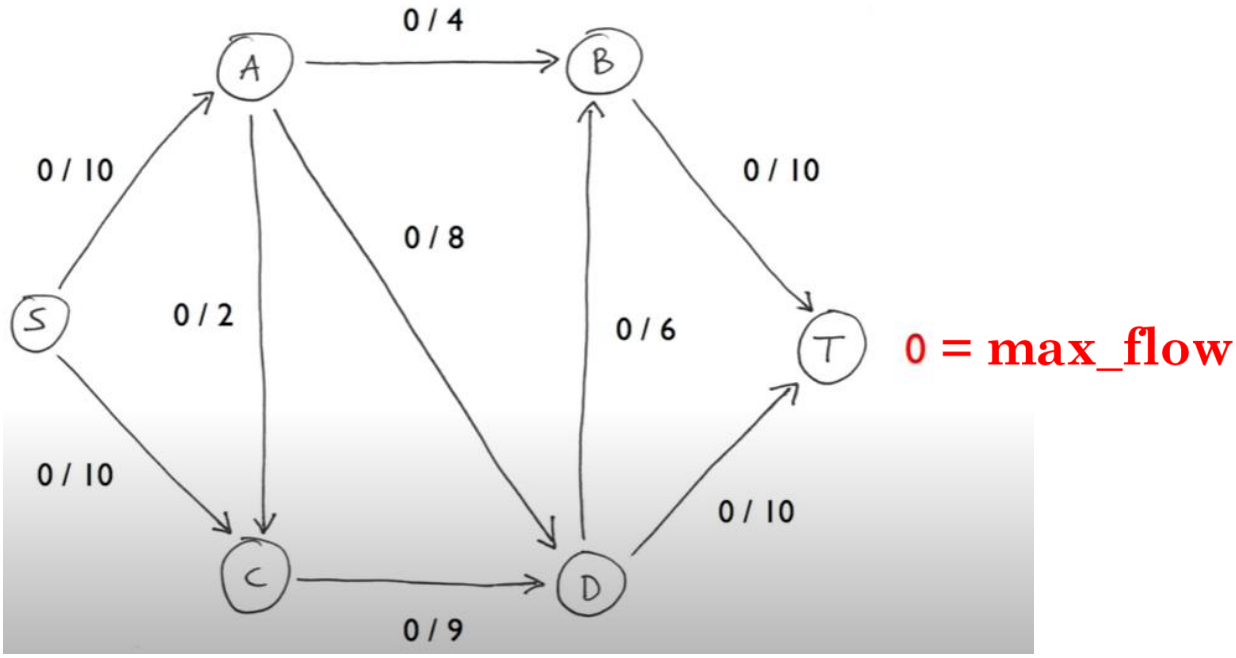
➢ Update *max_flow* by adding the amount of material flowing through the path, i.e. the bottle neck capacity of the path presently being considered, into *max_flow*.

}

3. Return *max_flow*.

For finding a path from source to sink in the residual graph, we use Breadth First Search (BFS) which does the job in linear time.

Example Illustration

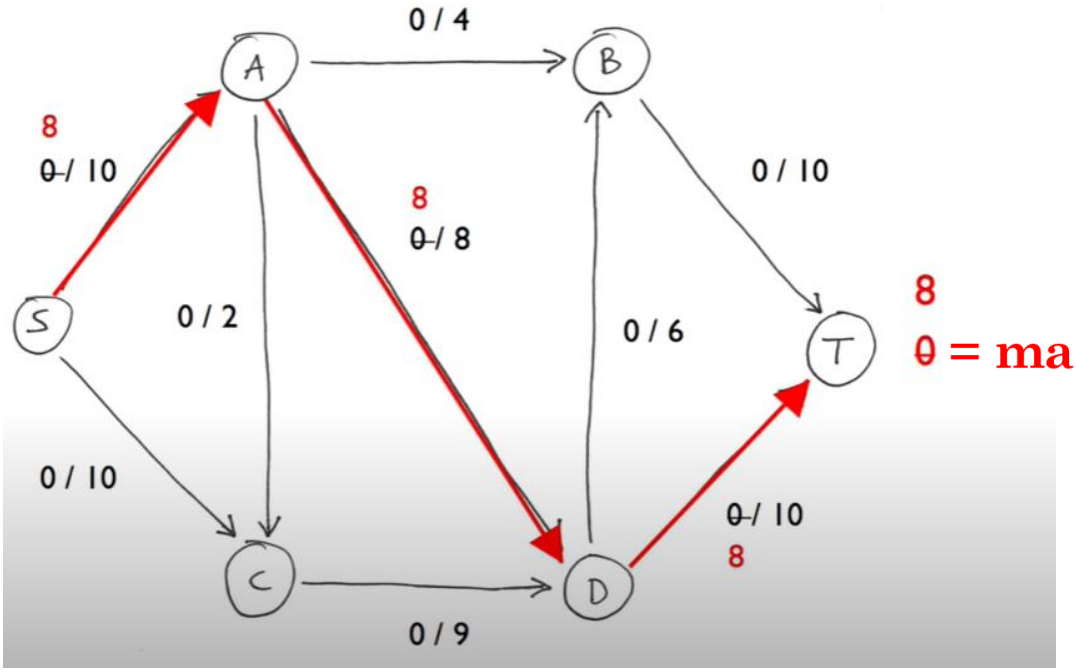


Let us consider the above graph and start choosing augmenting paths from s to t. Remember that on the left side of the slash ‘/’ is the present flow through the edge ‘e’ i.e. Flow(e), and on the right side of the slash is the initial capacity of the edge ‘e’ i.e. Cap(e). The numerical value in red colour on the right hand side of each figure is the value of max_flow after that iteration.

The first path that we choose is S→A→D→T.
The bottle-neck capacity for this path will be
Min{10 - 0, 8 - 0, 10 - 0} => Min{10, 8, 0} => 8.

Now, we will augment each edge on the path with the bottle-neck value. Note that we need to assure that the amount of material flowing into a vertex is equal to the amount of material flowing out of that vertex (except for the source and the sink).

The figure shown below illustrates the same.

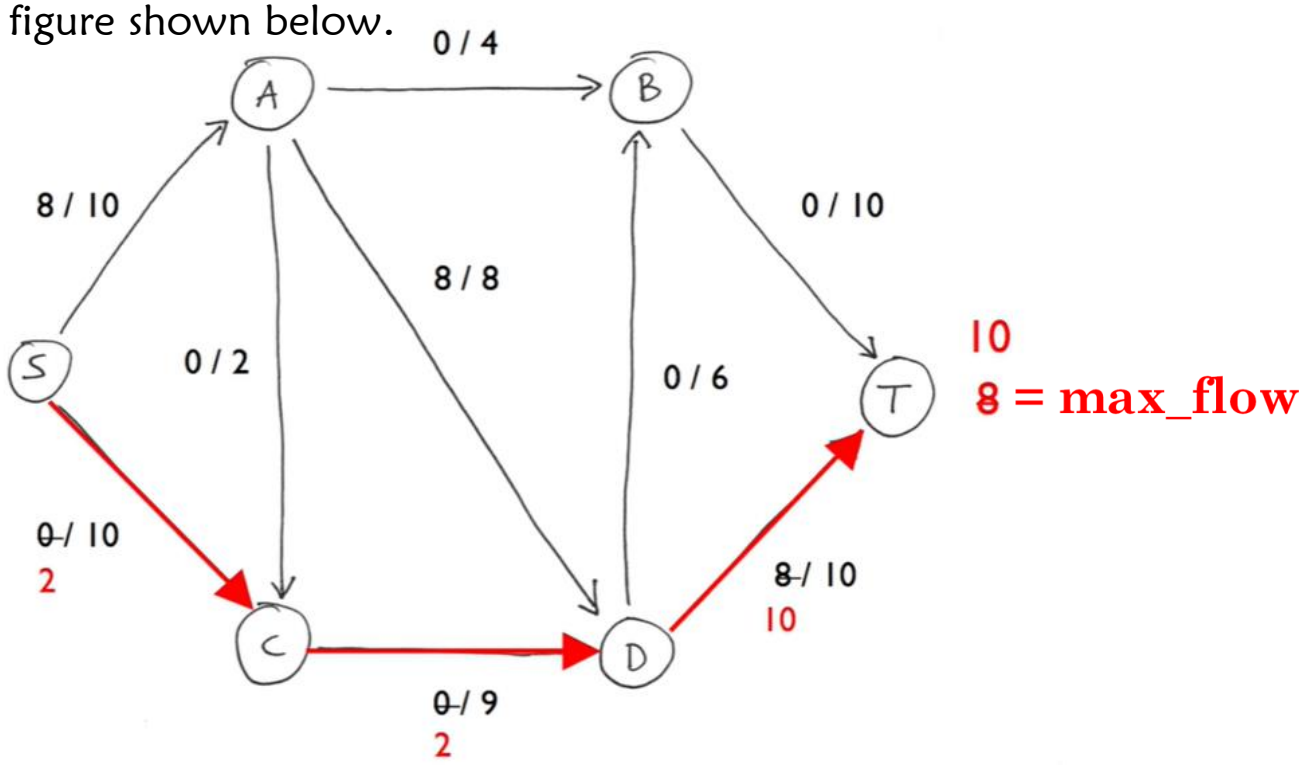


The next path that we choose is S→C→D→T.

The bottle-neck capacity for this path will be

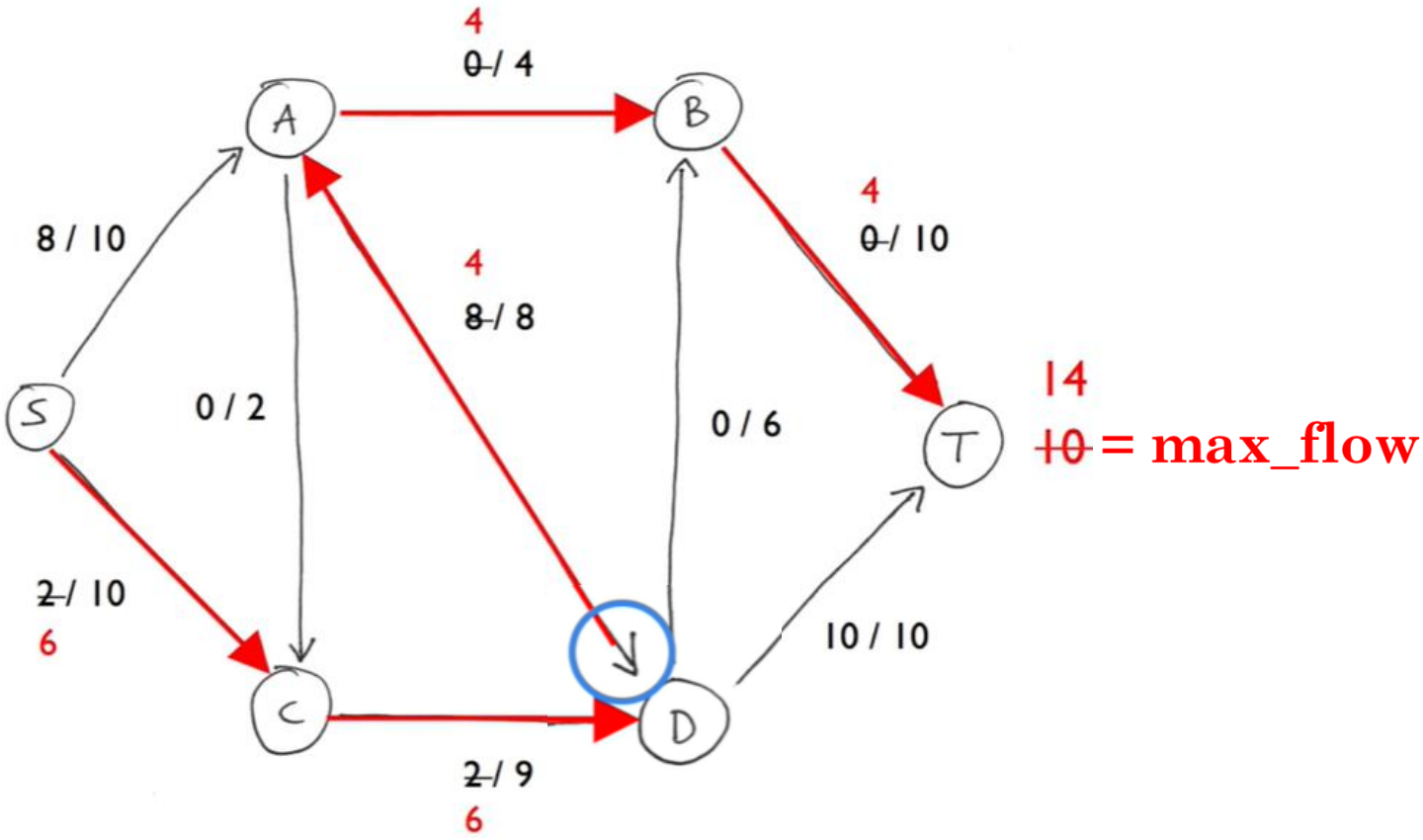
Min{10 - 0, 9 - 0, 10 - 8} => Min{10, 9, 2} => 2.

After augmenting the path S→C→D→T the flow graph looks somewhat like the figure shown below.

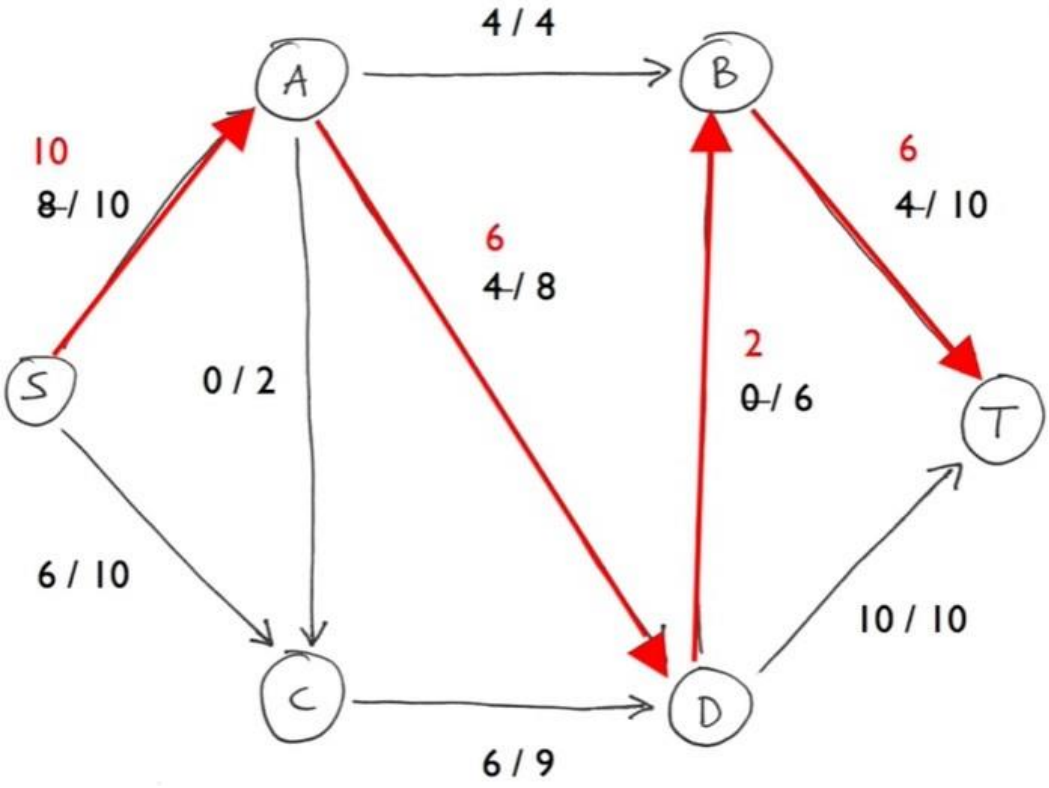


Next, we intentional choose a path with a backwards edge so as to get a clarification on how the algorithm works in this case.

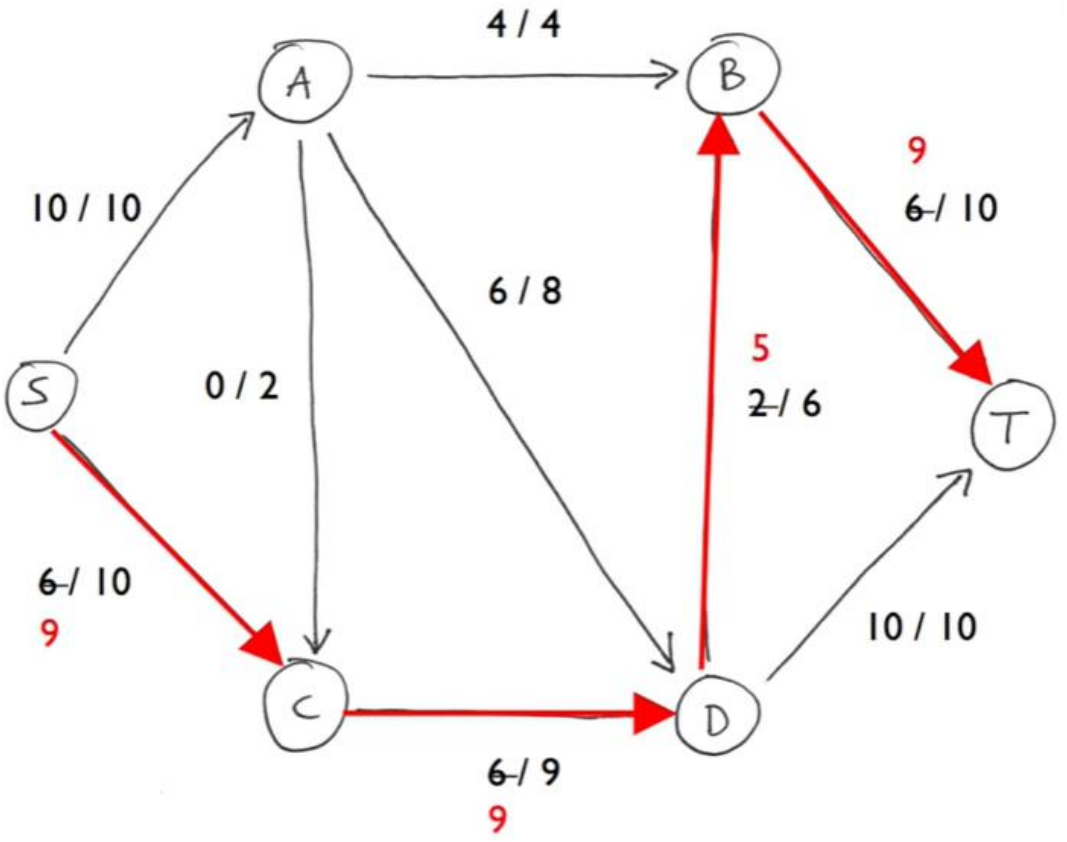
The path that we choose is S→C→D→A→B→T.
The bottle-neck capacity for this path will be
Min{10 - 2, 9 - 2, 8 - 0, 4 - 0, 10 - 0} => 4.
In case of backward edges, we consider the current flow of the edge for obtaining the bottle-neck capacity of the path.
After augmenting the path S→C→D→A→B→T the flow graph looks somewhat like the figure shown below. Add flow to the forward edges and take away flow from the backward edges.



The next path that we consider is S→A→D→B→T.
The bottle-neck capacity for this path will be
Min{10 - 8, 8 - 4, 6 - 0, 10 - 4} => Min{2, 4, 6, 6} => 2.
After augmenting the path S→A→D→B→T the flow graph looks somewhat like the figure shown below.



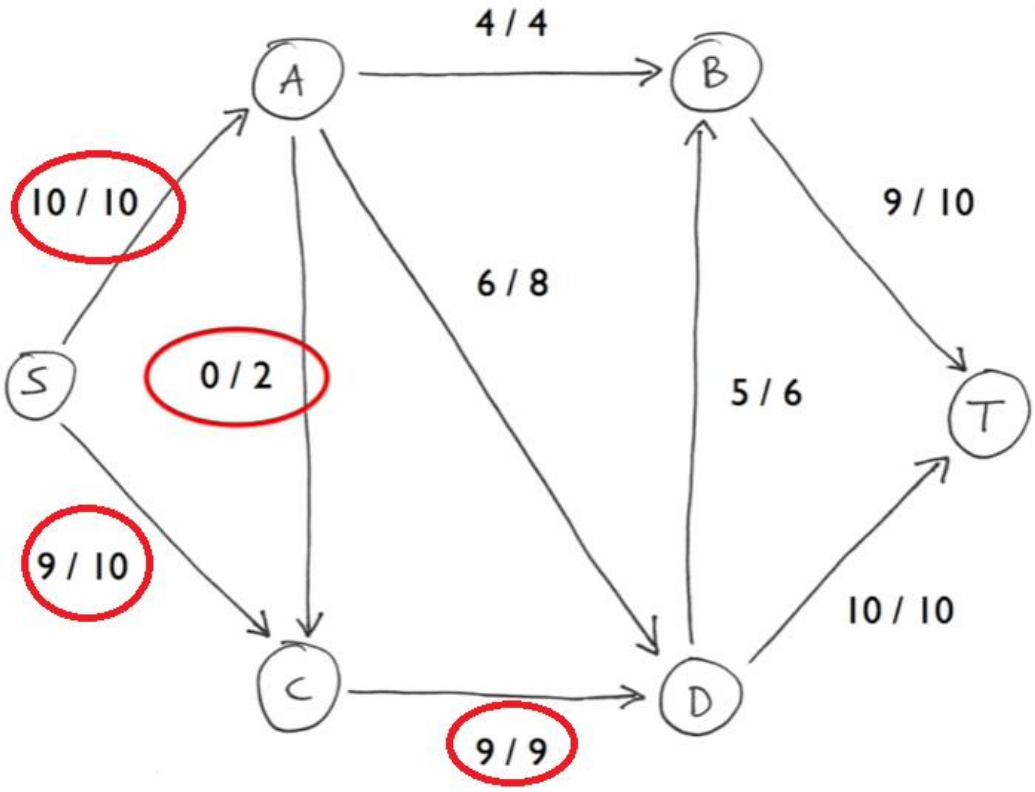
The next path that we consider is S→C→D→B→T.
The bottle-neck capacity for this path will be
Min{10 - 6, 9 - 6, 6 - 0, 10 - 4} => Min{4, 3, 6, 6} => 3.
After augmenting the path S→C→D→B→T the flow graph looks somewhat like the figure shown below.



Now, when we look at the edge S→A we realize that it's completely filled and no flow carrying capacity is left.

The edge S→C has a carrying capacity of 10 – 9 = 1 unit. However, the forward edge C→D is full and the backward edge C→A is empty. Therefore, it is not possible to find any other augmenting path from the source S to the sink T.

The maximum flow value return by our algorithm is 19 which is correct.



Implementation Idea of the Ford-Fulkerson Algorithm

- Residual capacity is 0 if there is no edge between two vertices of the residual graph.

- We initialize the residual graph as the original graph as there is no initial flow and the initial residual capacity is equal to original capacity for all the edges in the graph.

- Using BFS, we find out if there is an augmenting path from the source vertex s to the sink vertex t. This augmenting path will be the shorted possible path between the source vertex s and the sink vertex t in the present residual graph as the algorithm used for computing this path is breadth first search.

- BFS also builds a **Parent[]** array. Using the Parent[] array, we backtrack from the sink vertex t to the source vertex s and get the possible flow through this path by finding the minimum cut or the bottle-neck capacity along that path i.e. the minimum available capacity among all the edges of the path.

- Add the found path flow (the bottle-neck capacity of the path) to overall flow.

- Update residual capacities in the residual graph.
  - Subtract the path flow from all the forward edges along the path.
  - Add the path flow along all the backward edges. We need to **add path flow along backward edges because, later, we may need to send some flow (less than Flow(e)) in reverse direction along the edge**.

Time Complexity Analysis

- In practical scenarios, the maximum-flow problem arises with integral capacities of the edges in the directed flow graph.
  - If the capacities are rational numbers, an appropriate scaling transformation can be used to make them integers.

- Under this assumption, the Ford-Fulkerson algorithm has a time complexity of **O(f.E)**, where **f** is the maximum flow found by the algorithm.
  - Step 1, which is the initialization step, takes O(E) time.
  - Step 2 consists of the execution of the while loop.
    - Finding an augmenting path in a residual graph takes O(E) time.
    - Finding the bottle-neck capacity in the path takes O(E) time again.
    - Augmenting the path with the bottle-neck flow again takes O(E) time.

    Therefore, each iteration of the while loop takes **O(E)** time.

  Since the flow value increases by at least one unit in each iteration of the while loop, we can say that the **while loop gets executed at most f number of times**. Thus, we are able to justify the that the time complexity of a standard implementation of For-Fulkerson algorithm is O(E) {Initialization step} + O(f.E) {while loop execution} which is nothing but **O(f.E)**.

- Edmonds-Karp Algorithm – Time Complexity $O(V.E^2)$
  - The efficiency of the Ford-Fulkerson algorithm can be improved if the augmenting path from the source to the sink computed in each iteration of the while loop is the **shortest possible path from the source to the sink** in the present residual network.
  - Each edge between the source and the sink can be considered to have a unit distance. Therefore, the shortest path between the source and sink is the one which passes through the least number of vertices between the source and the sink.

- The augmented path computed in each iteration of the while loop will be the shorted path in the residual network if we use Breadth First Search to compute the path instead of Depth First Search.
- The time complexity of the Edmonds-Karp algorithm is $O(V.E^2)$.
  - Each edge 'e' in the set of edges of the graph G_f can only be the bottleneck for a path P at most O(V) times.
  - The number of augmenting paths p used in the Edmonds–Karp algorithm is at most O(V.E) because only O(E) pairs of vertices can ever be edges in Gf and each edge can only be the bottleneck O(V) times.
  - Since each iteration of the while loop, including the Breadth First Search takes O(E) time, the total running time for the Edmonds–Karp algorithm is $O(V.E^2)$

Proof of correctness of the Ford-Fulkerson Algorithm

For proving that the Ford-Fulkerson algorithm gives the maximum flow we need to take the help of the max-flow min-cut theorem. This theorem directly implies the correctness of the Ford-Fulkerson algorithm. The proof of this theorem will indeed prove the correctness of the Ford-Fulkerson algorithm for us.

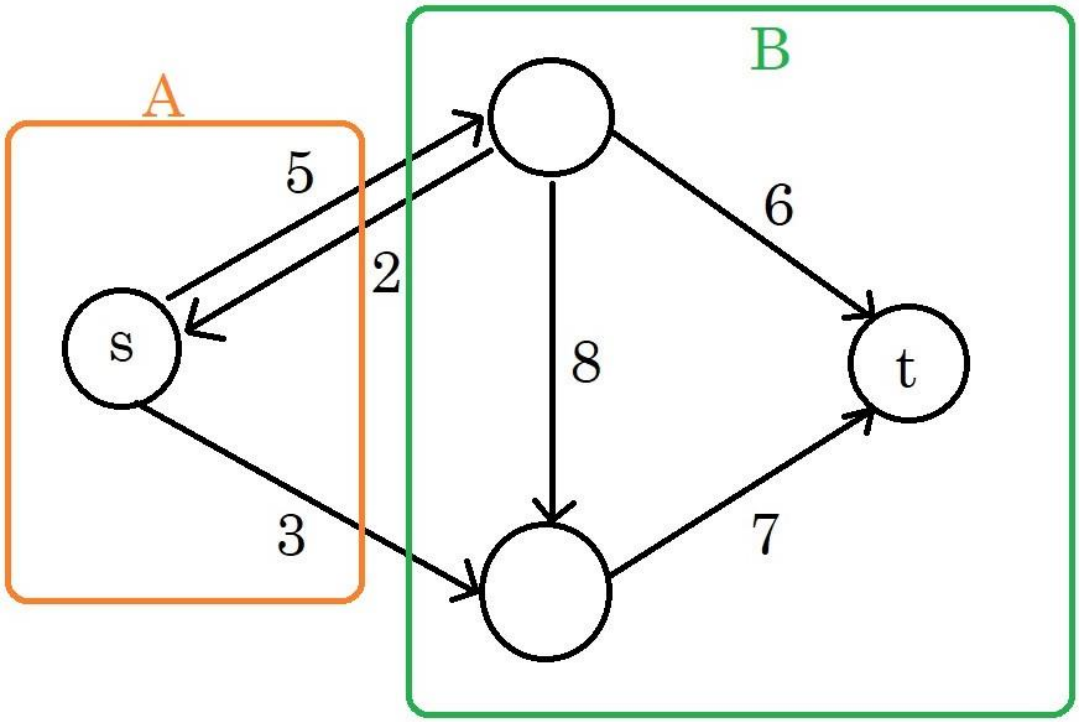
The theorem states that the ***maximum value of an s-t flow is equal to the minimum capacity over all s-t cuts***.

What is an s-t cut?

An s-t cut is a partition of all the vertices of the given graph such that the source vertex s is in set A and the sink t is in set B.

The capacity of an s-t cut is equal to the sum of the capacities of all the edges which are going out of the set A. We ignore the edges which stick into the set A while calculating the capacity of the s-t cut.

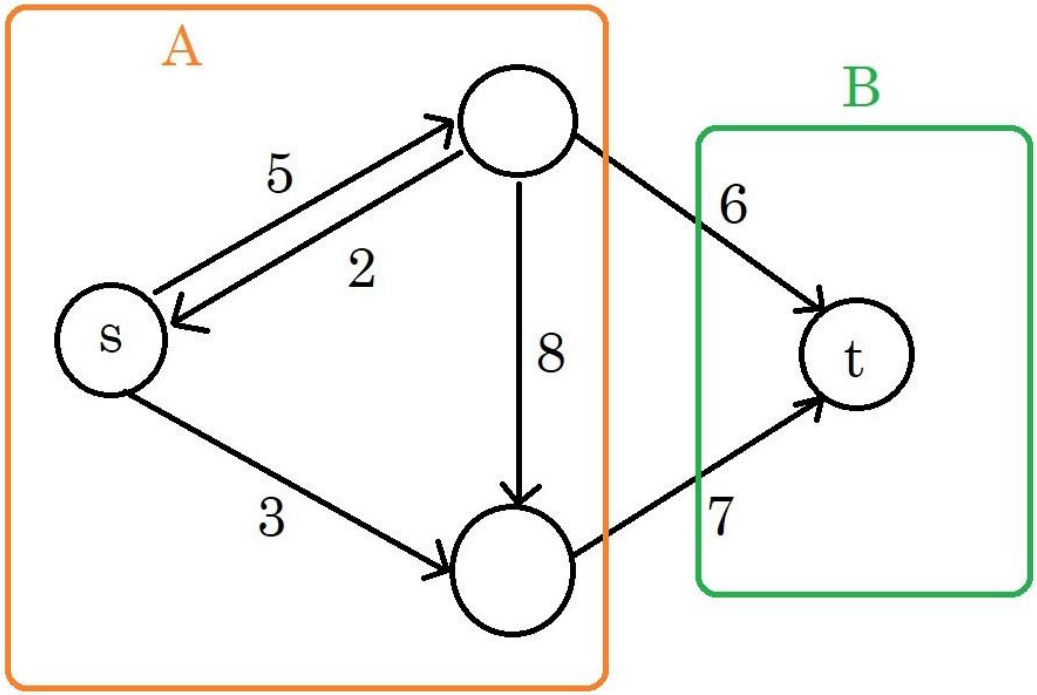
Let us take the example of this graph to understand what an s-t cut is.



The capacity of the s-t cut shown in the image above

Capacity(A,B) = Sum of the capacities of the edges going out of set A

= 5 + 3 = 8



In the above shown diagram, Capacity(A,B) = 6 + 7 = 13

Special Theorem:

For a flow f in a graph G, the following are equivalent.

1. There exists an s-t cut (A,B), such that f = capacity(A,B).
2. f is the maximum flow.
3. G_f has no s-t path.

If this special theorem is true then the Ford-Fulkerson algorithm indeed calculates the maximum flow. The terminating condition of the algorithm is stated in point 3 of the special theorem and the claim is that point 3 is equivalent to point 2 of the special theorem.

We will show the equivalence of the 3 statements of the special theorem by creating a cycle of implications. First we show that 1 implies 2, then we show that 2 implies 3 and then we show that 3 implies 1 which finally proves the max-flow min-cut theorem and also the correctness of the Ford-Fulkerson algorithm.

Implication 1: Point 1 => Point 2:

We assume that there is an s-t cut (A,B) such that the value of f is equal to capacity of the cut (A,B). We need to derive that f is a max flow. This can be done by showing that for all s-t cuts (A,B) and all flows f,

capacity(A,B) >= f.

If no flow can ever exceed the capacity if the s-t cut then by assumption we have a cut whose capacity is equal to the value of the flow. Thus, the inequality holds with equality and f is the maximum flow.

We start with a value of f which is equal to the flow in all the edges going out of the source vertex s. The flow in all the edges sticking into the source is ignored. We can rewrite this sum by looking at all the vertices in the set A.

$$f = \sum_{e \in E_{out}(s)} Flow(e) - \sum_{e \in E_{in}(s)} Flow(e)$$

Remember that A contains the source and not the sink. For each vertex in A, we sum up the flow leaving the vertex and deduct the sum of the flow going into the vertex. Due to the conservation of flow, which is stated in the problem statement of the maximum flow problem, we can say that this summation only added a bunch of zeroes.

$$f = \sum_{e \in E_{out}(s)} Flow(e) - \sum_{e \in E_{in}(s)} Flow(e) = \sum_{v \in A} \underbrace{\left[\sum_{e \in E_{out}(v)} Flow(e) - \sum_{e \in E_{in}(v)} Flow(e) \right]}_{\text{Zero for all other vertices in A except s due to conservation of flow.}}$$

The intent behind this was to correlate the value of the flow with the value of the capacity of the cut. We will rewrite the sum again considering how each of the 4 types of edges which impact the sum.

- Edges with both the start point and the end point in A contribute once positively when we look at v as it sticks out of v. And once negatively with w because if sticks into w. So we see that the orange edge will show twice in the sum, once positively and once negatively which cancels out in the end.

- Edges with both the start point and the end point in B don't show up in the sum at all as we are considering only the edges of vertices in set A.

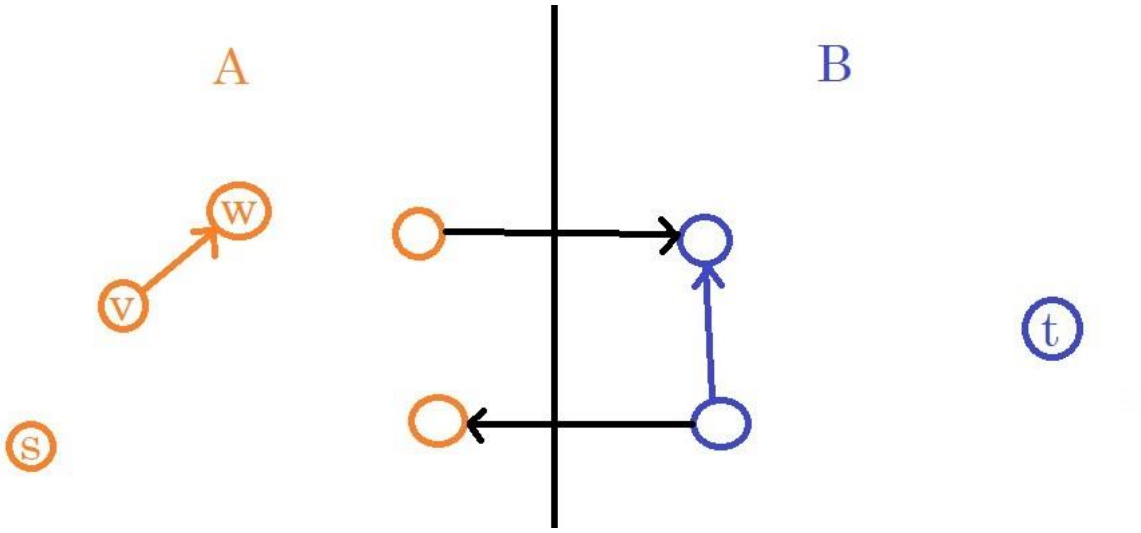
- Edges from A to B will contribute only positively to the sum because they are outgoing edges from the vertices in A to the vertices in B and we are summing over all the vertices in A and not in B.

- Edges from B to A will contribute only negatively to the sum because they are the edges having their end points in A and start points in B and we are summing over all the vertices in A and not in B.

Therefore, edges with both end points in A or with end point in B contribute 0 to the sum. We can rewrite the sum as:

Flow through all the edges going out of A

– Flow through all the edges going into A.



All the terms in the left sum are smaller than or equal to the capacities of the respective edges whose flow is being considered. As flow cannot be negative, each term in the right sum is greater than or equal to 0. So if we omit the right sum and replace the left sum with the sum of the capacities of the edges going out of A, the resulting sum must be greater than or equal to the capacity(A,B) of the s-t cut (A,B).

$$f = \sum_{e \in E_{out}(s)} Flow(e) - \sum_{e \in E_{in}(s)} Flow(e) = \sum_{v \in A} \left[\sum_{e \in E_{out}(v)} Flow(e) - \sum_{e \in E_{in}(v)} Flow(e) \right]$$

$$= \sum_{e \in E_{out}(A)} Flow(e) - \sum_{e \in E_{in}(A)} Flow(e) \leq \sum_{e \in E_{out}(A)} Cap(e) = Capacity(A, B)$$

<=Cap(e) >= 0

We have shown that for any flow and any s-t cut the value of the flow is less than the capacity of the s-t cut. This proves our first implication.

Implication 2: Point 2 => Point 3:

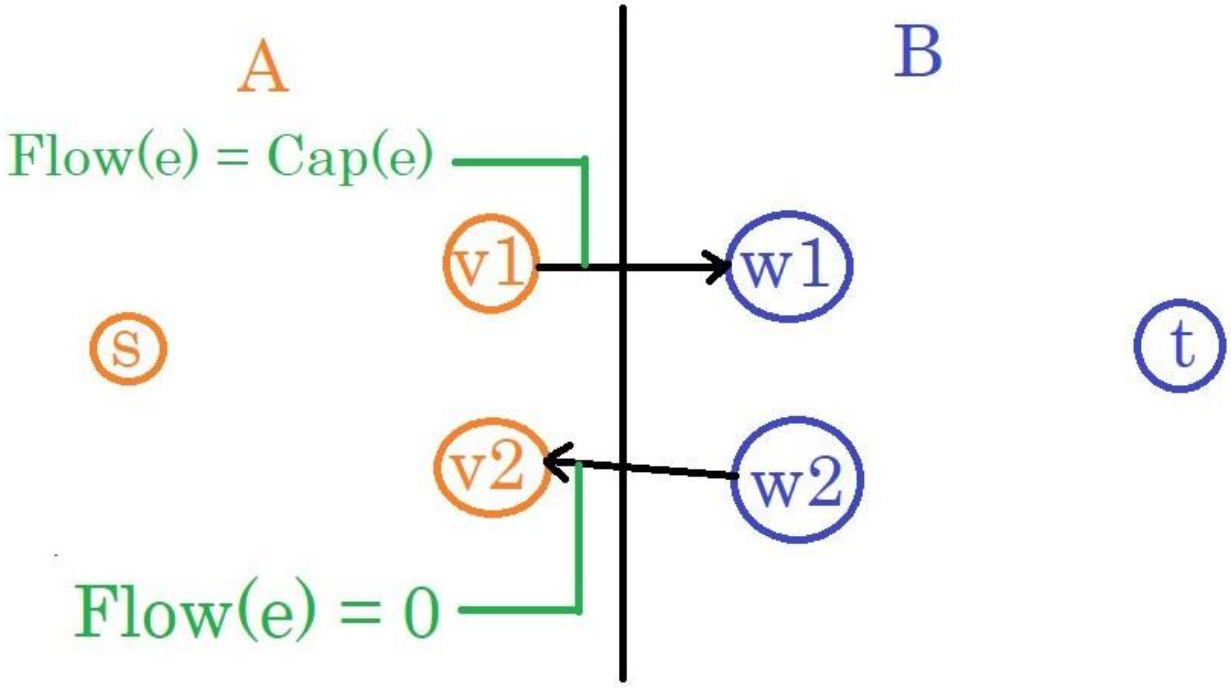
If f is the max flow, then the residual graph has no augmenting path from the source vertex s to the sink vertex t.

This implication can be proved using contrapositive. If there is an augmenting path from the source vertex s to the sink vertex t, then the Ford-Fulkerson algorithm does not stop and the value of the max flow can be increased further. Therefore, f cannot be the maximum flow in this case.

Implication 3: Point 3 => Point 1:

If the residual graph has no augmenting path from s to t, then there exists an s-t cut (A,B) such that f = capacity(A,B).

To prove this implication, we will construct a cut by putting all the vertices that are reachable from the source into set A and all the remaining vertices into set B. Reachable, here, means that we may take only the edges which have strictly positive residual capacity or, in other words, we only take the edges which are not fully saturated. Note that this partition is indeed an s-t cut because by assumption there is no augmenting s-t path in the residual graph so the sink is not reachable from the source. Together with the fact that the source is, trivially, reachable from itself we get that the source vertex s is in set A and the sink vertex t is in set B and, hence, (A,B) is a valid s-t cut.



An important observation is that all the edges going from A to B must be saturated because, if they are not, then, by construction, the vertex at the end of those edges should be in A as the vertex w_1 will be reachable from the

source and we have all the vertices reachable from the source into set A. But that's not the case as the vertex w_1 is in set B.

For all edges going from B to A the flow must be 0. If that's not the case then we will have a reverse edge in the residual graph with a strictly positive residual capacity going back from v_2 to w_2. w_2 will then be reachable from the source s and hence should have been in set A which it isn't.

Let us have another look at the inequality from the first implication where we showed that the flow is bounded above by the capacity of s-t cut. We will observe that for this particular s-t cut the inequality will hold with equality as all edges going out of A are fully saturated and all the edges sticking into A from B have 0 flow. Therefore, we have successfully constructed a cut with the capacity of the cut being equal to the value of the flow.

$$f = \sum_{e \in E_{out}(s)} Flow(e) - \sum_{e \in E_{in}(s)} Flow(e) = \sum_{v \in A} \left[\sum_{e \in E_{out}(v)} Flow(e) - \sum_{e \in E_{in}(v)} Flow(e) \right]$$

$$= \sum_{e \in E_{out}(A)} Flow(e) - \sum_{e \in E_{in}(A)} Flow(e) = \sum_{e \in E_{out}(A)} Cap(e) = Capacity(A, B)$$

$= Cap(e)$ $= 0$ Hold with equality

This completes the proof of the theorem. We have shown that if a residual graph has no s-t path, which is the break condition for the Ford-Fulkerson algorithm, then the value of the flow is indeed the max flow. We have also shown that all flows are bounded above by the capacities of all cuts and that if f is the max flow then we can construct a cut of the residual graph such that the capacity of the cut is equal to the maximum flow in the graph.

As no cut may have smaller capacity than any flow this cut must also be the minimum cut which is exactly what the max-flow min–cut theorem states.

Practical Life Applications

- The edges of the flow network directed graph can represent roads with cars. The capacity can be considered as the number of cars which the roads can sustain in traffic.
- The edges of the flow network directed graph can be pipes carrying water. The capacity of the edge will represents the maximum volume of water that can flow the pipe.
- The edges of the flow network directed graph can be wires carrying electric current. The capacity of the edge will represents the maximum electric current that can drift through the wire at any time.

References

- YouTube Videos:
  - https://www.youtube.com/watch?v=Tl90tNtKvxs&t=246s&ab_channel=MichaelSambol
  - youtube.com/watch?v=Iwc3Uj4aaF4&ab_channel=TutorialsPoint%28India%29Ltd.
  - https://www.youtube.com/watch?v=3LG-My_MoWc&ab_channel=TutorialsPoint%28India%29Ltd.
  - https://www.youtube.com/watch?v=LdOnanfc5TM&t=427s&ab_channel=WilliamFiset
  - https://www.youtube.com/watch?v=YD3heMex_uw&t=111s&ab_channel=SimonReuter

- Books:
  - Introduction to Algorithms – CLRS
  - Algorithm Design – Tardos

- Research papers:

- http://cs.williams.edu/~shikha/teaching/spring20/cs256/lectures/Lecture19.pdf
- https://courses.engr.illinois.edu/cs473/fa2010/Lectures/lecture17.pdf
- http://people.qc.cuny.edu/faculty/christopher.hanusa/courses/Pages/634fa09/notes/flows2.pdf
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec13.pdf
- https://people.cs.umass.edu/~sheldon/teaching/mhc/cs312/2013sp/Slides/Slides19%20-%20Ford-Fulkerson.pdf

➢ Other valuable online sources:

- https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/
- https://www.geeksforgeeks.org/max-flow-problem-introduction/
- https://en.wikipedia.org/wiki/Ford%E2%80%93Fulkerson_algorithm

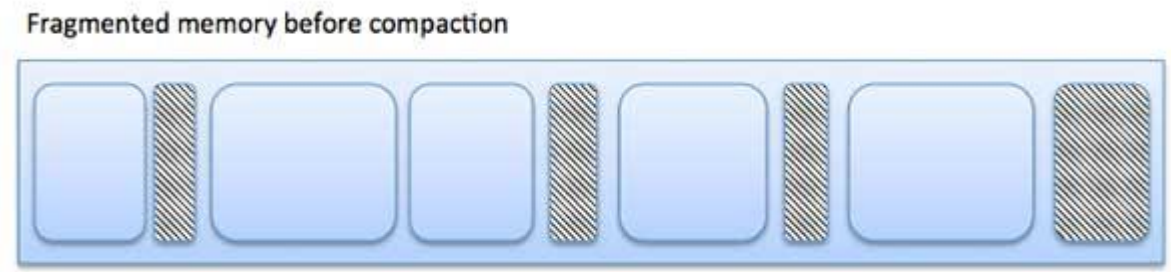
APPLICATION–3

COPYING GARBAGE COLLECTION USING CHENEY'S ALGORITHM

What is Garbage Collection?

It is a type of memory management. It automatically cleans up unused objects and pointers in memory, allowing the resources to be used again. *A garbage collector* , or just *collector*, attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program.

Problem



- The shaded fragments of memory are unused/unwanted(garbage) pieces of memory.
- The issue is that this memory cannot be used/occupied by a file having size greater than the size of each fragment(shaded).
- This is technically a waste of memory.

Solution

All the useful files (occupied useful memory) is stored in continuous manner in the memory leaving a single continuous chunk of unused/unwanted memory, allowing us to store a file in the memory which was not possible earlier.

Copying Garbage Collection – Cheney's Algorithm

- **Cheney's algorithm** was first described in a 1970 ACM paper by C.J. Cheney.
- It is a method of *tracing garbage collection* in computer software systems.
- In this scheme, the heap is divided into two equal halves, only one of which is in use at any one time. Garbage collection is performed by copying live objects from one semispace (the from-space) to the other (the to-space), which then becomes the new heap. The entire old heap is then discarded in one piece.

Some Useful Definitions

- **Heap Memory** - Portion of memory where *dynamically allocated* memory resides.

- **Tracing garbage collection** – is a form of *automatic memory management* that consists of determining which objects should be deallocated ("garbage collected") by tracing which objects are *reachable* by a chain of references from certain "root" objects, and considering the rest as "garbage" and collecting them.

- **Live Objects** - Objects still in use .Objects that are currently present on Run-Time Stack or being referenced from somewhere else.

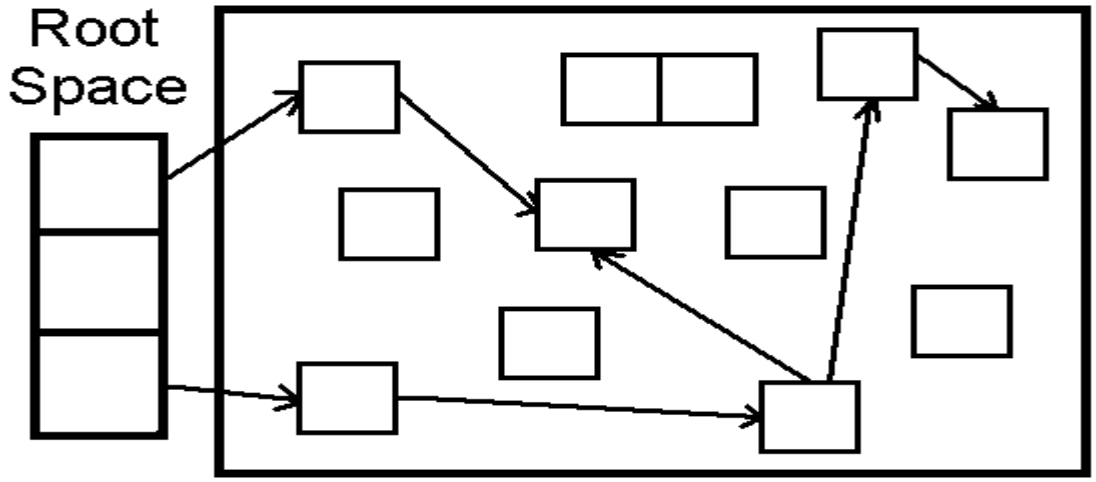
- **Root Space/Root Set** – Stack storing references to live objects which may have references to other live objects.

- **Dead/Garbage Objects** : objects are no longer used or referenced from anywhere.

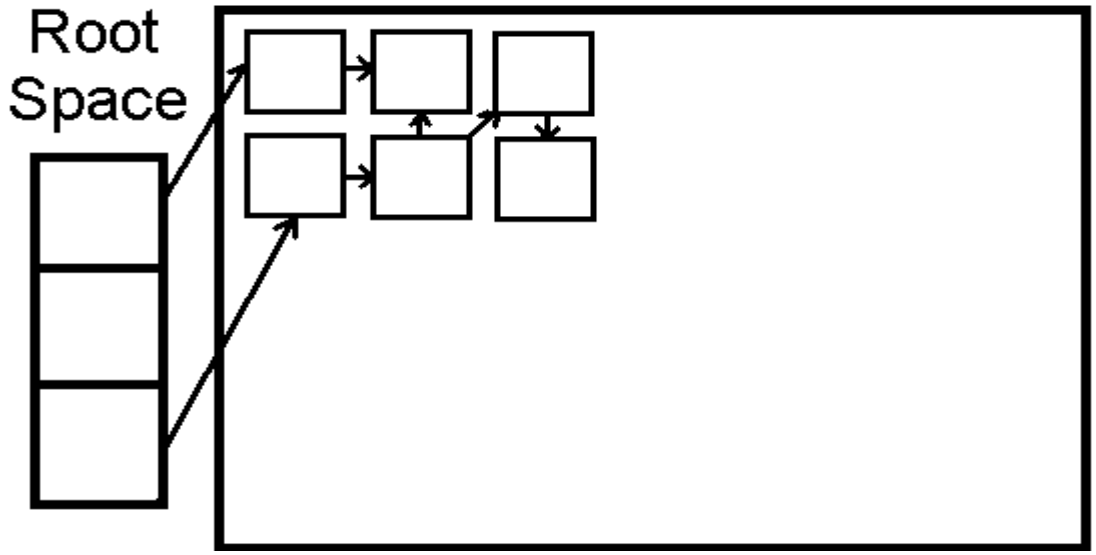
Aim of Algorithm

Compaction, *copy live objects* next to each other .to effectively use the unused memory / memory claimed by garbage objects.

Memory –



After performing copying garbage collection using Cheney's algorithm



- Live objects are copied next to each other and Garbage objects are removed(left behind/not copied) , freeing up more space in the memory.

- Challenges : Object Addresses Change
  - Incompatible with pointer arithmetic.
  - Must track location Changes:
    - May have multiple references to same object
    - Must all point to same address after the COPY.

✓ Cheney's algorithm addresses all these Challenges.

Cheney's Algorithm Overview

- Uses two heaps(semi-spaces)
  - From-Space, where current objects are
  - To-Space, where objects will be after Compaction.

- Maintain two pointers in To-Space.
  - NEXT: Start of unallocated Space.
  - SCAN: Next location to trace/scan.

- Whenever we find a live Object (in root-set):
  - Copy referenced Object to To-Space, grow NEXT , also copy it's references.

- Once we are Done with root-set:
  - Set Scan to the start of To-Space.
  - While SCAN < NEXT:

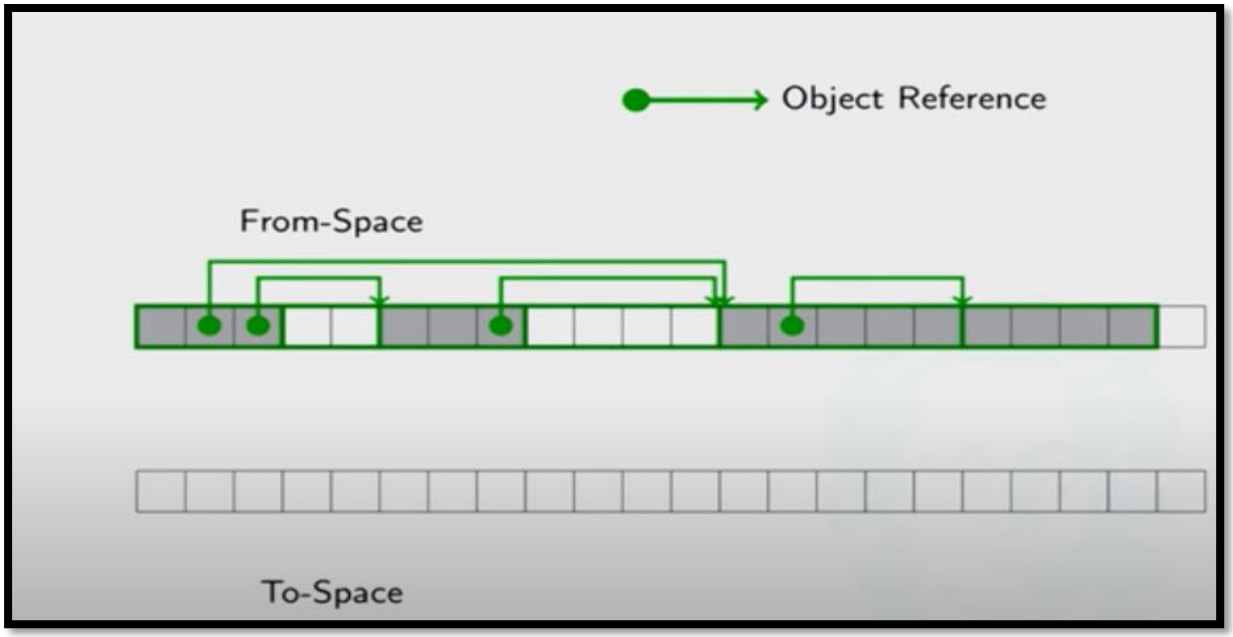
    Points to From-Space object? Copy Object, grow NEXT.

    Advance SCAN.
- Use *forwarding pointer* to avoid copying more than once.

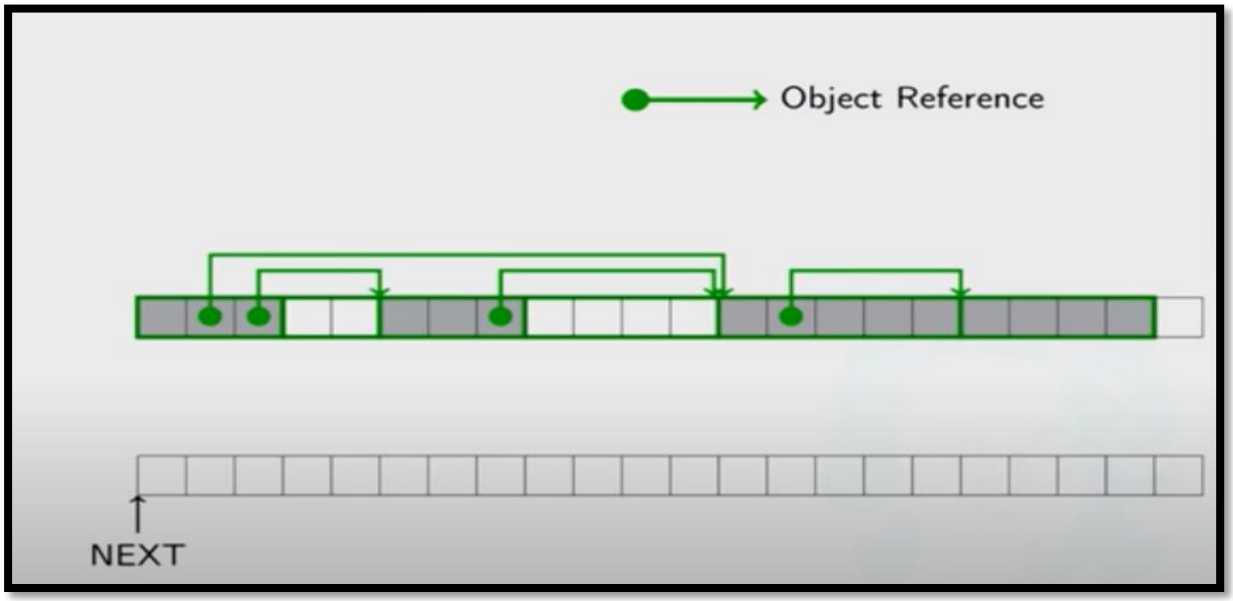
❖ *Forwarding pointer – Pointer that points from old location (in From Space) to new location (in To-Space) for each object that is copied.*

Example Illustration of Algorithm

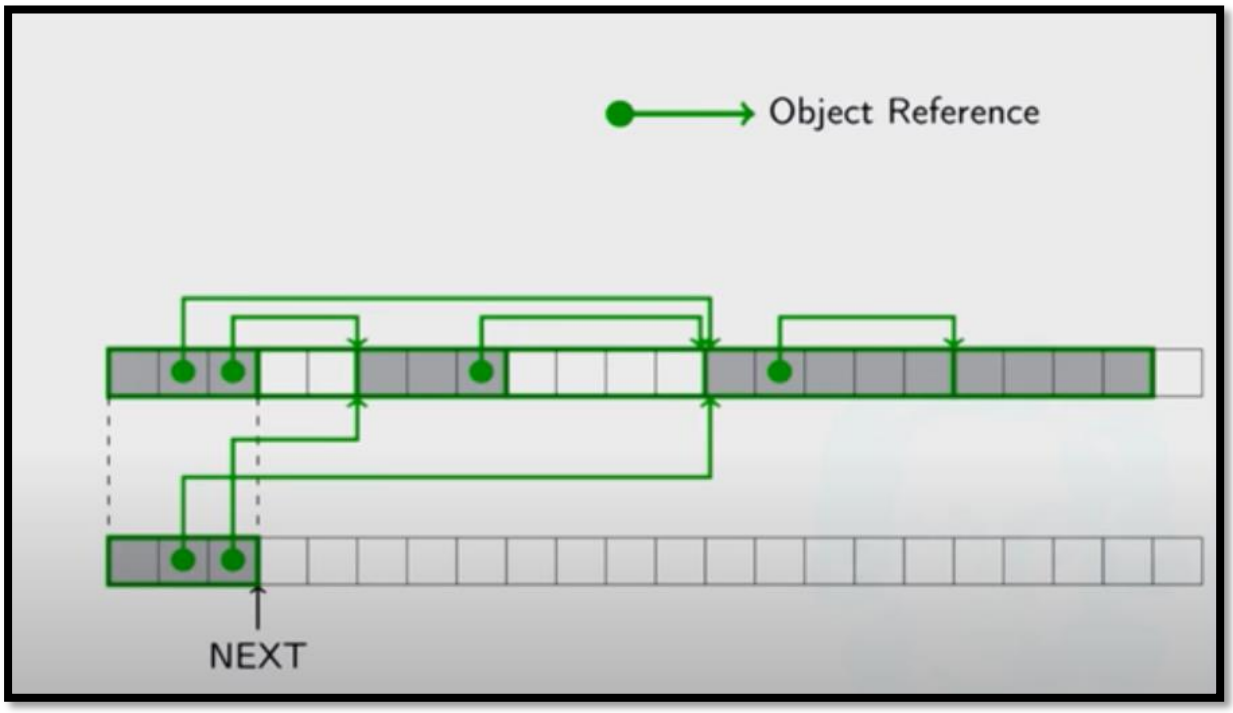
- First From-Space and To-Space are swapped.



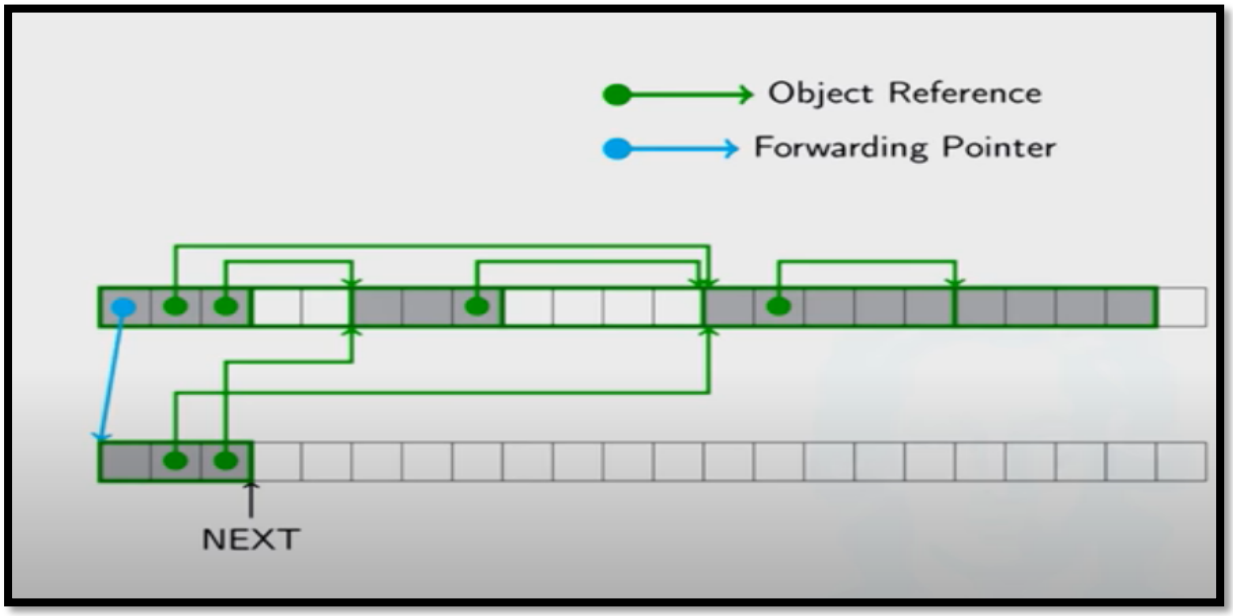
- *NEXT pointer points to Start of To-Space.*



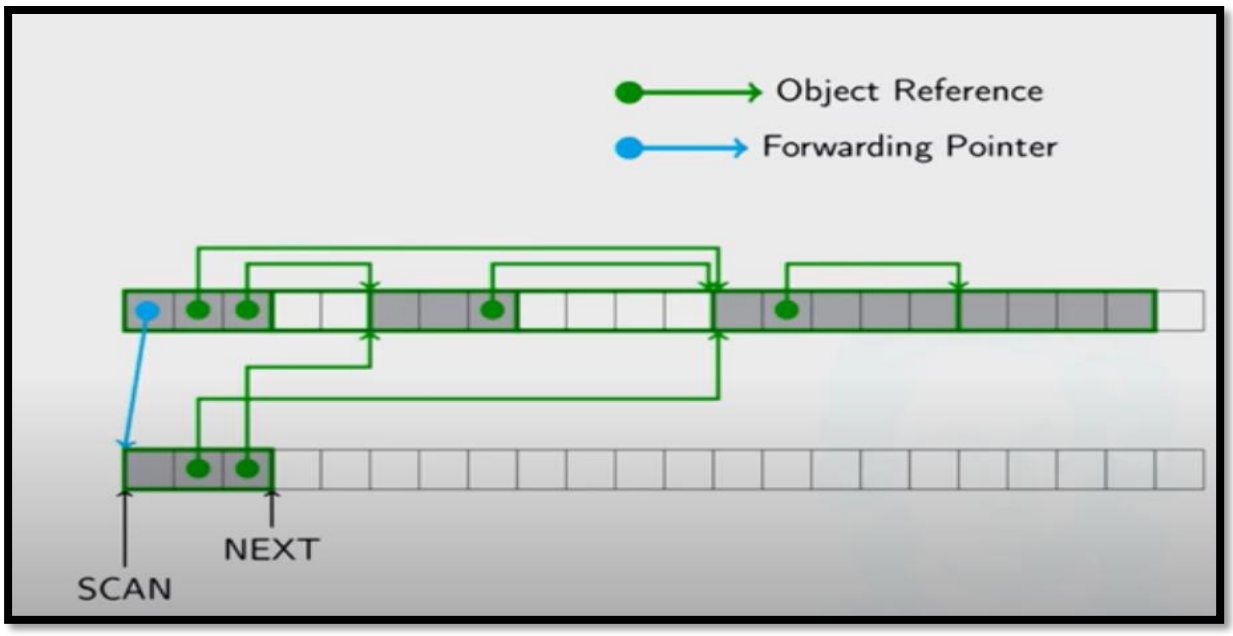
- Object(s) from Root-Set is(are) Copied.
- While Copying an object it also copies all the references of that object.
- **Note that** *these references are from the To-Space to the From-Space.*



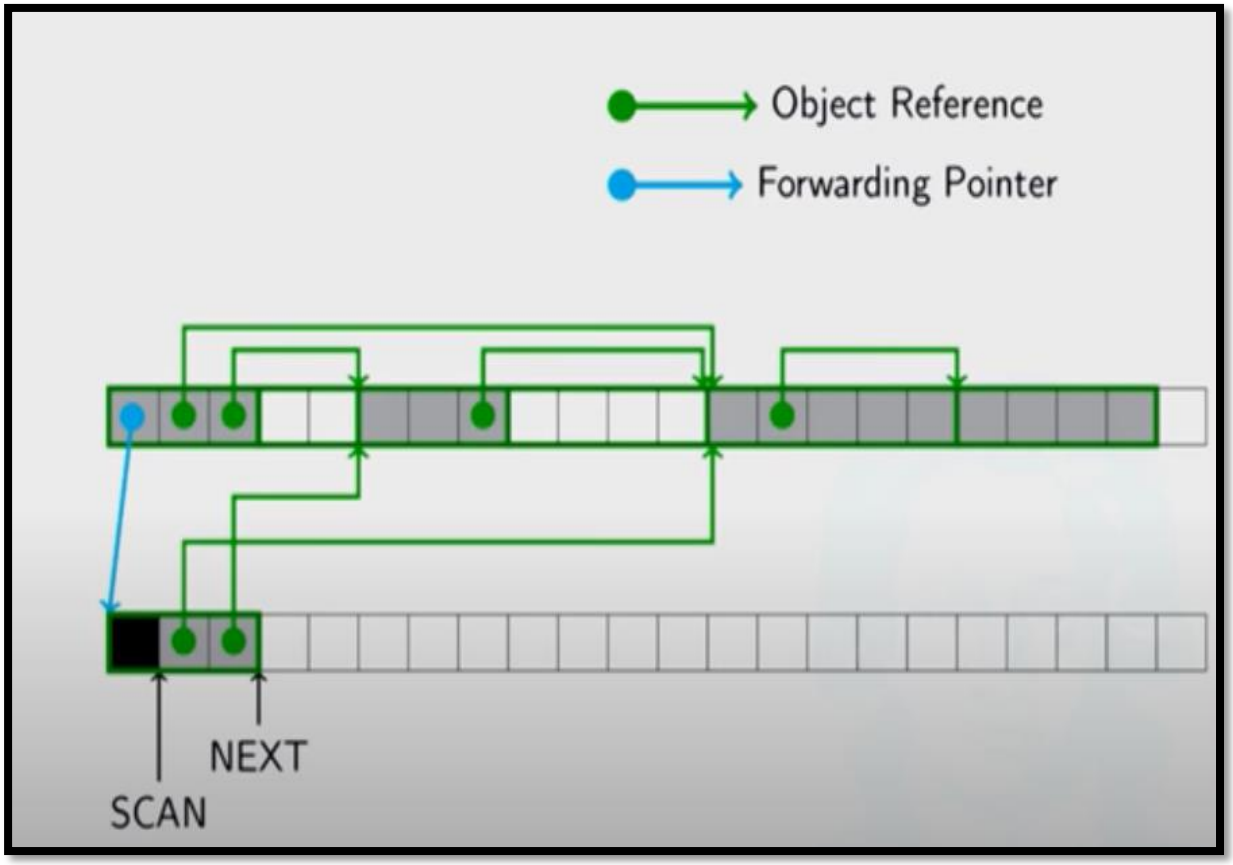
- Forwarding pointer is Added.



- When all live objects from the root-set are copied, *SCAN* pointer starts to the start of the To-Space.



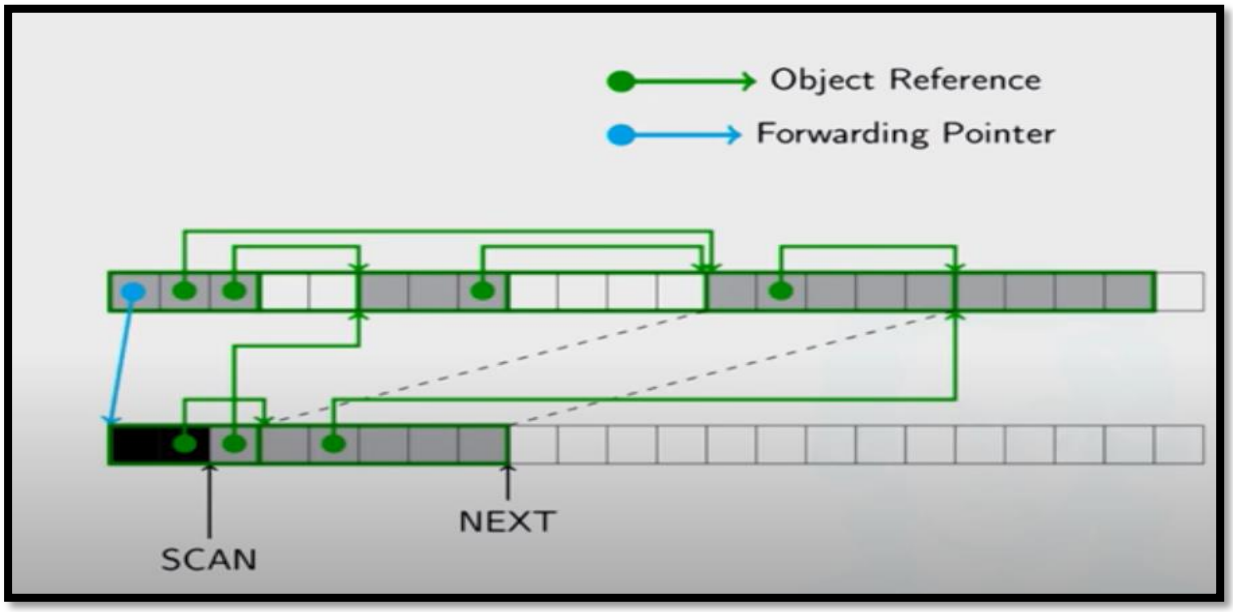
- *SCAN* pointer iterates over the next word/field.
- *Here First word does not have any references, so nothing needs to be done. SCAN pointer moves to next word.*



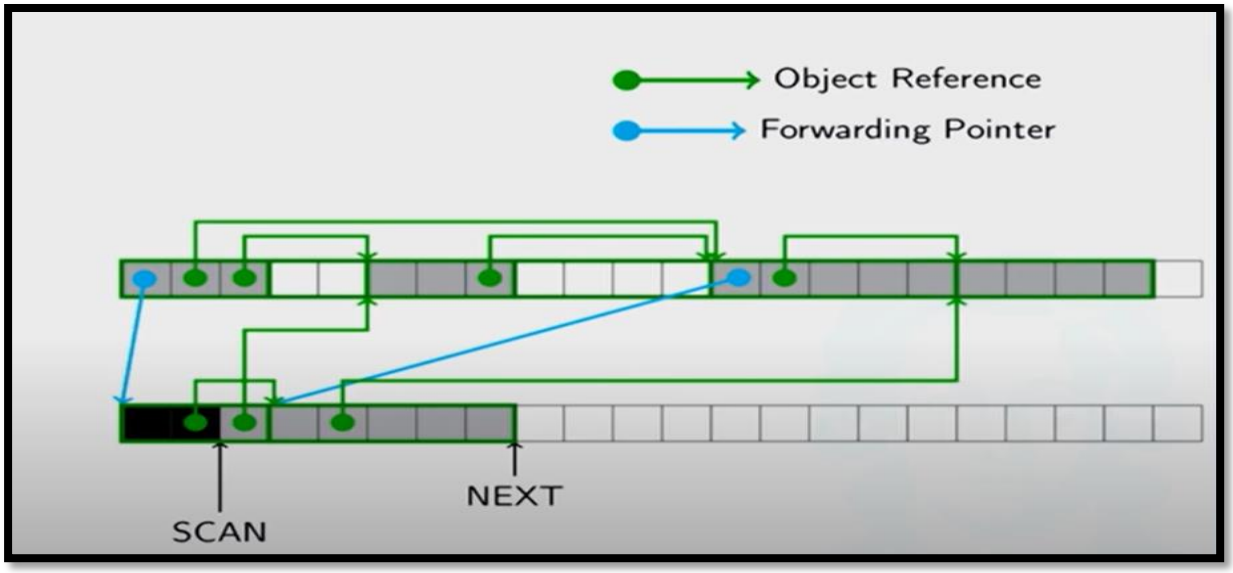
- The 2nd word has a reference to an object in From-Space, so that object is copied into the To-Space.

- Reference that was earlier from the To-Space to the From-Space, is updated and is now within the To-Space only.

- *NEXT* pointer keeps advancing to the first empty slot (next to last occupied slot).



- Forwarding pointer is added for the newly copied object.

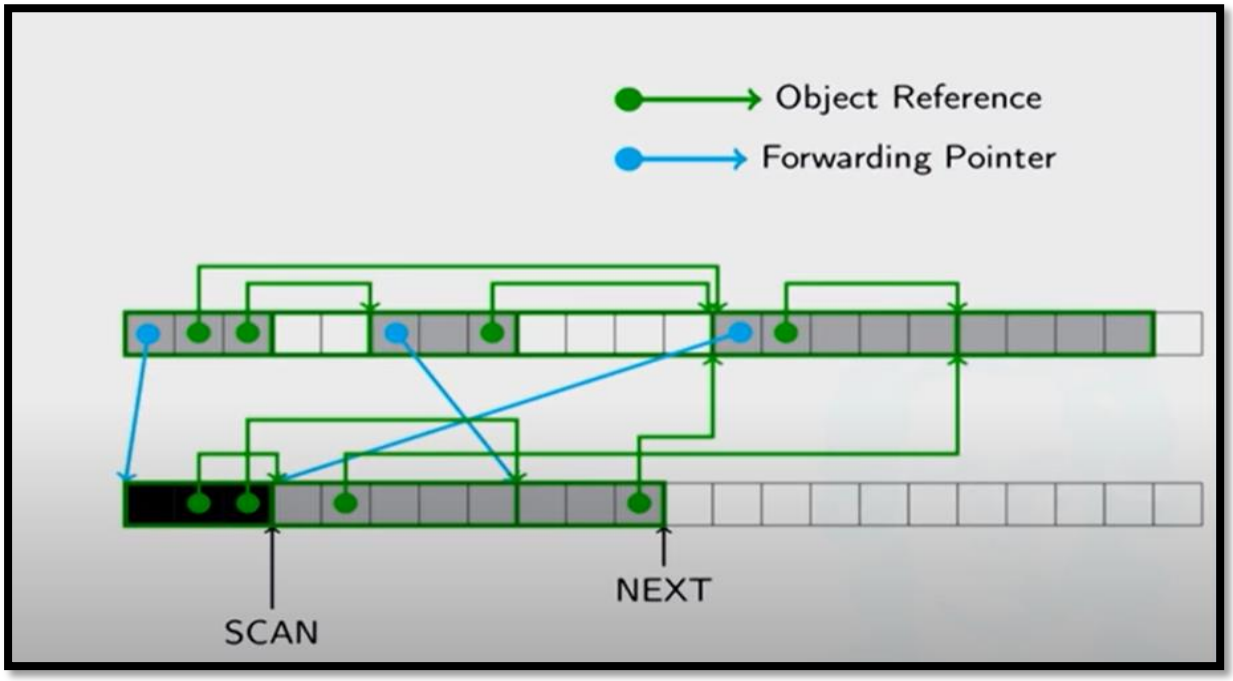


- Now *SCAN* pointer points to 3rd word of the first object .

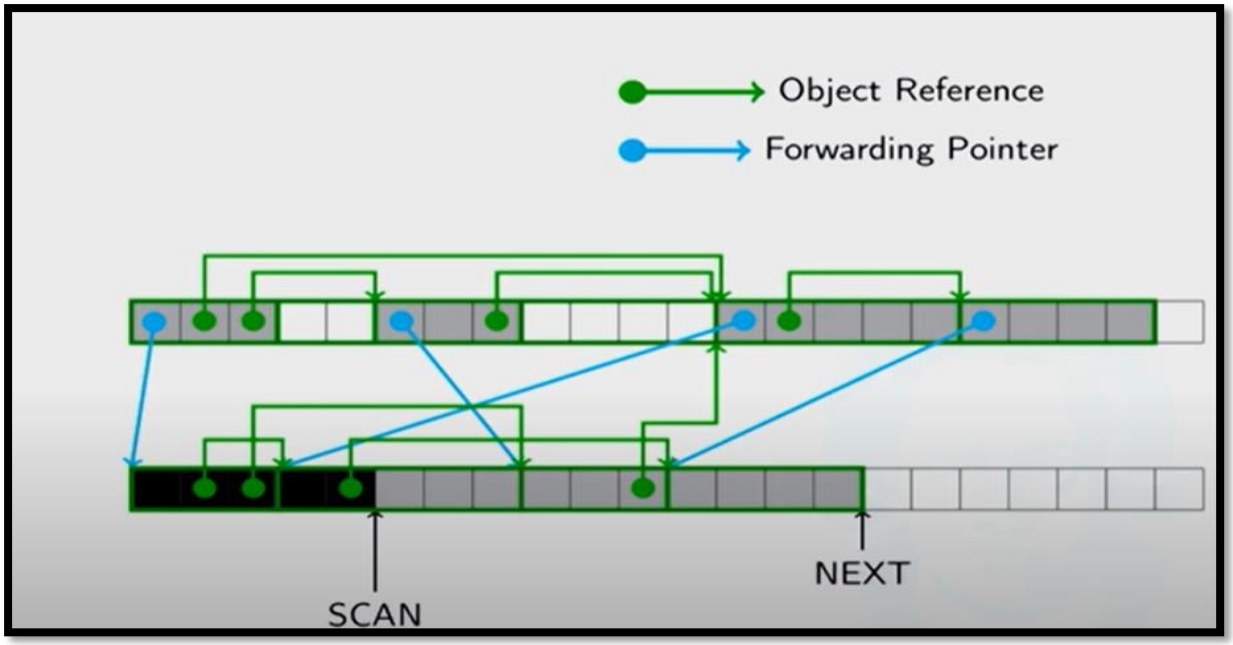
- *Same steps repeat.*
  - The object it was pointing to is copied into the To-Space along with its references.

  - Forwarding pointer is added.

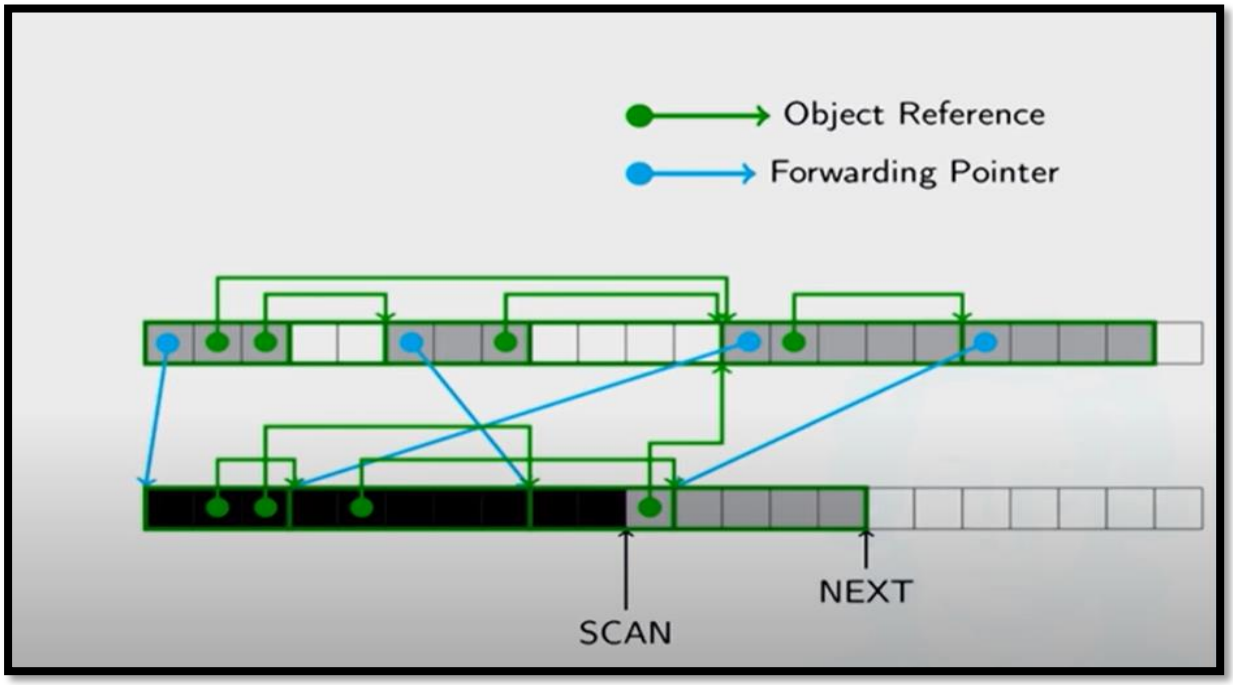
  - The references are updated (Earlier reference was from the To-Space to From-Space, now it is within To-Space only as Object is copied in To-Space).



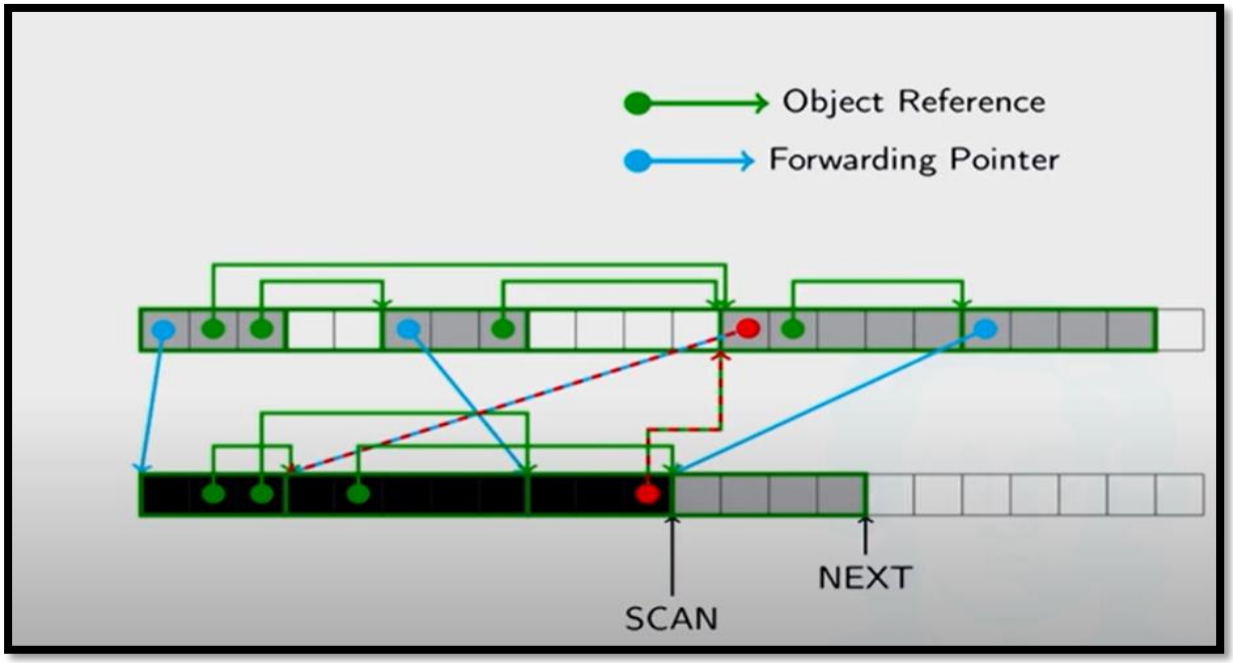
- *SCAN* pointer simply keeps iterating forward until it points to a word/field which has a reference to an object.
- *SCAN* pointer now points to 5th word, Same steps repeat (same as for 3rd word).



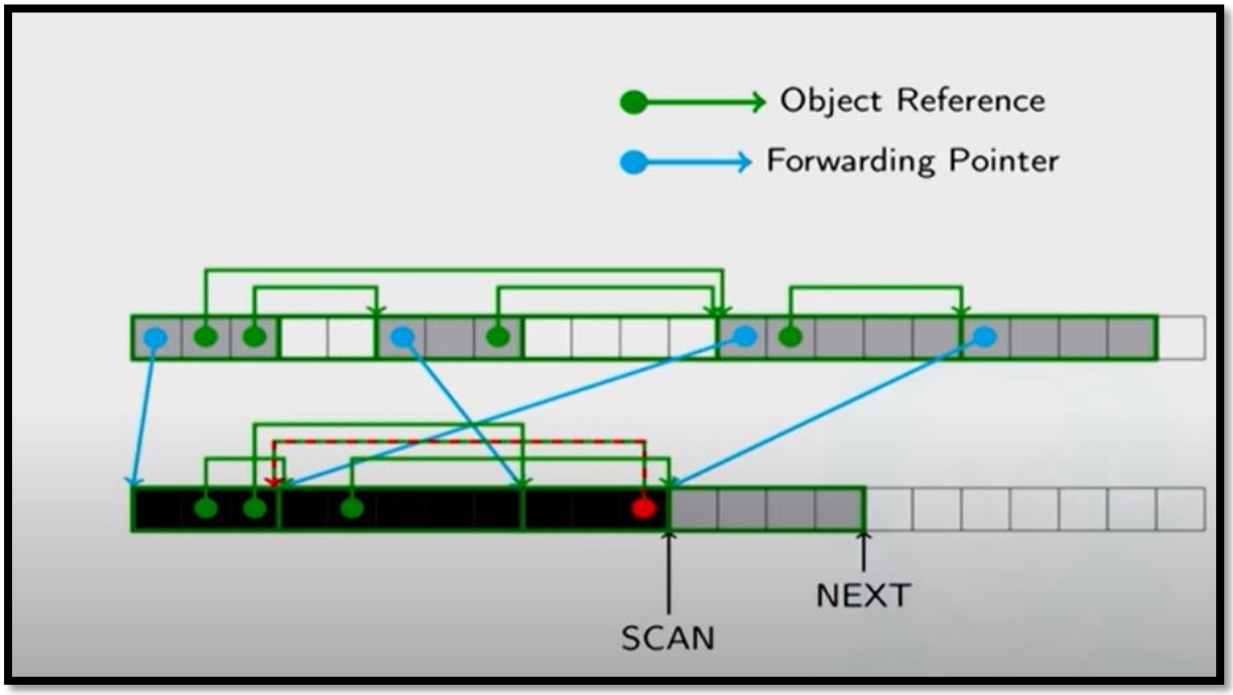
- *SCAN* pointer simply keeps iterating forward until it points to a word/field which has reference to an object.



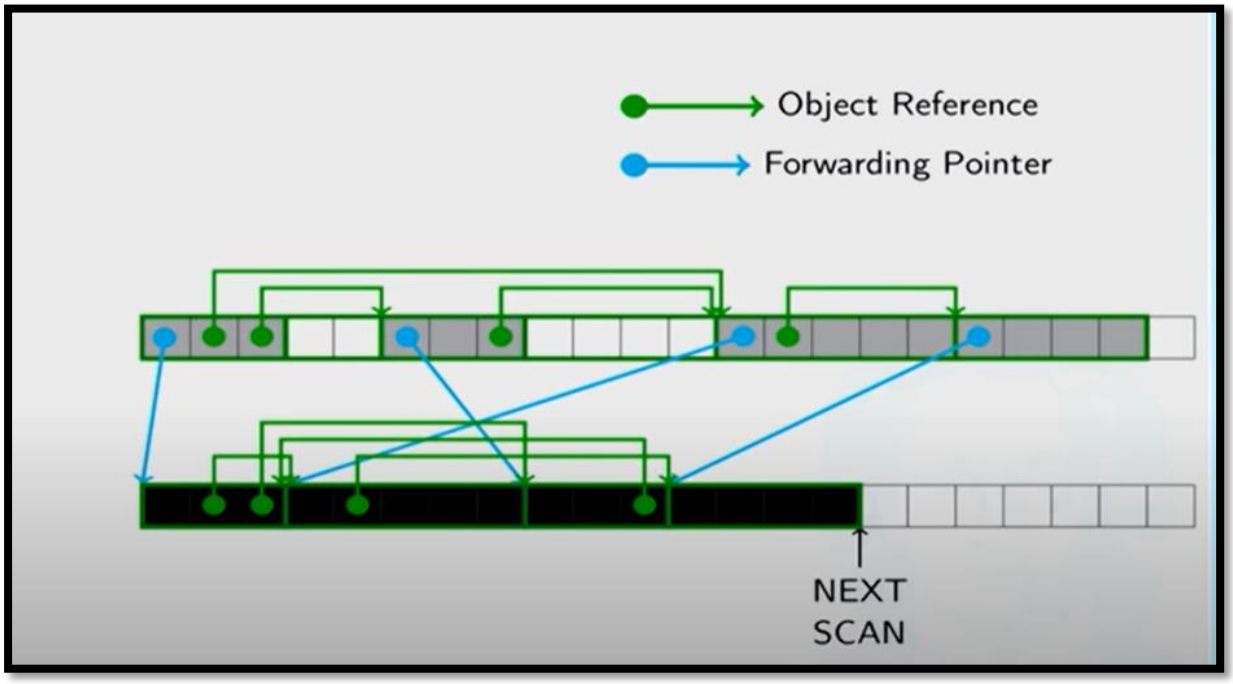
- Now *SCAN* pointer points to an Object which already has a forwarding Pointer (Object is already copied in To-Space), so we do not copy that object.



- Referenced object already has a forwarding pointer, so forwarding pointer is followed (Which gives location of copy of object in To-Space) and reference is updated accordingly.



- Algorithm terminates when *SCAN = NEXT*.



Pseudo-Code

COPY(Address a)

```
{
        If (Has-forwarding-pointer(a)) //object is present in To-Space.
                return Get-forwarding-pointer(a);
         new_a = NEXT;  // new address of object.
        NEXT = NEXT + object-size(a); // increment NEXT pointer.
        Set-forwarding-pointer (a, new_a);
       return new_a;
}
```

DoScan ()

```
{
        While (SCAN < NEXT)
        {
                For each reference r from o (object pointed to by SCAN)
                        r = COPY(r);
                 SCAN += Object_size (SCAN); //Increment SCAN.
        }
}
```

Set-forwarding-pointer (Address a, Address b)

```
{
        Obj(a).Fptr = b;  // sets the forwarding pointer of the object at address a.
}
```

Get-forwarding-pointer (Address a)

```
{
        // All objects must have a field for storing forwarding pointer.
        F_Ptr = Obj(a).Fptr //Forwarding pointer of the object at address a.
        return F_ptr;
}
```

Has-forwarding-pointer (Address a)

```
{
        If (Obj(a).Fptr != NULL) // Object has a forwarding pointer.
                return true;
         else
                return false;
}
```

Object_size (Address a)

```
{
        return size of the object at address a;
}
```

Cheney's_Algorithm ()

```
{
        // Initially Memory is allocated from the To-Space.
         Swap (From-Space, To-Space);
         SCAN = Beginning of To-Space.
         NEXT = Beginning of To-Space.
        // All the objects of root-set are copied to To-Space.
         For Each root in the root-set
                        root = COPY (root);
        DoScan ();
        //Takes care of all the objects referenced by Objects of root-set
        updates all the references/pointers correctly.
}
```

Some Facts

- The algorithm needs no stack and only two pointers : a pointer to the beginning of free space in the to-space, and a pointer to the next word in to-space that needs to be examined. For this reason, it is sometimes called a *"two-finger" algorithm* .

- It only needs "two fingers" pointing into the to-space to keep track of its state. The data between the two fingers represents work remaining for it to do.

- The forwarding pointer (or "broken heart") is used only during the garbage collection process; when a reference to an object already in to-space (thus having a forwarding pointer in from-space) is found, the reference can be updated quickly simply by updating its pointer to match the forwarding pointer.

- Because the strategy is to exhaust all live references, and then all references in referenced objects, this is known as a *breadth-first* list copying garbage collection scheme.

Equivalence to Tri-Color Abstraction

(Similar to BFS)

- White - Object not visited and is garbage at end of tracing phase. Objects in the from-space without forwarding pointers.

- Grey - Object is visited but its descendants may not have been scanned ,Collector must visit it again. Objects in the To-Space that are yet not scanned.

- Black – Object & its immediate descendants are visited.

  GC finished with black objects, will not visit them again. Objects in the To-Space that are scanned.

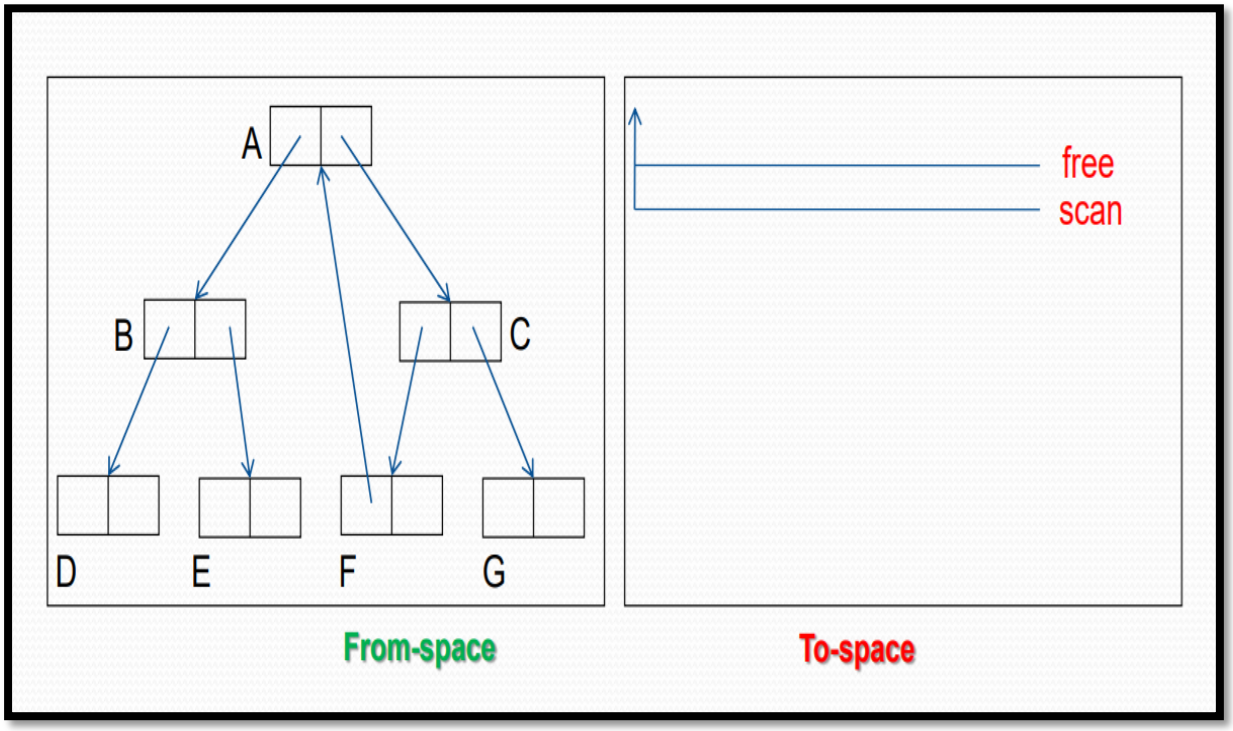
❖ The algorithm COPYs any white objects (equivalent to the grey set by copying them to the to-space.

❖ Objects that are between the SCAN pointer and the NEXT pointer on the to-space area are members of the grey set still to be scanned .

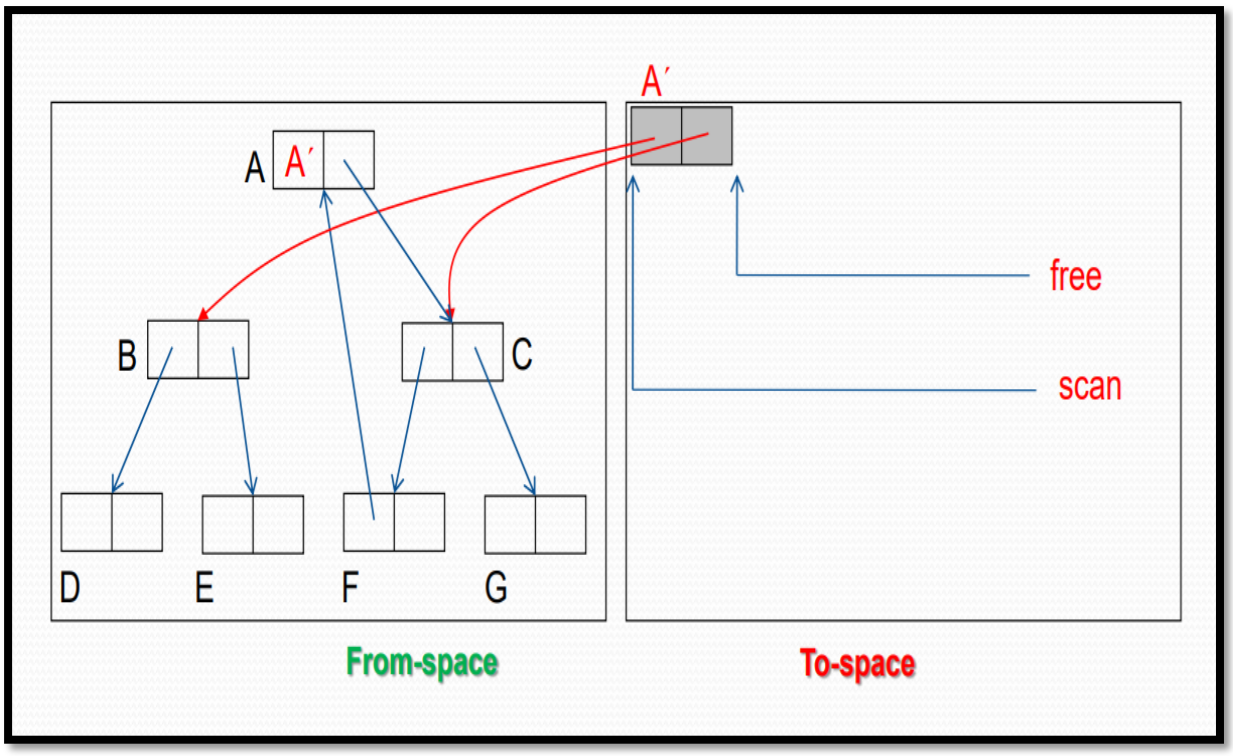
❖ Objects below the scanning pointer belong to the black set .Objects are COPY to the black set by simply moving the scanning pointer over them.

❖ When the scanning pointer reaches the free-space pointer, the grey set is empty, and the algorithm ends.

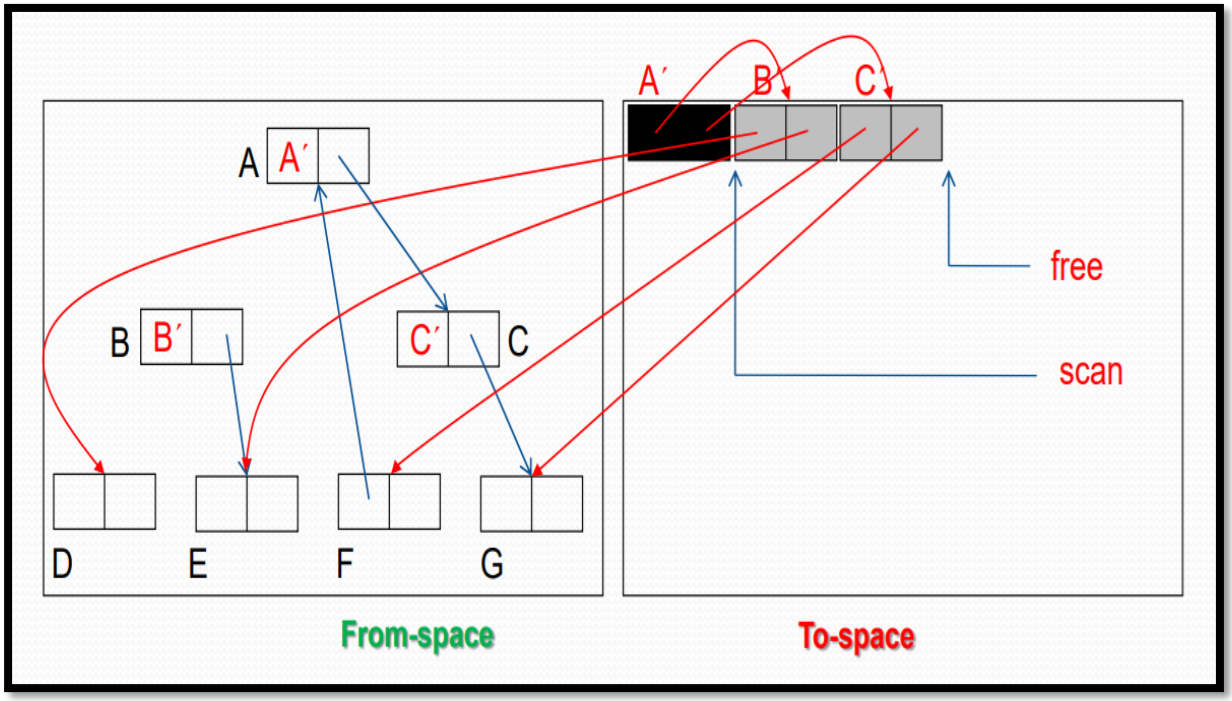
Example Illustration



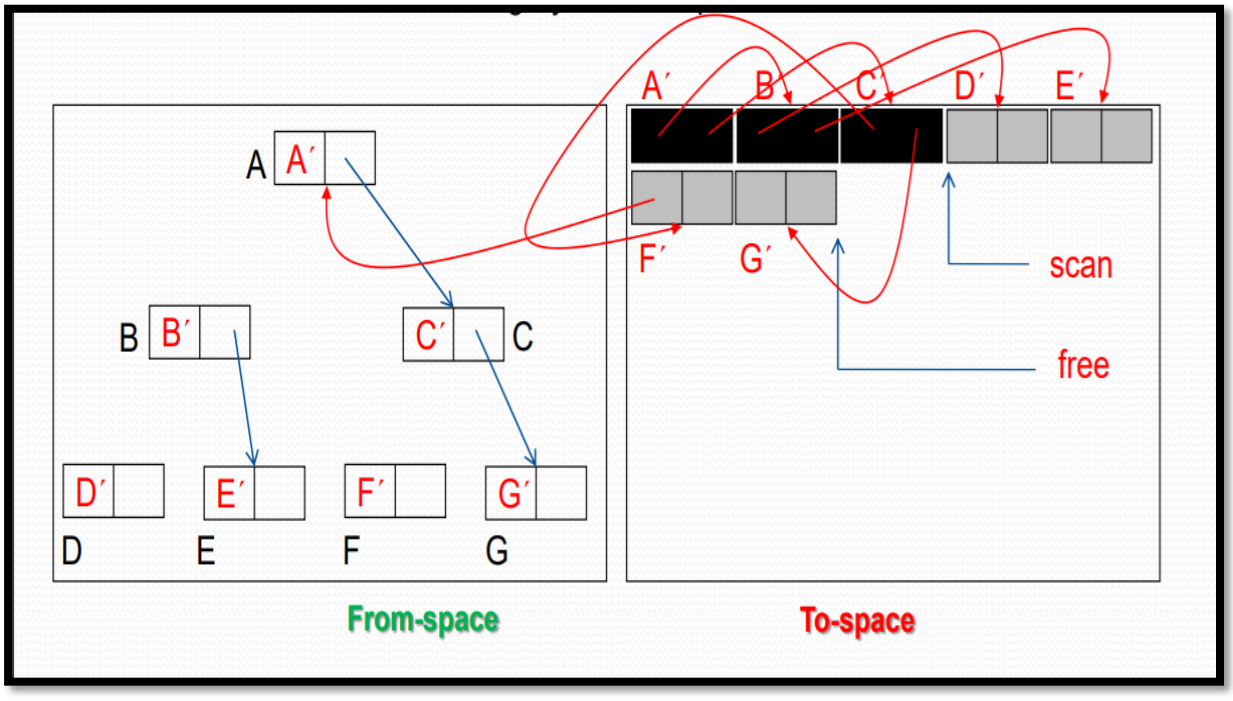
- Roots(live objects) from the root-set are Copied.



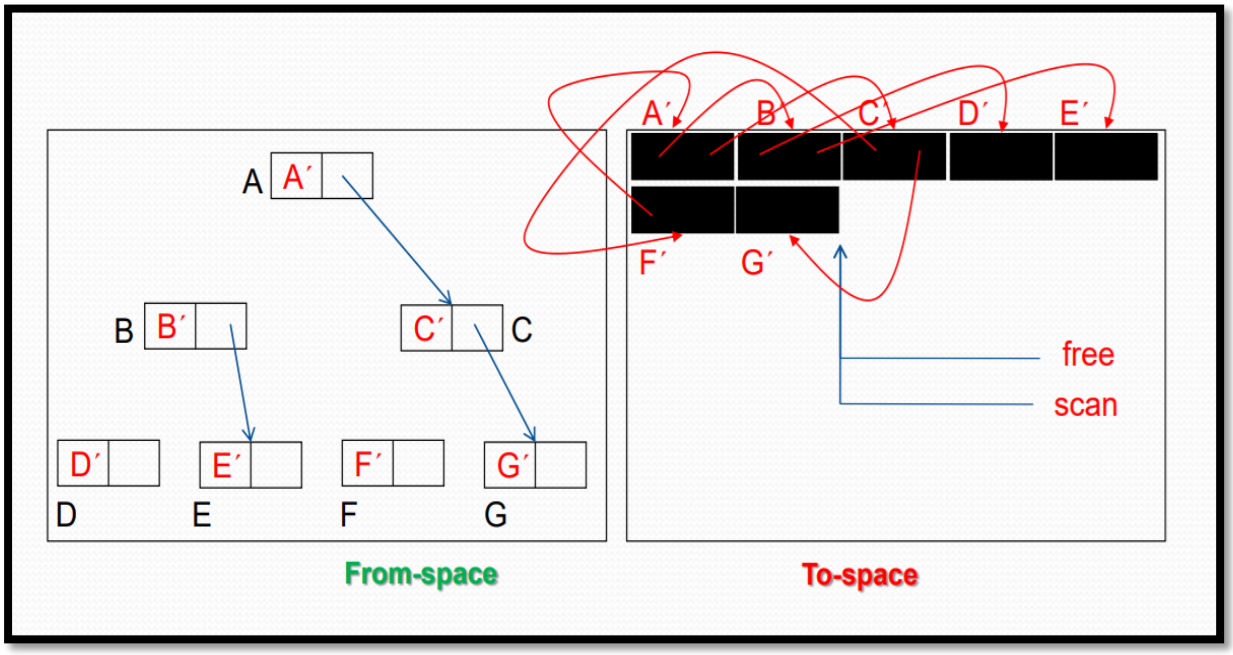
- A′ scanned, copying B and C.
- Black nodes have been scanned; grey nodes copied but not scanned.



- All From-State objects are copied.



- Algorithm terminates(When grey set becomes empty).



Why not DFS? Why BFS?

Problems With DFS (Depth First Search)

- Such a traversal uses memory linear in the depth of the traversal.
- Putting this memory on the stack is a problem because a deep memory graph would cause a stack overflow.
- Putting it on the heap would require having another memory manager just to do memory management, which is possible, but undesirable for runtime, and memory complexity reasons.

BFS (Breadth First Search) – a Smart Choice

- ✓ Using BFS replaces the linear memory usage of a typical DFS with a single extra pointer.
- ✓ This pointer is called the *SCAN* pointer, or sometimes the grey pointer.

- **This is intersecting to note that naive BFS also requires additional memory(a queue) like DFS(a stack) but in case of Cheney's algorithm memory between SCAN and NEXT pointers is the BFS-Queue, resulting in no additional memory requirement.**

References

- https://www.rose-hulman.edu/class/csse/csse490-dsr/201030/Slides/CheneyCopyingGC.pdf
- https://youtu.be/P1rU_9IB414
- https://en.wikipedia.org/wiki/Cheney%27s_algorithm
- https://www.cs.princeton.edu/courses/archive/spring16/cos320/lectures/13-GC.pdf
- https://www.andrew.cmu.edu/course/15-501-819/files/gc-assn.pdf