CS448f: Image Processing For Photography and Vision

Graph Cuts

Seam Carving

- Video
- Make images smaller by removing “seams”
- Seam = connected path of pixels
  - from top to bottom
  - or left edge to right edge
- Don’t want to remove important stuff
  - importance = gradient magnitude

Finding a Good Seam

- How do we find a path from the top of an image to the bottom of an image that crosses the fewest gradients?

Finding a Good Seam

- Recursive Formulation:
- Cost to bottom at pixel x =

  gradient magnitude at pixel x +

  min(cost to bottom at pixel below x,

  cost to bottom at pixel below and right of x,

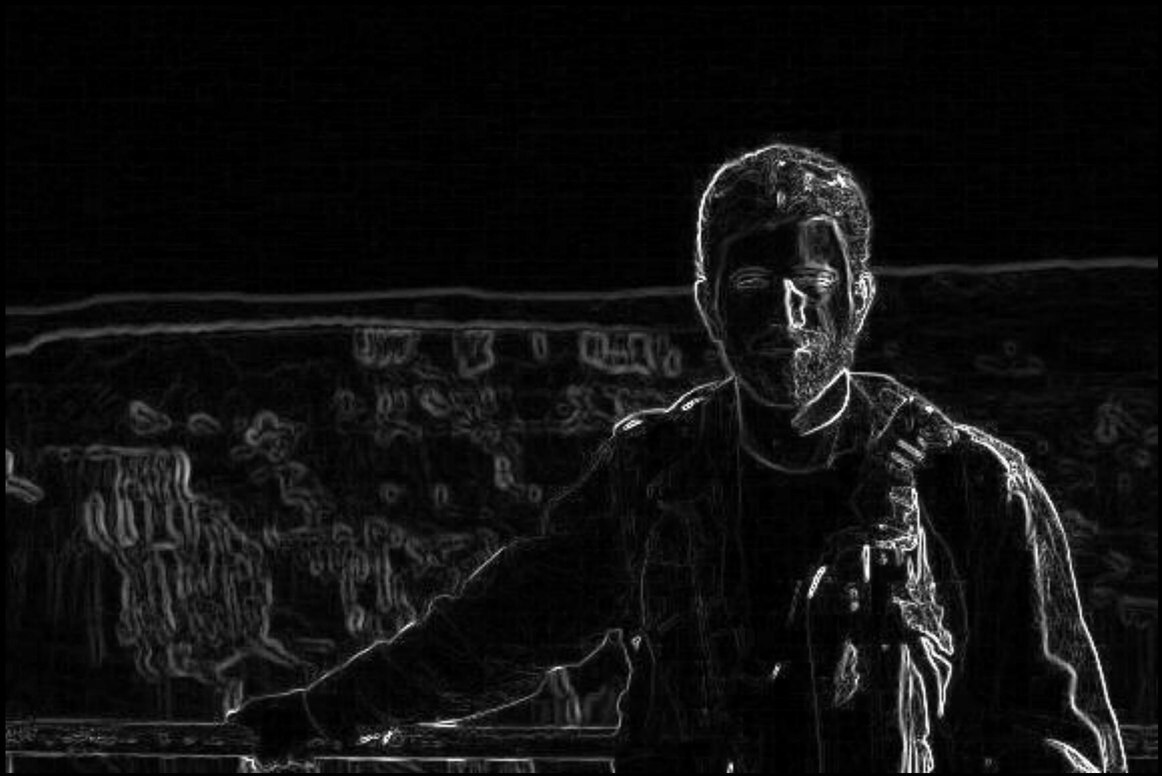
  cost to bottom at pixel below and left of x)

Dynamic Programming

- Start at the bottom scanline and work up, computing cheapest cost to bottom
  - Then, just walk greedily down the image

```
for (int y = im.height-2; y >= 0; y--) {
   for (int x = 0; x < im.width; x++) {
        im(x, y)[0] += min3(im(x, y+1)[0],
                            im(x+1,y+1)[0],
                            im(x-1, y+1)[0]);
   }
}
```

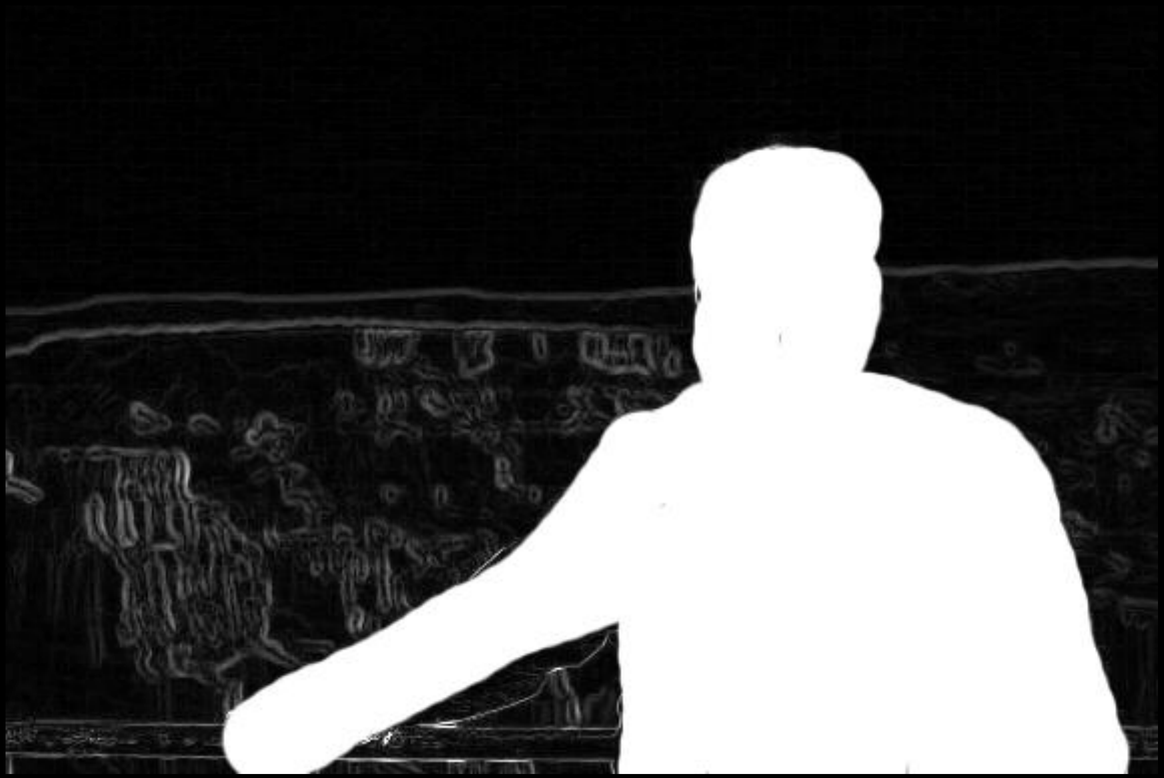
Instead of Finding Shortest Path Here:



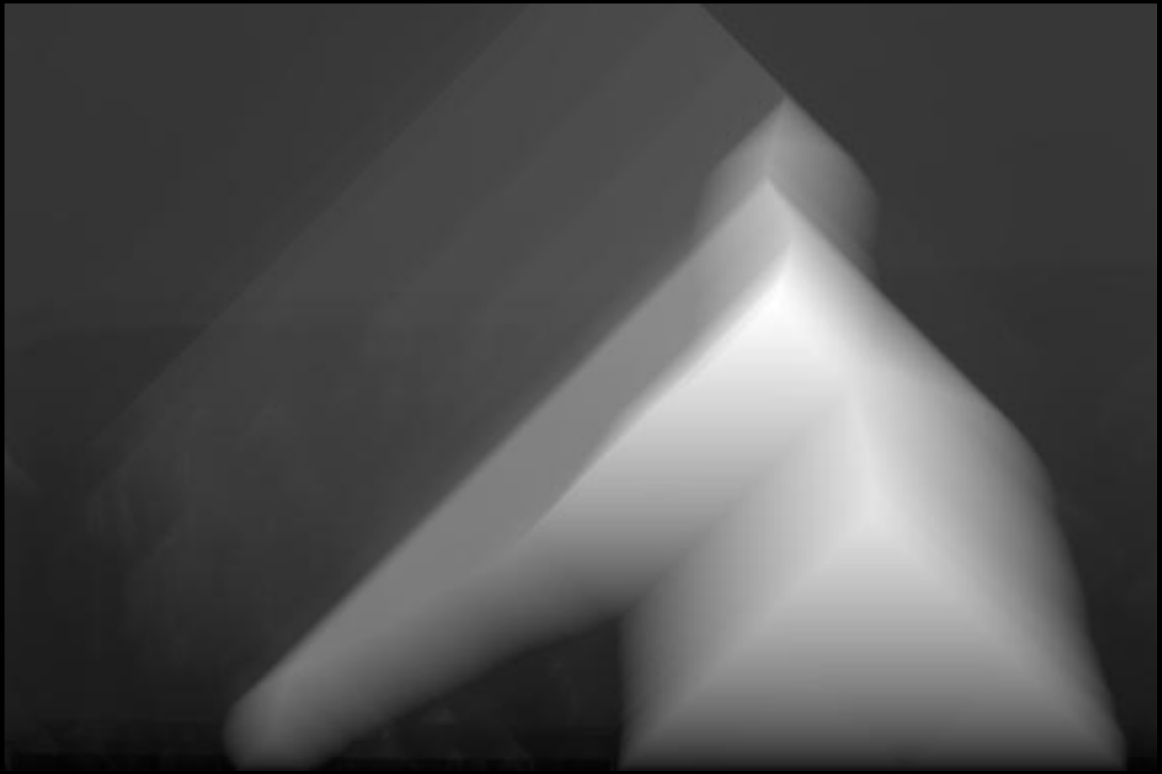
We greedily walk down this:

We greedily walk down this:

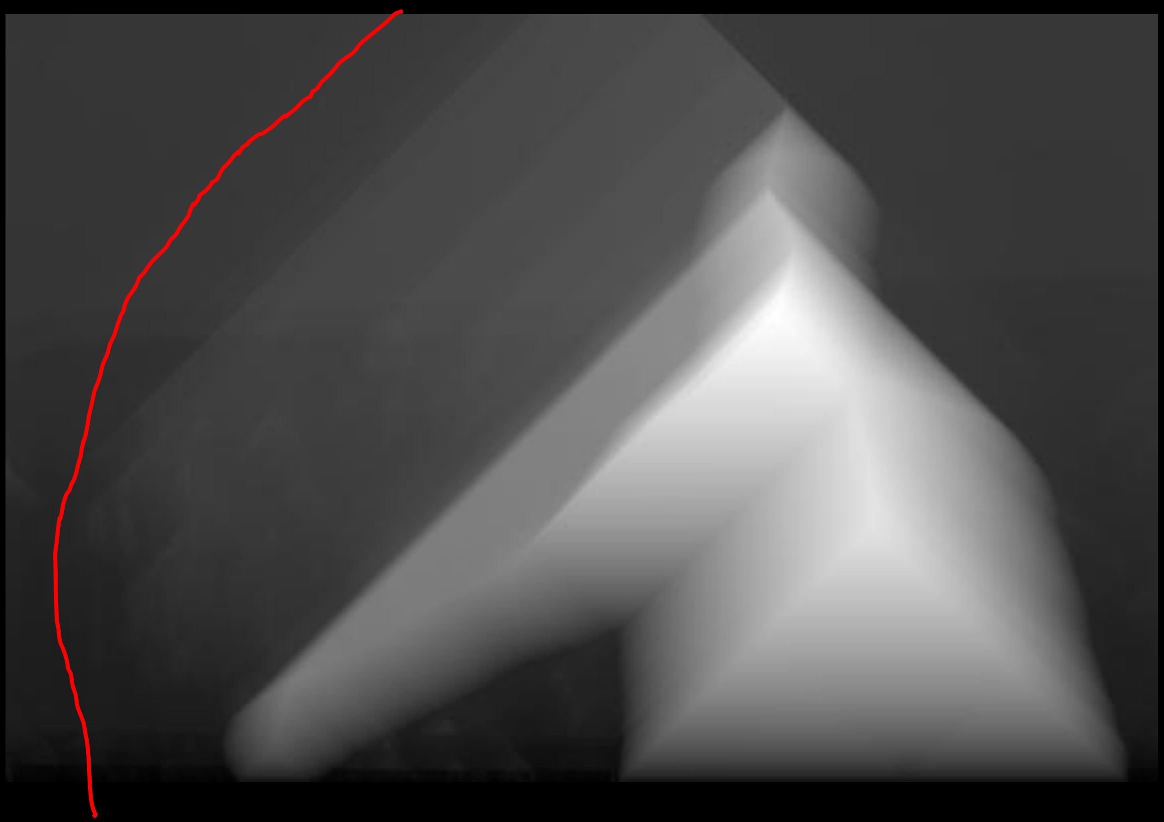
Protecting a region:



Protecting a region:



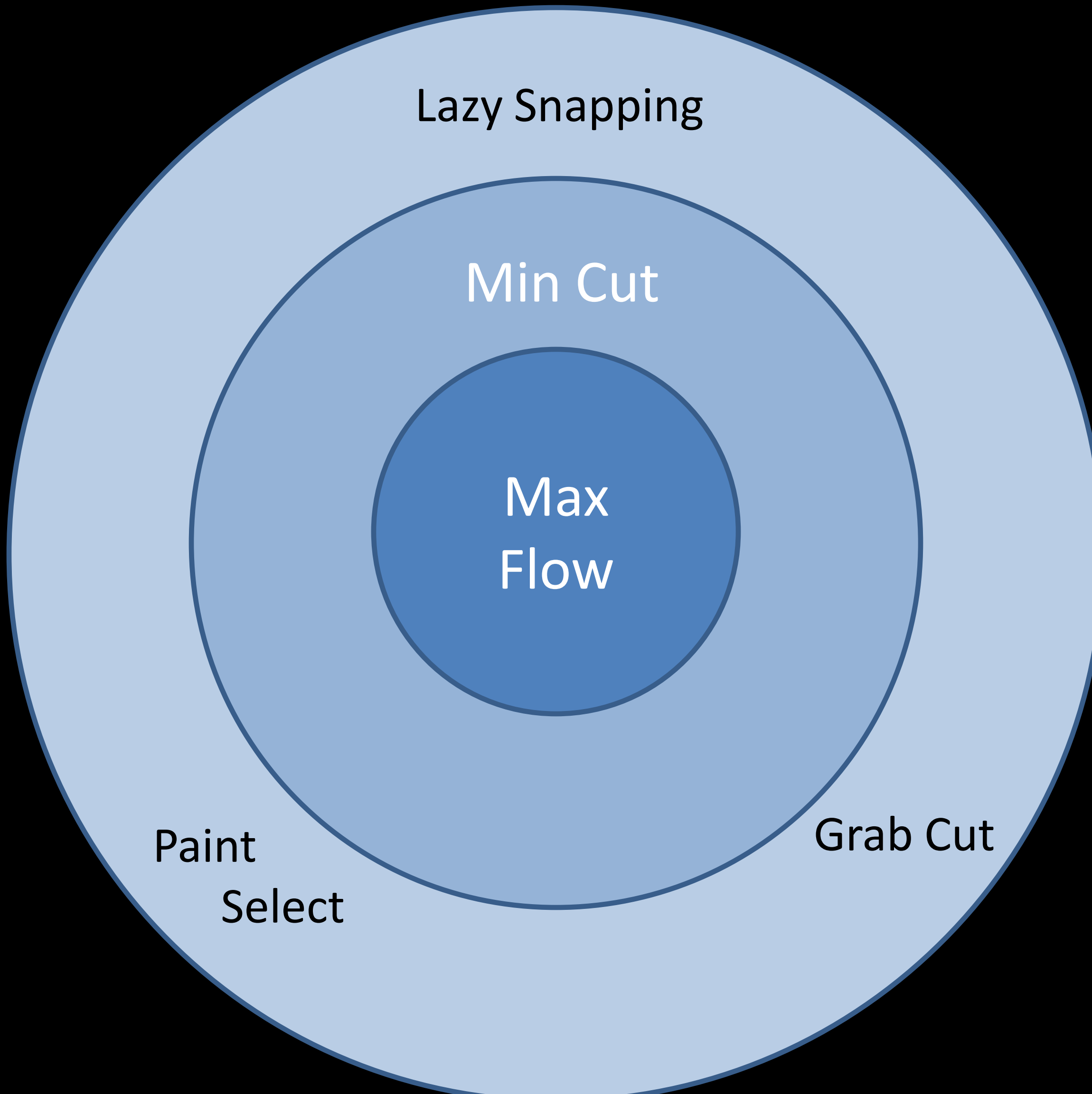
Protecting a region:



Demo

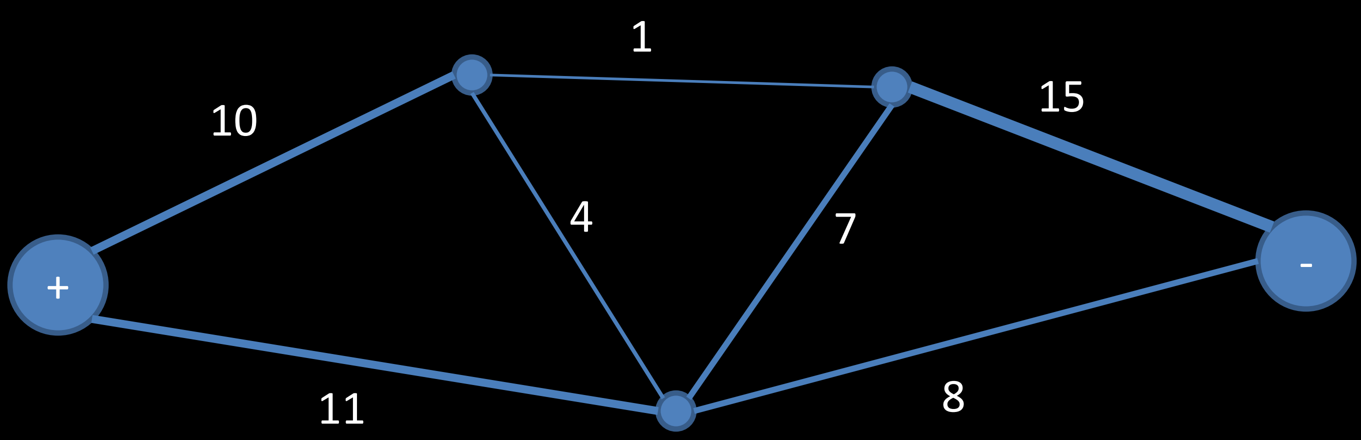
How Does Quick Selection Work?

- All of these use the same technique:
  - picking good seams for poisson matting
    - (gradient domain cut and paste)
    - pick a loop with low contrast
  - picking good seams for panorama stitching
    - pick a seam with low contrast
  - picking boundaries of objects (Quick Selection)
    - pick a loop with high contrast


Lazy Snapping
Min Cut
Max
Flow
Paint
Select
Grab Cut

Max Flow

- Given a network of links of varying capacity, a source, and a sink, how much flows along each link?



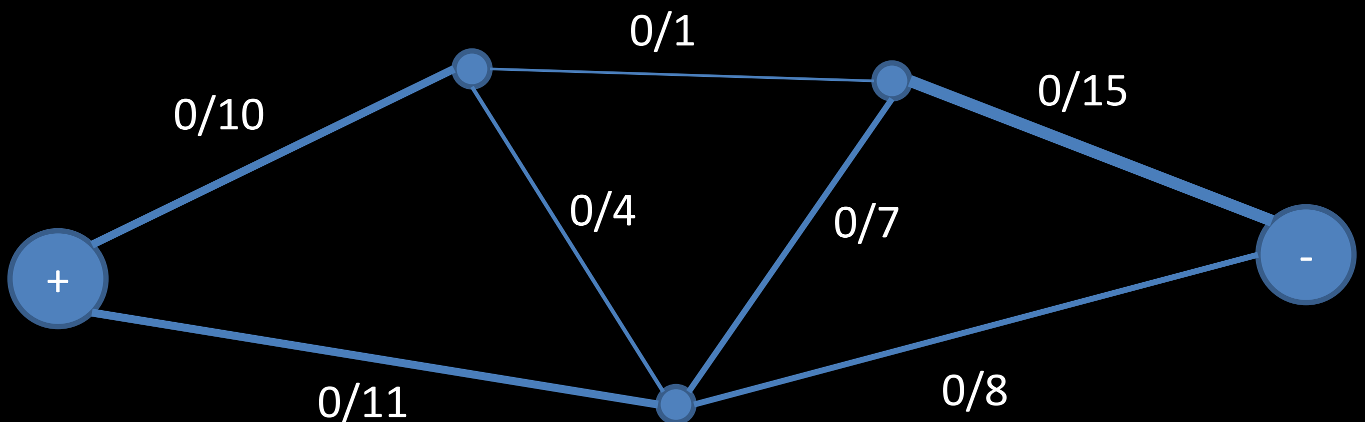
Aside: It’s Linear Programming

- One variable per edge (how much flow)
- One linear constraint per vertex
  - flow in = flow out
- Two inequalities per edge
  - -capacity < flow < capacity
- One linear combination to maximize
  - Total flow leaving source
  - Equivalently, total flow entering sink

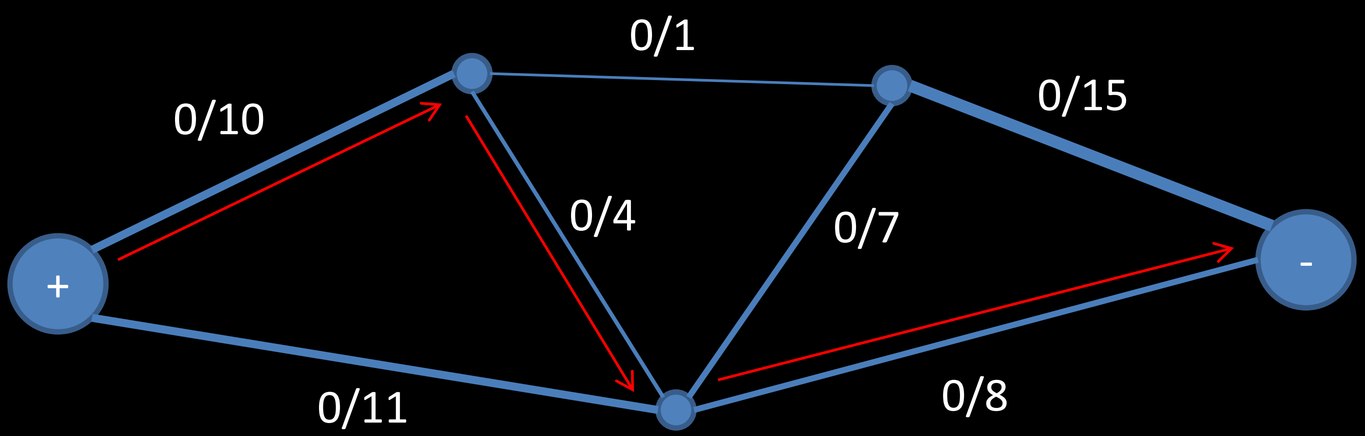
Aside: It’s Linear Programming

- The optimimum occurs at the boundary of some high-D simplex
  - Some variables are maxed out, the others are then determined by the linear constraints
- The Simplex method:
  - Start from some valid state
  - Find a way to max out one of the variables in an attempt to make the solution better
  - Repeat until convergence

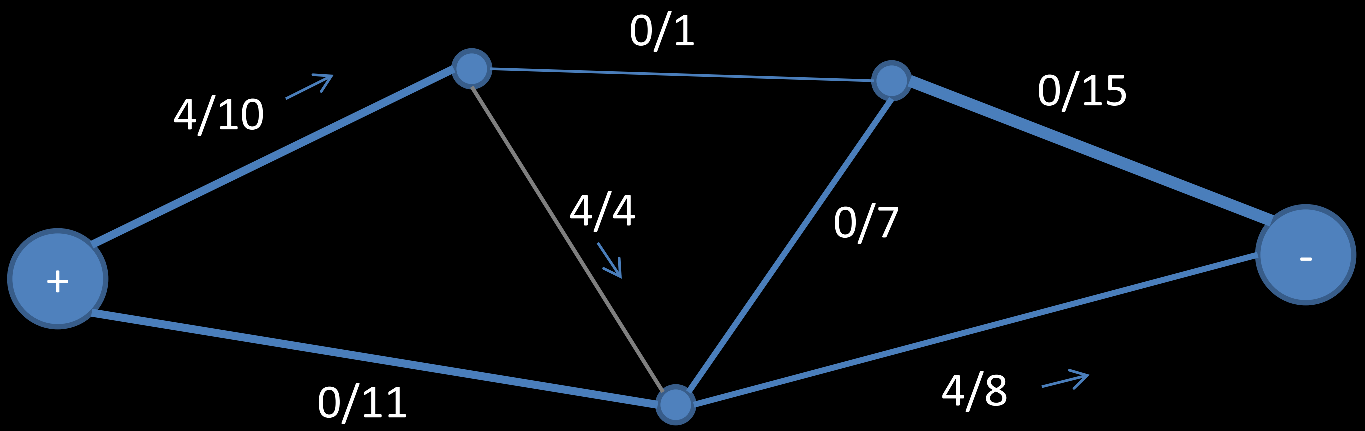
Start with no flow



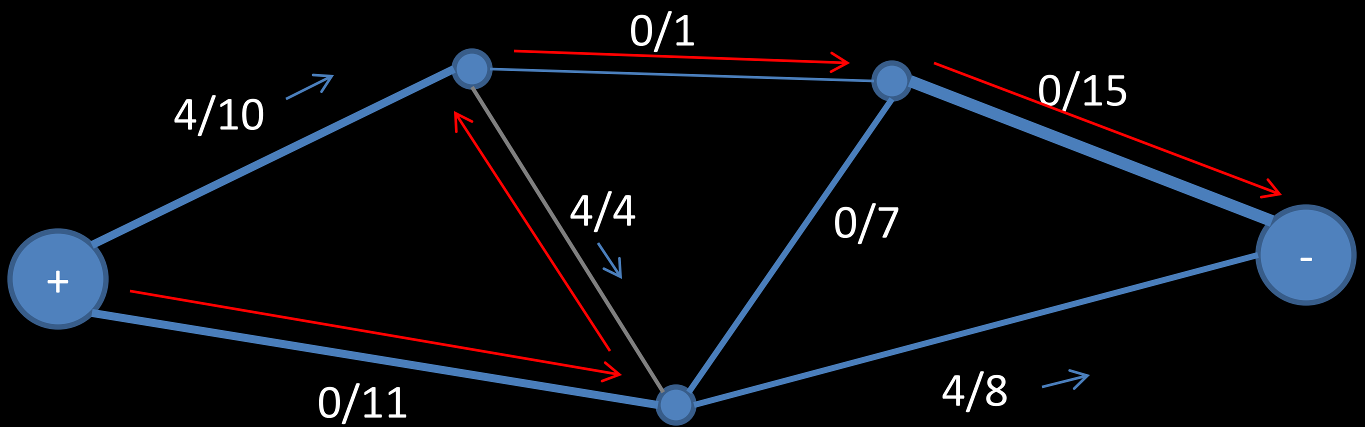
Find path from source to sink with capacity



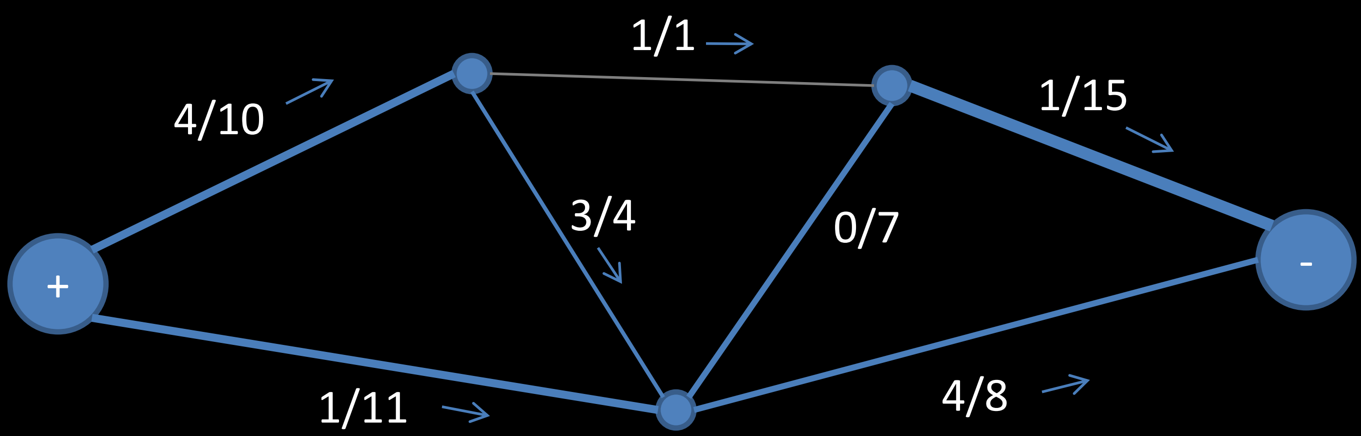
Max out that path
Keep track of direction



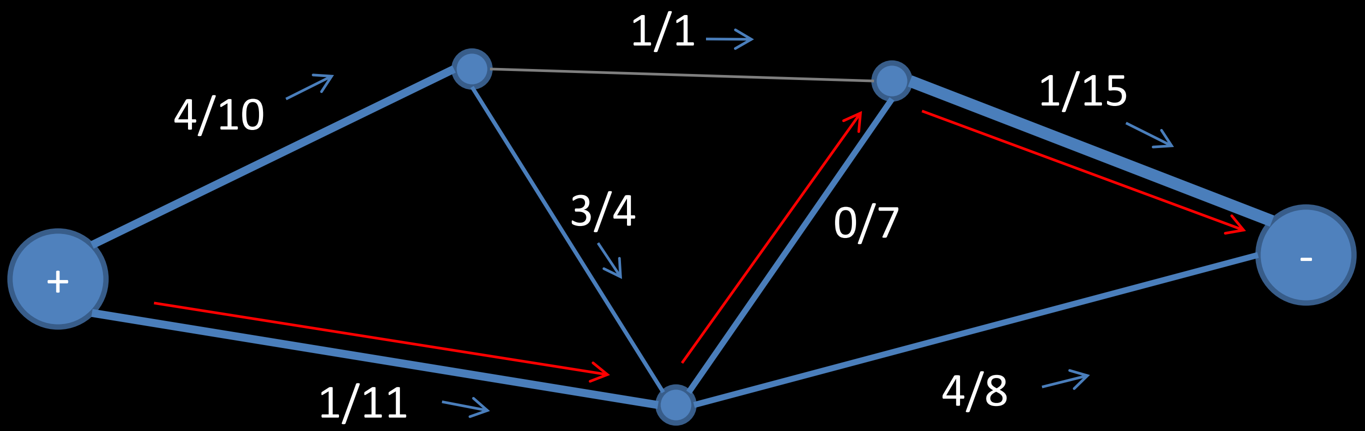
Repeat



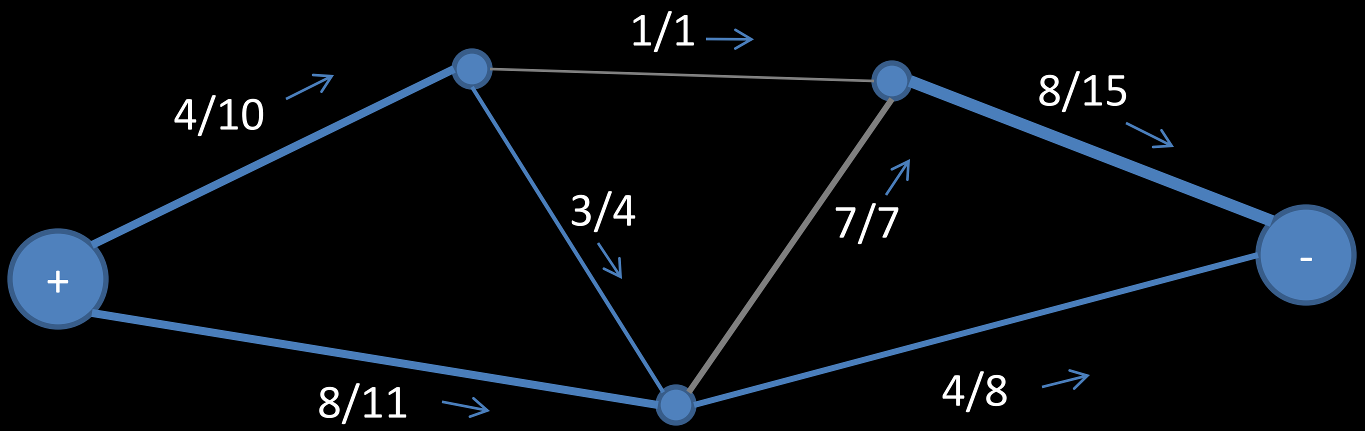
A maxed out edge can only be used in the other direction



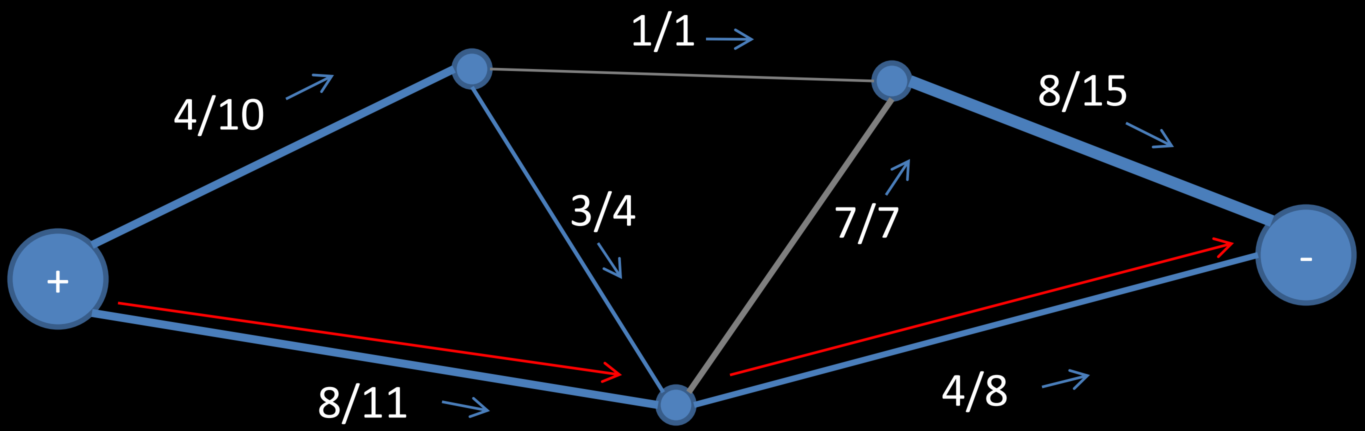
Continue...



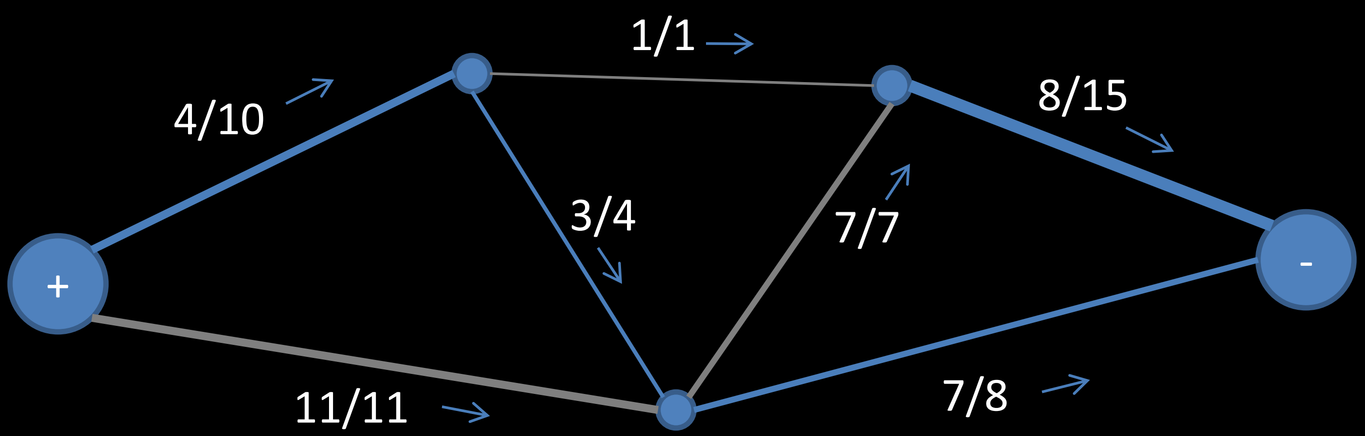
Continue...



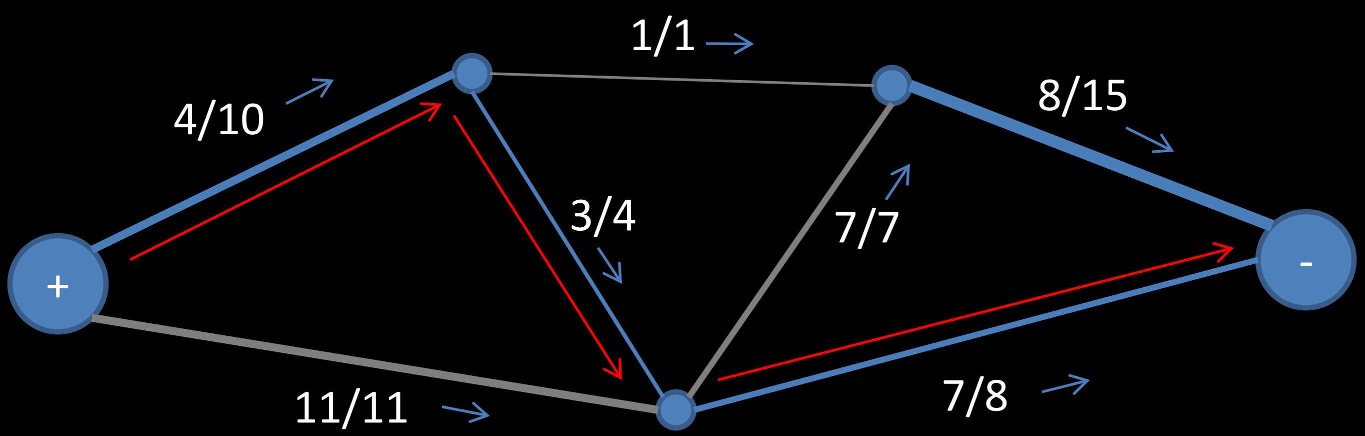
Continue...



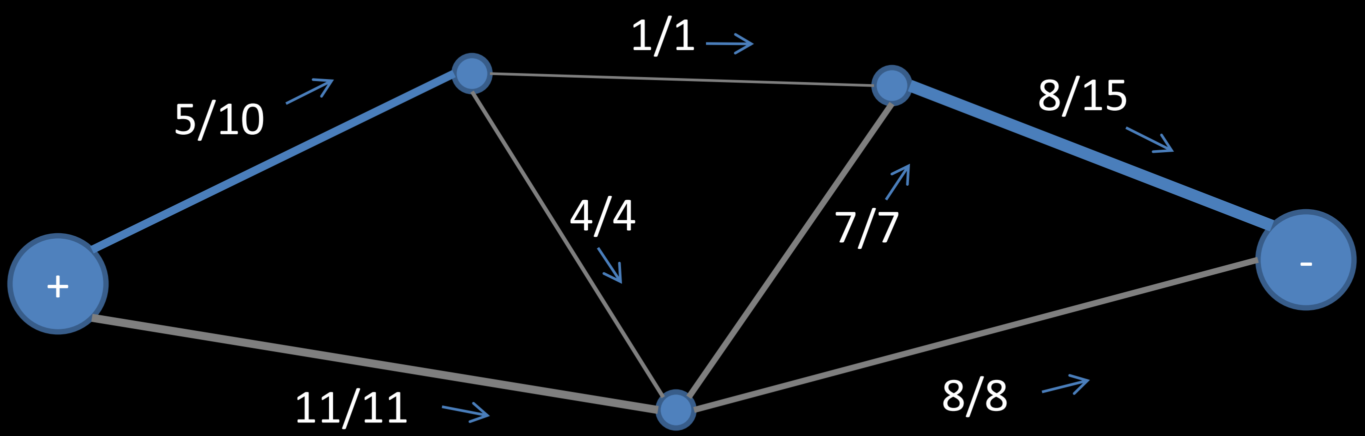
Continue...



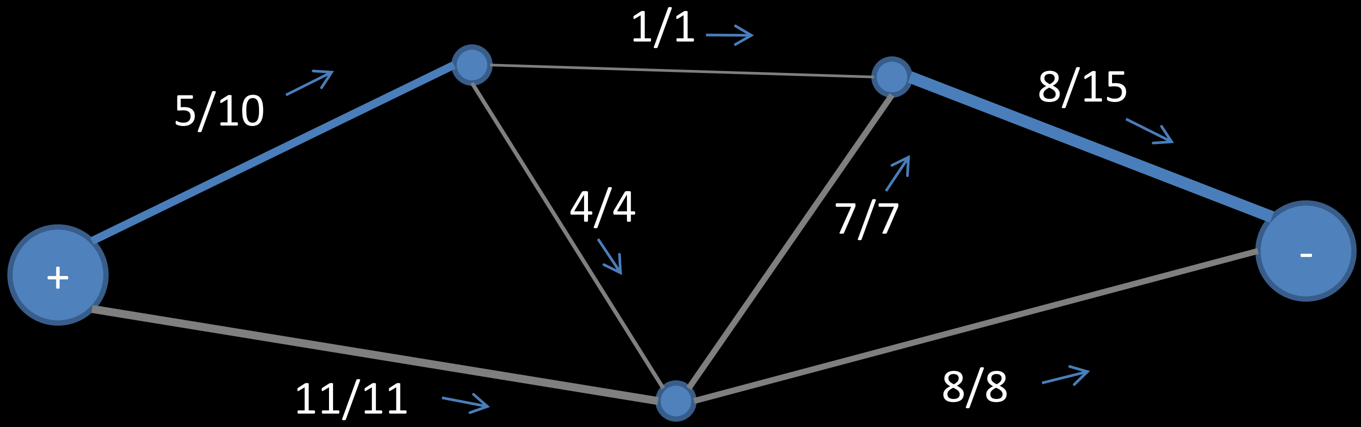
Only one path left...



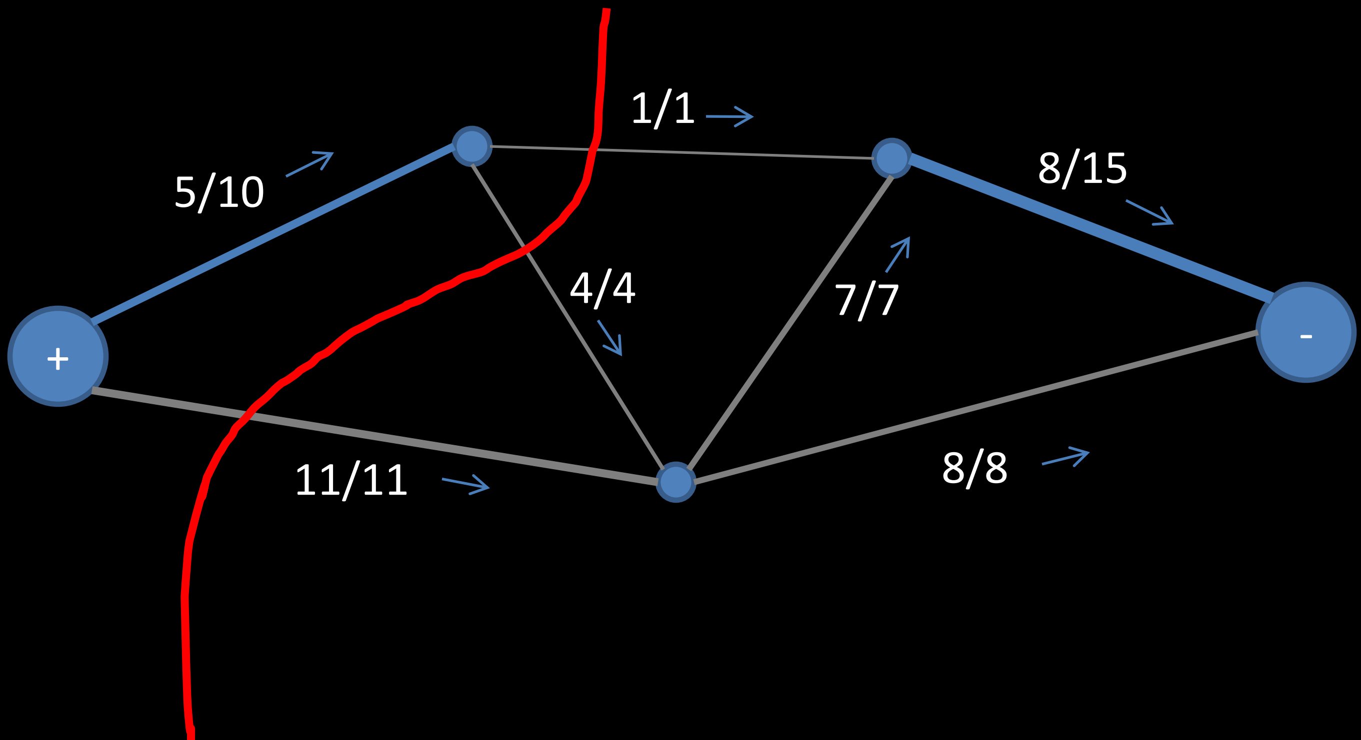
No paths left. Done.



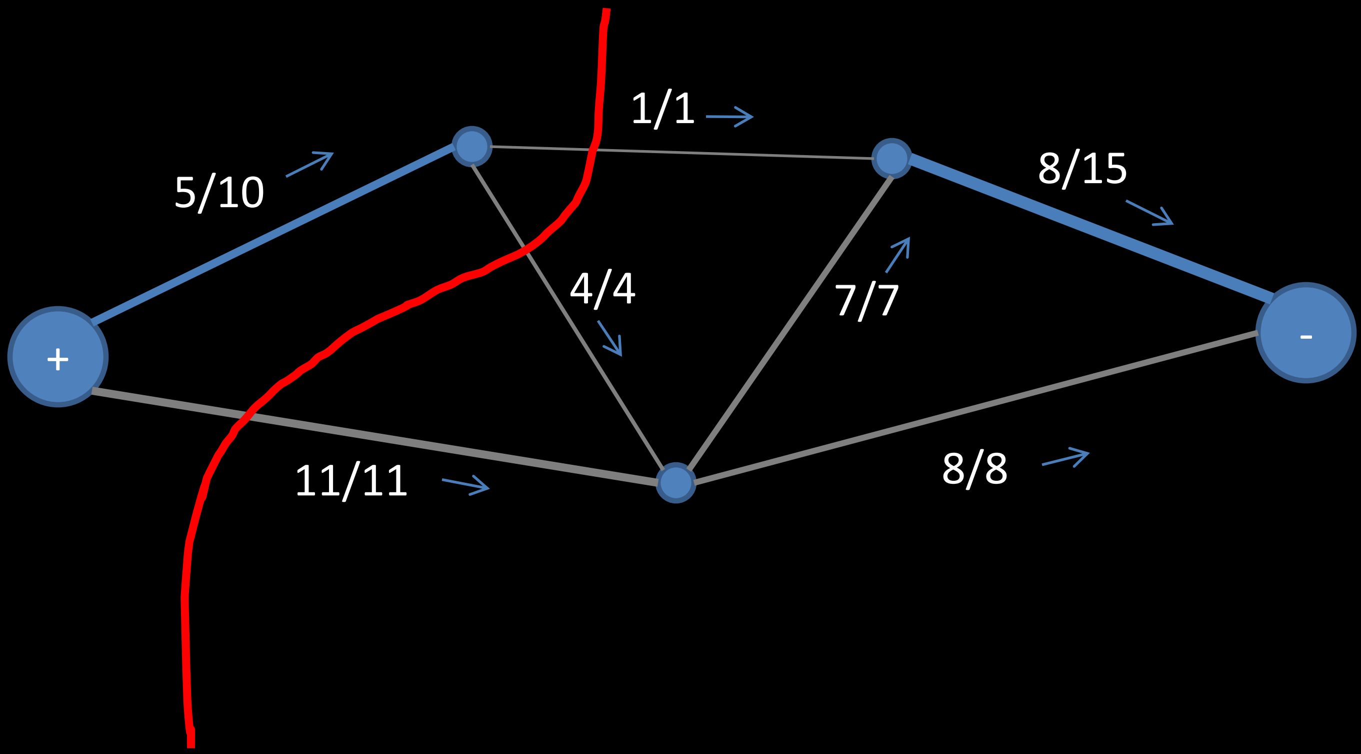
The congested edges represent the bottleneck



Cutting across them cuts the graph while removing the minimum amount of capacity



Max Flow = Min Cut

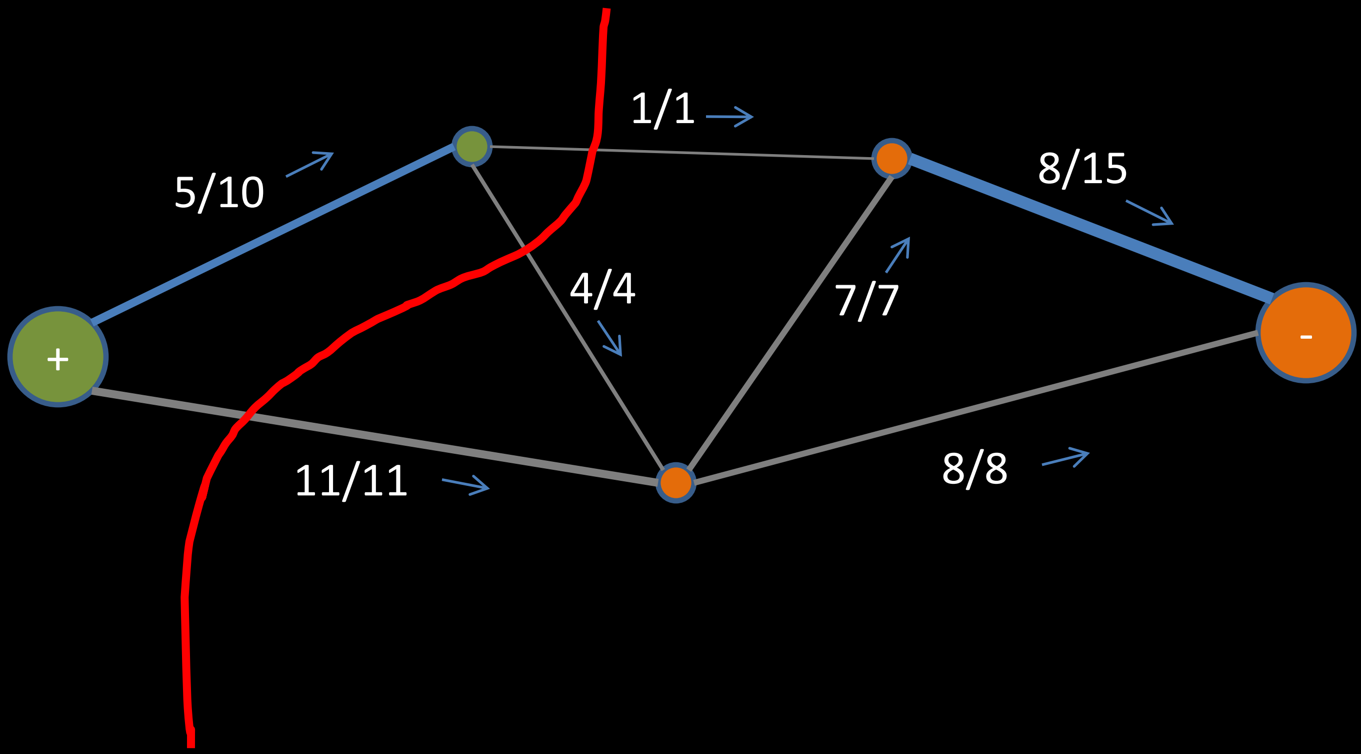


Cut Cost = 1 + 4 + 11 = 16

Everything Reachable from Source

vs

Everything Else

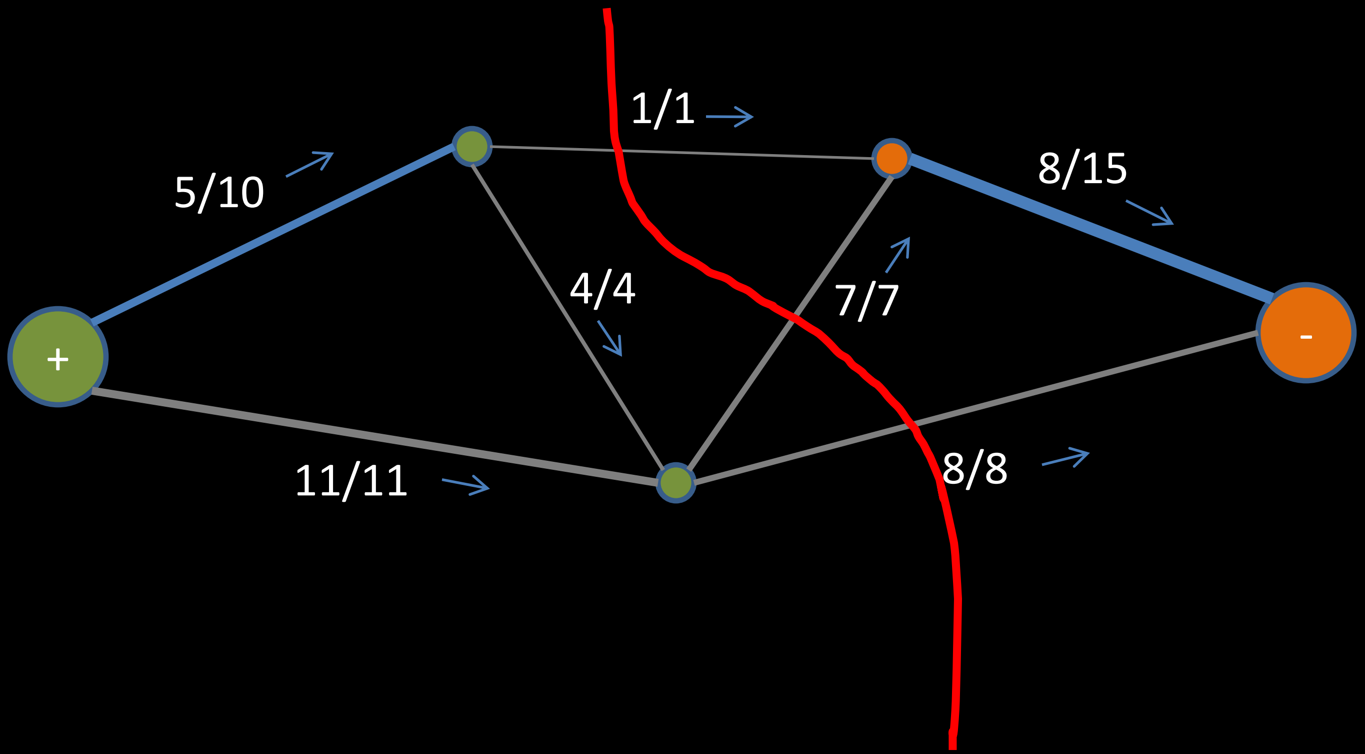


Cut Cost = 1 + 4 + 11 = **16**

Everything Reachable from Sink

vs

Everything Else



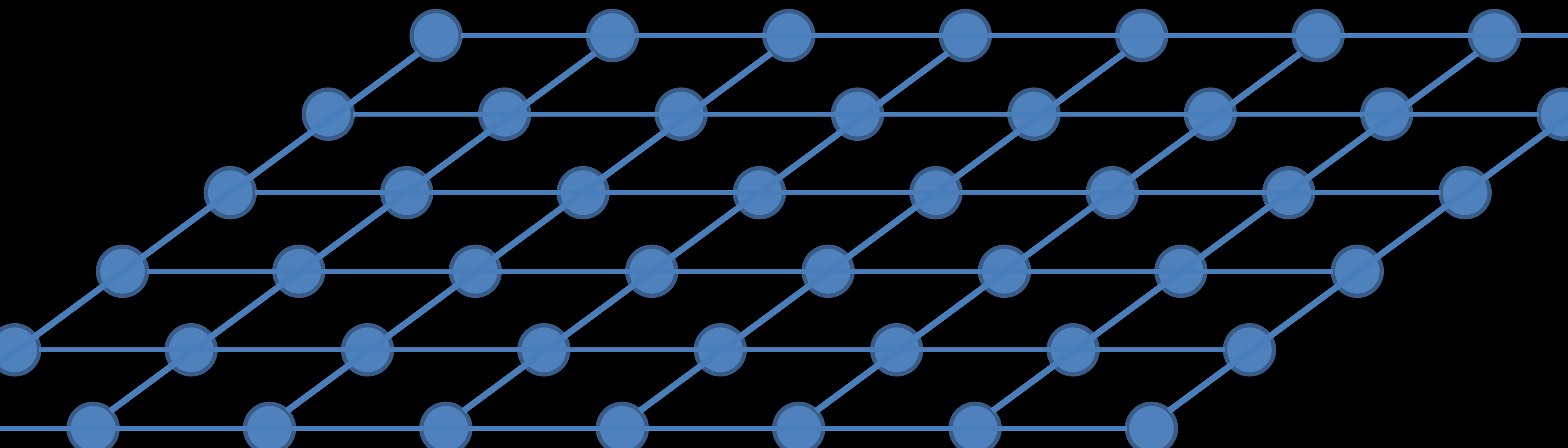
Cut Cost = 1 + 7 + 8 = **16**

Aside: Linear Programming

- It turns out min-cut is the dual linear program to max-flow
- So optimizing max flow also optimizes min-cut

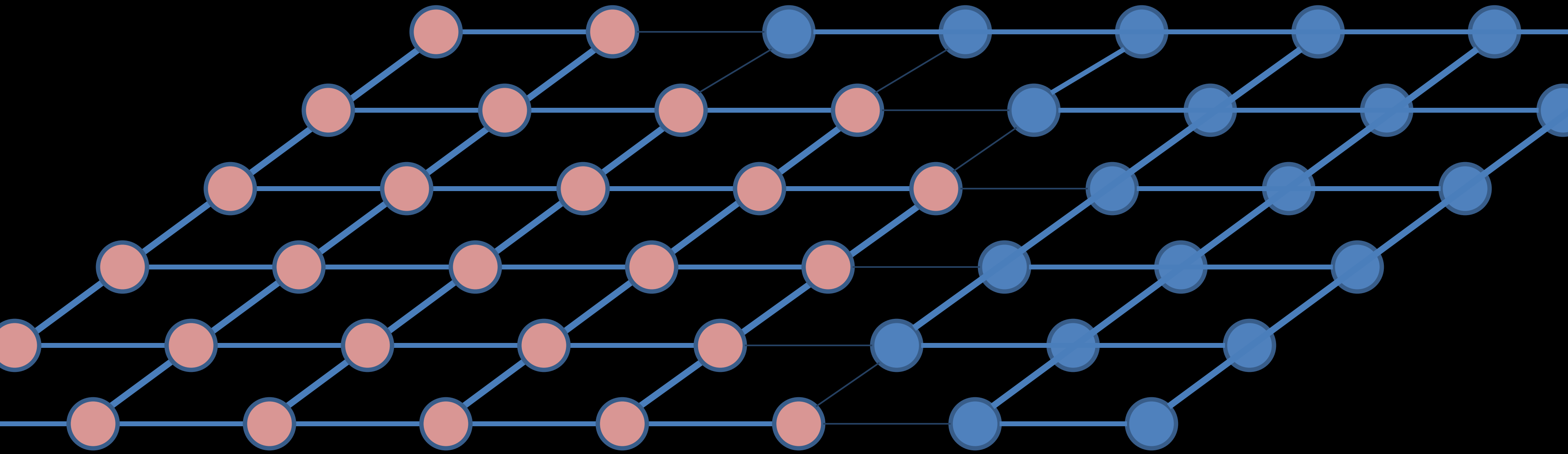
How does this relate to pixels?

- Make a graph of pixels. 4 or 8-way connected



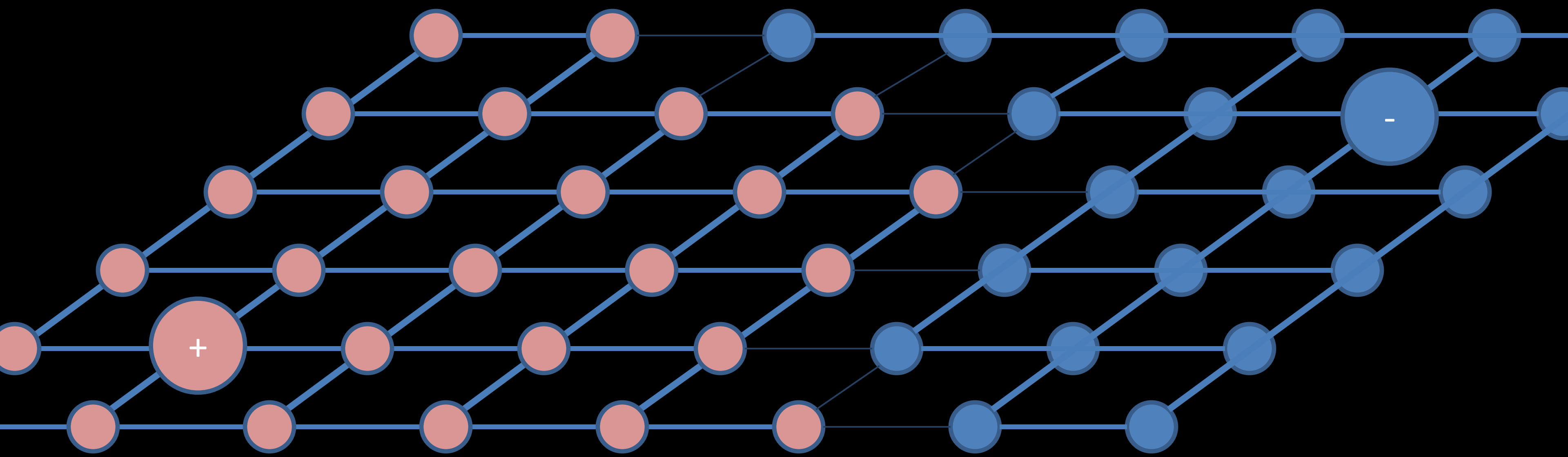
Foreground vs Background

- Edge Capacity = Similarity
  - So we want to cut between dissimilar pixels



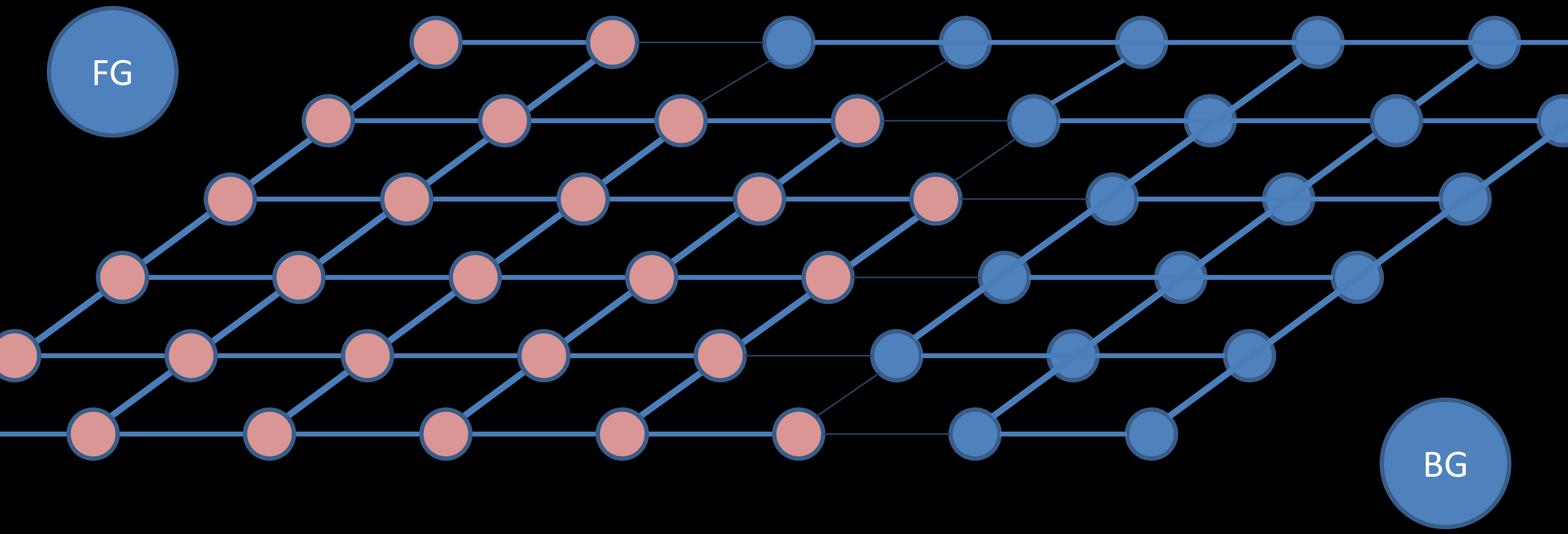
Source and Sink?

- Option A: Pick two pixels



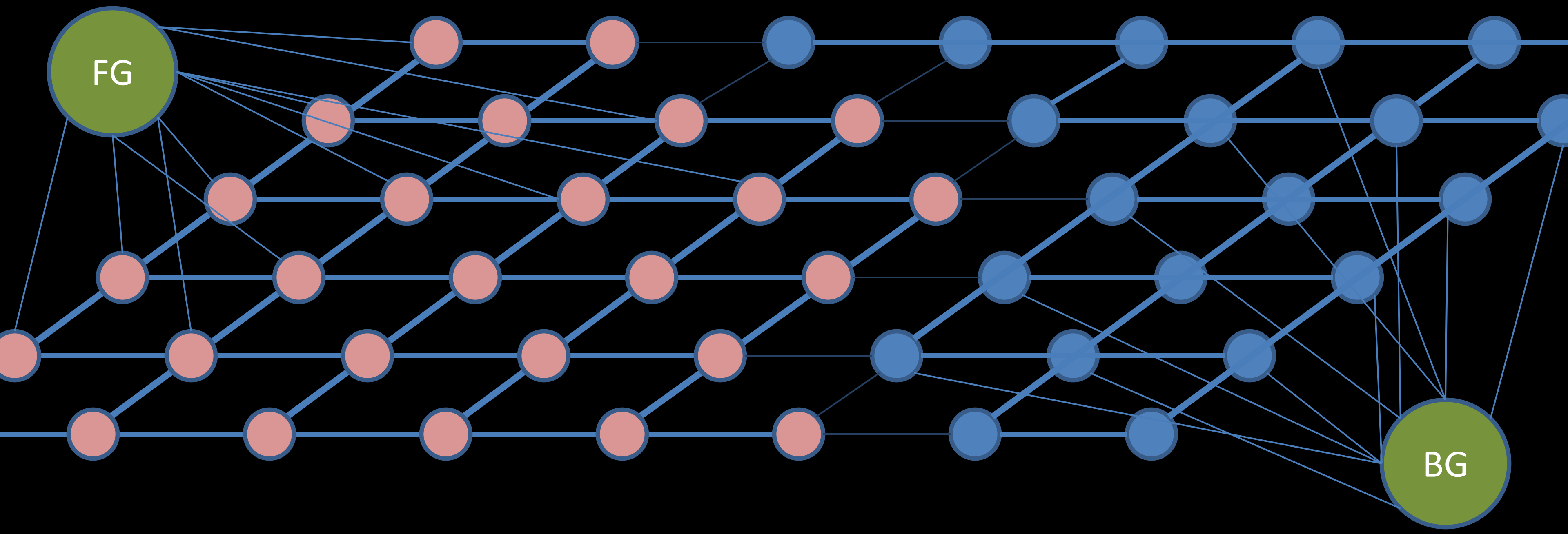
Source and Sink?

- Option B (better): Add extra nodes representing the foreground and background



Source and Sink?

- Connect them with strengths corresponding to likelihood that pixels below to FG or BG



Switch to 1D

- Edges between pixels

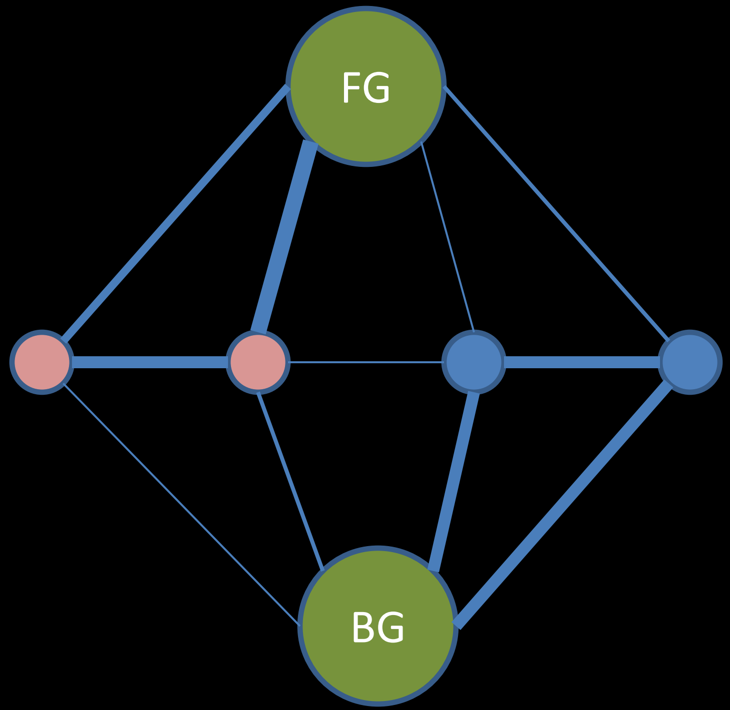
= similarity

- Edges from FG to pixels

= likelihood that they belong to FG

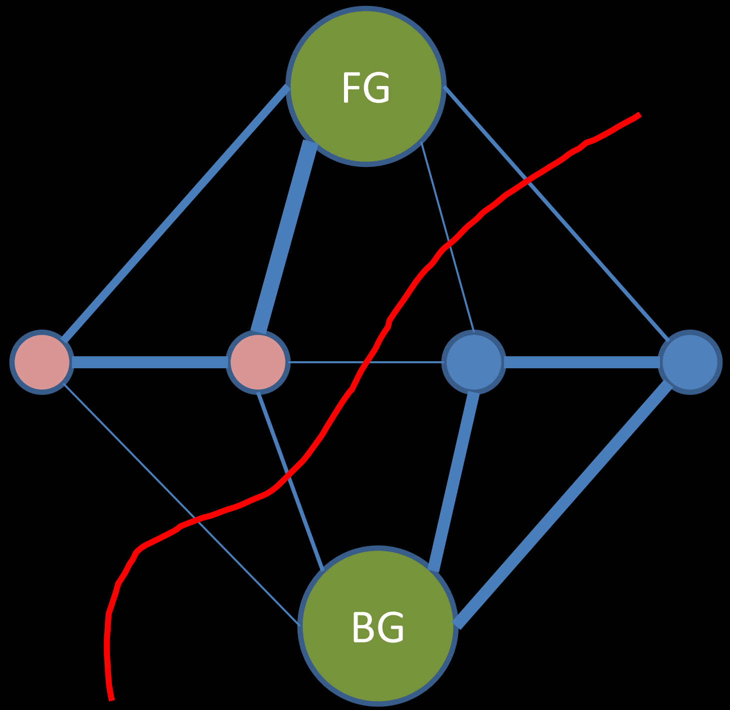
- Edges from BG to pixels

= likelihood that they belong to BG



Switch to 1D

- The min cut leaves each pixel either connected to the FG node or the BG node



Edge strengths between pixels

- Strength = likelihood that two pixels should be in the same category
- likelihood = -log(1-probability)
- probability = ?
  - Gaussian about color distance will do
  - $P_{xy} = \exp(-(I(x) - I(y))^2)$
  - When colors match, likelihood is infinity
  - When colors are very different, likelihood is small

Edge strengths to FG/BG

- If a pixel was stroked over using the tool
  - Strength to FG = large constant
  - Strength to BG = 0
- Otherwise
  - Strength to FG/BG = likelihood that this pixel belongs to the foreground/background
  - likelihood = -log(1-probability)
  - probability = ?

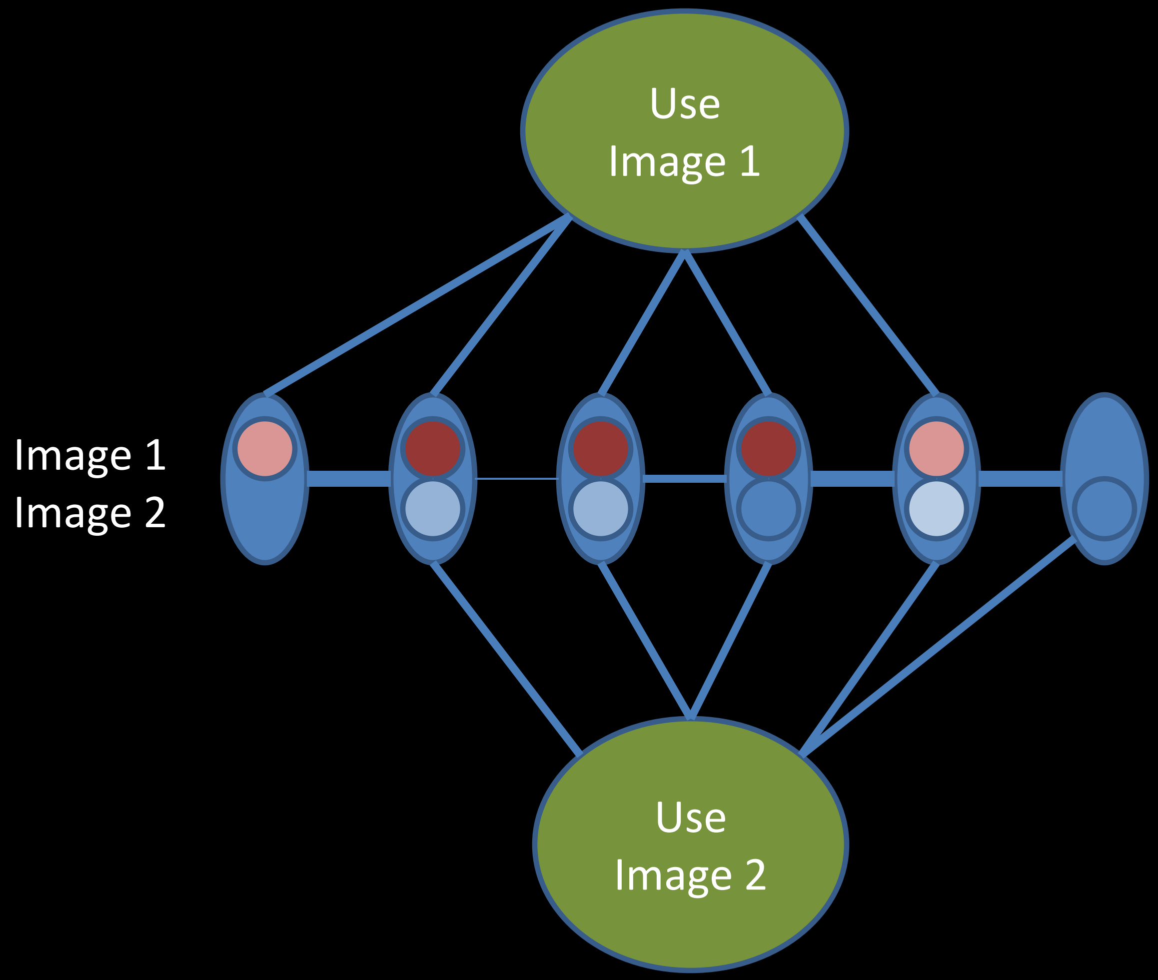
Probability of belonging to FG/BG

- Here's one method:
- Take all the pixels stroked over
  - Compute a histogram
  - FG Probability = height in this histogram
- Do the same for all pixels not stroked over
  - Or stroked over while holding alt
  - BG Probability = height in this histogram
- So if you stroked over red pixels, and a given new pixel is also red, FG probability is high.

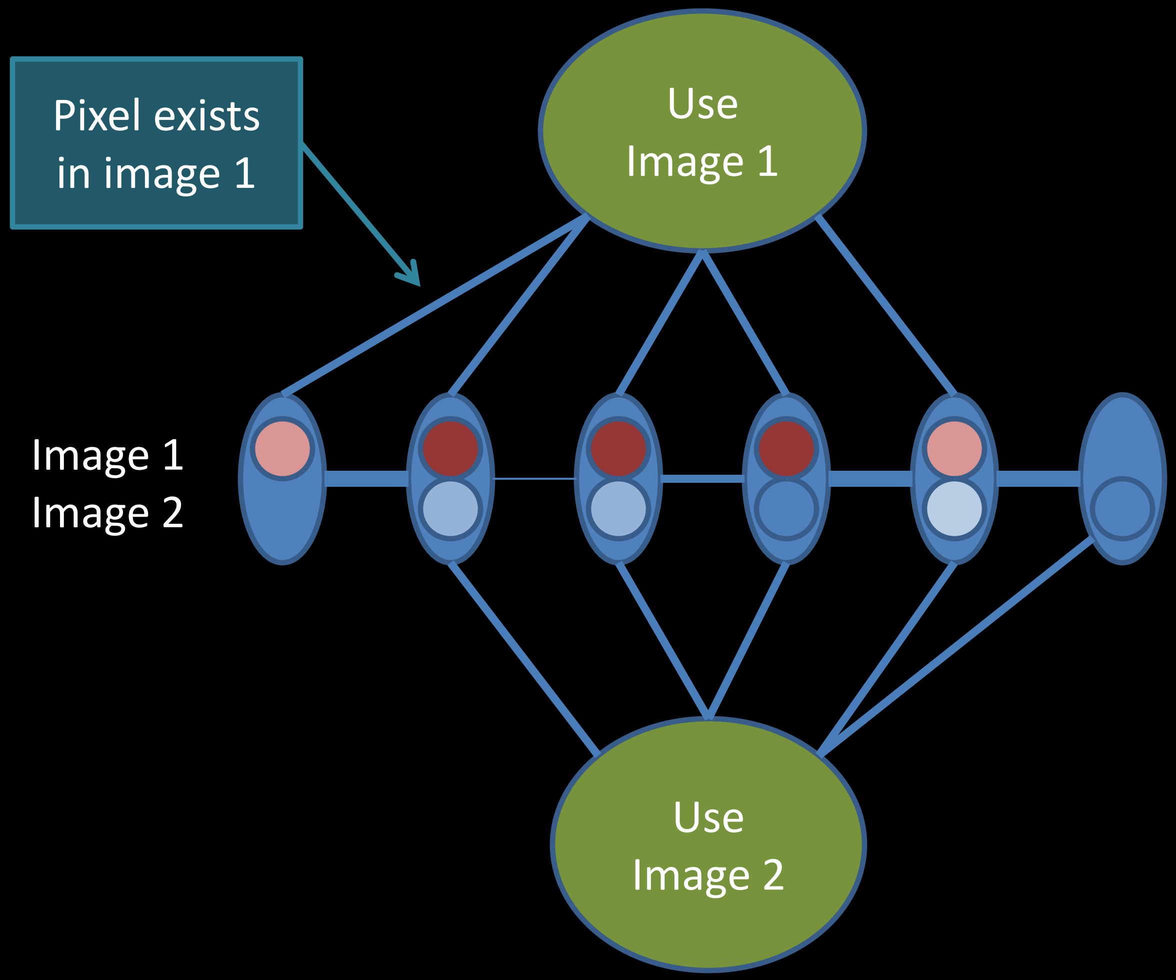
In terms of minimization:

- Graph cuts minimizes the sum of edge strengths cut
  - sum of cuts from FG/BG + sum of cuts between pixels
  - penalty considering each pixel in isolation + penalty for pixels not behaving like similar neighbours
  - data term + smoothness term
- Much like deconvolution

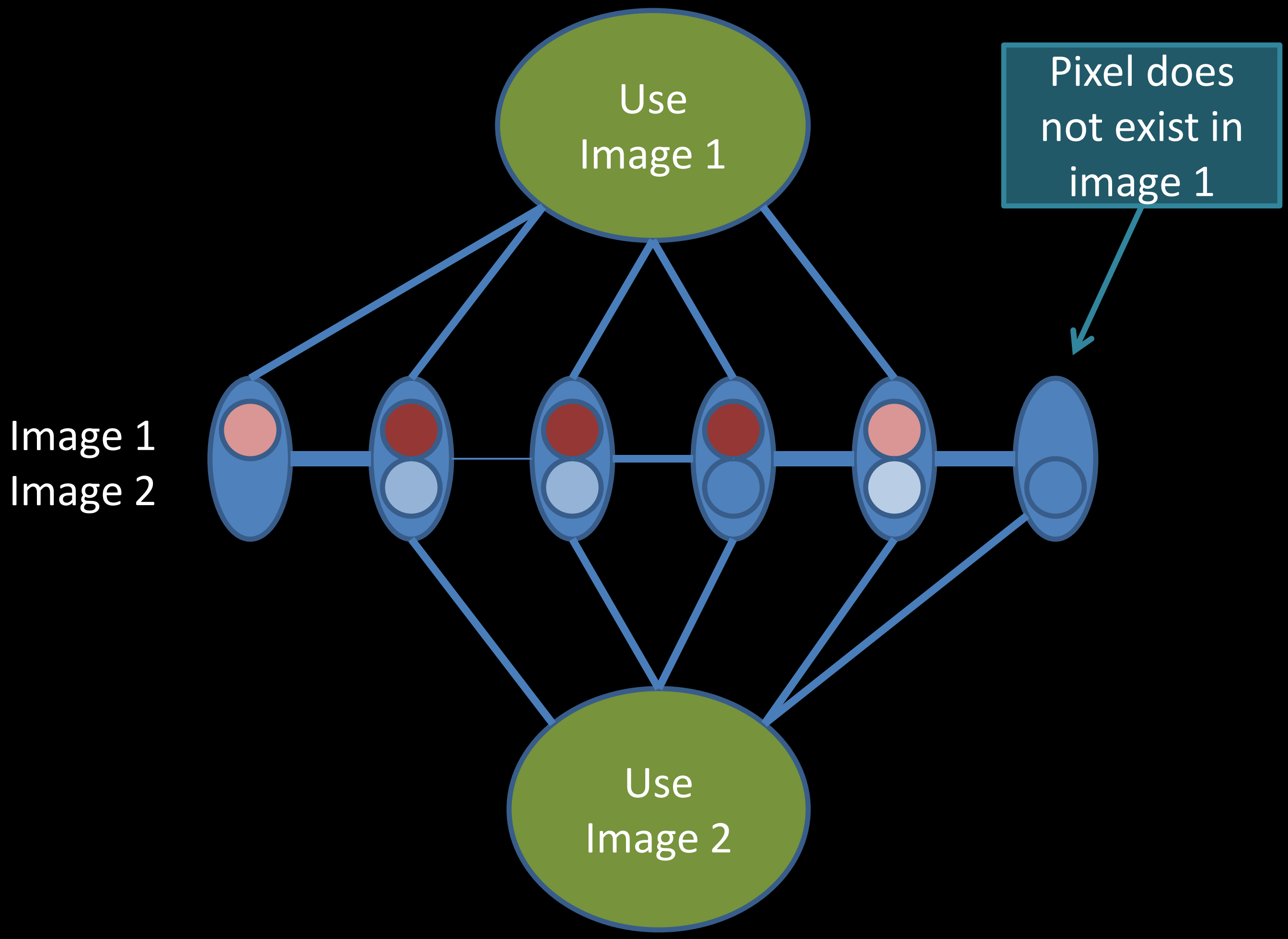
Picking seams for blending



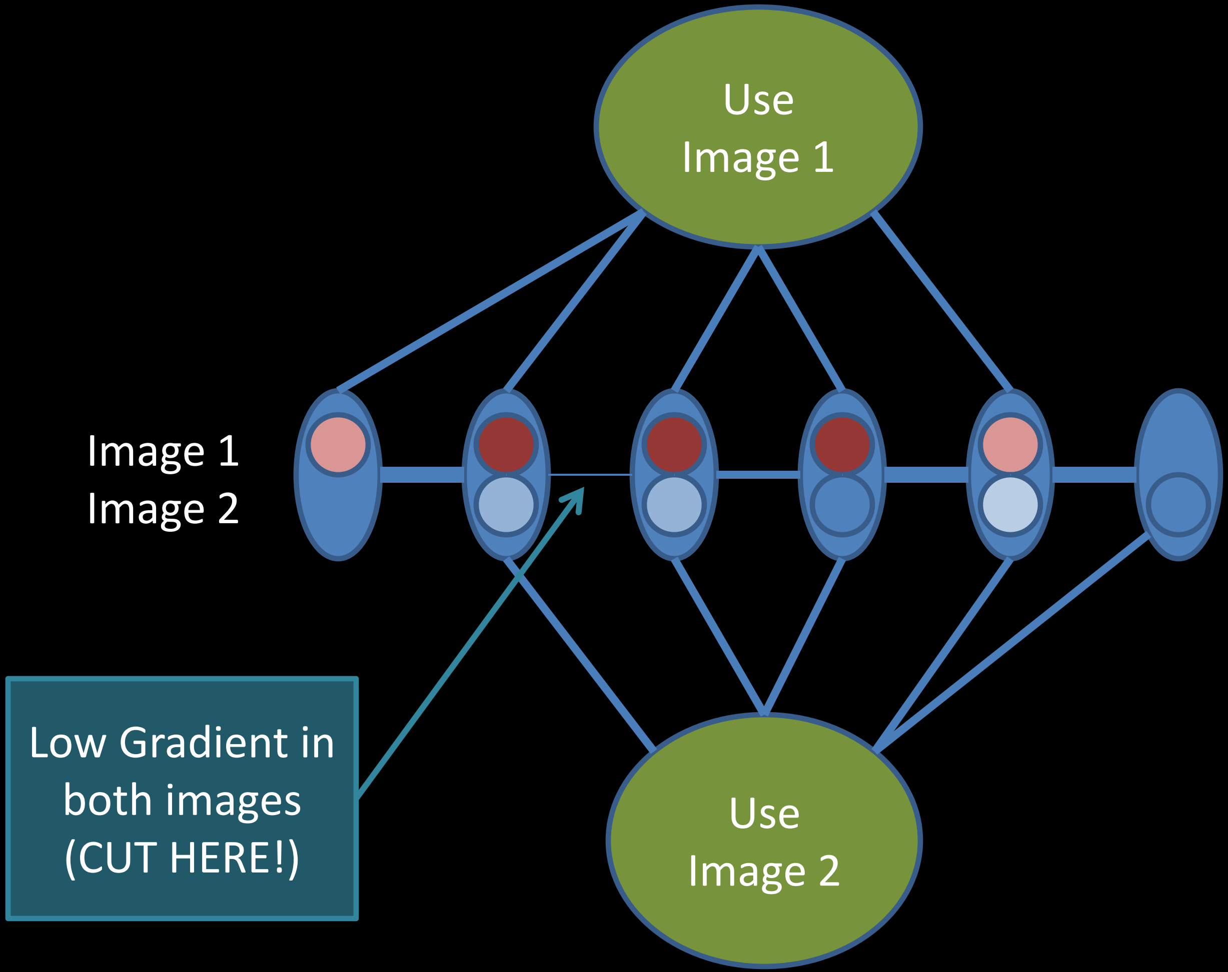
Picking seams for blending



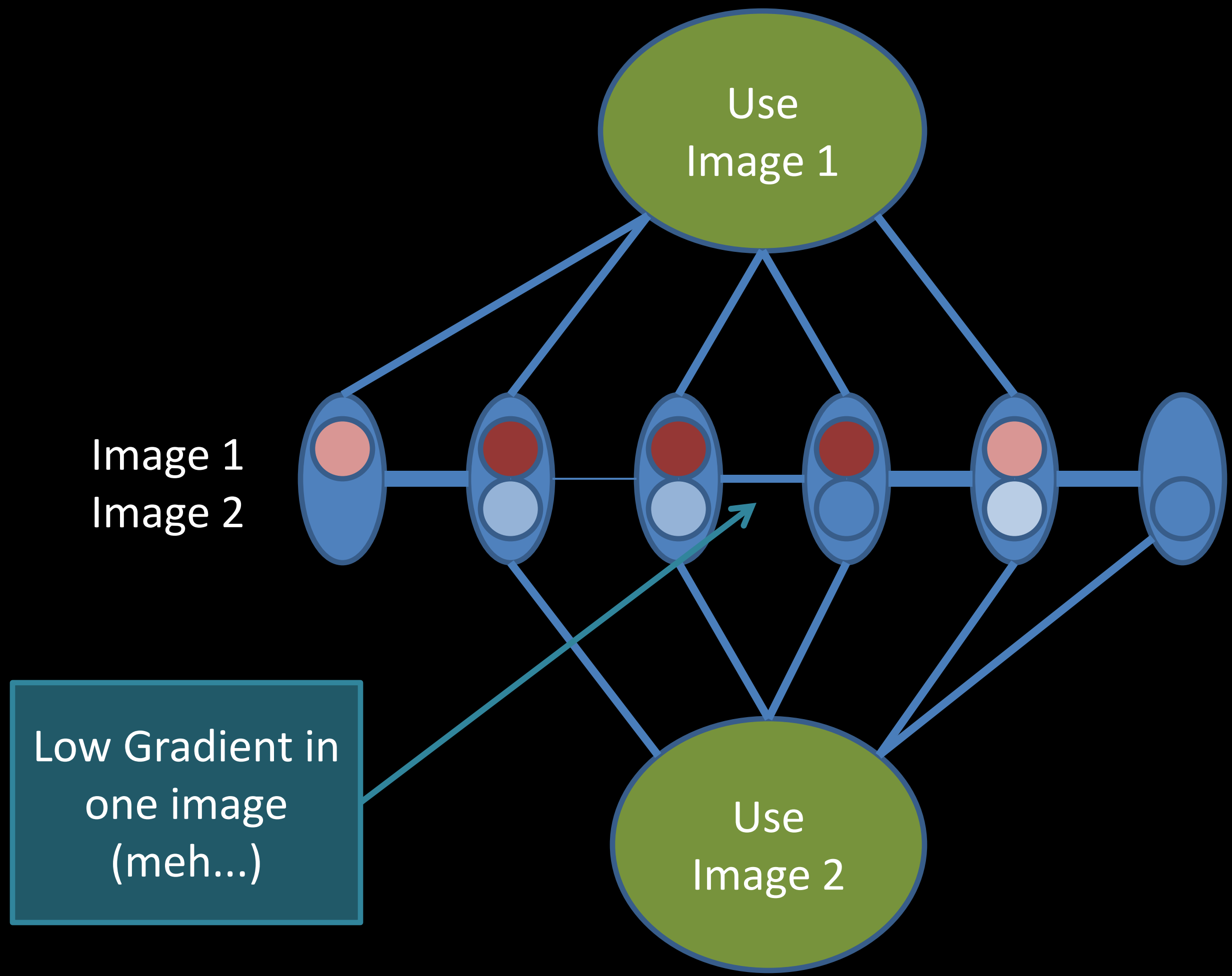
Picking seams for blending



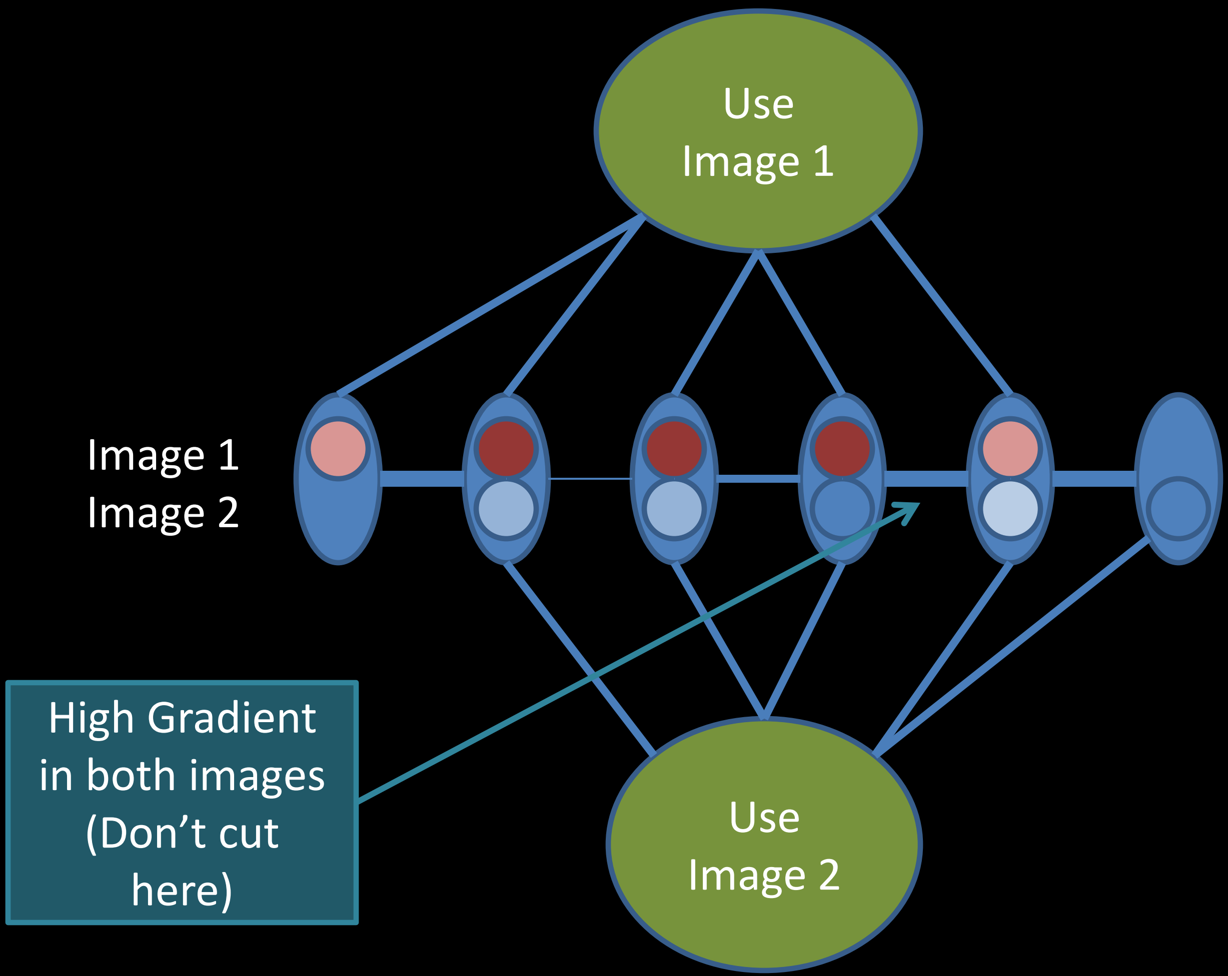
Picking seams for blending



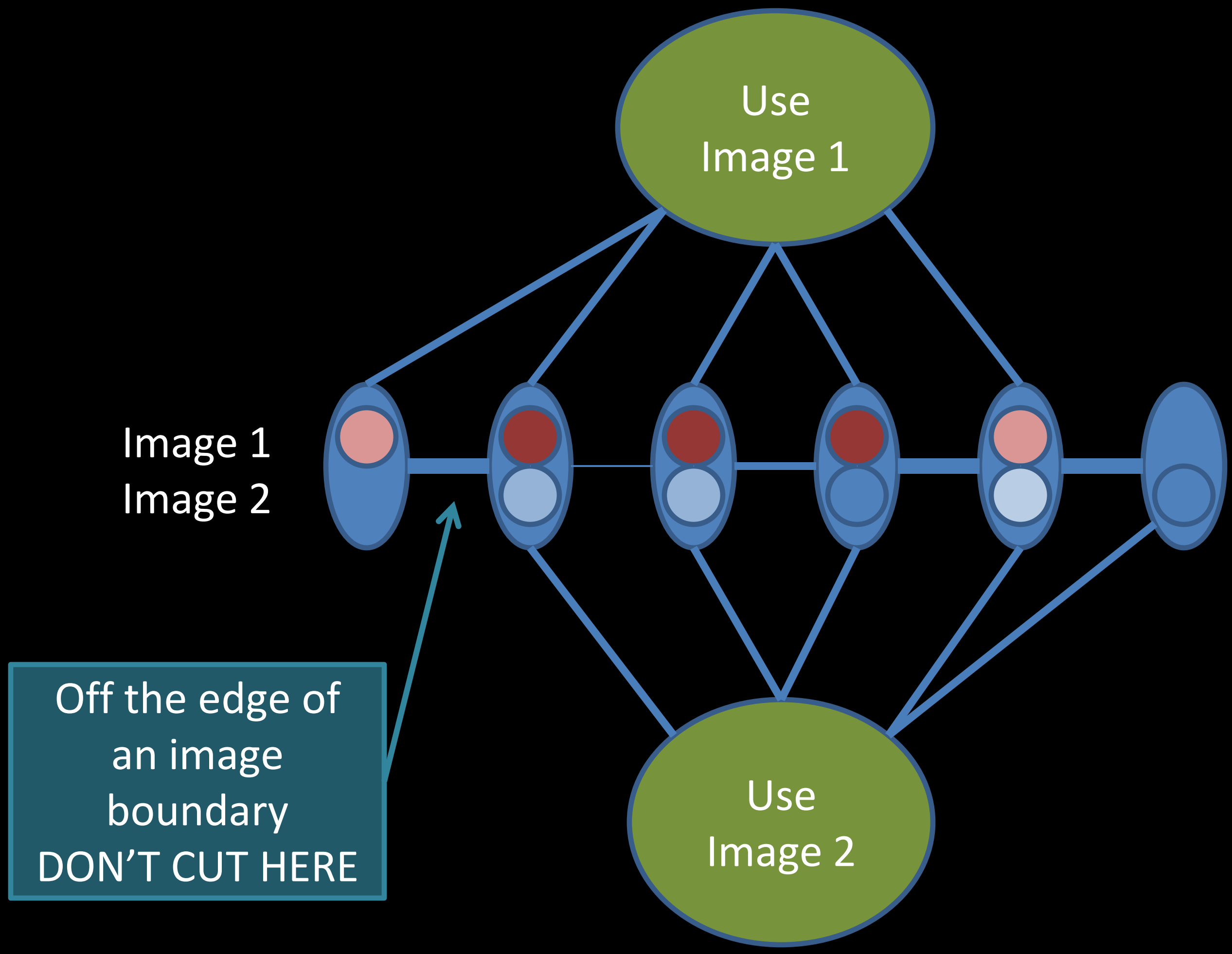
Picking seams for blending



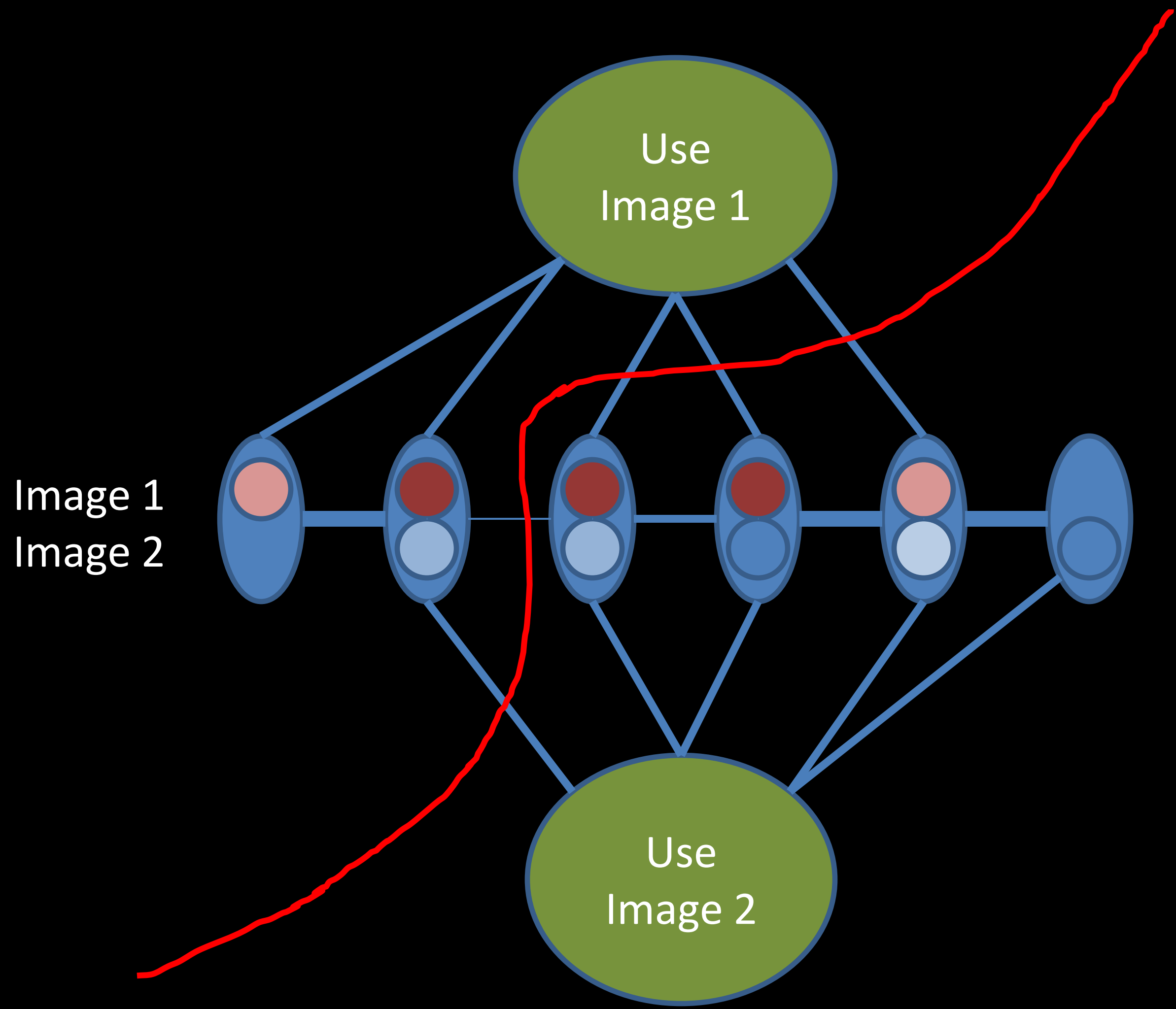
Picking seams for blending



Picking seams for blending



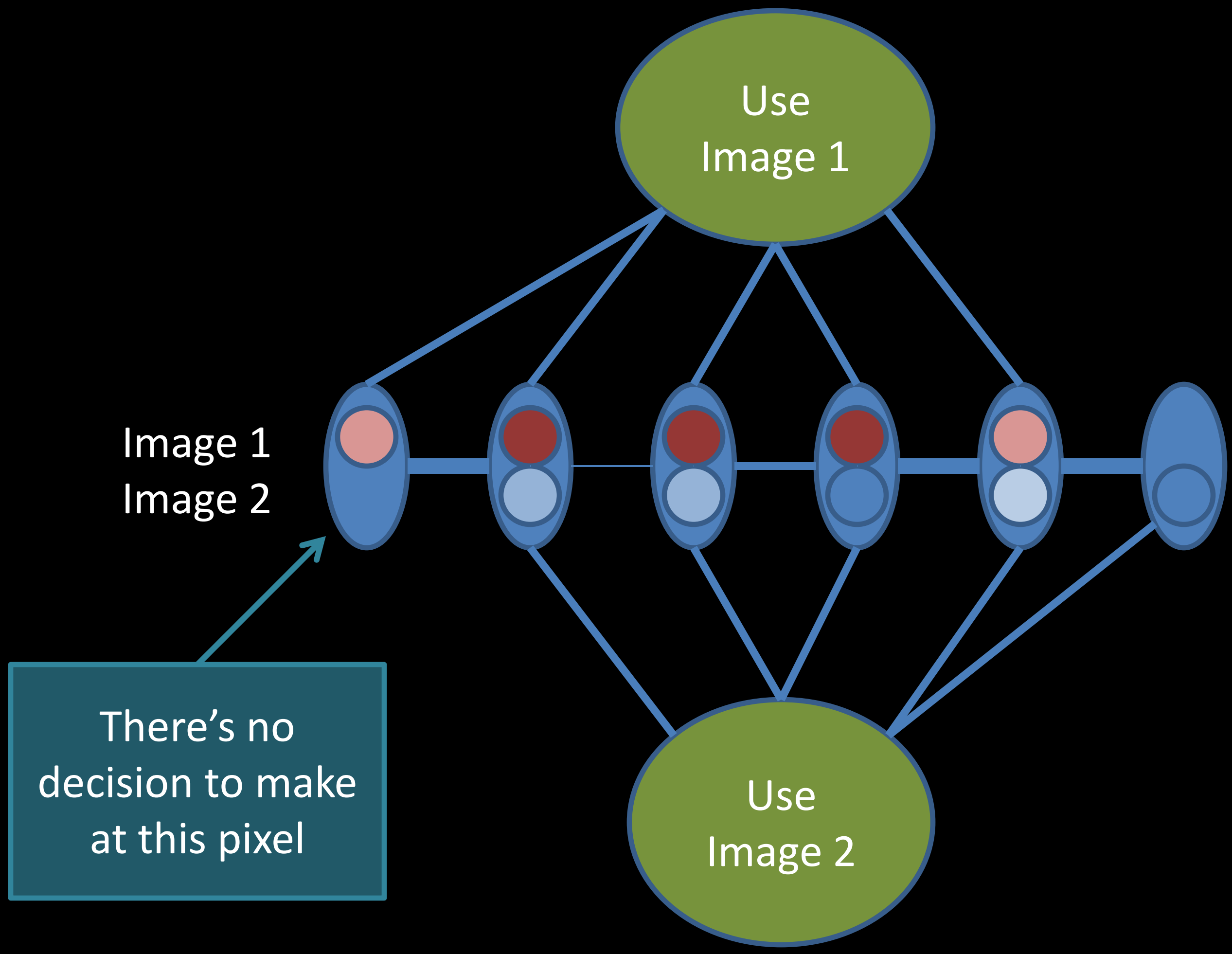
Picking seams for blending



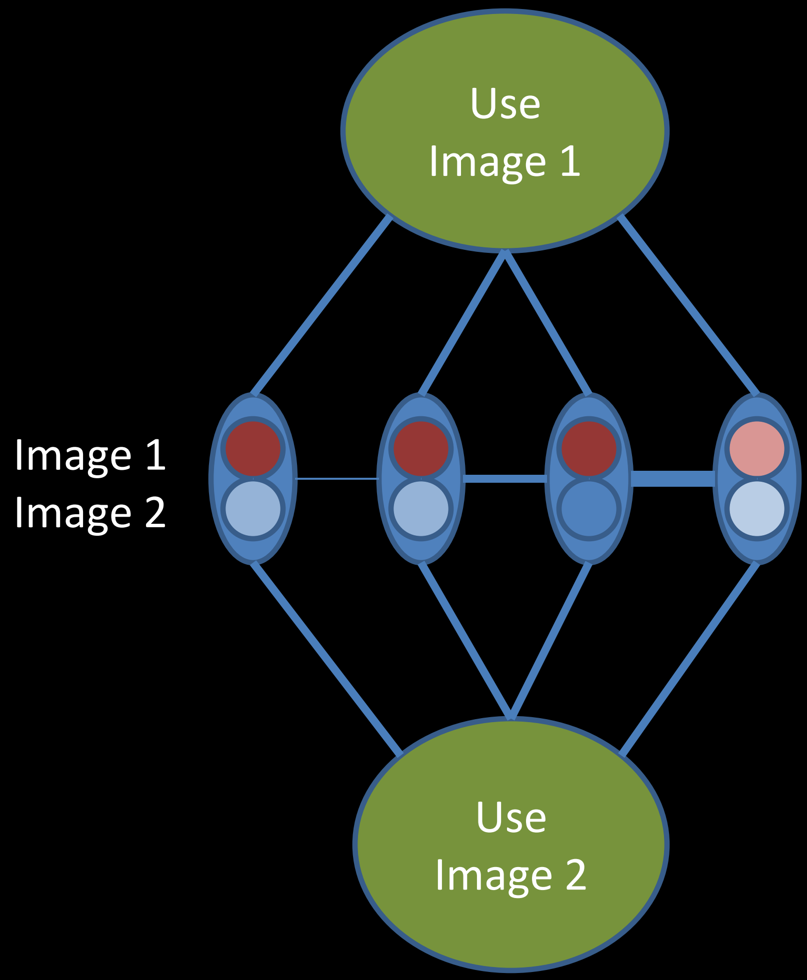
Speeding up Graph Cuts

- Use a fancy max-flow algorithm
  - e.g. tree reuse
- Use a smaller graph

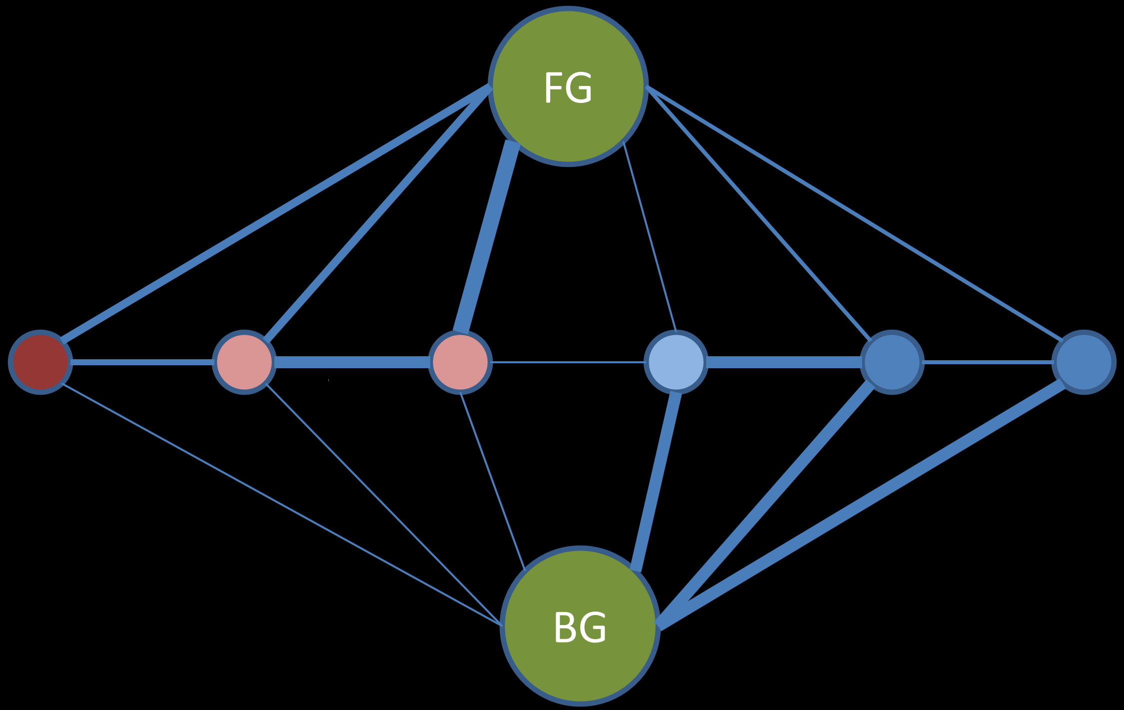
Speeding up Graph Cuts



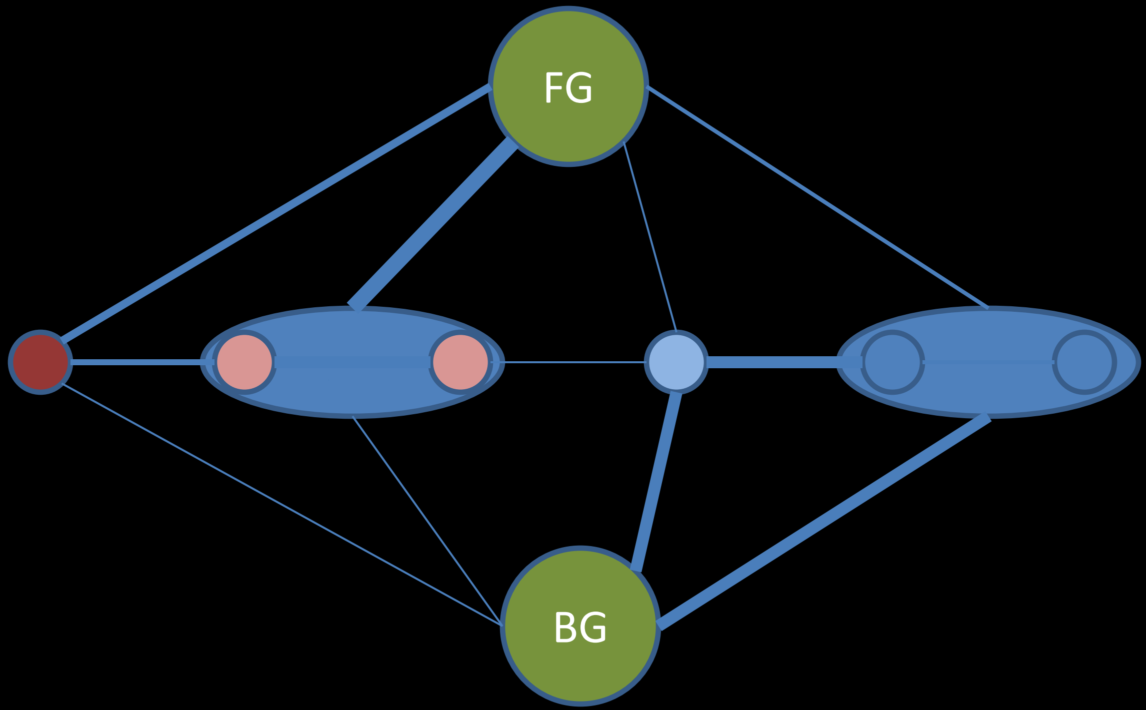
Only include the relevant pixels



Consider selection again



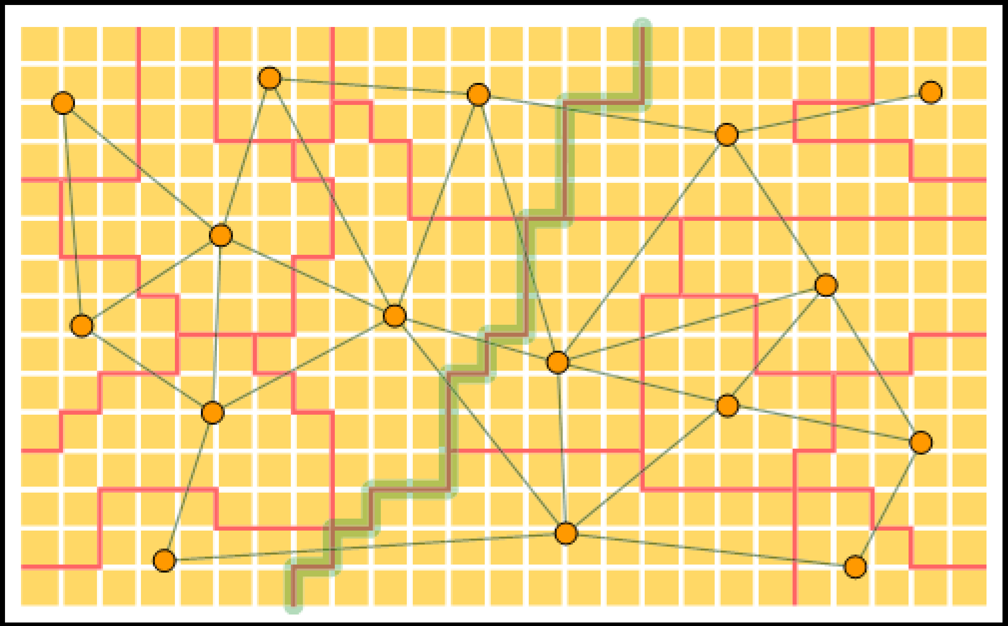
Clump pixels of near-constant color



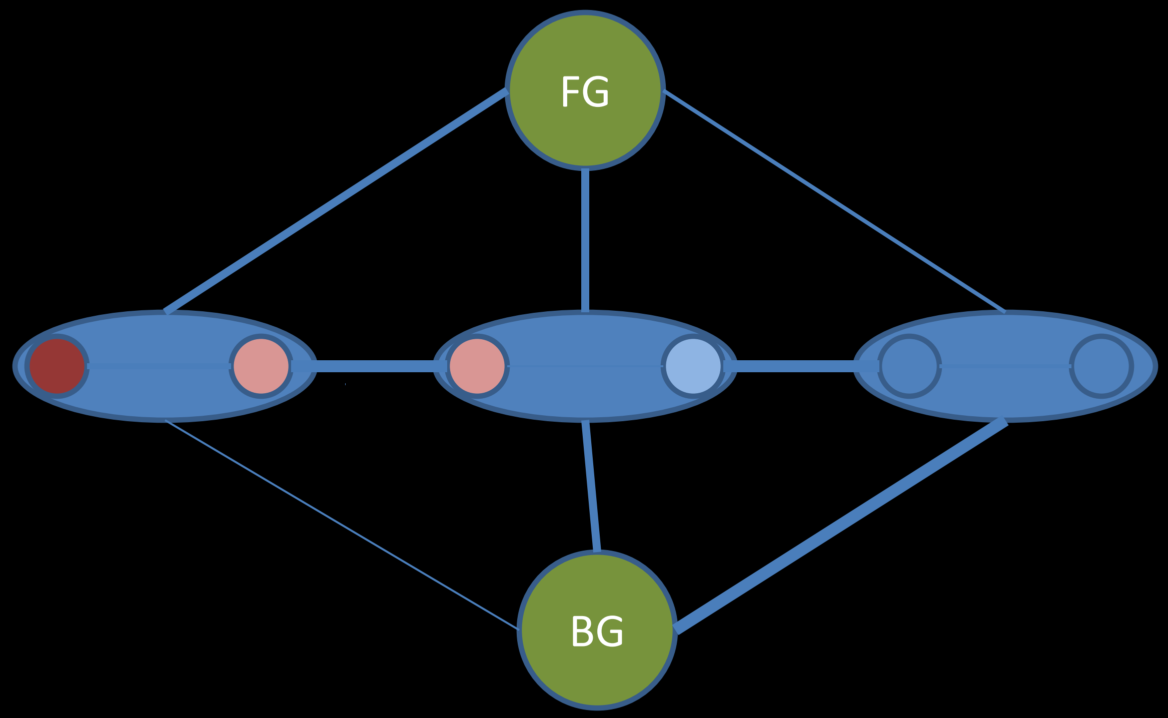
Clump pixels of near-constant color

Lazy Snapping does this

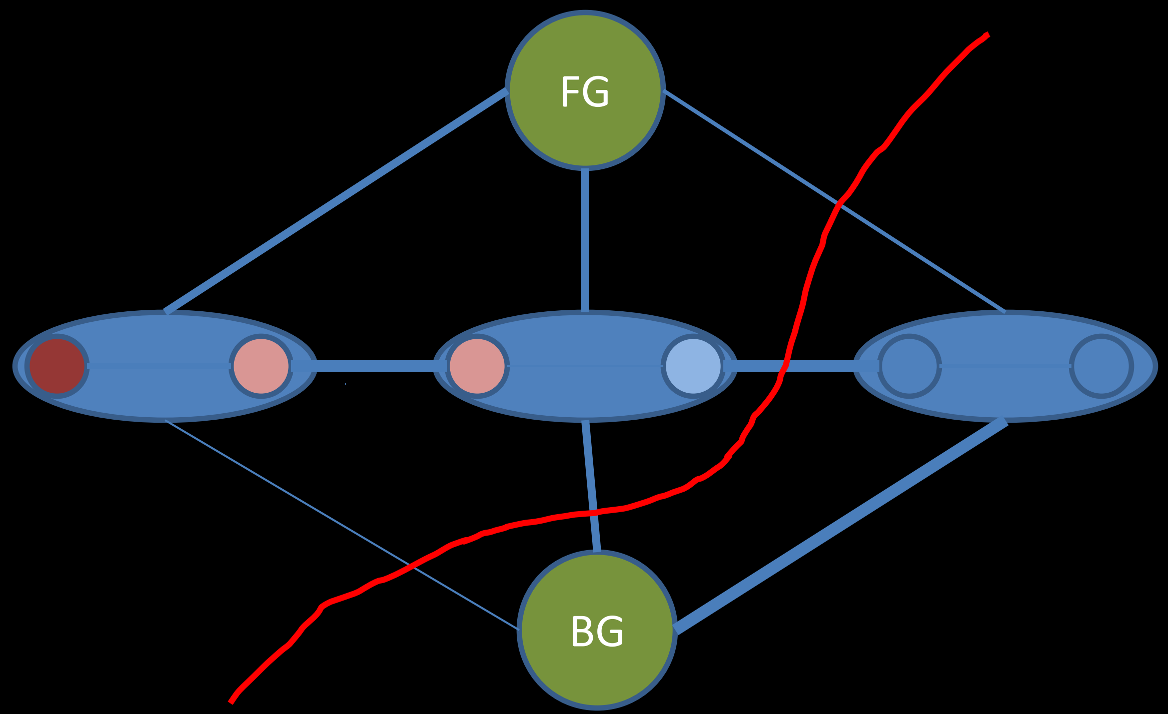
(Li et al. SIGGRAPH 04)



Coarse to Fine
1) Solve at low res.



Coarse to Fine
1) Solve at low res.

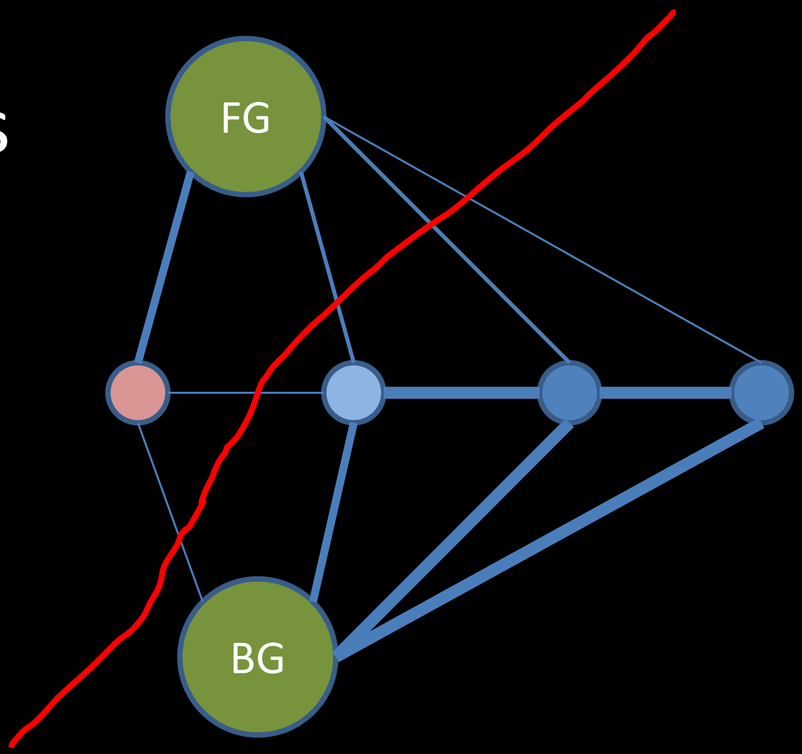


Coarse to Fine
2) Refine the boundary

Paint Selection does this

Liu et al. SIGGRAPH 2009

(and uses joint bilateral upsampling to determine the boundary width)



Videos

- GrabCut (SIGGRAPH 04)
  - http://research.microsoft.com/en-us/um/cambridge/projects/visionimagevideoediting/segmentation/images/Video.avi
- Paint Selection (SIGGRAPH 09)
  - http://research.microsoft.com/en-us/um/people/jiansun/videos/PaintSelection.wmv