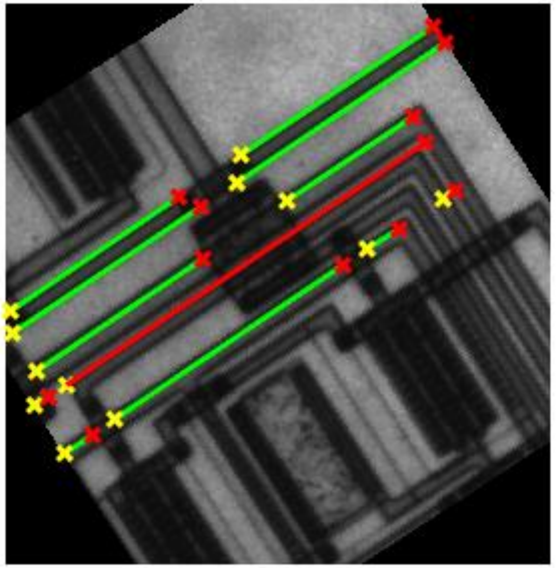
# CMPT 732-G200 Practices for Visual Computing

Ali Mahdavi Amiri

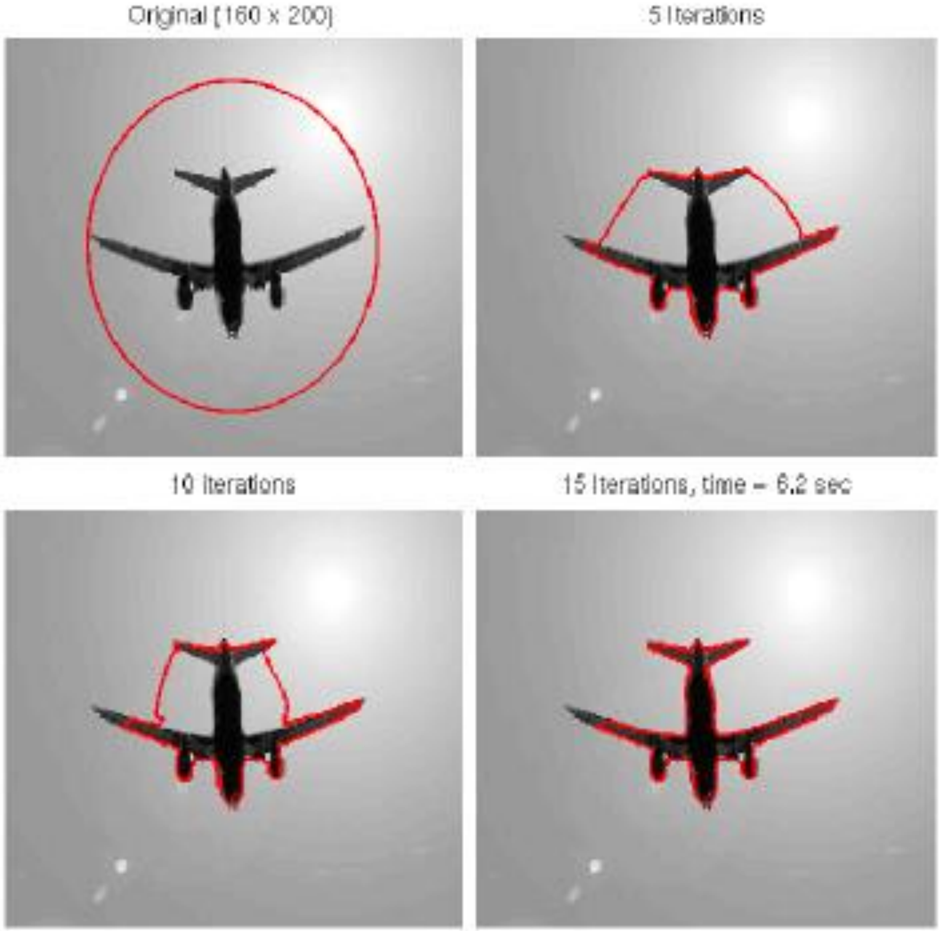
# Image Segmentation

- Hough Transform



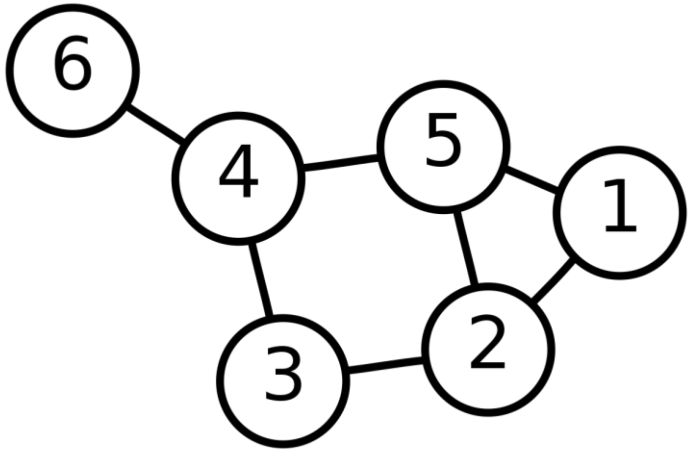
# Image Segmentation

- Hough Transform
- Active Contours



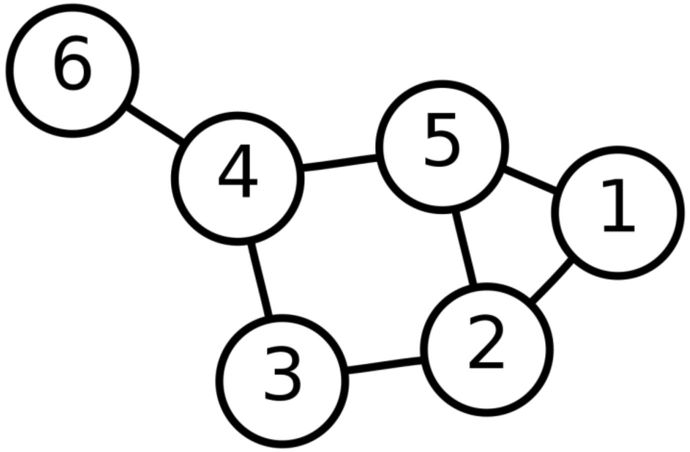
# Image Segmentation

- Hough Transform
- Active Contours
- Graphs



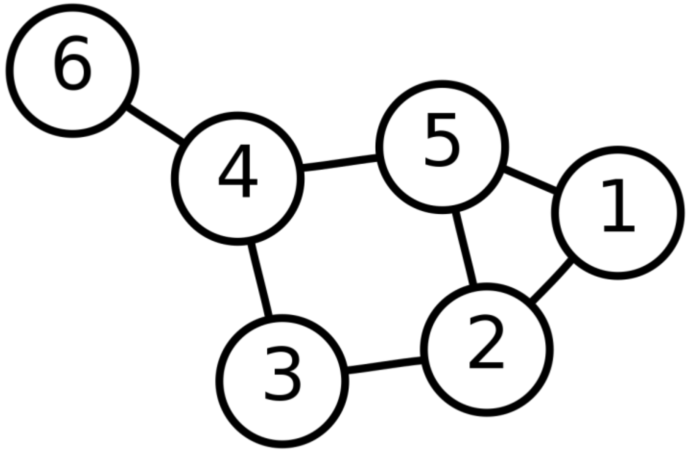
# Graph

- Graph $G$ is a set of vertices $V$ connected by edges $E$ and we use $G(V, E)$ notation to refer to it.



# Graph

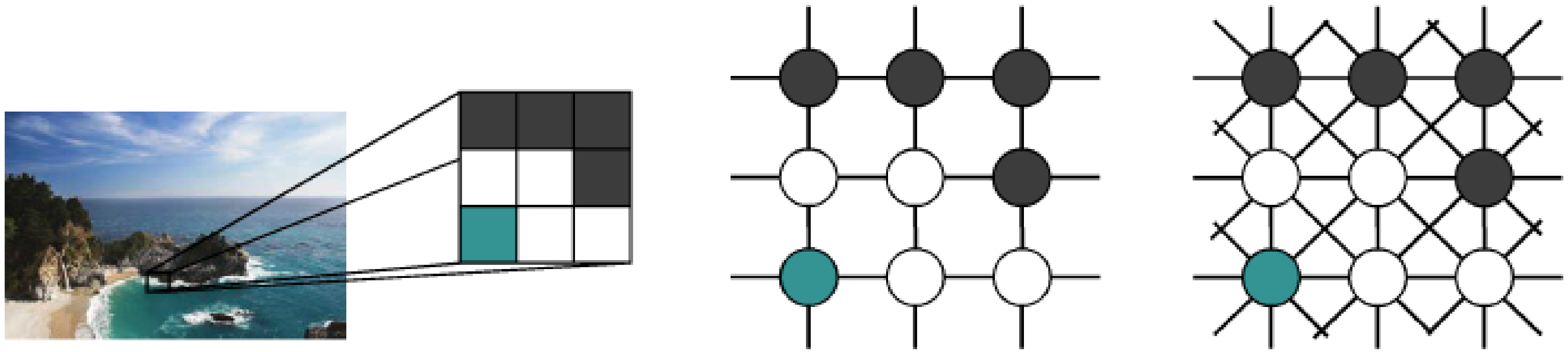
- Graph $G$ is a set of vertices $V$ connected by edges $E$ and we use $G(V,E)$ notation to refer to it.




What is the difference between graphs and trees?

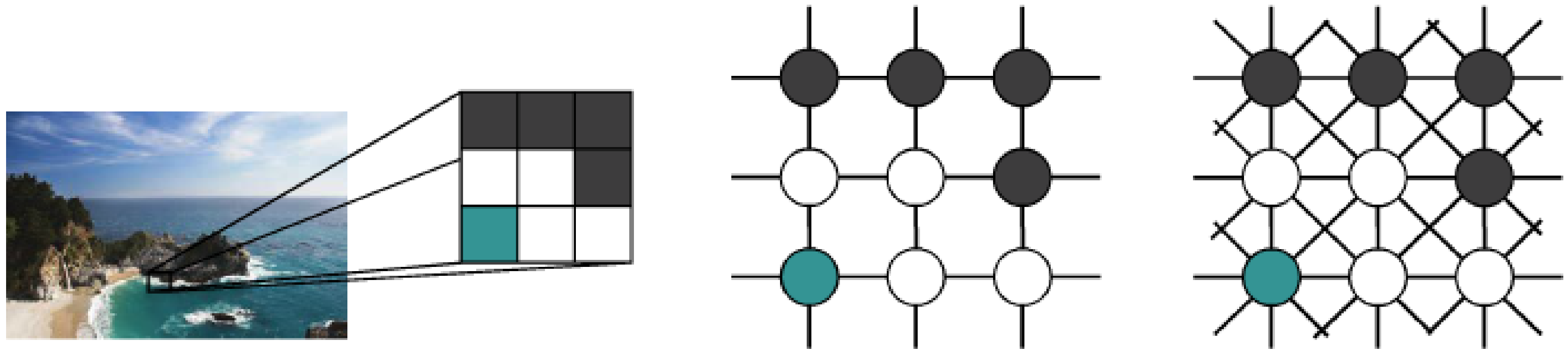
# Graph

- Graphs can be used to represent images



# Graph

- We want to use this representation to segment an image into a set of regions.



# Graph

- We discussed that the segments are usually separated by edges.

What is an edge?

# Graph

- An edge appears in sudden movements of the color of one pixel to its neighbor.

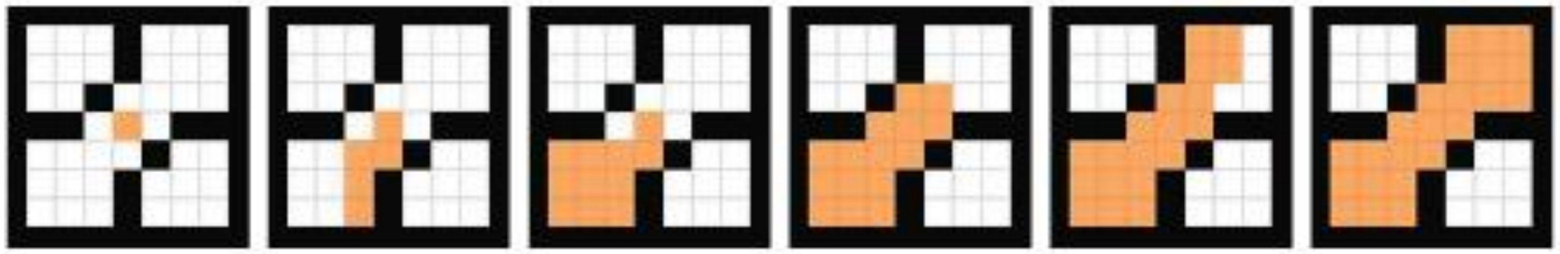
# Graph

- An edge appears in sudden movements of the color of one pixel to its neighbor.

Can we use these properties to design a segmentation technique?

# Flood Fill

- Think about starting from a node and add its neighbors if their neighbors have the same color.



# Flood Fill

- Simple algorithm

```
FloodFill( i, j ) {
    Pixel p = I(i,j)
    If not (stopTest(p) or isVisited(p)) {
        Visit(p)
        FloodFill(i, j+1)
        FloodFill(i, j−1)
        FloodFill(i+1, j)
        FloodFill(i−1, j)
    }
}
```

# Flood Fill

- Simple algorithm

When should we stop?

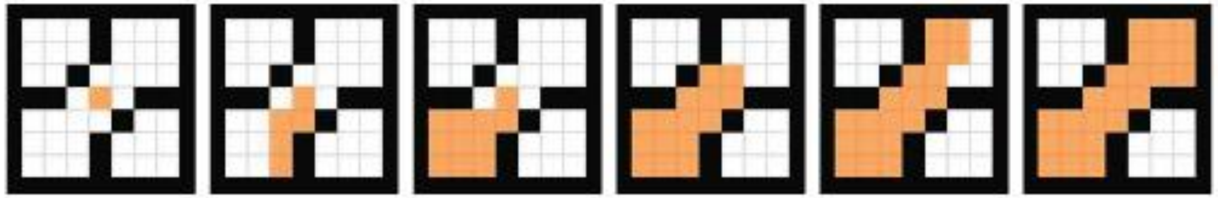
```
FloodFill( i, j ) {
    Pixel p = I(i,j)
    If not (stopTest(p) or isVisited(p)) {
        Visit(p)
        FloodFill(i, j+1)
        FloodFill(i, j-1)
        FloodFill(i+1, j)
        FloodFill(i-1, j)
    }
}
```

# Flood Fill

- Simple algorithm

When should we stop?

```
FloodFill( i, j ) {
    Pixel p = I(i,j)
    If not (stopTest(p) or isVisited(p)) {
        Visit(p)
        FloodFill(i, j + 1)
        FloodFill(i, j − 1)
        FloodFill(i + 1, j)
        FloodFill(i − 1, j)
    }
}
```



# Flood Fill

- Simple algorithm

```
FloodFill( i, j ) {
    Pixel p = I(i,j)
    If not (stopTest(p) or isVisited(p)) {
        Visit(p)
        FloodFill(i, j + 1)
        FloodFill(i, j − 1)
        FloodFill(i + 1, j)
        FloodFill(i − 1, j)
    }
}
```

Assign a color or label to the pixel

# Region Grow

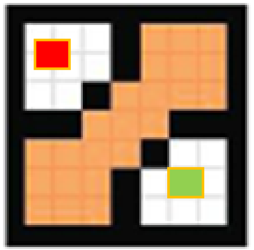
- More general that works for a general graph

```
RegionGrow( Node v, Graph G ) {
   If not (stopTest(v,G) or isVisited(v)) {
      Visit(v)
      For all neighbors of u of v do
         RegionGrow(u, G)
   }
}
```

# Region Grow

- We cannot segment the entire image, we need more seeds:

```
RegionGrow( Node v, Graph G ) {
    If not (stopTest(v,G) or isVisited(v)) {
        Visit(v)
        For all neighbors of u of v do
            RegionGrow(u, G)
    }
}
```



# Watershed Algorithm

- Partition algorithm repeats until no unassigned vertex is remained:

```
Partition( Graph G ) {
   Loop until there are no more free nodes in G {
      Choose a free node s as seed
      RegionGrow(s,G)
   }
   Clean up small regions
}
```

# Watershed Algorithm

- Partition algorithm repeats until no unassigned vertex is remained:

```
Partition( Graph G ) {
    Loop until there are no more free nodes in G {
        Choose a free node s as seed
        RegionGrow(s,G)
    }
    Clean up small regions
}
```

Distribute it to neighboring segments

# Watershed Algorithm

- Partition algorithm repeats until no unassigned vertex is remained:

```
Partition( Graph G ) {
    Loop until there are no more free nodes in G {
        Choose a free node s as seed
        RegionGrow(s,G)
    }
    Clean up small regions
}
```

Randomly

# Watershed Algorithm

- Random chosen seeds are not good ideas

```
Partition( Graph G ) {
   Loop until there are no more free nodes in G {
      Choose a free node s as seed
      RegionGrow(s,G)
   }
   Clean up small regions
}
```

Randomly

# Watershed Algorithm

- Random chosen seeds are not good ideas

```
Partition( Graph G ) {
    Loop until there are no more free nodes in G {
        Choose a free node s as seed
        RegionGrow(s,G)
    }
    Clean up small regions
}
```

Randomly

Why?

# Watershed Algorithm

- The idea of watershed algorithm is to start from a local minima. Therefore you need to have a height function $h$.

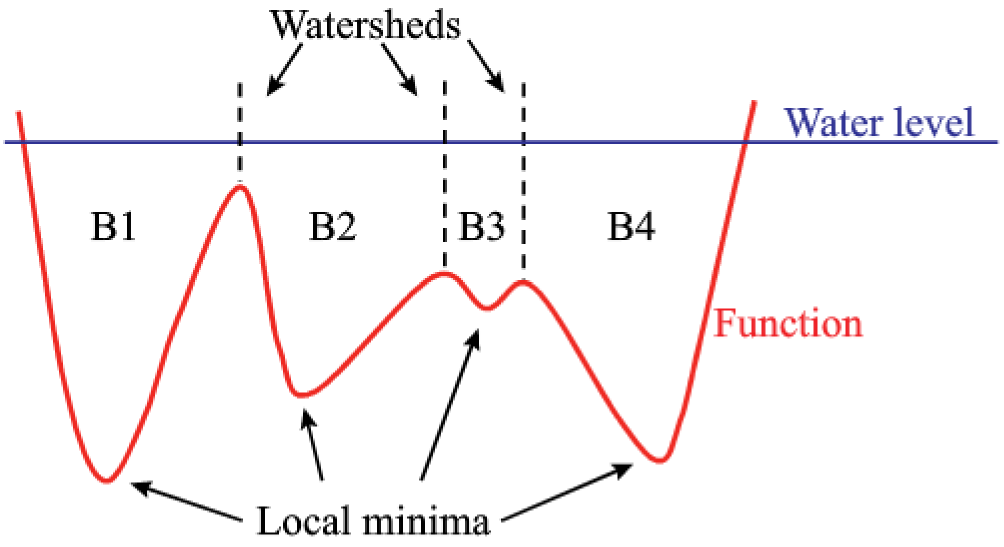
# Watershed Algorithm

- The idea of watershed algorithm is to start from a local minima.

- One possibility is to convert images to grey scale and define the minimum on the pixel gradient. Refer to book for more info on $h$.

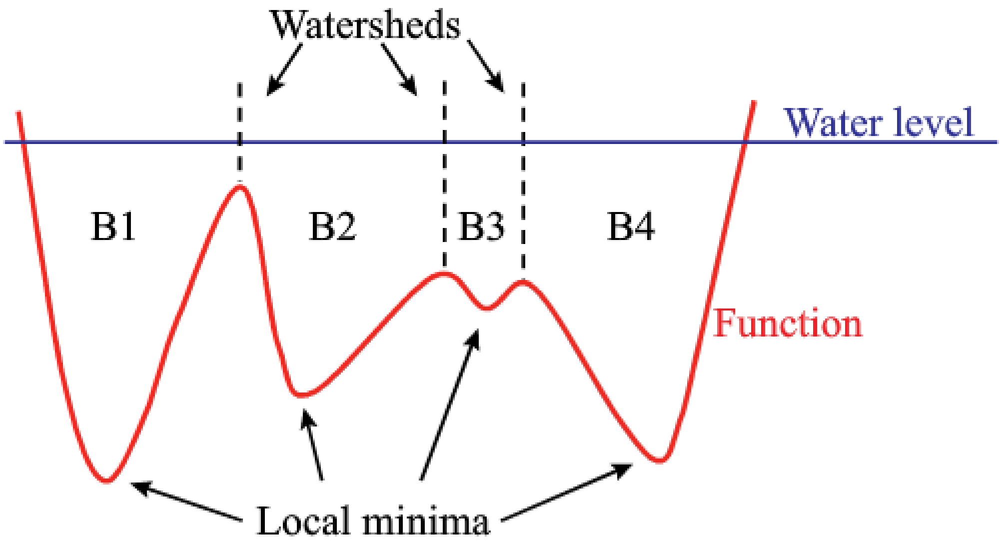
# Local Minima

- Local Minima $M$ of $I$ at altitude $h$ is a connected plateau of pixels with the value $h$ from which it is impossible to reach a point of lower altitude without having to climb.



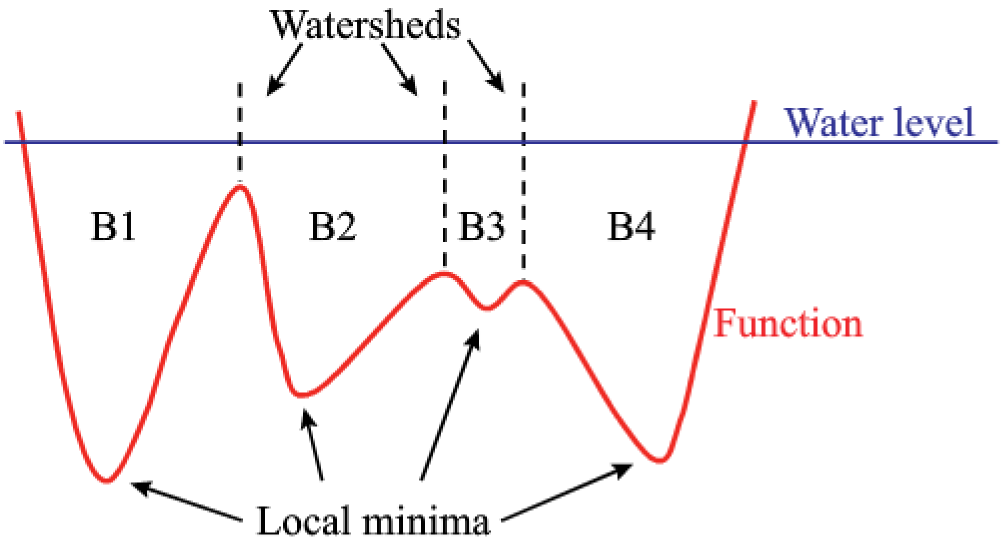
# Watershed Algorithm

- The idea of watershed is to start from a local minimum and try to fill it with water.



# Watershed Algorithm

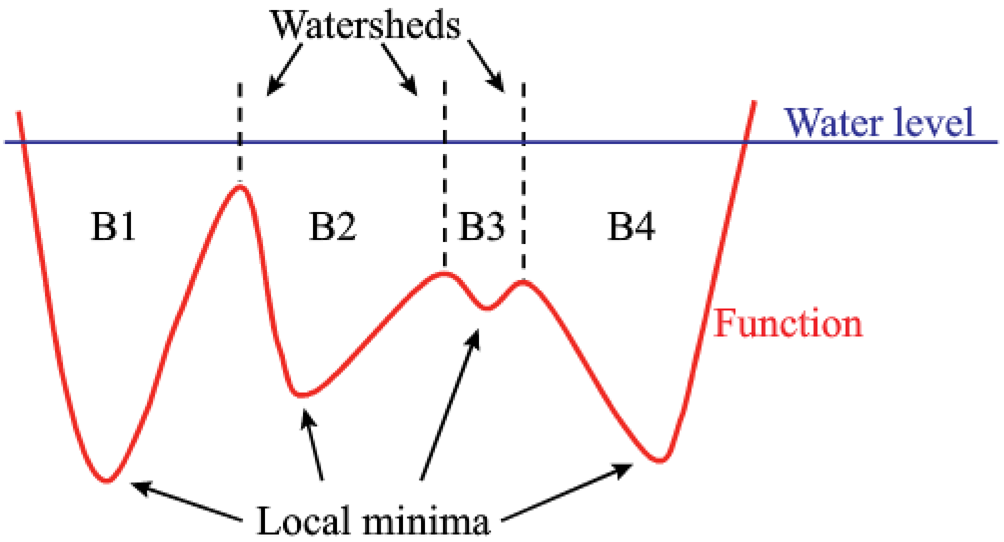
- The idea of watershed is to start from a local minimum and try to fill it with water.
- Wherever two basins are full of water and start to flood, make a barrier (watershed).



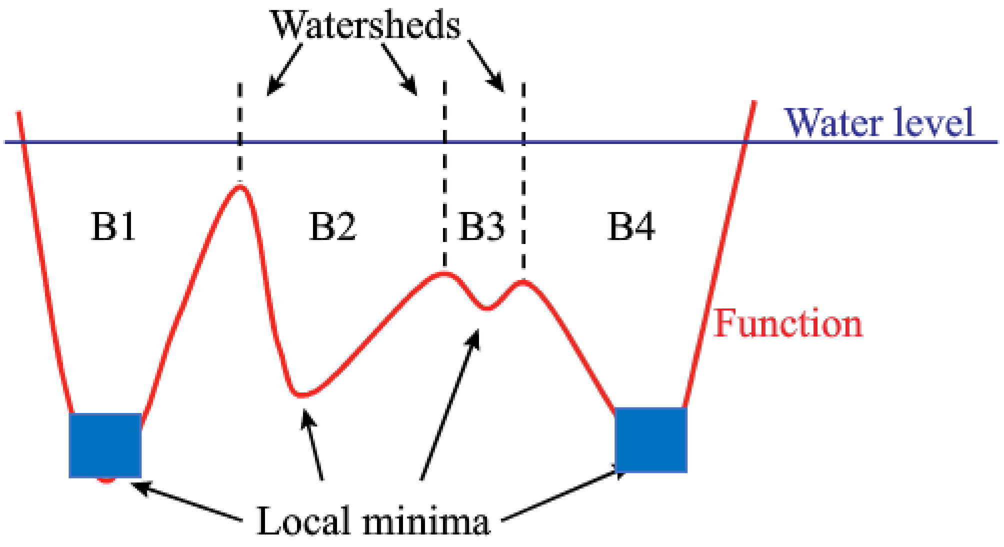
# Watershed Algorithm

- The idea of watershed is to start from a local minimum and try to fill it with water.
- Wherever two basins are full of water and start to flood, make a barrier (watershed).

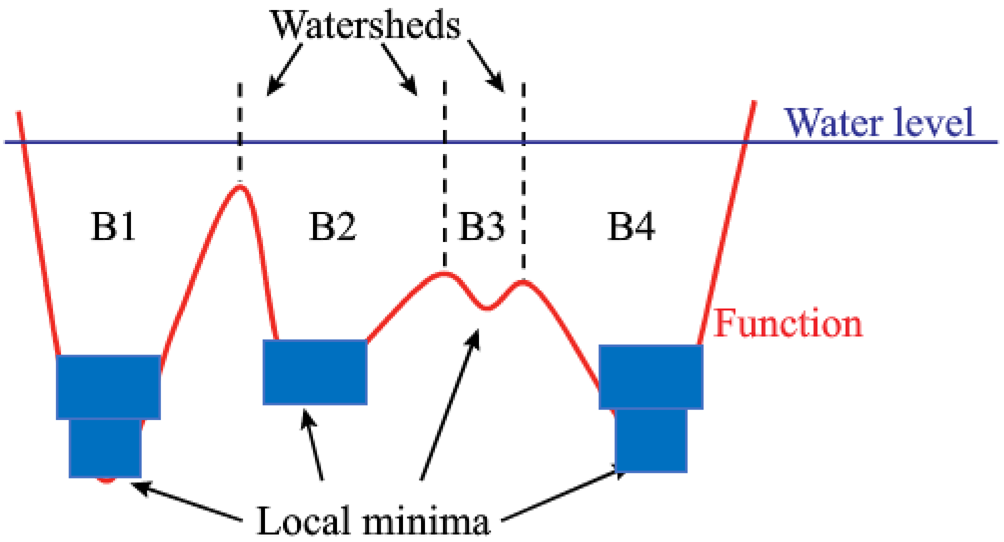
Label neighbors



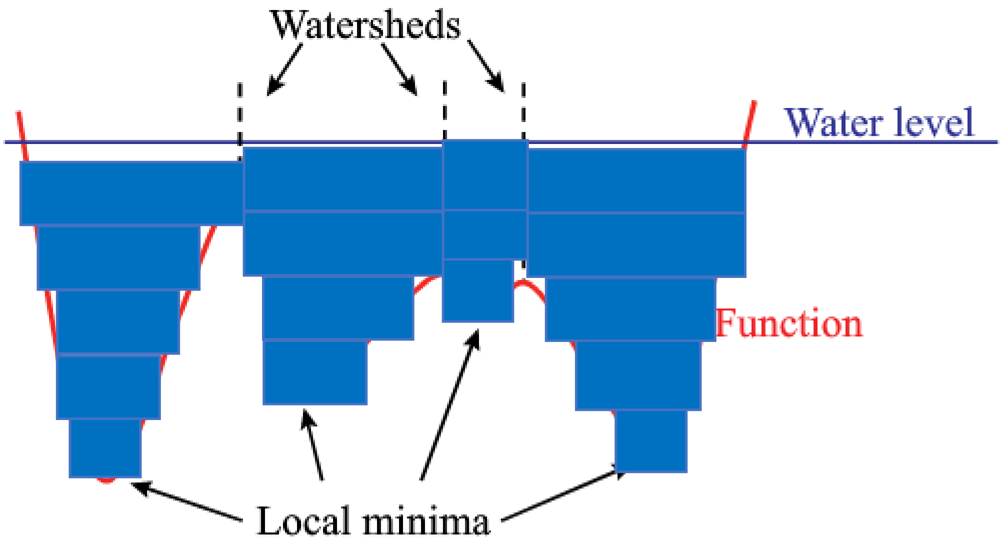
# Watershed Algorithm



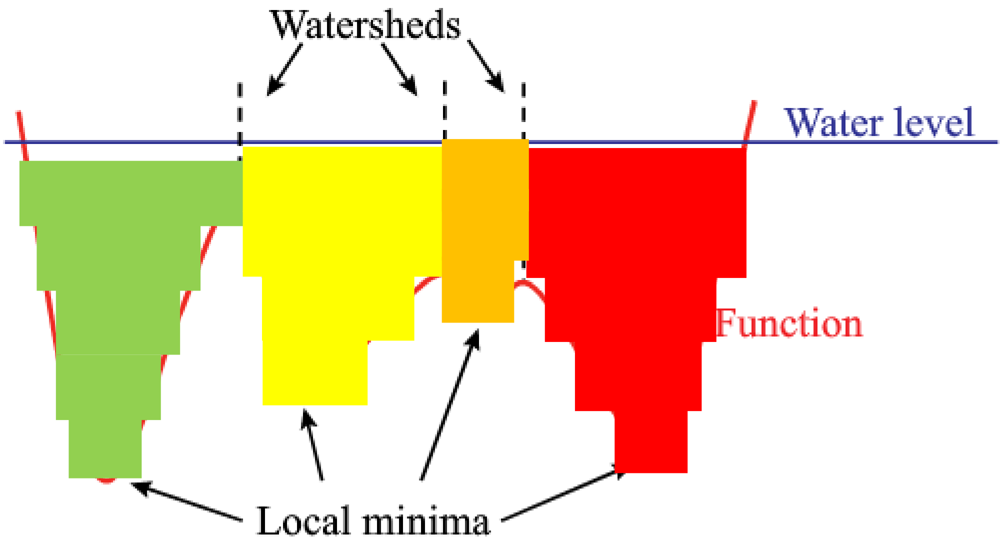
# Watershed Algorithm



# Watershed Algorithm

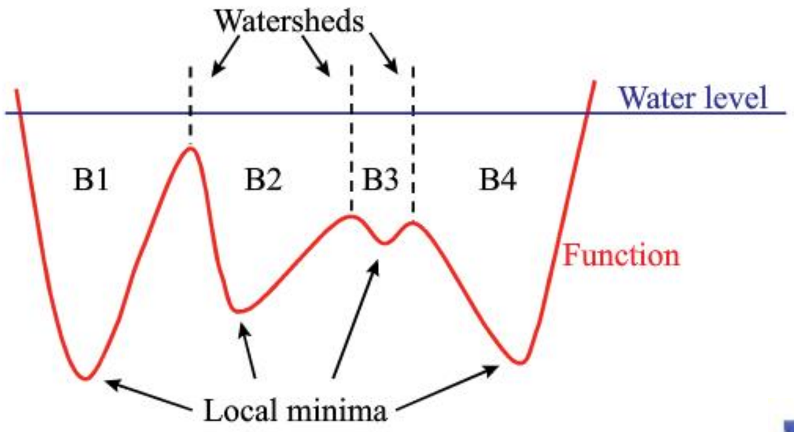


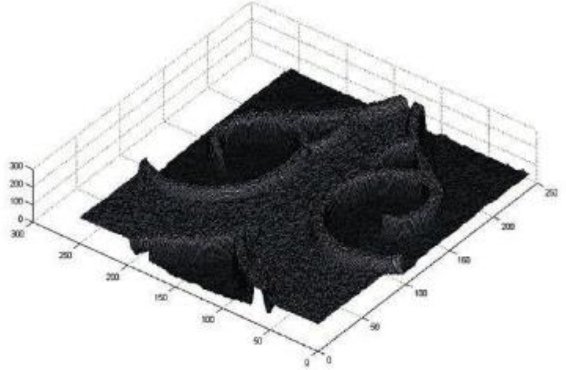
# Watershed Algorithm

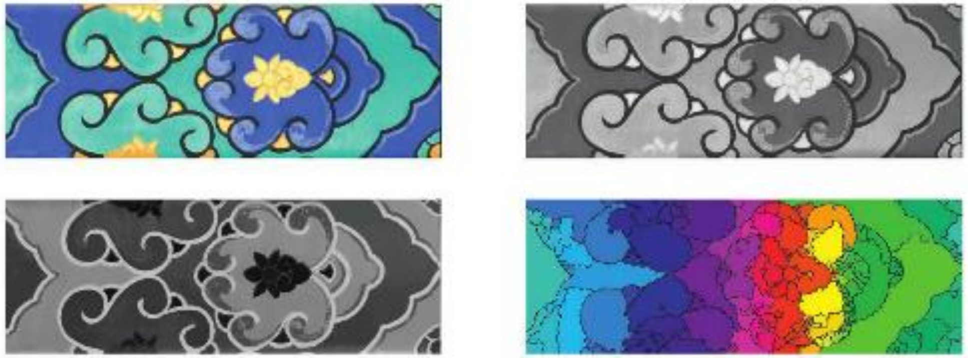


# Watershed Algorithm

- Local minima and basins can be defined on 2D domain



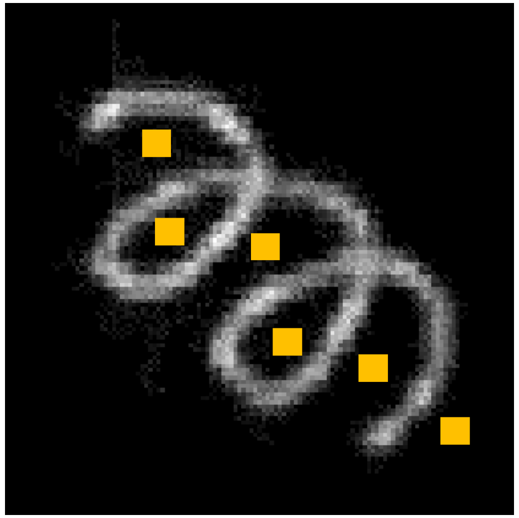


# Watershed Algorithm

```
Watershed( Graph G, function h ) {
   Find all local minima of h(G) as seeds {
   Create a priority queue Q
      the highest priority is the lowest height
   Label each seed with a unique label
   Insert all un-labeled neighbors of all seeds into Q
   Loop until Q is empty {
      Get the node v with highest priority from Q
      If all labeled neighbors of v have the same label
         Then label v with the same label
      Else
         Label v as a watershed node
      Insert all un-labeled neighbors of v to Q
   }
}
```

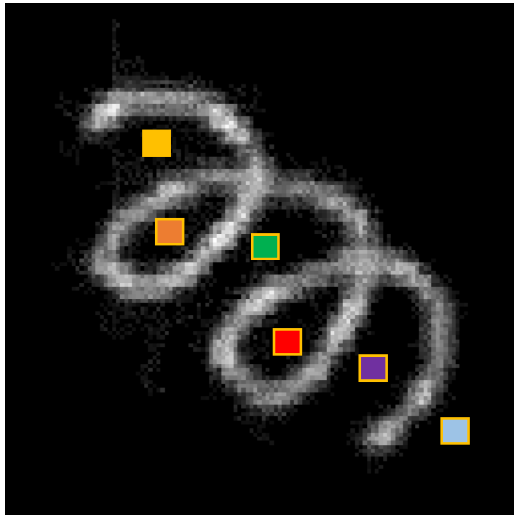
# Watershed Algorithm

```
Watershed( Graph G, function h ) {
  Find all local minima of h(G) as seeds {
  Create a priority queue Q
    the highest priority is the lowest height
  Label each seed with a unique label
  Insert all un-labeled neighbors of all seeds into Q
  Loop until Q is empty {
    Get the node v with highest priority from Q
    If all labeled neighbors of v have the same label
      Then label v with the same label
    Else
      Label v as a watershed node
    Insert all un-labeled neighbors of v to Q
  }
}
```



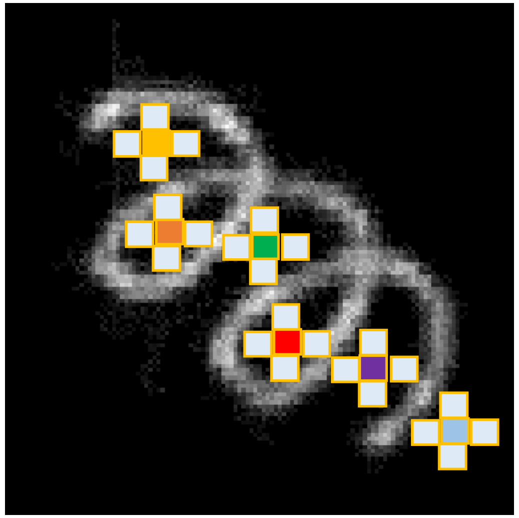
# Watershed Algorithm

```
Watershed( Graph G, function h ) {
  Find all local minima of h(G) as seeds {
  Create a priority queue Q
    the highest priority is the lowest height
  Label each seed with a unique label
  Insert all un-labeled neighbors of all seeds into Q
  Loop until Q is empty {
    Get the node v with highest priority from Q
    If all labeled neighbors of v have the same label
      Then label v with the same label
    Else
      Label v as a watershed node
    Insert all un-labeled neighbors of v to Q
  }
}
```



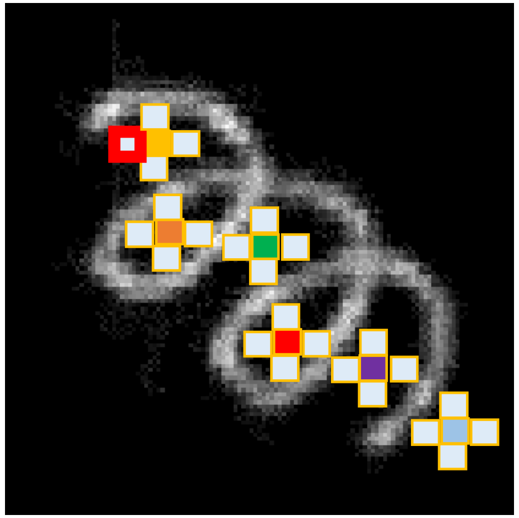
# Watershed Algorithm

```
Watershed( Graph G, function h ) {
  Find all local minima of h(G) as seeds {
  Create a priority queue Q
    the highest priority is the lowest height
  Label each seed with a unique label
  Insert all un-labeled neighbors of all seeds into Q
  Loop until Q is empty {
    Get the node v with highest priority from Q
    If all labeled neighbors of v have the same label
      Then label v with the same label
    Else
      Label v as a watershed node
    Insert all un-labeled neighbors of v to Q
  }
}
```



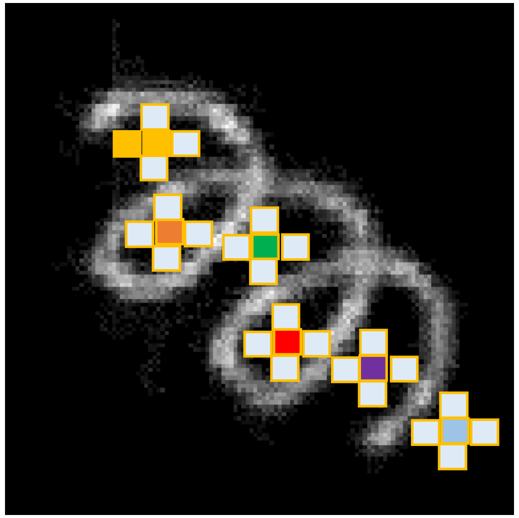
# Watershed Algorithm

```
Watershed( Graph G, function h ) {
   Find all local minima of h(G) as seeds {
   Create a priority queue Q
      the highest priority is the lowest height
   Label each seed with a unique label
   Insert all un-labeled neighbors of all seeds into Q
   Loop until Q is empty {
      Get the node v with highest priority from Q
      If all labeled neighbors of v have the same label
         Then label v with the same label
      Else
         Label v as a watershed node
      Insert all un-labeled neighbors of v to Q
   }
}
```



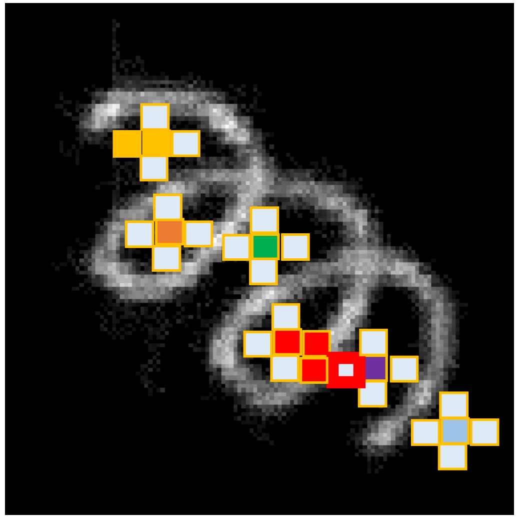
# Watershed Algorithm

```
Watershed( Graph G, function h ) {
    Find all local minima of h(G) as seeds {
    Create a priority queue Q
        the highest priority is the lowest height
    Label each seed with a unique label
    Insert all un-labeled neighbors of all seeds into Q
    Loop until Q is empty {
        Get the node v with highest priority from Q
        If all labeled neighbors of v have the same label
            Then label v with the same label
        Else
            Label v as a watershed node
        Insert all un-labeled neighbors of v to Q
    }
}
```



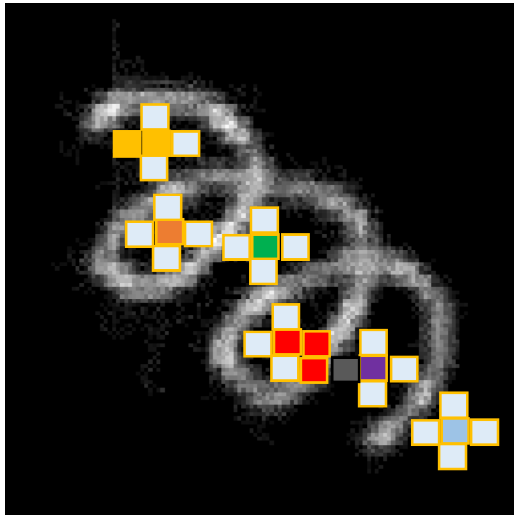
# Watershed Algorithm

```
Watershed( Graph G, function h ) {
    Find all local minima of h(G) as seeds {
    Create a priority queue Q
        the highest priority is the lowest height
    Label each seed with a unique label
    Insert all un-labeled neighbors of all seeds into Q
    Loop until Q is empty {
        Get the node v with highest priority from Q
        If all labeled neighbors of v have the same label
            Then label v with the same label
        Else
            Label v as a watershed node
        Insert all un-labeled neighbors of v to Q
    }
}
```



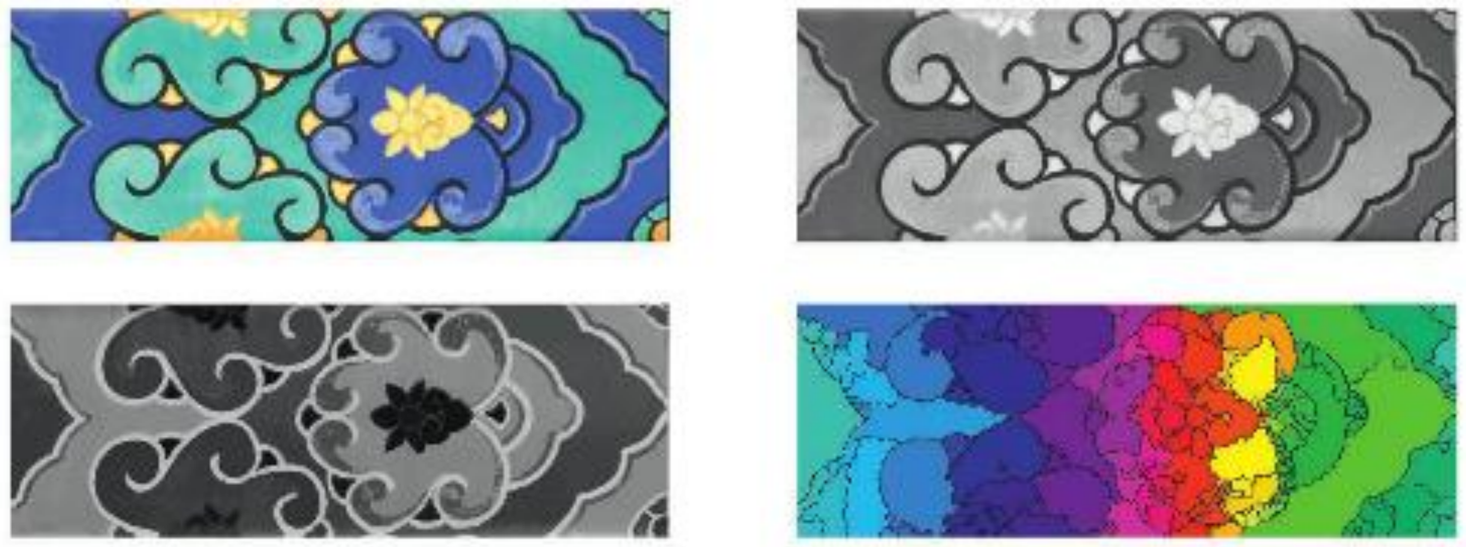
# Watershed Algorithm

```
Watershed( Graph G, function h ) {
   Find all local minima of h(G) as seeds {
   Create a priority queue Q
      the highest priority is the lowest height
   Label each seed with a unique label
   Insert all un-labeled neighbors of all seeds into Q
   Loop until Q is empty {
      Get the node v with highest priority from Q
      If all labeled neighbors of v have the same label
         Then label v with the same label
      Else
         Label v as a watershed node
      Insert all un-labeled neighbors of v to Q
   }
}
```



# Recap

- Watershed algorithm starts from pixels with local minima and tries to add pixels in their neighborhood until two regions collide (watershed).



# Graph Cut

- We can see the image segmentation problem as an optimization problem.

# Graph Cut

- We can see the image segmentation problem as an optimization problem.

- We define an energy that measures the cost of assigning a label (F,B) to a pixel.

# Graph Cut

- We can see the image segmentation problem as an optimization problem.

- We define an energy that measures the cost of assigning a label (F,B) to a pixel.

What is a proper energy function?

# Energy Function

- We should assign B/F labels to the pixel with color similar to background or foreground.

# Energy Function

- We should assign B/F labels to the pixel with color similar to background or foreground.

- How do we know which colors are foreground and which colors are background?

# Energy Function

- We can use an interactive system to define B/F colors.

# Energy Function

- Now we are ready to define the first cost function

$$D(L) = \sum_{p \epsilon I} cost(p, L(p))$$

# Energy Function

- Now we are ready to define the first cost function

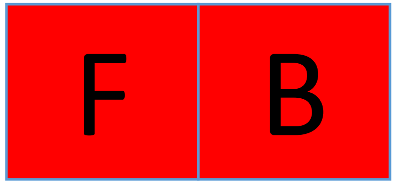
$$D(L) = \sum_{p \epsilon I} cost(p, L(p))$$

$$L(p) = \begin{Bmatrix} 0 & foreground \\ 1 & background \end{Bmatrix}$$

$cost(p, 1)$ is small if the color of $p$ is close to the color distribution of background pixels.

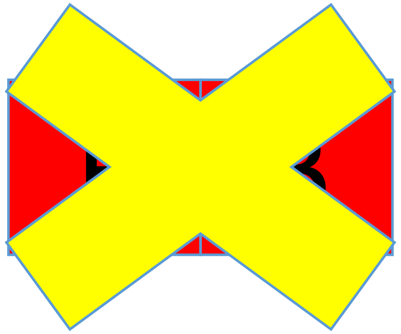
# Energy Function

- We should define another energy function (smoothness). This energy function is high when two neighboring pixels with similar color receive two different labels.



# Energy Function

- We should define another energy function (smoothness). This energy function is high when two neighboring pixels with similar color receive two different labels.



# Energy Function

- We should define another energy function (smoothness). This energy function is high when two neighboring pixels with similar color receive two different labels.

$$S(L) = \sum_{p,q\epsilon N} B(p,q).\,|L(P) - L(q)|$$

# Energy Function

- We should define another energy function (smoothness). This energy function is high when two neighboring pixels with similar color receive two different labels.

$$S(L) = \sum_{p,q \epsilon N} B(p,q).\,|L(P) - L(q)|$$

$$|L(p) - L(q)| = \begin{Bmatrix} 0 & if\ p\ and\ q\ have\ the\ same\ label \\ 1 & otherwise \end{Bmatrix}$$

# Energy Function

- We should define another energy function (smoothness). This energy function is high when two neighboring pixels with similar color receive two different labels.

$$S(L) = \sum_{p,q \epsilon N} \boxed{B(p,q).}\, |L(P) - L(q)|$$

$$|L(p) - L(q)| = \begin{Bmatrix} 0 & if\ p\ and\ q\ have\ the\ same\ label \\ 1 & otherwise \end{Bmatrix}$$

Measures the color similarity

# Energy Function

- Overall energy

$$E(L) = D(L) + \lambda S(L)$$

# Energy Function

- Overall energy

$$E(L) = D(L) + \lambda S(L)$$

- The solution space to this problem for all the pixels to get F/B label is exponential ($2^n, n$: is he number of pixels)

# Energy Function

- Overall energy

$$E(L) = D(L) + \lambda S(L)$$

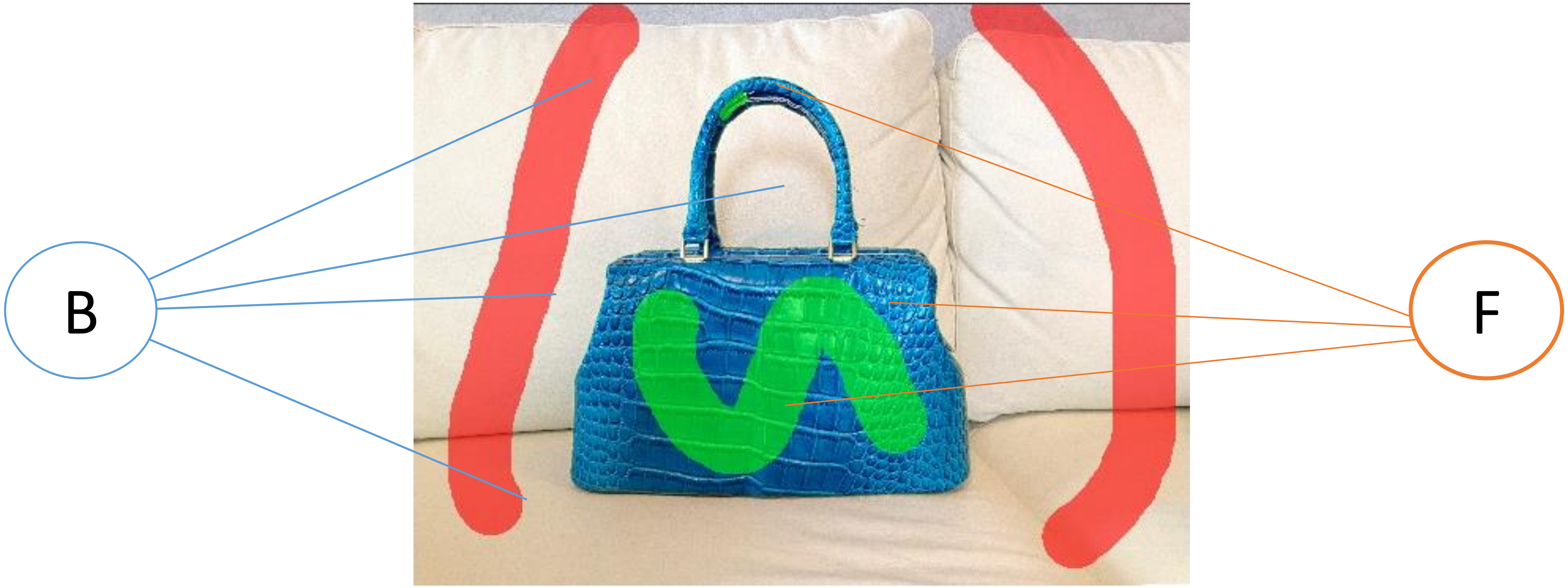
- The solution space to this problem for all the pixels to get F/B label is exponential ($2^n, n$: is he number of pixels)
  - Why?

# Graph Cut

- We can usually cut the foreground from a background and paste it somewhere else.

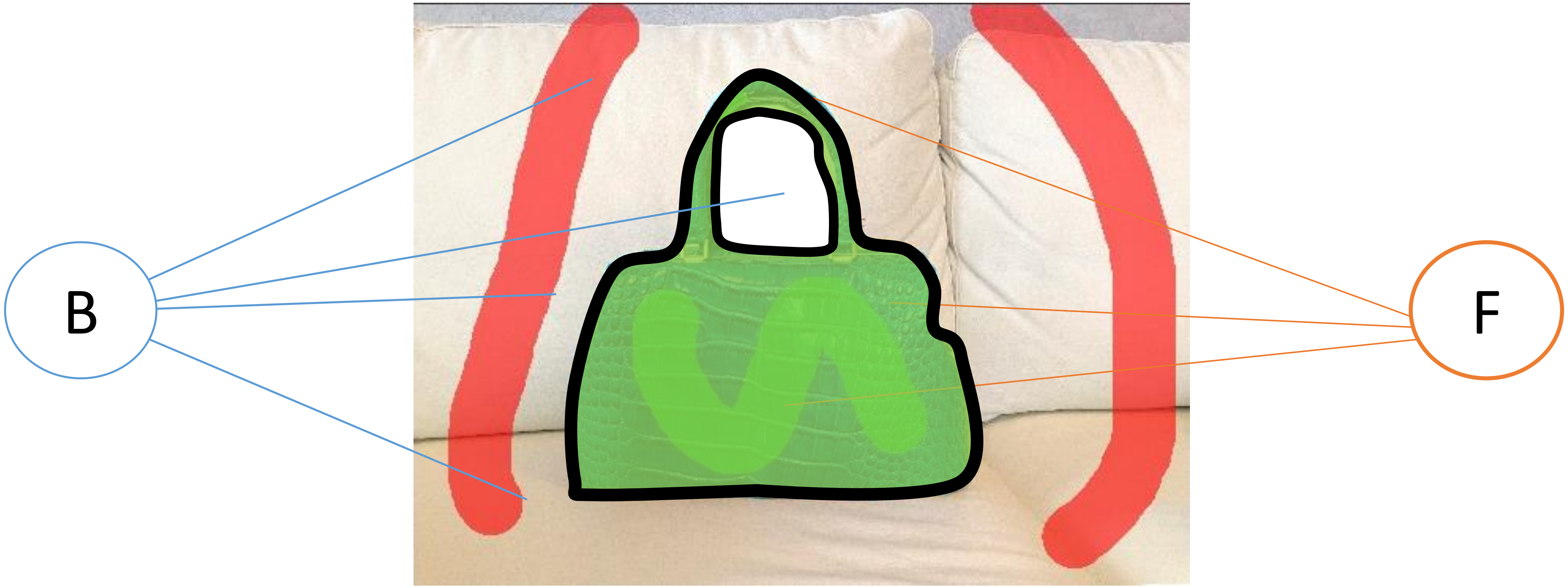
# Graph Cut

- Therefore we can see this as two separated graphs that are connected to two dummy nodes (B/F).



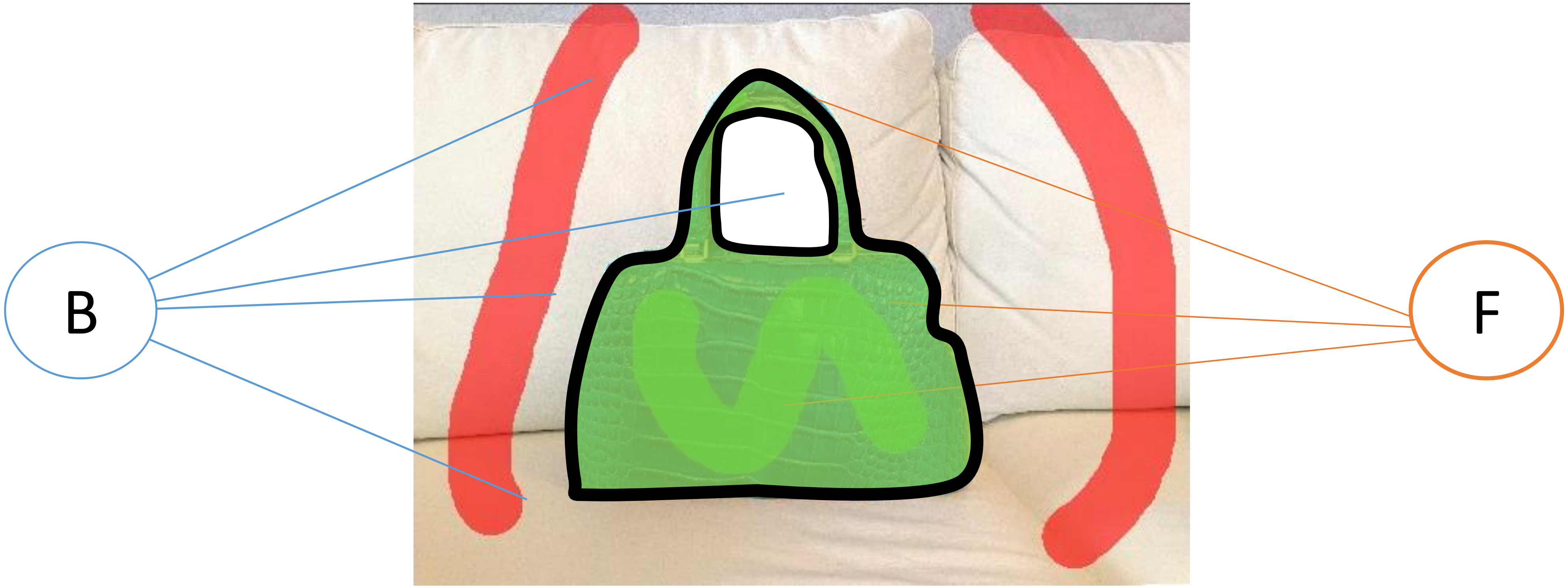
# Graph Cut

- If we can find a way to make these two subgraphs separated, then we have found a segmentation.



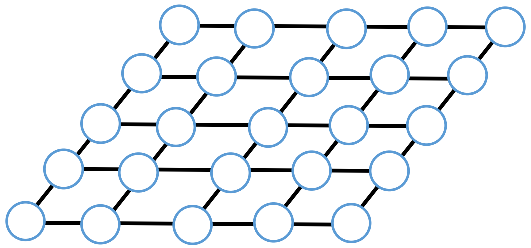
# Graph Cut

- Fortunately, there is a well know algorithm for doing this task and it is called graph cut.



# Graph Cut

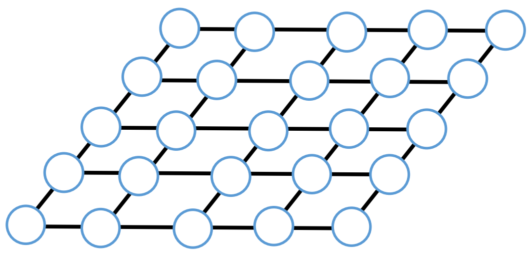
- Imagine the image as a graph whose vertices are pixels of an image and its edges connect two neighboring pixels.



# Graph Cut

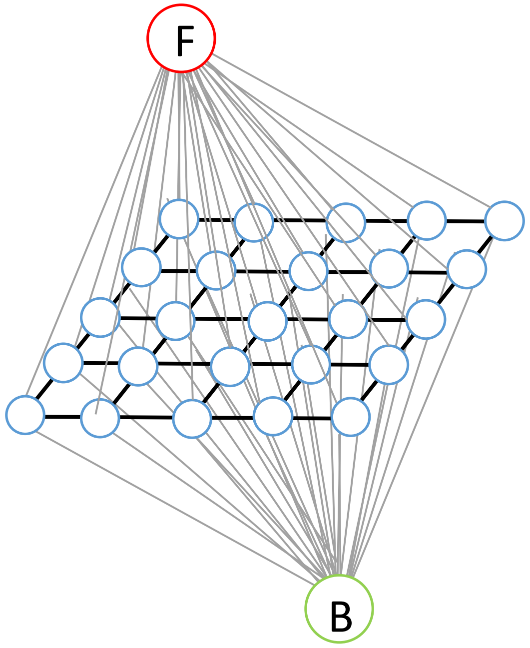
- Imagine the image as a graph whose vertices are pixels of an image and its edges connect two neighboring pixels.

- These edges are called n-links



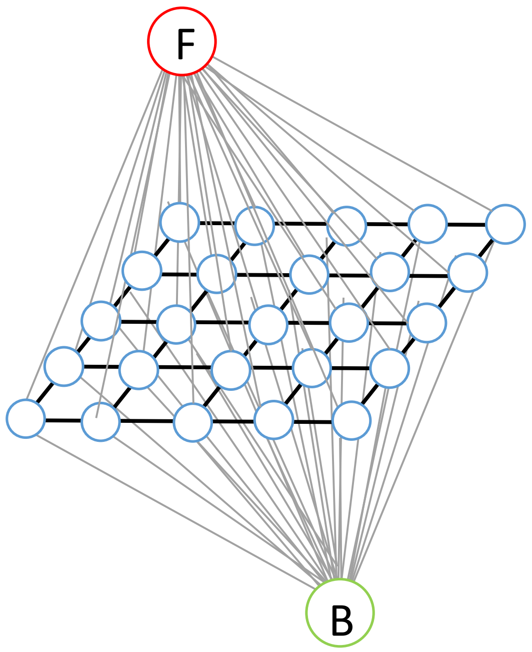
# Graph Cut

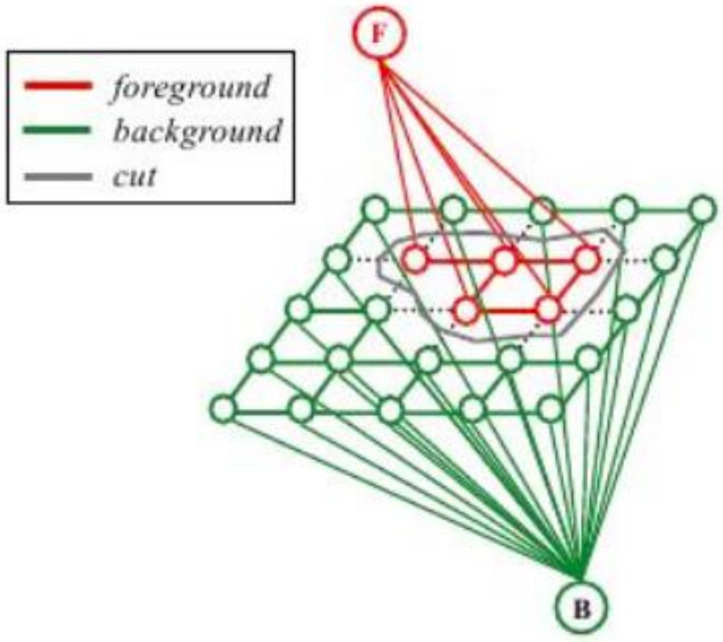
- Now consider that all the vertices (pixels) are connected to two dummy terminal nodes through a set of t-links



# Graph Cut

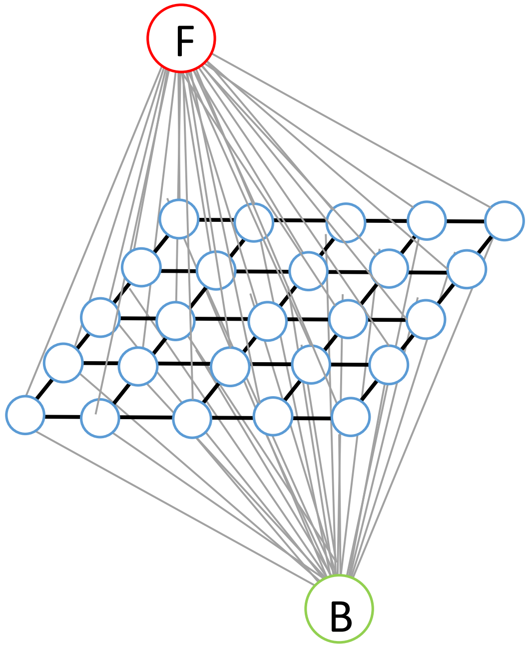
- The main idea is to find a cut in the graph that separates the two terminal nodes.

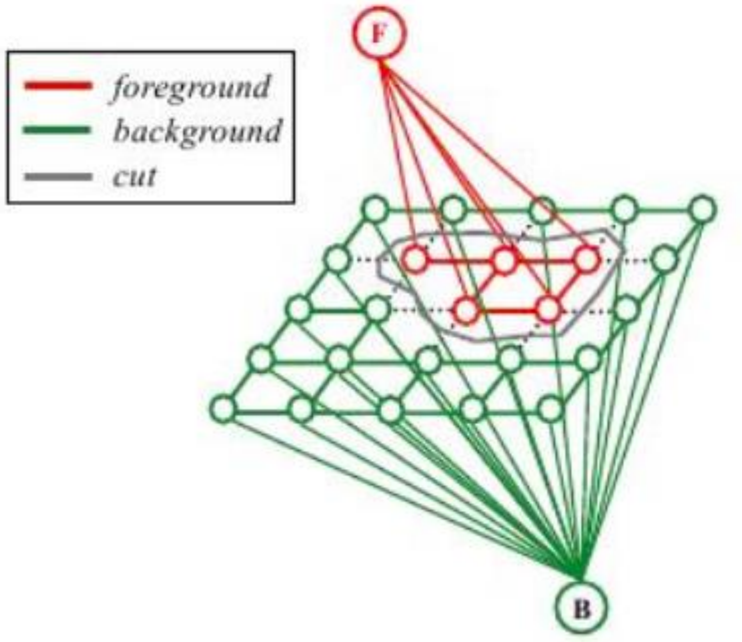


# Graph Cut

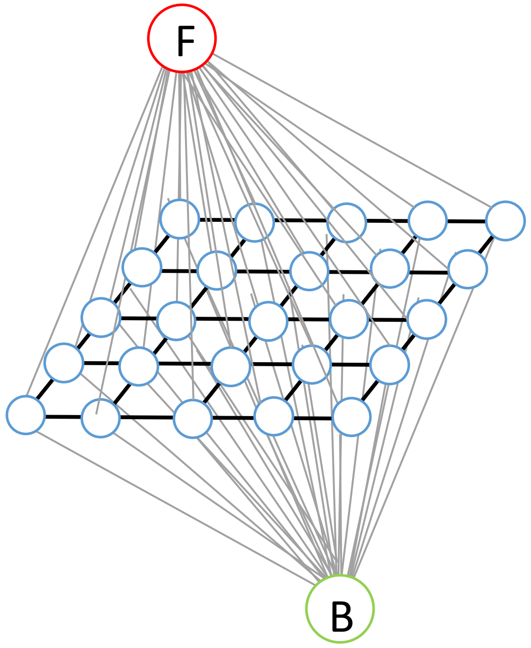
- A cut is a set of edges that are removed from the graph.

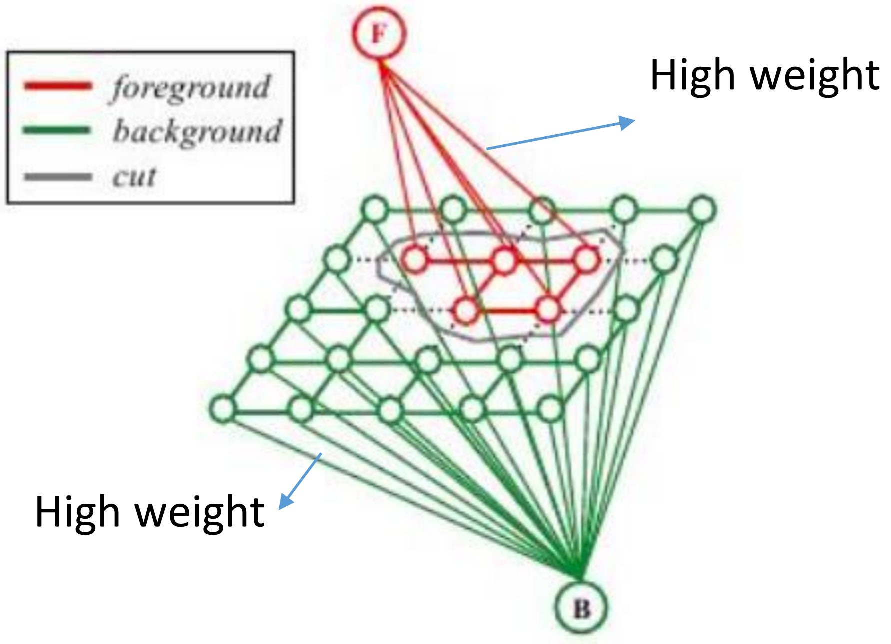


# Graph Cut

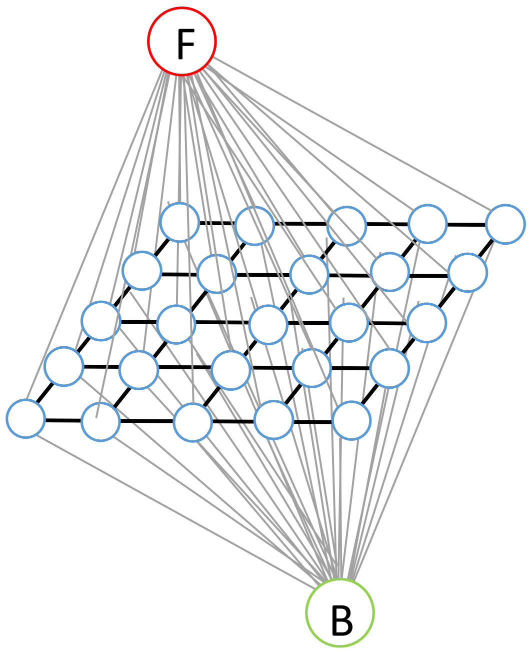
- If we assign a high weight to the edges that we want to keep, we are looking for a minimal cut (set of edges with low weights)



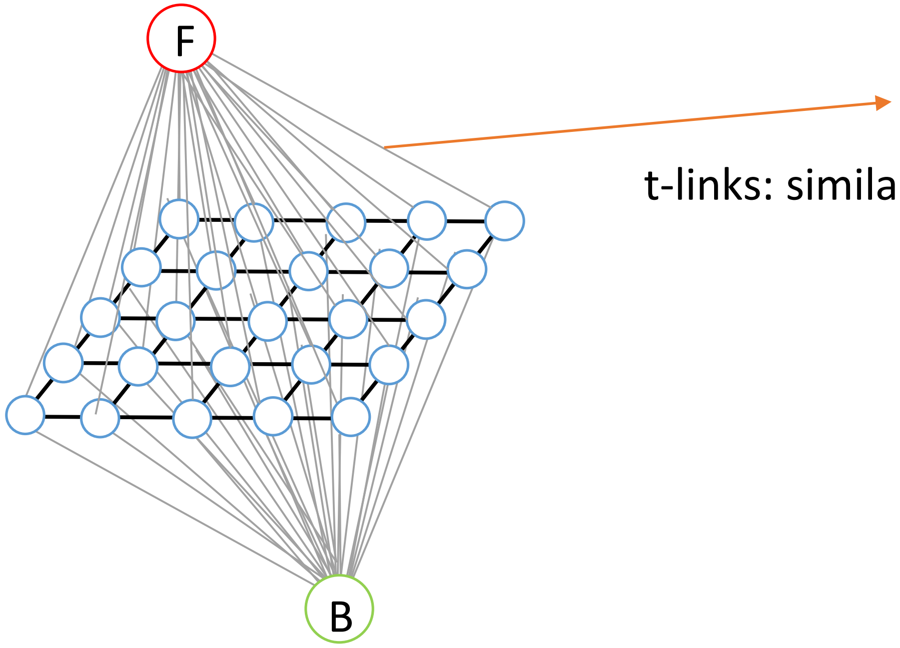
# Graph Cut

- What function do you suggest for weights of the edges?



# Graph Cut

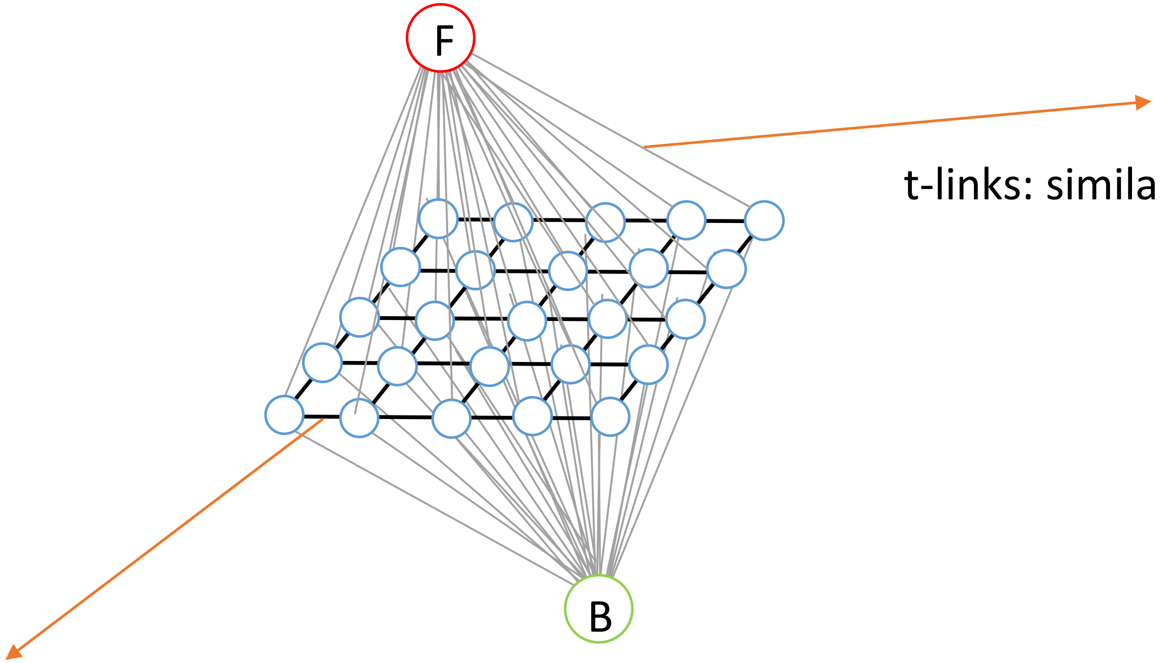
- What function do you suggest for weights of the edges?




t-links: similarity measure to the distribution of terminals

# Graph Cut

- What function do you suggest for weights of the edges?

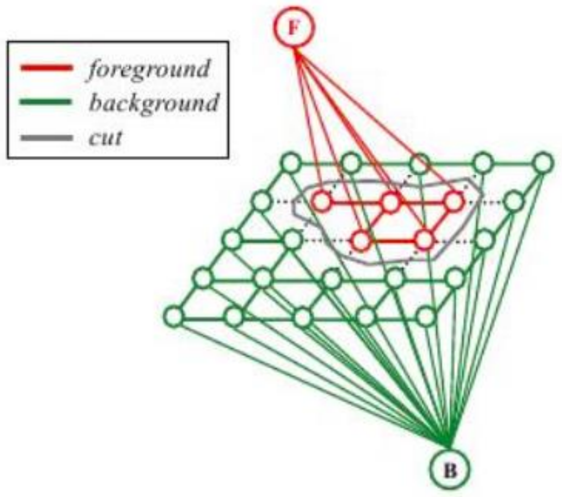



t-links: similarity measure to the distribution of terminals

n-links: $B(p, q)$ color similarity of two neighbors

# Graph Cut

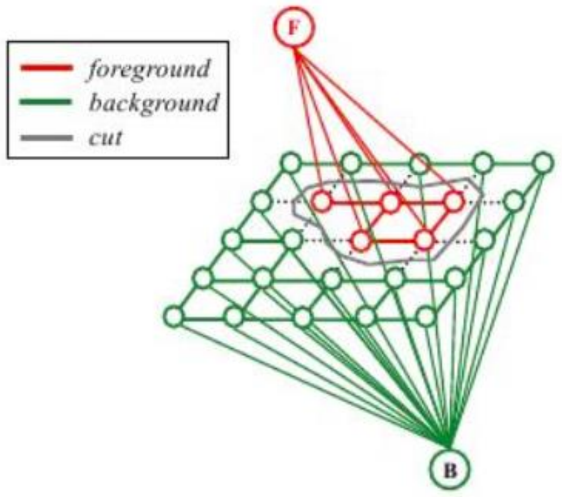
- We are looking to collect edges with minimum weights that separate terminal nodes F and B.



# Graph Cut

- We are looking to collect edges with minimum weights that separate terminal nodes F and B.

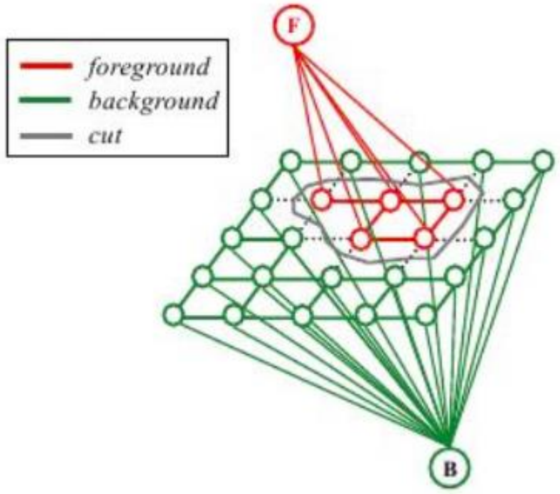
- Again, there is an exponential number of cuts.



# Graph Cut

- We are looking to collect edges with minimum weights that separate terminal nodes F and B.

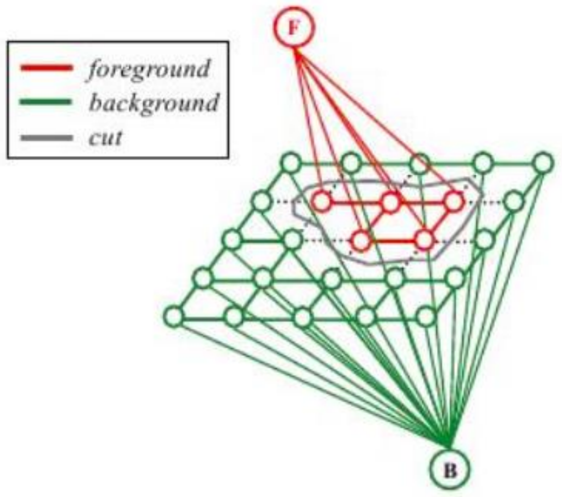
- Again, there is an exponential number of cuts. Why?



# Graph Cut

- We are looking to collect edges with minimum weights that separate terminal nodes F and B.

- Again, there is an exponential number of cuts. Why?
  - $2^{|E|}$



# Ford Fulkerson

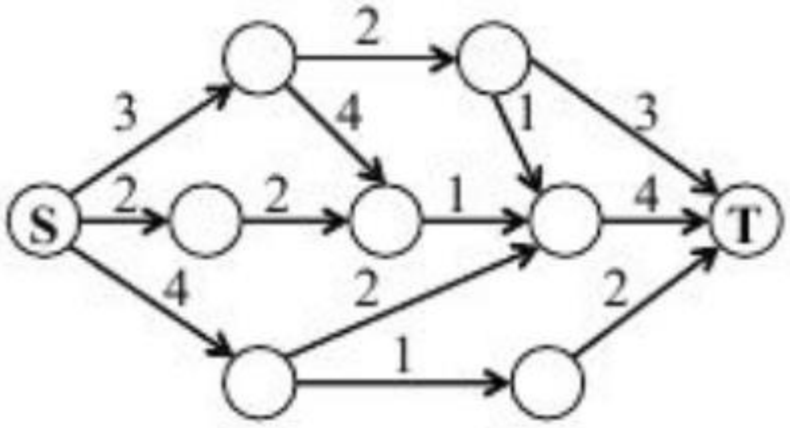
- This algorithm finds the minimum cut in $\mathrm{O}(|E|f)$, $f$ is the maximum number of flows.

# Ford Fulkerson

- This algorithm finds the minimum cut in $\mathrm{O}(|E|f)$, $f$ is the maximum number of flows.
- Let’s see what are flows.

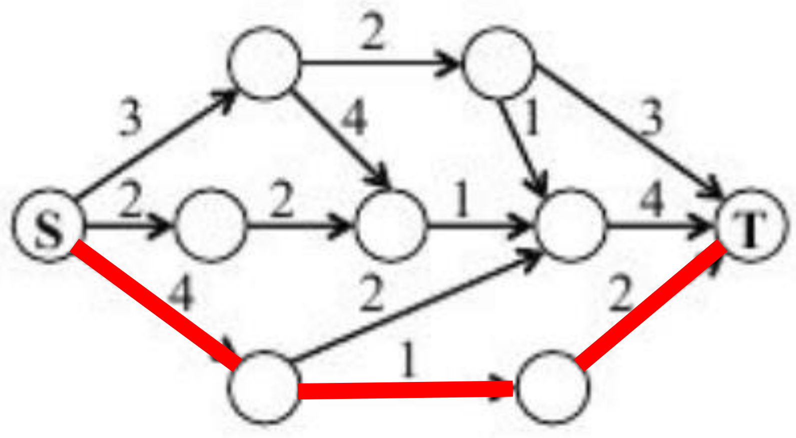
# Ford Fulkerson

- Let's say we have the following graph and we want to find its minimum cut from S to T.



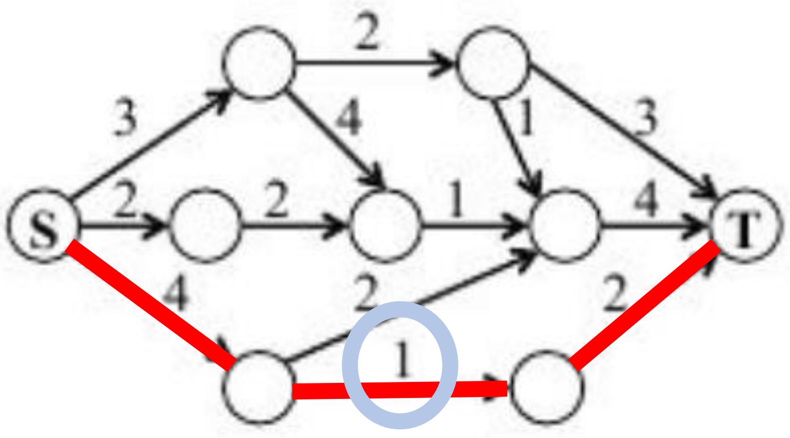
# Ford Fulkerson

- In the first step, we find the augmenting path which is a shortest path form S to T with nonzero weights on it.



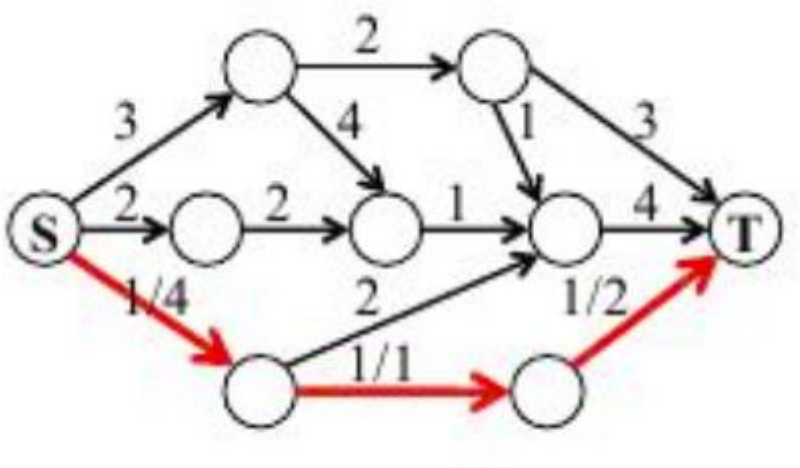
# Ford Fulkerson

- Second Step, we find the minimum edge



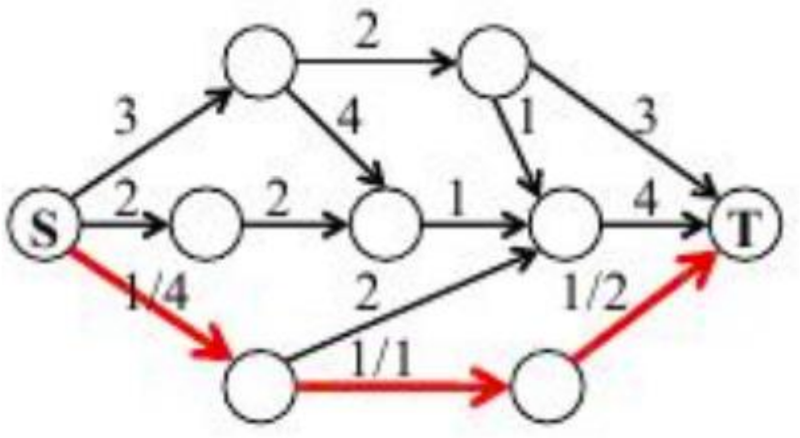
# Ford Fulkerson

- Second Step, we find the minimum edge and flow the path with that Value.



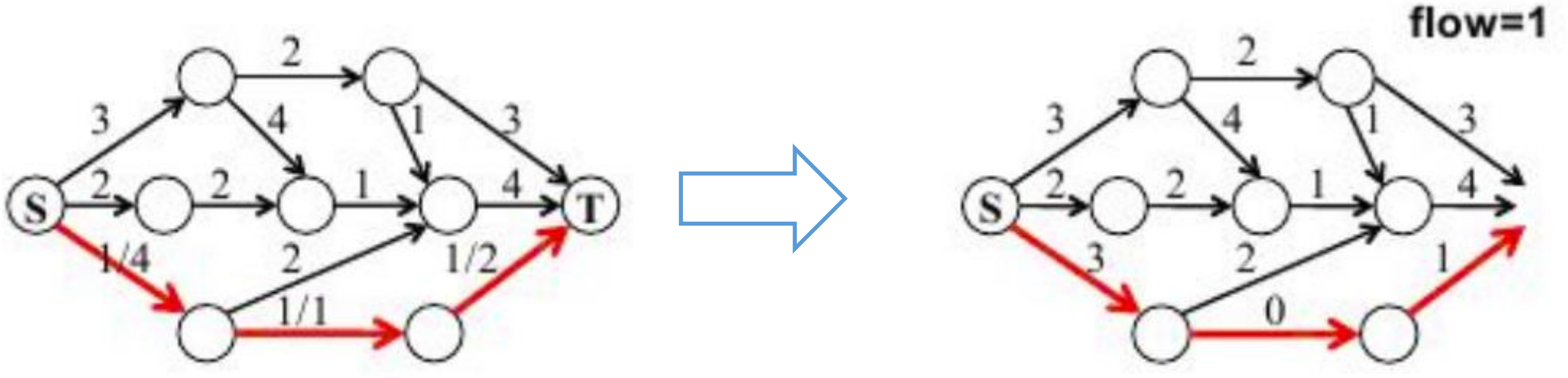
# Ford Fulkerson

- Third Step, we update the weight of all the nodes participating in the path by subtracting the amount of flow from the initial weights.



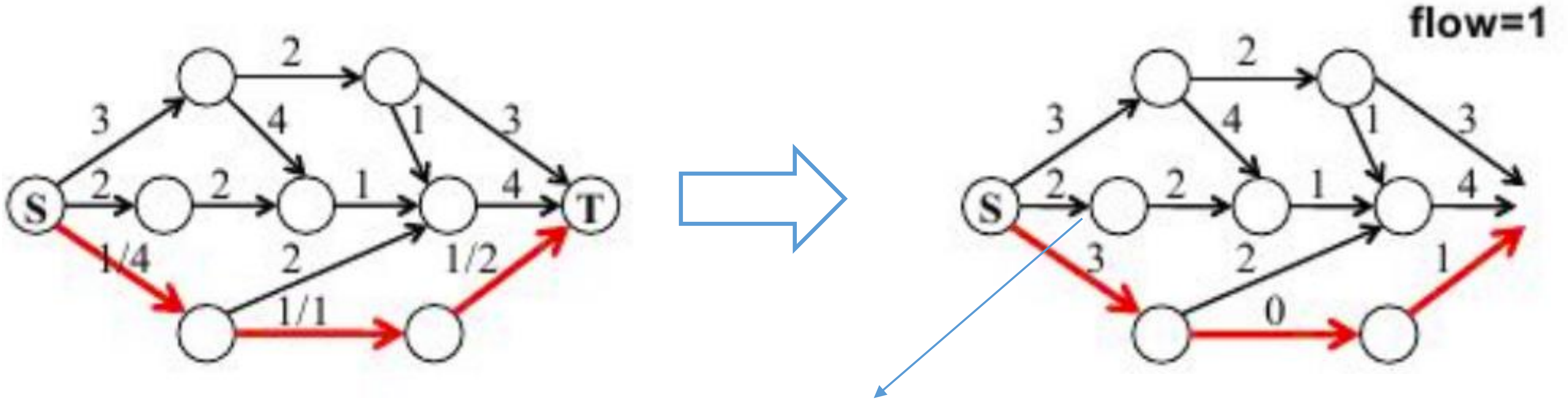
# Ford Fulkerson

- Third Step, we update the weight of all the nodes participating in the path by subtracting the amount of flow from the initial weights.



# Ford Fulkerson

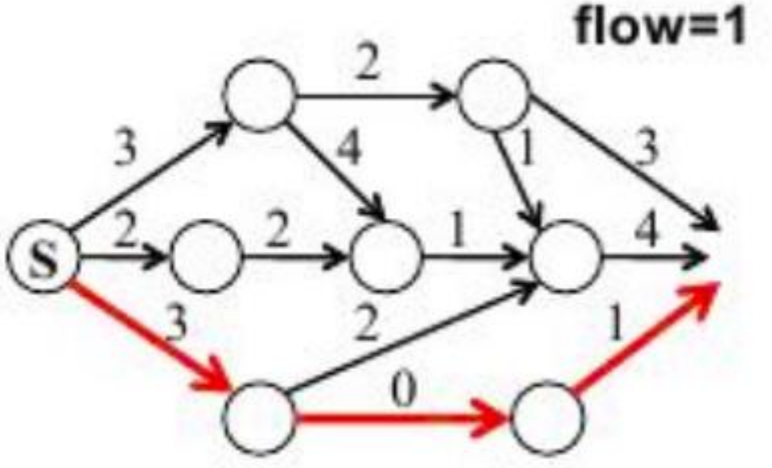
- Third Step, we update the weight of all the nodes participating in the path by subtracting the amount of flow from the initial weights.




Residual graph

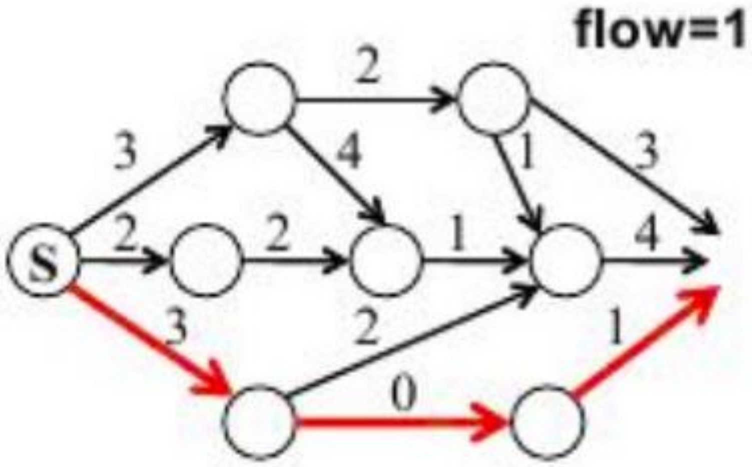
# Ford Fulkerson

- We repeat the same process on the residual graph.



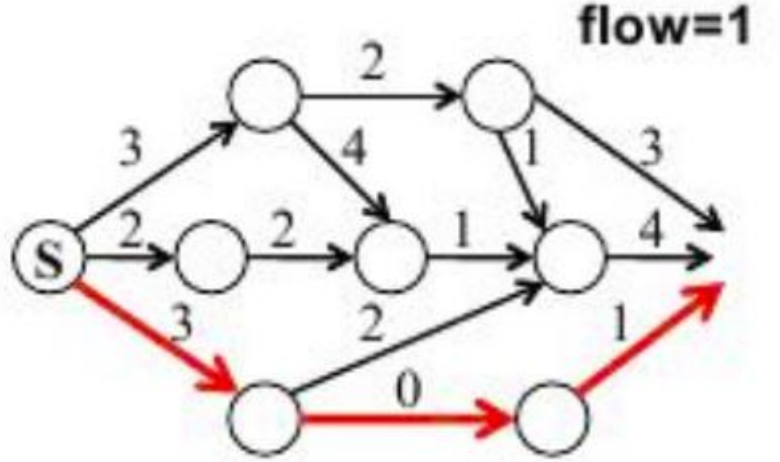
# Ford Fulkerson

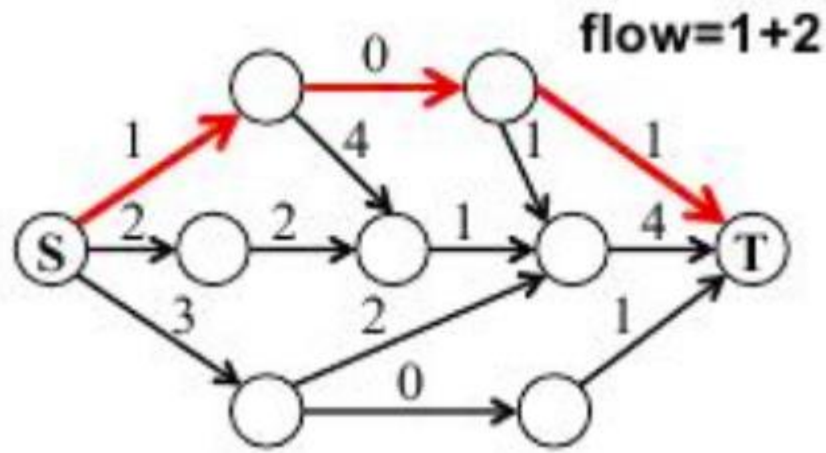
- We repeat the same process on the residual graph.
- The path that we went through is not picked again. why?



# Ford Fulkerson

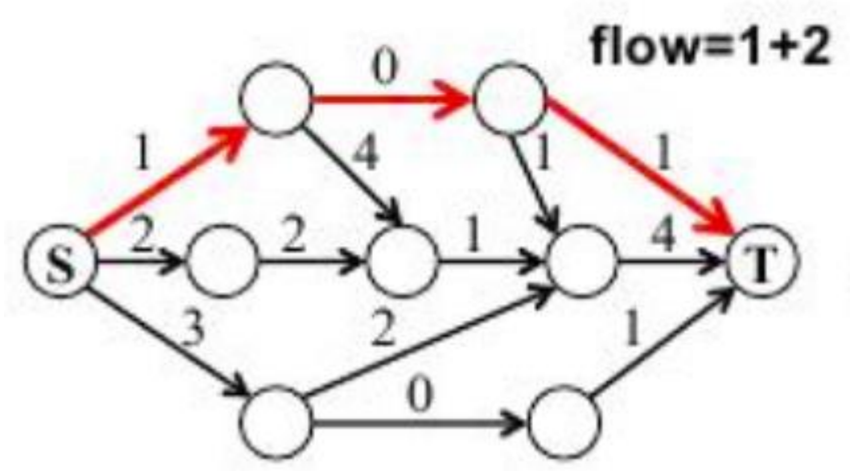
- Applying the same process:

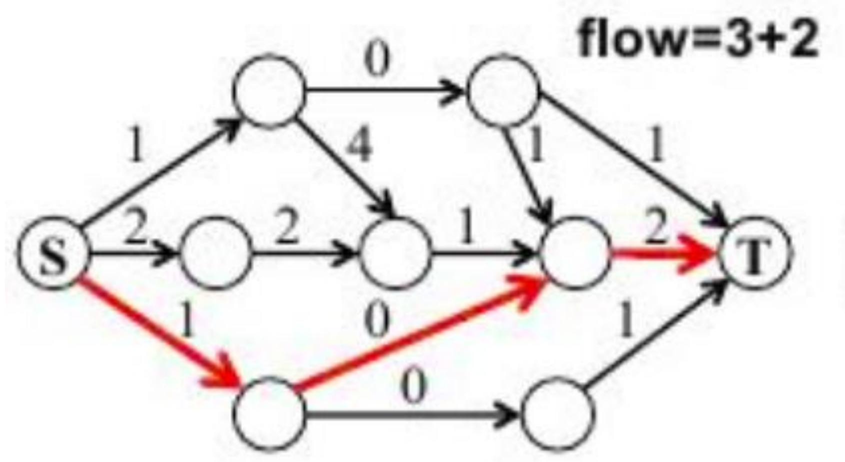


# Ford Fulkerson

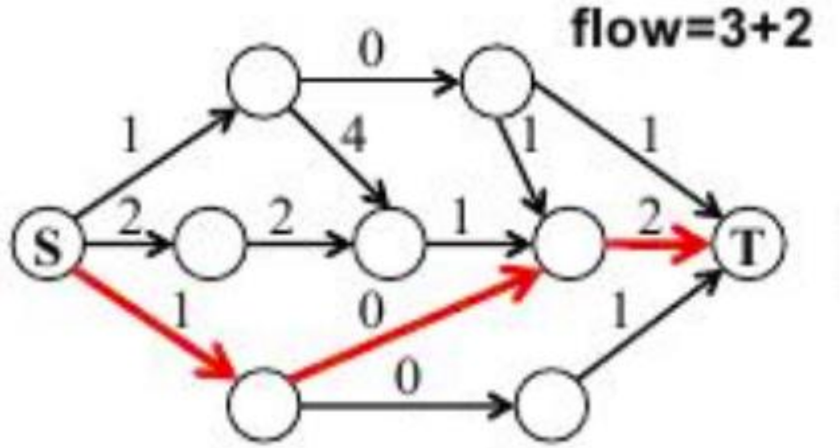
- Applying the same process:

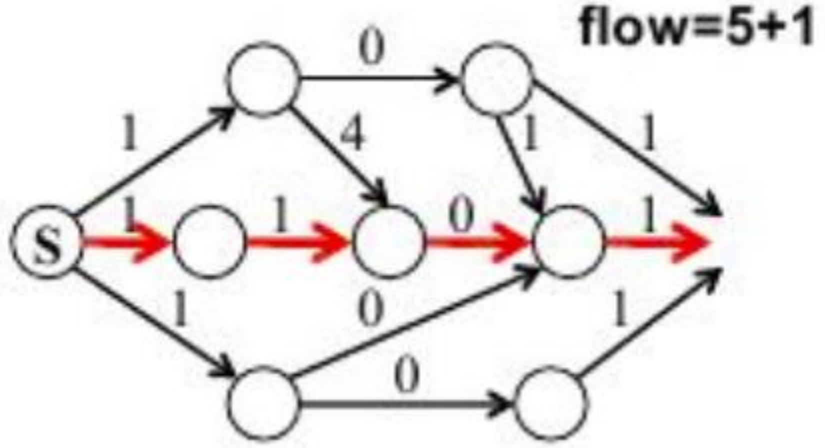


# Ford Fulkerson

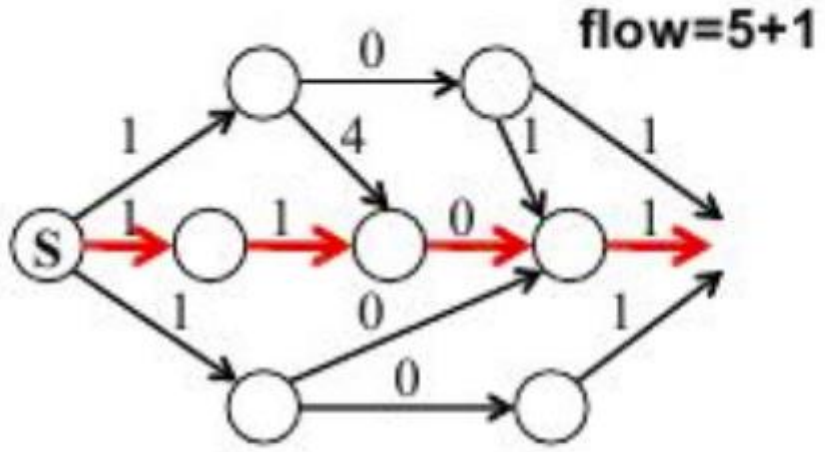
- Applying the same process:



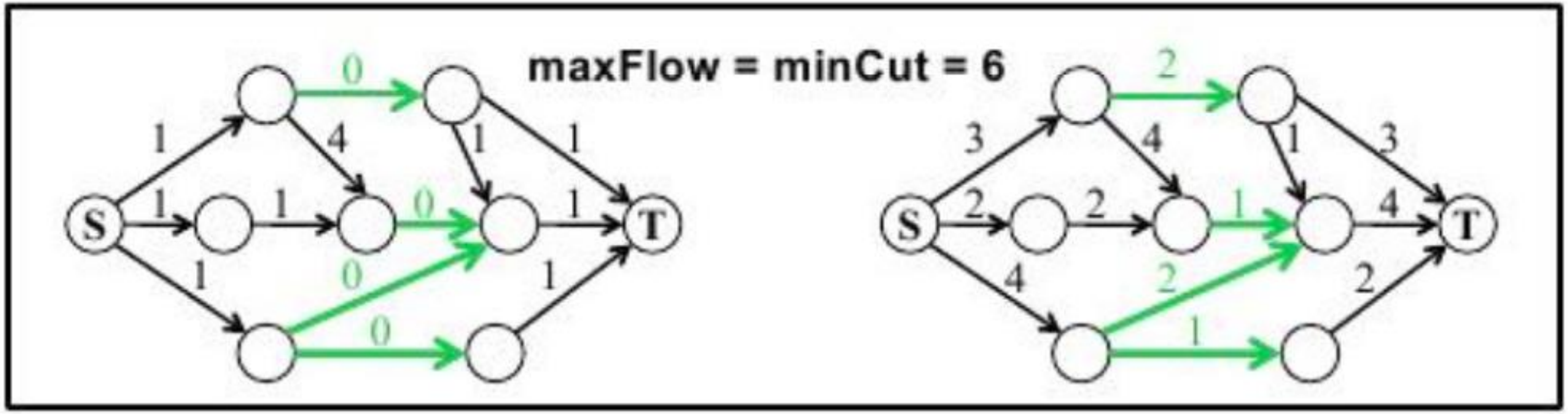
# Ford Fulkerson

- No augmenting path anymore, the cut is found



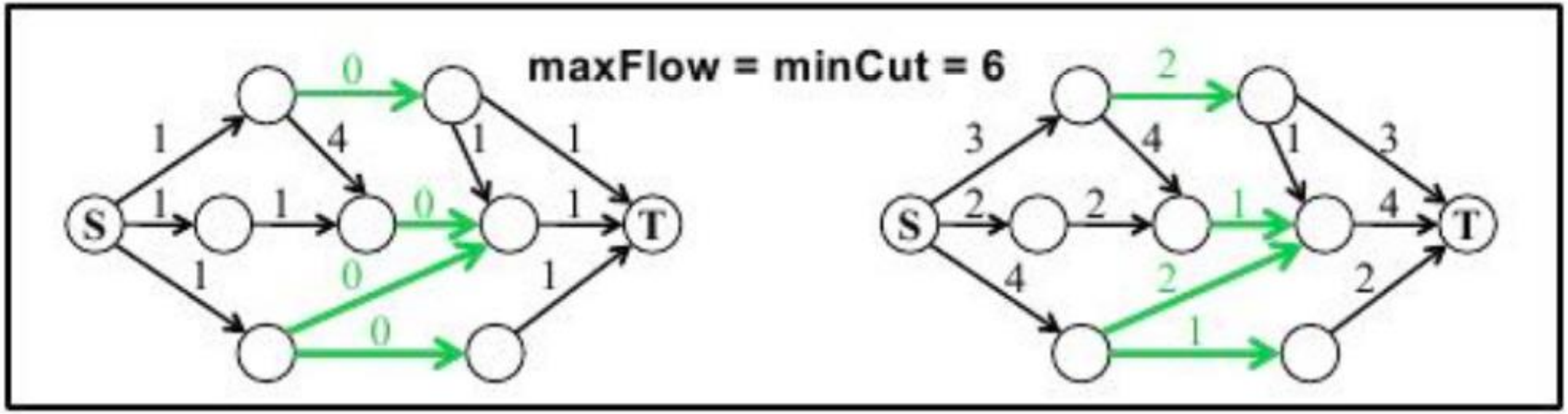
# Ford Fulkerson

- No augmenting path anymore, the cut is found



# Ford Fulkerson

- In fact, each iteration of this algorithm finds one edge belonging to the cut.



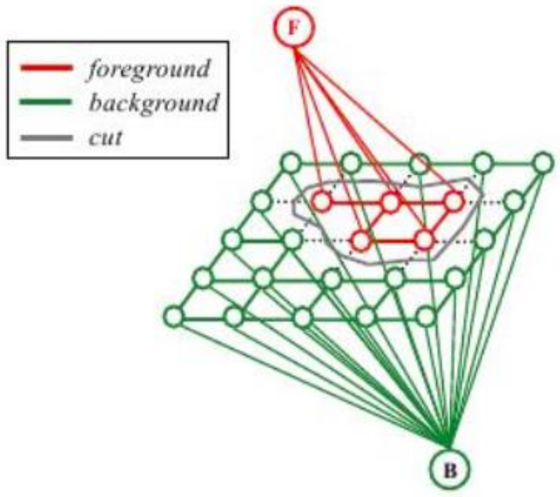
# Ford Fulkerson

- Algorithm

```
Ford--Fulkerson( graph G,node s, node t ) {
  Initialize f(e) = 0 for all e ∈ E.
  Repeat {
     Find an augmenting path A in G between s and t
     Augment f(e) for each edges e ∈ A
     G = calculate residual graph using A
  } Until no more augmenting paths found in G
}
```

# Recap

- Connect all pixels to two terminal nodes representing F/B.
- Assign proper weights to the edges.
- Apply Ford Fulkerson algorithm to separate F/B nodes.



# Questions?