Fundamental Network Algorithms

Snehal M. Shekatkar

General remarks

Calculate run times even for simple quantities!

- More than one method to calculate the same thing
- Allows estimation of the actual amount of time needed on a computer

Degree

- Degrees are stored along with the adjacency list
- $O(1)$ to look up a degree value
- For adjacency matrix, degree calculation takes $O(n)$, so anyway keeping the degree array makes sense

Local clustering coefficient

$$C_i = \frac{\text{Number of edges between the neighbours of } i}{\text{Total possible edges between the neighbours}}$$

- Go through every pair only once ($u < v$).
- Denominator can be calculated in $O(1)$
- For adjacency matrix, this takes $O(n \times m^2/n^2) = O(m^2/n)$
- For the adjacency list, the calculation takes $O(m/n \times m^2/n^2 \times m/n) = O(m^4/n^4)$ time
- Only $O(m/n)$ in the hybrid representation
- Average clustering coefficient takes $O(m^2)$, $O(m^4/n^3)$ and $O(m)$ in the three cases

Global clustering coefficient

$$C = \frac{\text{Number of triangles} \times 3}{\text{Number of triples}}$$

- Calculate the number of triangles attached to each vertex (Factor of 3 is taken care of)
- This takes $O(m/n)$ in hybrid representation
- Denominator is $\frac{1}{2}\sum_i k_i(k_i - 1) = \frac{1}{2}n(\langle k^2 \rangle - \langle k \rangle^2)$
- Takes large time for scale-free networks!

Assortativity coefficient

- $r = \frac{\sum_{ij}(A_{ij} - k_i k_j/2m)k_i k_j}{\sum_{ij}(k_i \delta_{ij} - k_i k_j/2m)k_i k_j}$
- $S_e = \sum_{ij} A_{ij} k_i k_j = 2 \sum_{\text{edges}(i,j)} k_i k_j$
- $S_1 = \sum_i k_i \quad , \quad S_2 = \sum_i k_i^2 \quad , \quad S_3 = \sum_i k_i^3$
- $r = \frac{S_1 S_e - S_2^2}{S_1 S_3 - S_2^2}$
- Much better run time, because $m << n^2$

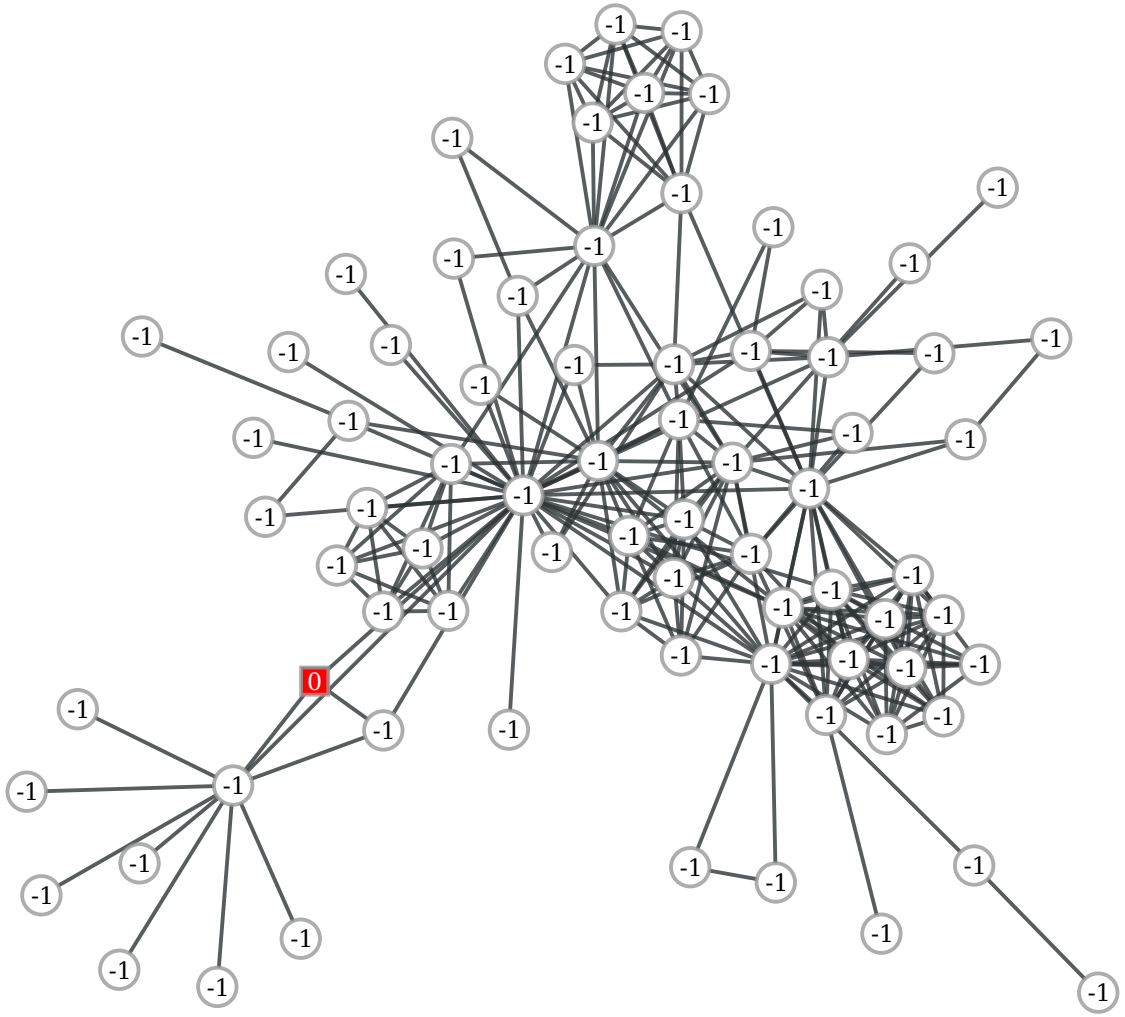
Shortest paths

- Distance between i and j is the length of a shortest path between them
- Algorithm of choice: ***Breadth-first-search***
- One run finds the shortest distance from a single vertex i to all other vertices
- Also lets us find all the shortest paths between given two vertices
- Works for both undirected and directed networks
- Also used to find the components

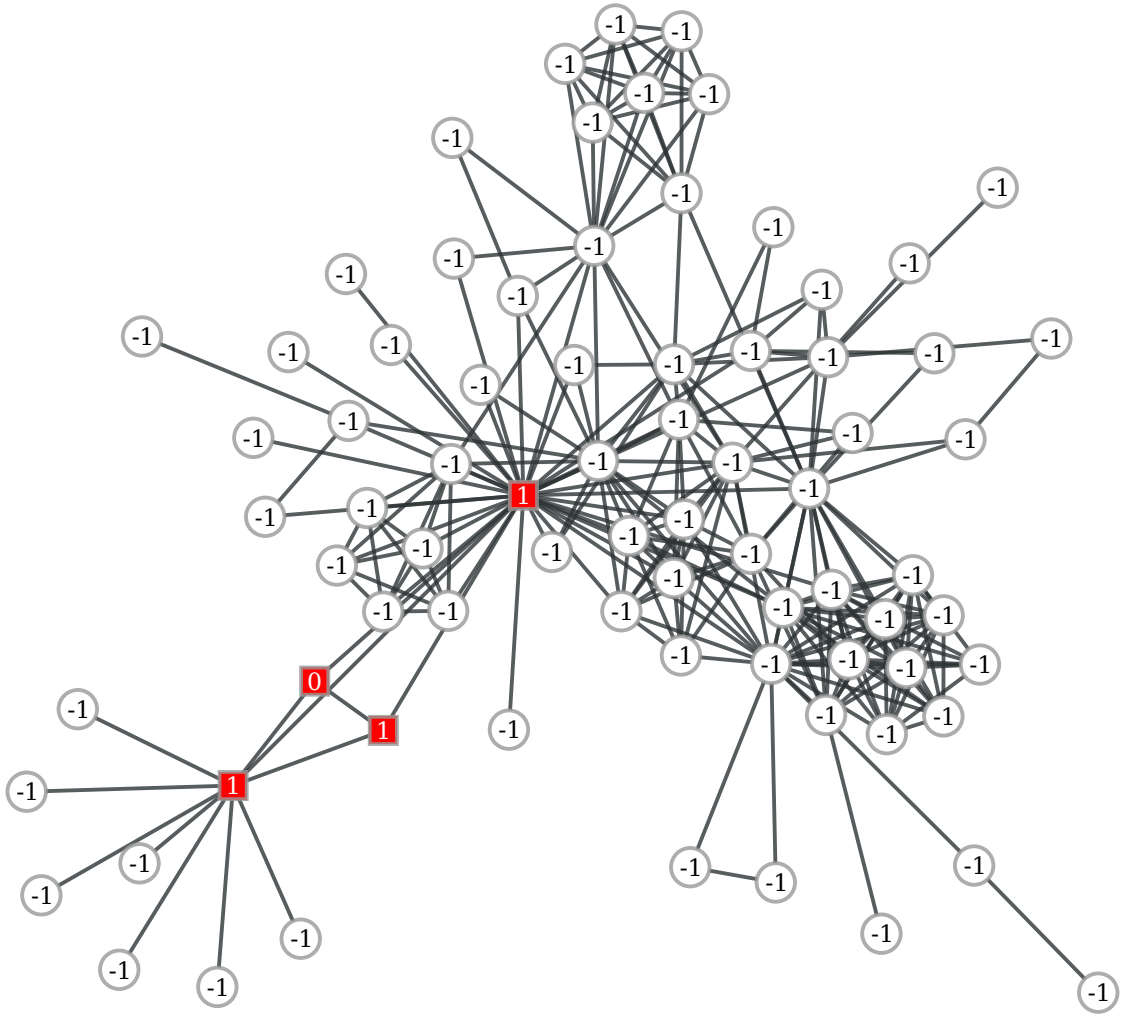
Breadth-first search

- Assign distance 0 to the starting node
- For each $d \geq 0$, locate vertices with distance d
- Find neighbours of these vertices
- If neighbours have no assigned distance, assign them $d + 1$
- Stop when there are no vertices with the next value of d

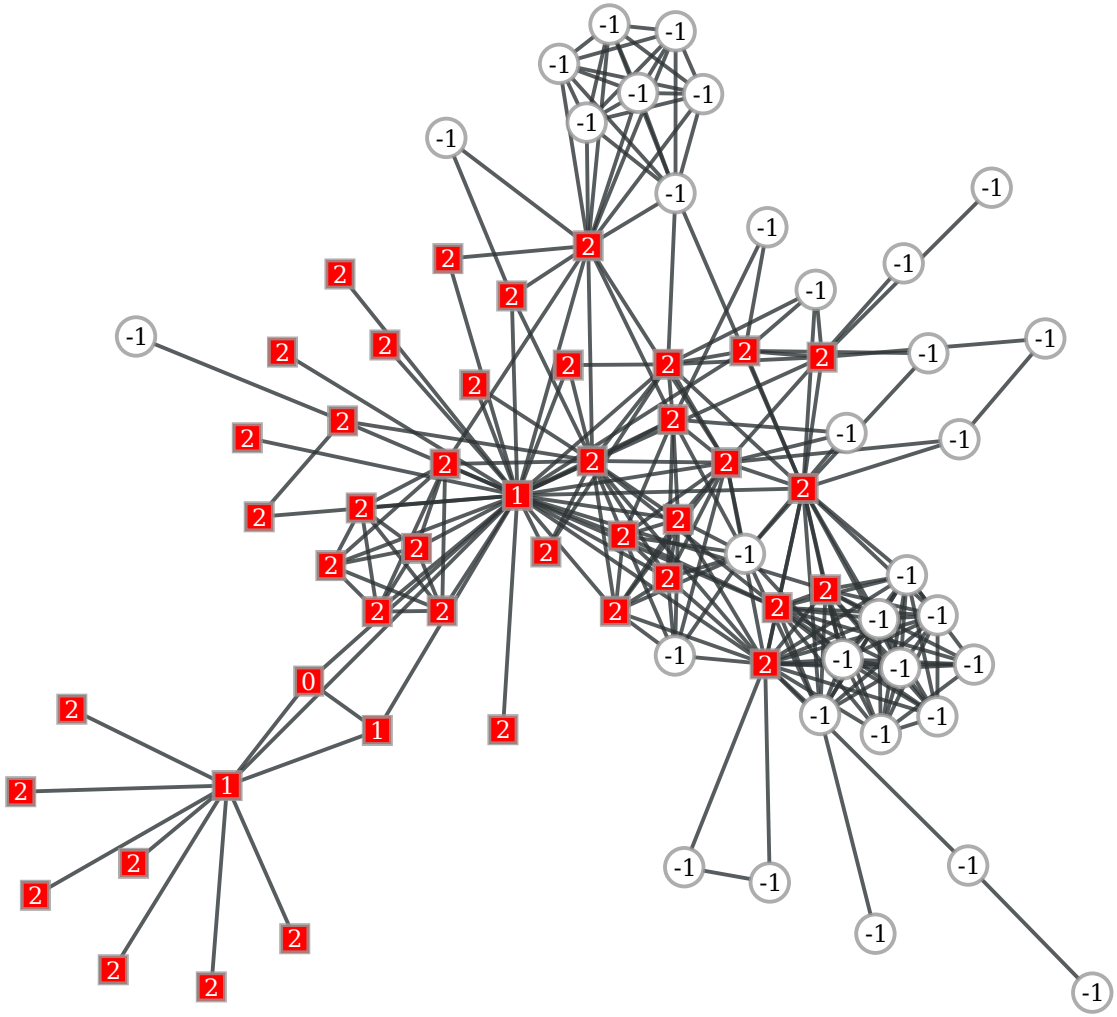
$d = 0$



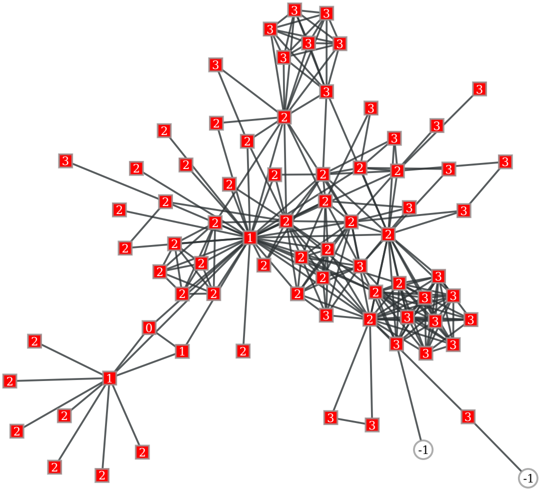
$d = 1$



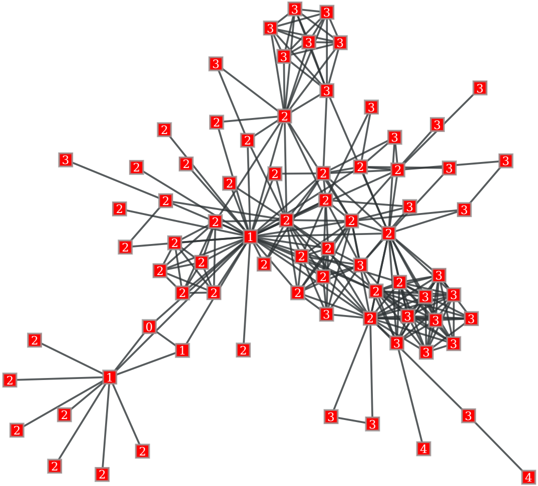
$d = 2$



$d = 3$



$d = 4$



Naive-implementation of the algorithm

- Create an array of size n to store the distances. Unknown distances are set to -1
- Create a distance variable d to keep track of the maximum distance so far, and set it to 0
- Repeat:
  1. Find vertices at a distance d by going through the distance array
  2. Find neighbours of these vertices, and check if their distances are unknown. If yes, assign distance $d+1$ to them
  3. Increase d by 1

Running time of the naive implementation

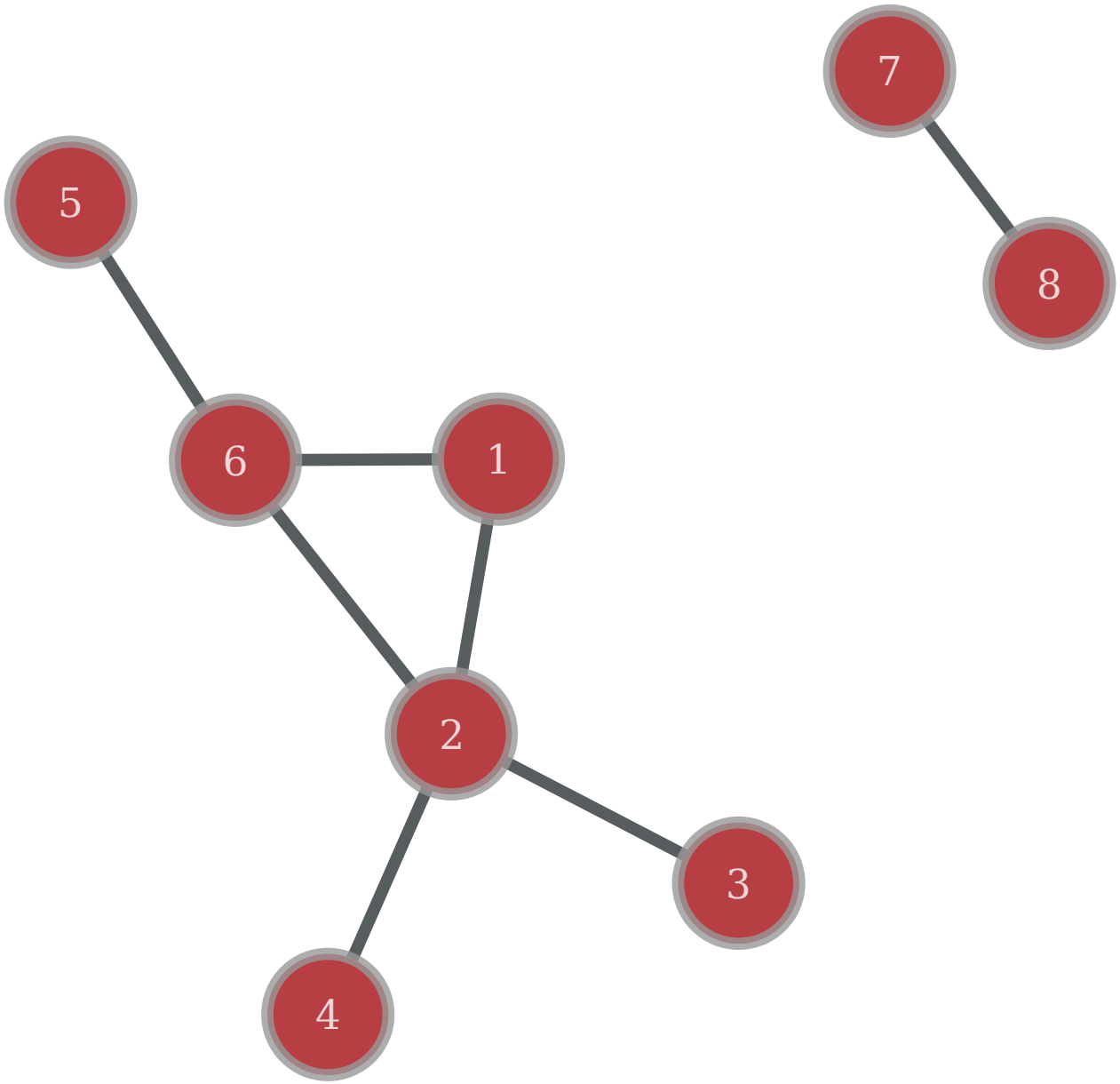
- Finding vertices with distance d takes $O(n)$
- If total r runs happen, then this takes $O(rn)$
- If the network is stored in the form of an adjacency list, finding neighbour for each vertex takes $O(m/n)$
- This is to be done for each vertex exactly once
  $\implies O(n \times m/n) = O(m)$
- Total time: $O(rn + m)$
- In the worst case, $r = n$, so the total time is $O(n^2 + m)$

A better implementation

- Most time is spent to find the vertices at a distance d at each stage
- But this set of vertices is precisely the set found in the previous step!
- This set can be kept ready using an array and two pointers
- Write pointer indicates the next empty location in the array
- Read pointer indicates the next item to be read

A better implementation

- Assign the first element of the queue array the label of the starting vertex i, set the read pointer to it, and set the write pointer to the second element. In the distance array, record the distance of i as 0
- If the read and write pointers point to the same element, the algorithm is over. If not, read the element using read pointer, and increase the pointer by 1
- Find the distance d for that vertex from the distance array
- Go through the neighbours of this vertex, and look up their distances. Those with unassigned distances get value $d + 1$, and their labels are stored using the write pointer
- Repeat


7
5
8
6
1
2
3
4

Running time

- Setting up arrays takes $O(n)$ in total
- For each vertex, locating neighbors takes $O(m/n)$
- In the worst case, this has to be done for all n vertices, hence this takes in total $O(n \times m/n) = O(m)$ time
- Thus, the total running time for a better implementation of BFS is $O(m + n)$. This is much better than $O(m + n^2)$ in the naive implementation.

Finding the distance between all the pairs

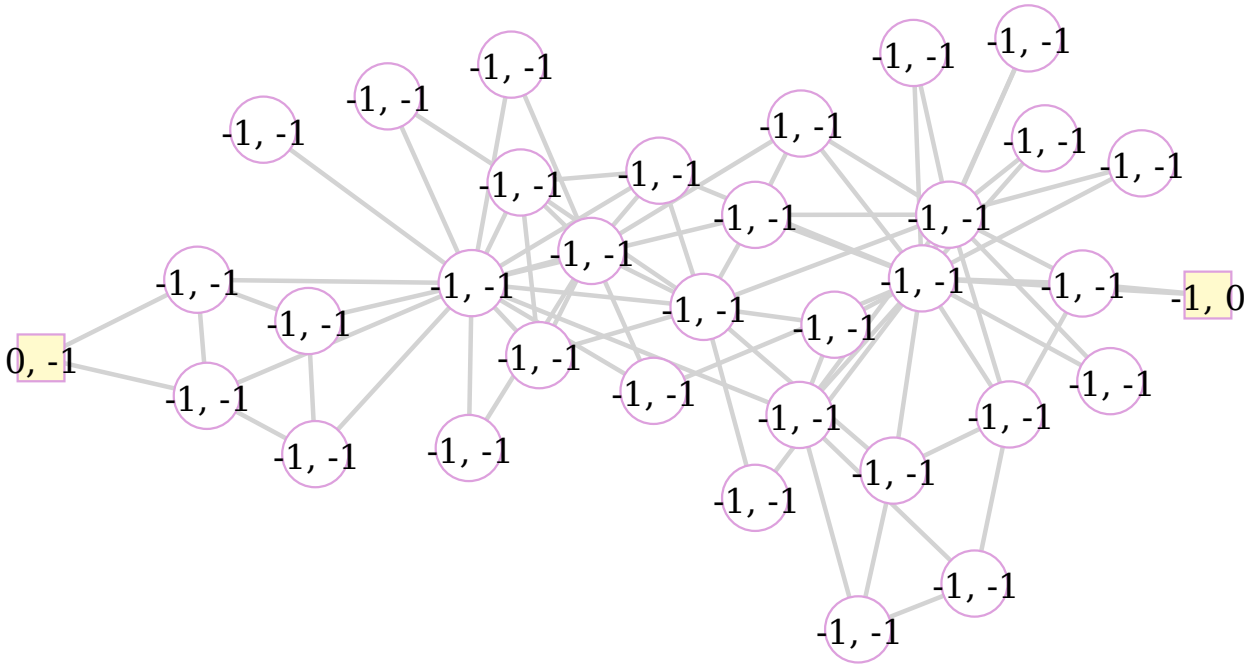
- To find the distance between every pair of vertices, we use every vertex as a source in turn.
- This can be done in $O(n \times (m + n))$, or in $O(n^2)$ on sparse graphs.
- This is optimal since n^2 numbers cannot be calculated in less than n^2 time.

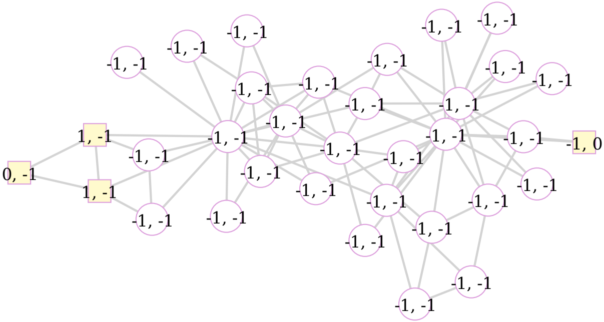
Finding the distance between given two vertices

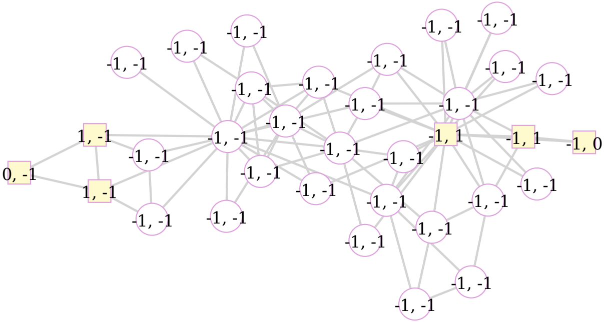
- Often the distance between only two given vertices s and t is needed.
- Calculate distances from s to all other vertices, and then read the distance to t
- Even better, we could run the breadth-first search only till the vertex t is found

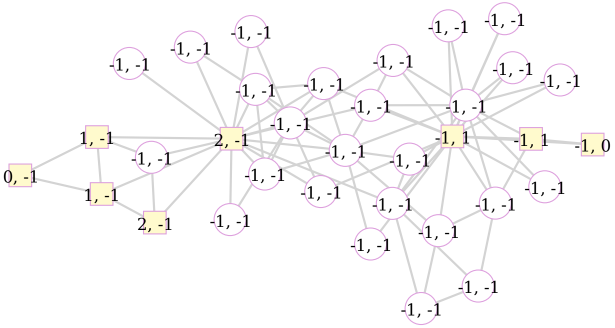
Two-source BFS

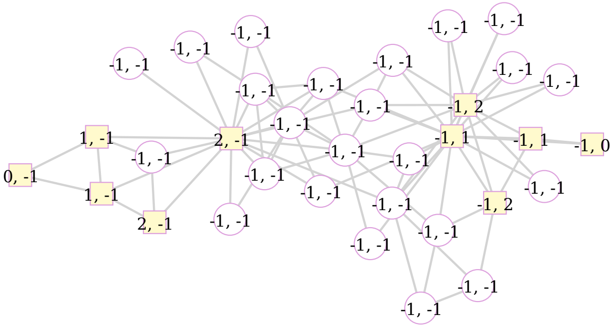
- The best strategy however, is to run two breadth-first searches starting from the two vertices!
- the point at which one BFS assigns distance to the vertex which has already been assigned a distance by the other BFS, we stop both BFSs
- the sum of the distances assigned by the two BFSs to this common vertex is the shortest distance between the two vertices

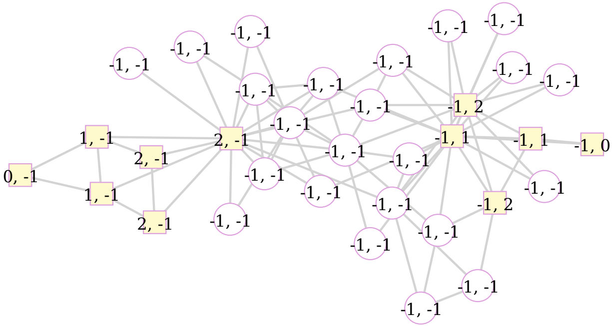

0, -1
-1, 0
-1, -1

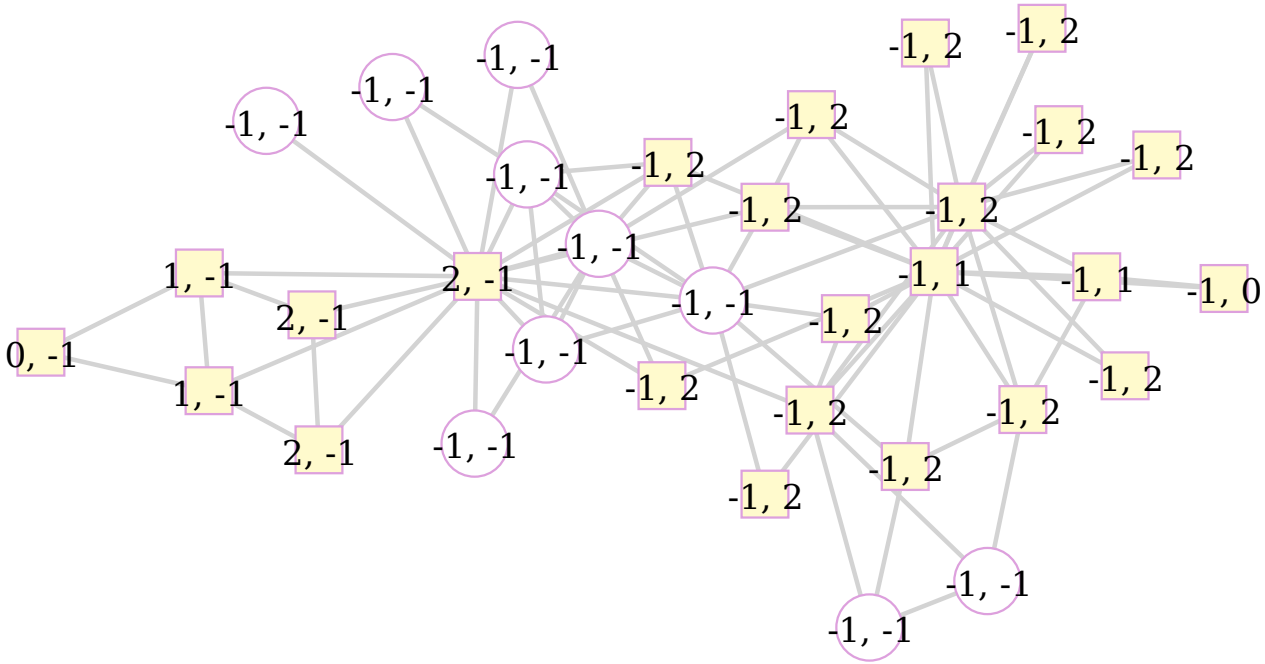

-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
1, -1
-1, -1
-1, -1
-1, -1
-1, 0
-1, -1
-1, -1
-1, -1
0, -1
-1, -1
-1, -1
-1, -1
1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1

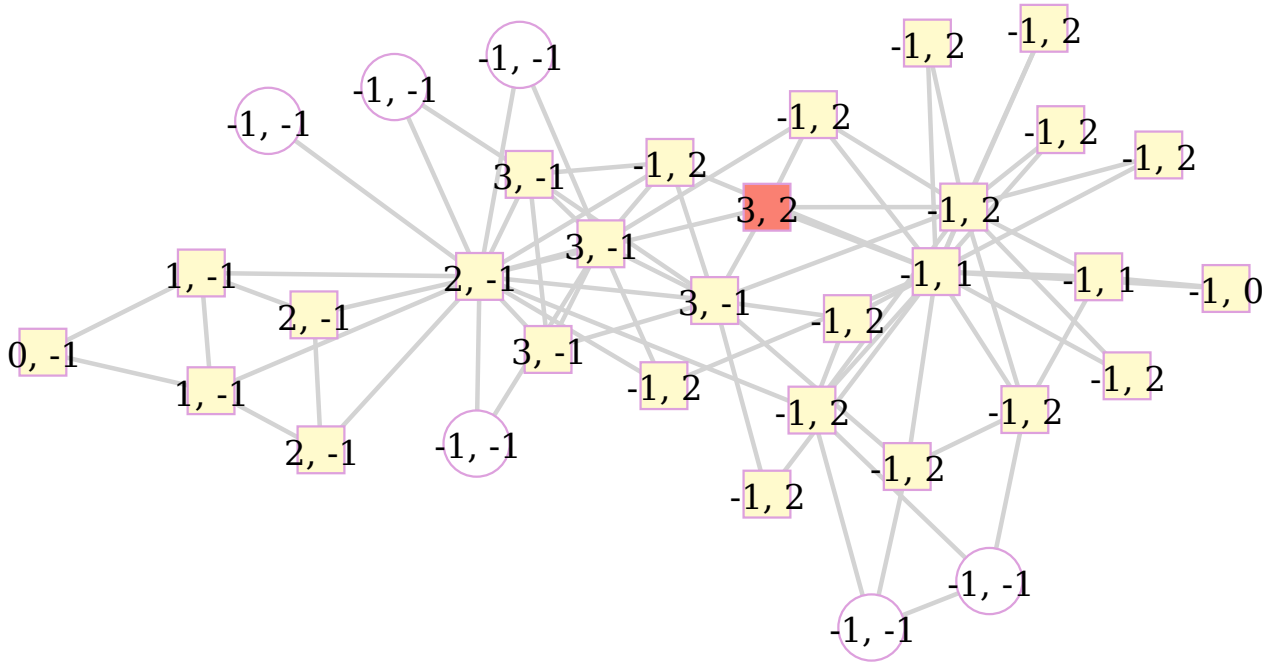

-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
1, -1
-1, -1
-1, 1
-1, 1
-1, 0
-1, -1
-1, -1
-1, -1
0, -1
-1, -1
1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1


0, -1
1, -1
1, -1
2, -1
2, -1
-1, 1
-1, 1
-1, 0


-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, 2
-1, -1
-1, 1
1, -1
2, -1
-1, 1
-1, 0
-1, -1
-1, -1
-1, -1
-1, -1
0, -1
-1, -1
-1, -1
1, -1
-1, -1
-1, 2
-1, -1
2, -1
-1, -1
-1, -1
-1, -1
-1, -1


-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, 2
-1, -1
1, -1
2, -1
-1, 1
-1, 1
-1, 0
-1, -1
2, -1
-1, -1
-1, -1
0, -1
-1, -1
-1, -1
1, -1
-1, -1
-1, 2
-1, -1
2, -1
-1, -1
-1, -1
-1, -1
-1, -1

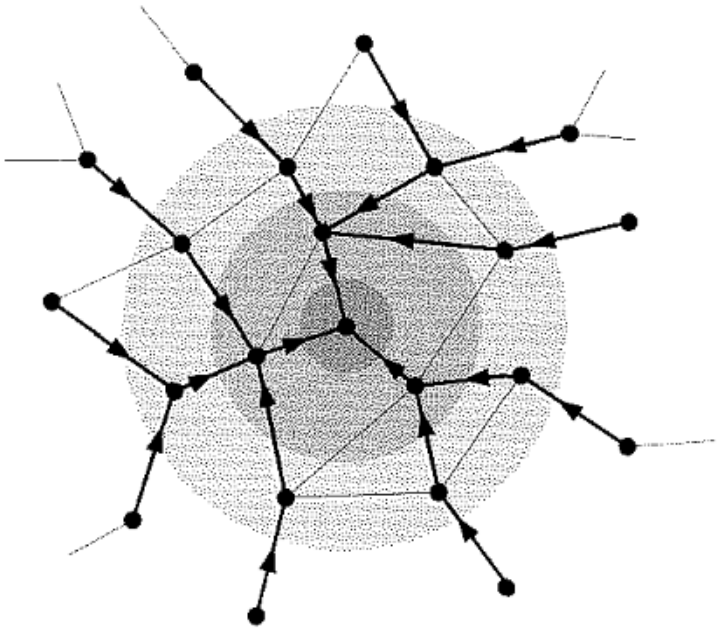

-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, -1
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 1
-1, 1
-1, 0
1, -1
1, -1
2, -1
2, -1
2, -1
2, -1
0, -1


-1, -1
-1, -1
-1, -1
3, -1
-1, 2
3, 2
3, -1
1, -1
2, -1
2, -1
3, -1
0, -1
1, -1
2, -1
-1, -1
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 1
-1, 1
-1, 0
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, 2
-1, -1
-1, -1

Running time of Two-Source BFS

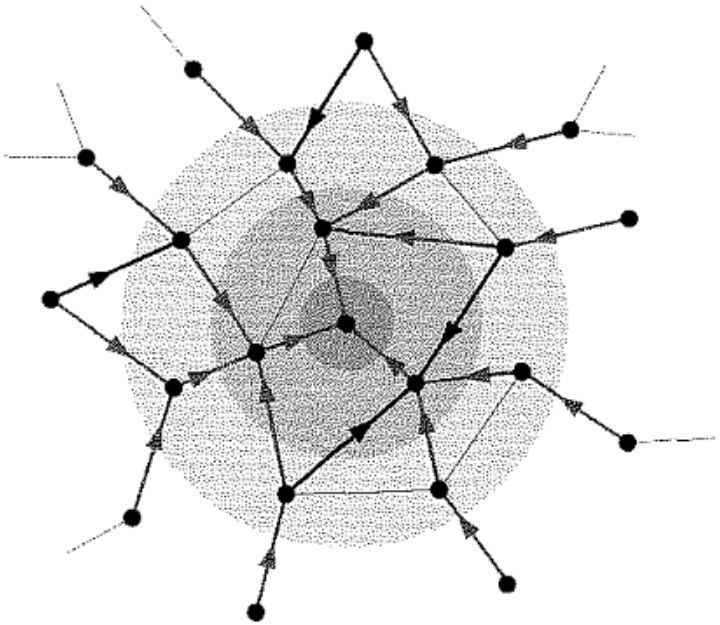
- In the single source case, about $n/2$ vertices are located till vertex t is found. For each of them, neighbors are located in $O(m/n) \implies$ total $O(n/2 \times m/n) = O(m)$ time
- In the two source case, each BFS stops after finding vertices up to distance about $d/2$
- With distance from the source, the number of vertices usually varies as c^d. Thus, in the single source case, the number of vertices visited $c^d \approx n/2$
- Hence, in the two-source case, the number of vertices visited is $c^{d/2} = \sqrt{n/2}$
- The total running time thus goes as $O(\sqrt{\frac{n}{2}} \times \frac{m}{n}) = O(m/\sqrt{n})$

Finding shortest paths



- The BFS doesn't output shortest paths, only the distance
- A small modification lets us find shortest paths also
- Each time an unseen vertex j is found, create a pointer to its predecessor i
- Pointers form a shortest-path tree for the source vertex, and a shortest paths from the source can be constructed using it
- A pointer can be created in $O(1)$, and all pointers in $O(n)$. Thus, total running time is still $O(m+n)$

Finding all shortest paths



- The algorithm finds only 1 path, whereas multiple usually paths exist between a given pair
- Thus, we must save all the predecessors for each newly found vertex
- If the neighbor j has already been assigned distance $d + 1$, there exist another shortest path via the current vertex i. Hence, i is a predecessor of j
- At the end, we get the shortest-path "tree", and all the shortest paths can be created using it

Finding all shortest paths

- Using two-source BFS, the predecessor tree can be computed in $O(m/\sqrt{n})$
- We must continue both BFSs till we find ALL common vertices
- All possible combinations of pointers should be used to find the paths