Bipartite Matching

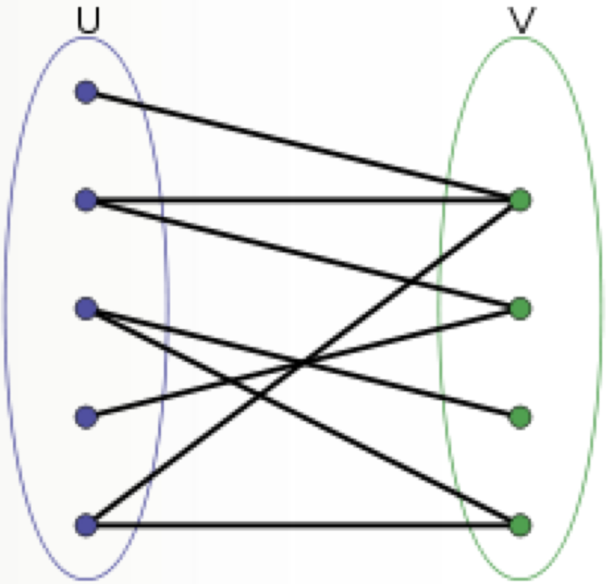
Brian A. Page
bpage1nd.edu

September 18, 2018

The College of Engineering
at the University of Notre Dame

Bipartite Graphs

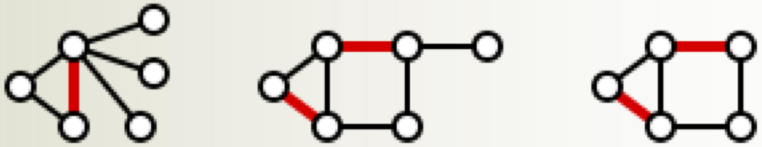
- Vertices of a graph G can be divided into two disjoint sets U and V.
- Every edge is of the form (u,v)
- No edges between vertices in same vertex set



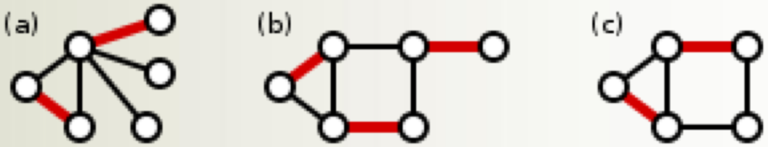
- Assignment problem, transportation problem, etc.

Graph Matching

- **Independent Edge Set (IES):** set of edges in which no two edges share a common vertex
- **Maximal**: adding an edge not in *M* destroys matching



- **Maximum** : Largest possible IES (can be many)



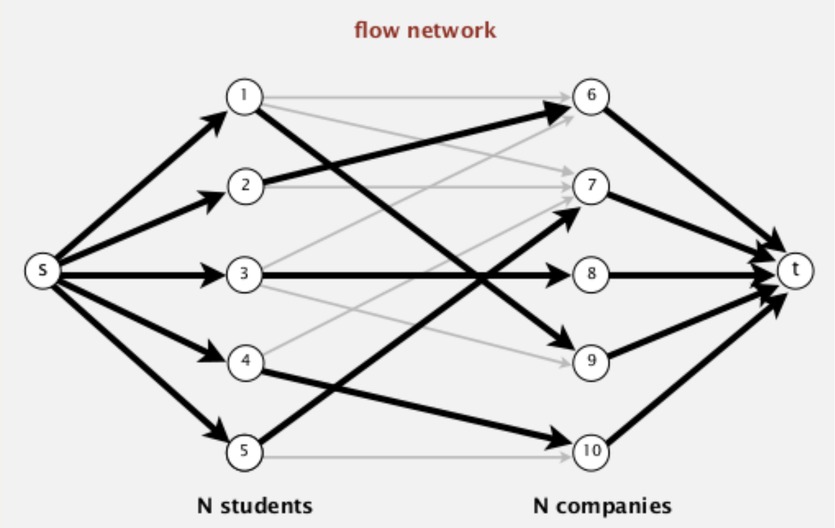
- **Perfect**: matches all V in G

Data Sets

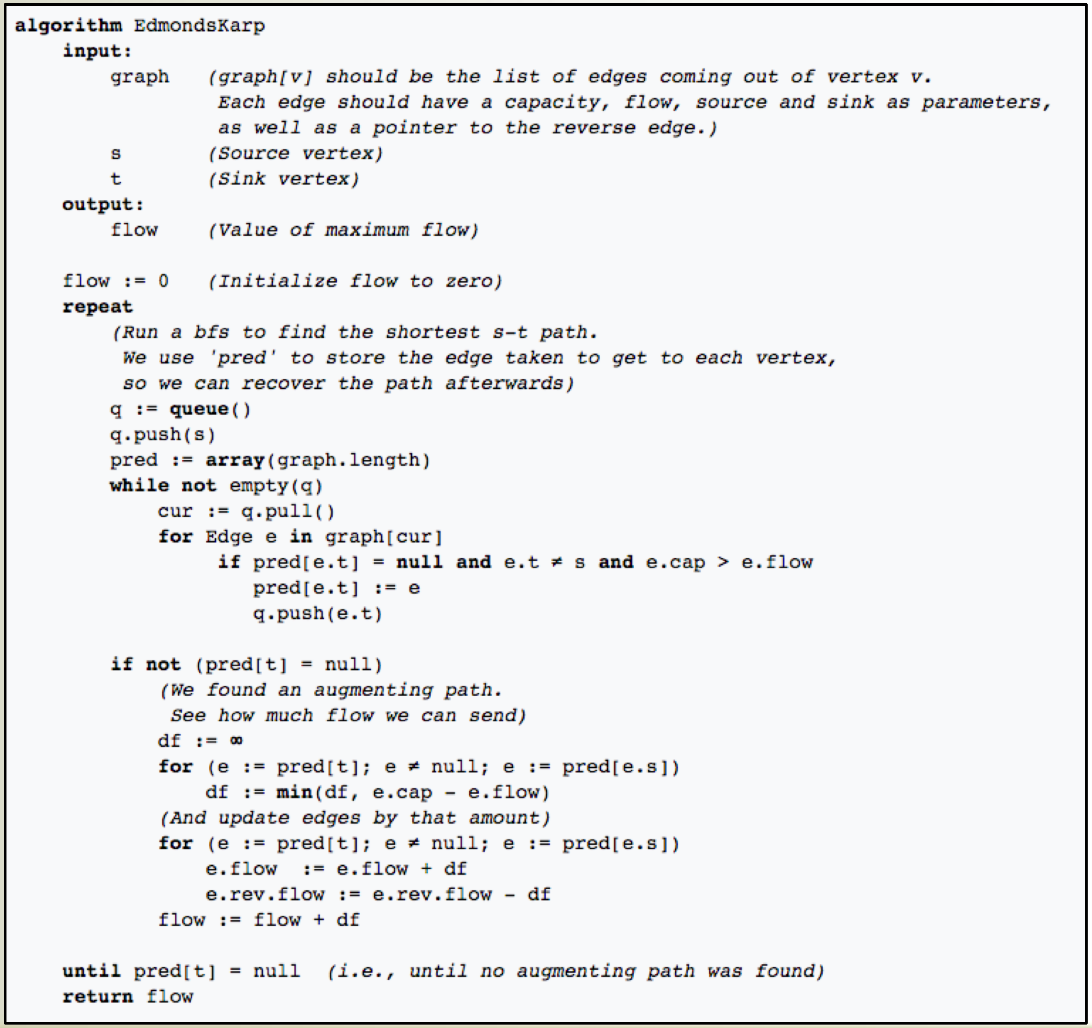
- Suite Sparse Matrix Collection
  - https://sparse.tamu.edu/
- SNAP (Stanford Large Network Dataset Collection)
  - http://snap.stanford.edu/data/
- Synthetic Graphs

Edmonds–Karp

- Based on Ford-Fulkerson maximum flow method
- *O*(VE^2) or *O*(V^2E) time depending on implementation



Edmonds-Karp

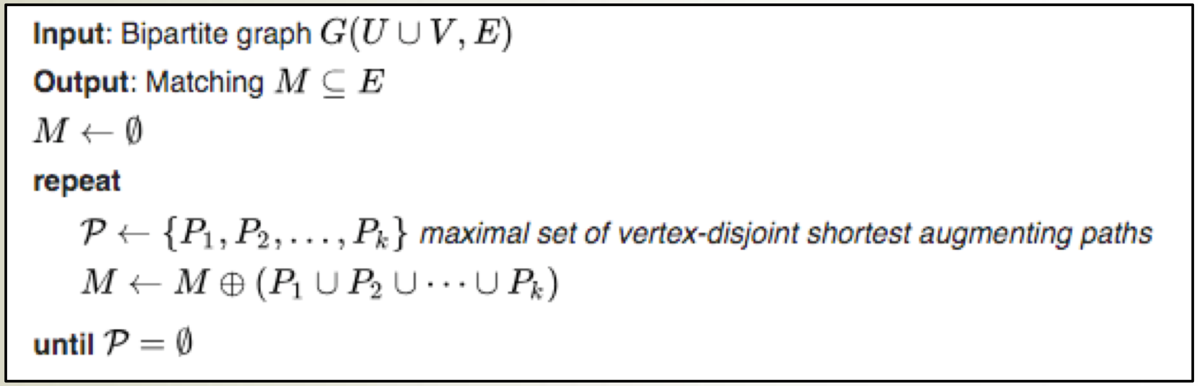


```
algorithm EdmondsKarp
    input:
        graph   (graph[v] should be the list of edges coming out of vertex v.
                 Each edge should have a capacity, flow, source and sink as parameters,
                 as well as a pointer to the reverse edge.)
        s       (Source vertex)
        t       (Sink vertex)
    output:
        flow    (Value of maximum flow)

    flow := 0   (Initialize flow to zero)
    repeat
        (Run a bfs to find the shortest s-t path.
         We use 'pred' to store the edge taken to get to each vertex,
         so we can recover the path afterwards)
        q := queue()
        q.push(s)
        pred := array(graph.length)
        while not empty(q)
            cur := q.pull()
            for Edge e in graph[cur]
                 if pred[e.t] = null and e.t ≠ s and e.cap > e.flow
                    pred[e.t] := e
                    q.push(e.t)

        if not (pred[t] = null)
            (We found an augmenting path.
             See how much flow we can send)
            df := ∞
            for (e := pred[t]; e ≠ null; e := pred[e.s])
                df := min(df, e.cap - e.flow)
            (And update edges by that amount)
            for (e := pred[t]; e ≠ null; e := pred[e.s])
                e.flow  := e.flow + df
                e.rev.flow := e.rev.flow - df
            flow := flow + df

    until pred[t] = null  (i.e., until no augmenting path was found)
    return flow
```

https://en.wikipedia.org/wiki/
Edmonds-Karp_algorithm

Hopcroft-Karp

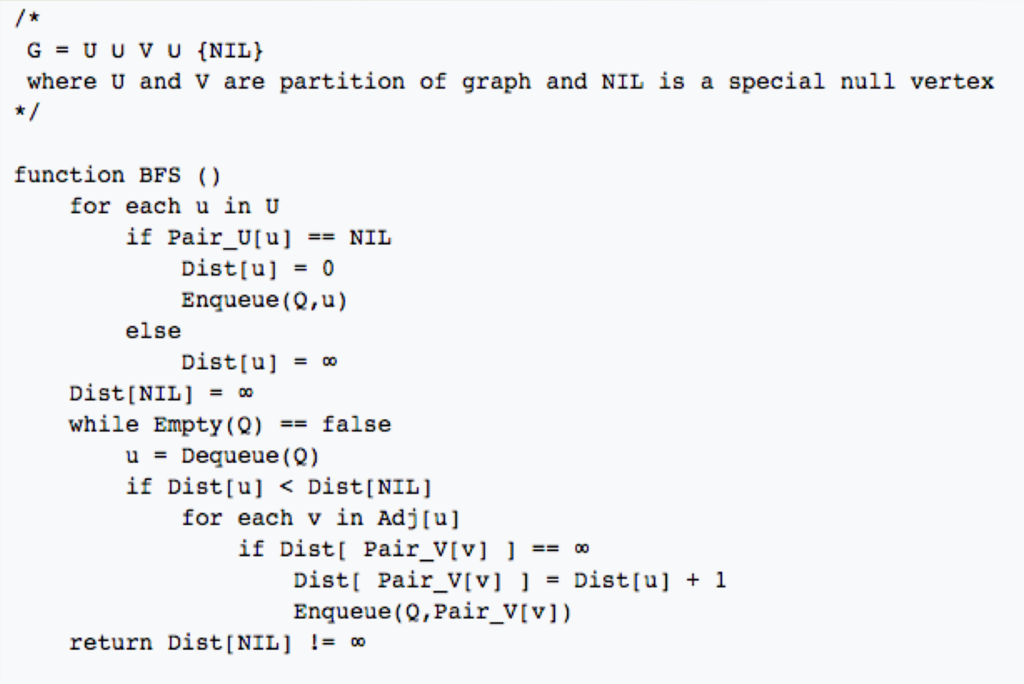
- Based on Push-relabel (maximum flow)
- Uses BFS to partition vertices into *matched* and *unmatched*
- Swaps edges in/out of matching
- Local vs global path augmentations



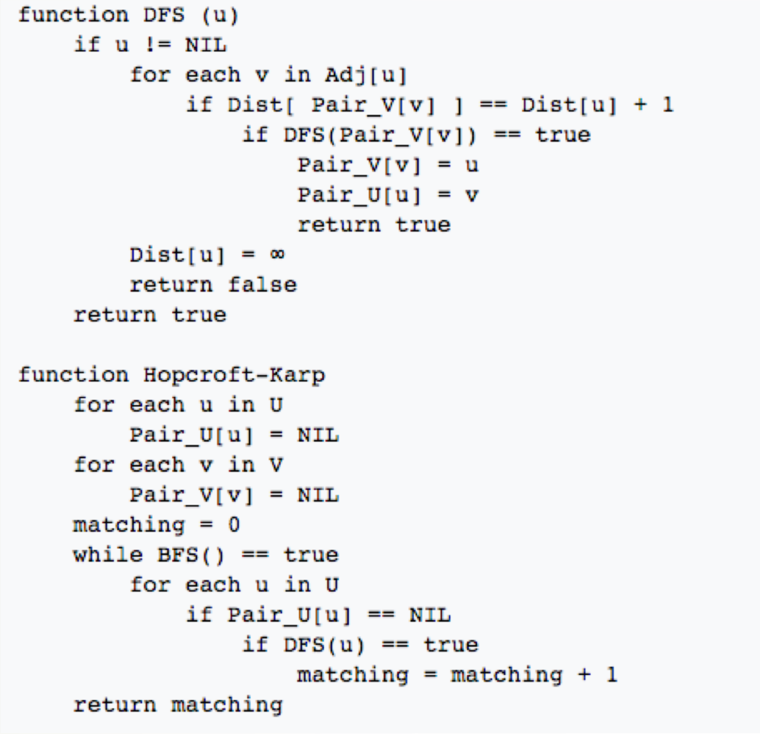
```
**Input**: Bipartite graph $G(U \cup V, E)$
**Output**: Matching $M \subseteq E$
$M \leftarrow \emptyset$
**repeat**
    $\mathcal{P} \leftarrow \{P_1, P_2, \dots, P_k\}$ maximal set of vertex-disjoint shortest augmenting paths
    $M \leftarrow M \oplus (P_1 \cup P_2 \cup \dots \cup P_k)$
**until** $\mathcal{P} = \emptyset$
```

- Runs in $O(|E|\sqrt{|V|})$

Hopcroft-Karp



```
/*
 G = U ∪ V ∪ {NIL}
 where U and V are partition of graph and NIL is a special null vertex
*/

function BFS ()
    for each u in U
        if Pair_U[u] == NIL
            Dist[u] = 0
            Enqueue(Q,u)
        else
            Dist[u] = ∞
    Dist[NIL] = ∞
    while Empty(Q) == false
        u = Dequeue(Q)
        if Dist[u] < Dist[NIL]
            for each v in Adj[u]
                if Dist[ Pair_V[v] ] == ∞
                    Dist[ Pair_V[v] ] = Dist[u] + 1
                    Enqueue(Q,Pair_V[v])
    return Dist[NIL] != ∞
```



```
function DFS (u)
    if u != NIL
        for each v in Adj[u]
            if Dist[ Pair_V[v] ] == Dist[u] + 1
                if DFS(Pair_V[v]) == true
                    Pair_V[v] = u
                    Pair_U[u] = v
                    return true
        Dist[u] = ∞
        return false
    return true

function Hopcroft-Karp
    for each u in U
        Pair_U[u] = NIL
    for each v in V
        Pair_V[v] = NIL
    matching = 0
    while BFS() == true
        for each u in U
            if Pair_U[u] == NIL
                if DFS(u) == true
                    matching = matching + 1
    return matching
```

Distributed Bipartite Matching

Pros

- Performance via strong scaling
- Larger graphs
- Shorter comp. time (hopefully)

Cons

- Greatly increase complexity
- Time now dependent on system and network

Distributed Bipartite Matching

Basic Distributed Sequential Algorithm:

```
Given $G(v,e)$ and process count $P$:
    assign vertices such that $|P_i(v)| = |P_j(v)|$
    distribute work
    Perform BM on G via some method
        if $P_i$ interacts with vertex st. $v \notin P_i(v)$
            alert $P_j$ st $v \notin P_j(v)$
            continue comp on $P_j$
    return matching
```

Distributed Bipartite Matching

- Parallel Distributed BM possibilities:
  - Connected components
  - Cut vertex separation / sub-graph assignment
  - **???**

- Questions to answer:
  - What can be parallelized?
  - How do we limit communication?
  - What other elements limit scalability and performance?

References

- Jeremy Kepner and John Gilbert. 2011. Graph Algorithms in the Language of Linear Algebra. Soc. for Industrial and Applied Math., Philadelphia, PA, USA.
- https://en.wikipedia.org/wiki/Bipartite_graph
- https://en.wikipedia.org/wiki/Matching_(graph_theory)#Bipartite_matching
- https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/
- https://en.wikipedia.org/wiki/Hungarian_algorithm#The_algorithm_in_terms_of_bipartite_graphs
- https://www.geeksforgeeks.org/hopcroft-karp-algorithm-for-maximum-matching-set-1-introduction/
- https://en.wikipedia.org/wiki/Assignment_problem