Lecture 3: Stable Matchings, Graphs and BFS

Sam McCauley

February 8, 2024

Welcome Back!

- New Assignment released
- Today: more names
- Also lots of review (which we'll go through quickly), so please ask questions!
- Any questions before we start?

Recall Stable Matching

- In stable matching: If a student s is matched to a hospital h, then for any hospital h' that s prefers to h, h' is already matched to someone they prefer to s
- And the reverse: if a hospital h is matched to a student s, any student s' that h prefers is matched to a hospital that s' prefers to h
- Intuitively: if a student calls up a hospital trying to improve their match, the hospital will always respond that they already are matched to a student they prefer

Stable Matching: First Attempt

Question: how would you match students to hospitals in practice? This looks a little like undergraduate admissions…

- Proceed in rounds
- Each unmatched hospital makes offer to its top unmatched candidate
- Each student accepts the top offer and rejects others

Let's try it with our example [On Board #1] and see if it works

It does not work. (Chris, MA) is an unstable pair! (Chris prefers MA to OH; MA prefers Chris to Beth.)

What went wrong?

- In a single round, MA offered to Aamir and OH offered to Chris
- Chris took OH since it was the best offer in that round
- But Aamir rejected MA! MA was free, but Chris never saw the offer.
- The issue with this algorithm is that if a hospital is rejected by a student, it might not be considered “on the market” by other students
- Solution: do not permanently commit to our matchings, allow students to trade up

Gale-Shapley Stable Matching Algorithm

Proceed in rounds. We keep track of “matched” pairs, but do not permanently match them until the end

- Each unmatched hospital offers to its top choice *that it has not offered to yet*
- Each student matches to top offer
- If a student gets a better offer, it rejects current offer and matches to best new offer

Gale-Shapley Pseudocode

```
To begin, all students and hospitals are unmatched
while ∃ unmatched hospital who has not proposed to every
    student:
     Let h be such a hospital
     Let s be highest-ranked student on h's list that h has not
         proposed to
     if s is unmatched:
         s and h become matched
     else:
         let h′ be the hospital s is currently matched to
         if s prefers h to h′:
             s and h become matched
             h′ becomes unmatched
return All matches
```

Let's do this out [On Board #2]

Gale-Shapley Running Time

- How long does it take to find an unmatched hospital?
- Can you come up with an $O(1)$ time solution?
    - Store unmatched hospitals in a queue (implemented with a linked list). Both of the following take $O(1)$ time: (1) find a new unmatched hospital and remove it from the list (2) add a new unmatched hospital to the list
- How long does it take to find the next student the hospital has not proposed to? Can you come up with an $O(1)$ time solution?
- One Solution: Store students in a stack (We always take the first item from the linked list.)

Gale-Shapley Running Time

```
1  To begin, all students and hospitals are unmatched
2  while ∃ unmatched hospital who has not proposed to every
       student:
3      Let h be such a hospital
4      Let s be highest-ranked student on h's list that h has not
           proposed to
5      if s is unmatched:
6          s and h become matched
7      else:
8          let h′ be the hospital s is currently matched to
9          if s prefers h to h′:
10             s and h become matched
11             h′ becomes unmatched
12 return All matches
```

- We showed: inner loop takes $O(1)$ time
- How many times can the outer loop run?
- Answer: n^2: each time a hospital proposes to a *new* student

Gale-Shapley Running Time

```
1  To begin, all students and hospitals are unmatched
2  while ∃ unmatched hospital who has not proposed to every
       student:
3      Let h be such a hospital
4      Let s be highest-ranked student on h's list that h has not
           proposed to
5      if s is unmatched:
6          s and h become matched
7      else:
8          let h′ be the hospital s is currently matched to
9          if s prefers h to h′:
10             s and h become matched
11             h′ becomes unmatched
12 return All matches
```

- Outer loop runs n^2 times, each time it runs takes $O(1)$
- Therefore: $O(n^2)$ running time
- Challenge question: why is this the best possible?

Gale-Shapley Correctness

Need to show:

- The returned matching is perfect
- The returned matching is stable

First, let's show the matching is perfect.

- Invariant: Once a student is proposed to, they will always stay in a match; furthermore they only swap if they get an offer from a hospital they prefer
- Claim: Every student eventually gets a match (intuitively why?)
  - If a student s is unmatched, some hospital h is unmatched.
  - h must not have proposed to s
  - So h is an unmatched hospital that has not proposed to each student

Goal: Use the above observations to show that the matching is stable

Gale-Shapley Correctness

Proof that the returned matching is stable by contradiction:

Assume for returned matching M, $\exists (h, s) \notin M$ such that $(h, s') \in M$ and $(h', s) \in M$, where h prefers s over s', and s prefers h over h'.

h must have offered to s before s'. Since $(h, s) \notin M$, either s broke the match to h, or s rejected the offer from h.

- A student only breaks a match if it receives a better offer. So if s broke the match to h, s must prefer h' to h; contradiction.
- If s rejected the offer from h, it must have been matched to a hospital h'' which it prefers to h. But again, then s must prefer h' to h'', and therefore prefer h' to h, a contradiction.

Stable Machings

- Since Gale-Shapley always finds a stable matching, that means a stable matching must *always exist*!

- Interesting further questions: are there multiple stable matching? Who benefits from this algorithm (do hospitals get their preference? Students?) What happens if there are ties, or partial preferences, or an uneven number of people?
  - Gale-Shapley is used for real-world medical student residency matching
  - Changed to student-proposing in 1997 due to classic algorithm favoring hospitals
  - Discussed in Algorithmic Game Theory

Gale-Shapley Wrapup

- Nobel prize in economics for this algorithm

- Traditionally called “stable marriage” problem

More Asymptotic Notation

Big Ω

- One can think of big-O as way to say $\leq$ (ignoring constants and for large n)
- Some true but not useful statements:
  - True: Insertion sort takes $O(n^{19})$ time
  - True: Finding the minimum element in an array using a linear scan takes $O(2^n)$ time
- Big-Ω notation: like big-O, but for $\geq$
- True: Insertion sort takes $\Omega(n^2)$ time
- True: Insertion sort takes $\Omega(n)$ time
- Wrong: Insertion sort takes $\Omega(n^3)$ time

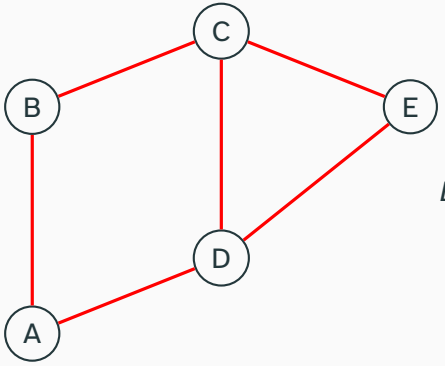
Big Θ

- We have $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- A version of "$=$" ignoring constants and for large n
- Insertion sort takes $\Theta(n^2)$ time
- True: Gale-Shapley takes $O(n^3)$ time
- Wrong: Gale-Shapley takes $\Theta(n \log n)$ time
- True: Gale-Shapley takes $\Theta(n^2)$ time

Questions about Running time and Correctness?

Graphs Review

Undirected Graph

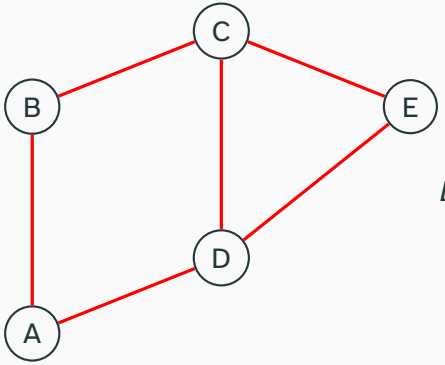


$V = \{A, B, C, D, E\}$

$E = \{(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)\}$

- Consists of a set of nodes (also called vertices) V and a set of edges E; each edge consists of a pair of vertices
- Captures *pairwise* relationships between objects
- Use $n = |V|$ and $m = |E|$ in this class.

Undirected Graph Terminology

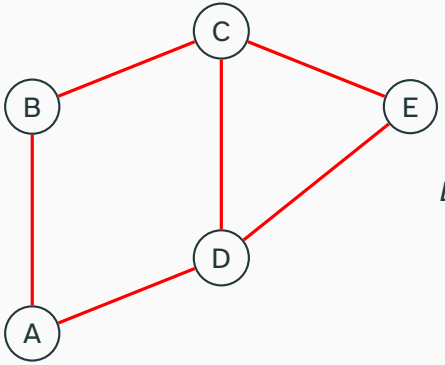


$$V = \{A, B, C, D, E\}$$

$$E = \{(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)\}$$

- Two nodes are neighbors if they share an edge
- A path is a sequence of nodes $u_1, u_2, \ldots, u_k$ such that every pair $(u_{i-1}, u_i) \in E$.
- A path is simple if all nodes in the path are distinct

Undirected Graph Terminology



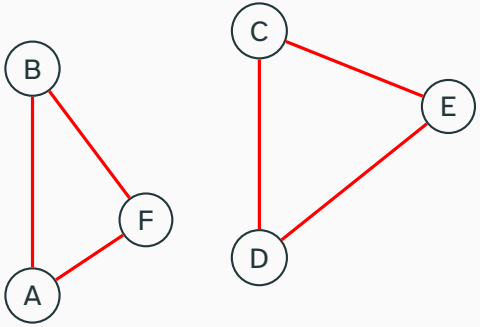
$$V = \{A, B, C, D, E\}$$

$$E = \{(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)\}$$

- The length of a path is the number of edges in the path
- A cycle is a path where $u_1 = u_k$
- A cycle is simple if all nodes other than $u_1 = u_k$ are distinct

Undirected Graph Terminology

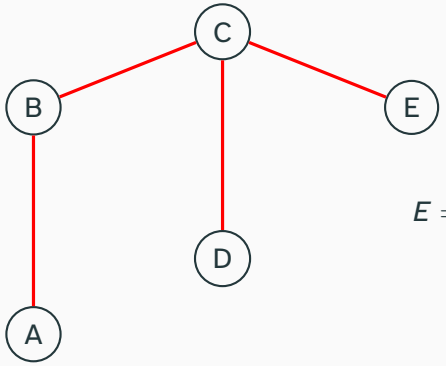
- A graph is connected if for any pair of nodes u and v there is a path from u to v.



$$V = \{A, B, C, D, E\}$$

$$E = \{(A, B), (A, F), (B, F), (C, D), (C, E), (D, E)\}$$

Tree



$V = \{A, B, C, D, E\}$

$E = \{(A, B), (B, C), (C, D), (C, E)\}$

- A graph is a tree if it is connected and does not contain a cycle
- A graph is a forest if it does not contain a cycle
- A tree may have a root node. If so, we define the height of a node v in the tree to be the number of edges in the path from the root node to v.

Graph Traversals

Graph Traversals

- Explore all the vertices and edges of a graph
- *Goal:* find out something useful about the graph
- Today we'll find the *shortest path* between two vertices u and v
  - The path with the smallest length that begins with u and ends with v

Graph Traversal Assumptions

- Today: assume that if a node has d neighbors, can find them in $O(d)$ time
- Can “traverse” an edge in time 1
- Can “mark” a vertex as visited in time 1
- We’ll revisit these assumptions next lecture when we discuss *adjacency list representation*

Breadth-First Search

Breadth-First Search (BFS)

- We’ll refer to as BFS
- Idea: start with some node s
- Slowly explore outwards from s
- “peeling one layer after another”

BFS Definition: Intuition

We define BFS using a sequence of layers

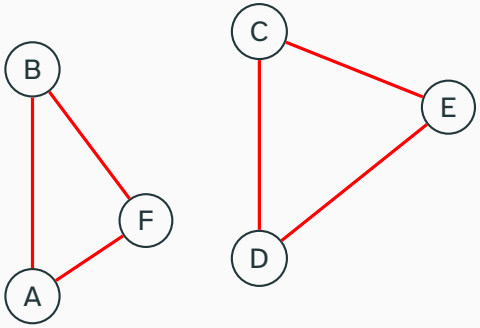
- Initialize $L_0 = \{s\}$, $i = 0$; mark s as visited
- if there exists a node in L_i with an unvisited neighbor:
    - Set L_{i+1} to be all unvisited neighbors of nodes in L_i; mark all nodes in L_{i+1} as visited; set $i = i + 1$

Let's do an example [On Board #3]

Any questions about this algorithm? We'll look at pseudocode for this algorithm next lecture

What does BFS Do?

- Keeps exploring until run out of nodes to explore
- Question: can you give an example of a graph (and a starting vertex s in the graph) where BFS does not visit all nodes?



$$V = \{A, B, C, D, E\}$$

$$E = \{(A, B), (A, F), (B, F), (C, D), (C, E), (D, E)\}$$

If the graph is not connected, BFS will not visit all nodes.

First key BFS Property

Theorem

If $(x, y) \in E$, and $x \in L_i$ and $y \in L_j$ for a BFS starting at some node s, then i and j differ by at most 1 (that is to say: $|i - j| \leq 1$)

Proof: Assume the contrary, that $i - j \geq 2$ or $j - i \geq 2$.

First, let's say $i \geq j + 2$. Since $y \in L_j$, all unvisited neighbors of y are added to L_{j+1}. Since x is not in level $L_{j'}$ for $j' \leq j$, x is unvisited, so x is added to L_{j+1}, a contradiction.

Second, let's say $j \geq i + 2$. (This case is basically identical.) Since $x \in L_i$, all unvisited neighbors of x are added to L_{i+1}. Since y is not in level $L_{i'}$ for $i' \leq i$, y is unvisited, so y is added to L_{i+1}, a contradiction.

BFS and Connected Graphs

I claim: in a connected graph G, BFS starting at vertex s will visit every vertex. Can we prove this using the BFS property we showed?

Consider some vertex v; we show that BFS visits v. Since G is connected there is a path from s to v; call this path $p = s, v_1, v_2, \ldots, v_k, v$.

Idea: We have that $s \in L_0$. Since v_1 is a neighbor of s, $v_1 \in L_1$. Let's generalize to all v_i using an induction.

Prove by induction: v_i is in level L_j for some $j \leq i$. Base case: $i = 1$ by above. Assume true for some i. Since v_{i+1} is a neighbor of v_i, then v_{i+1} must be in level $L_{j'}$ where $|j - j'| \leq 1$. Since $j \leq i$, we must have $j' \leq i + 1$.

Running BFS

- On disconnected graphs: if we run out of vertices, start again from a new unvisited vertex
- Cost for BFS to explore a node v with d_v neighbors?
- Answer: $O(1 + d_v)$
- Total running time:

$$\sum_{v \in V} O(1 + d_v) = O\left(n + \sum_{v \in V} d_v\right) = O(n + 2m) = O(n + m)$$

Recall that since each edge is adjacent to two vertices, $\sum d_v = 2|E|$.

The BFS Tree

- The levels explored by the BFS are the levels of a tree (i.e. the nodes at a particular height)
- If v' is a neighbor of v that we add to some level, then v is the parent of v'.
- Let's do an example [On Board #4]
- We can calculate the BFS tree while doing the BFS in $O(n + m)$ time
  - Useful for some applications!
  - And some algorithms assignments

BFS to find Shortest Path

- BFS gives the *shortest path* between the initial vertex s and any other vertex v in the graph
    - We call the length of the shortest path between two vertices u and v the *distance* betwen u and v
- How can we formalize?

Theorem

For any vertex v, if v is at height d of the BFS tree rooted at s, then the shortest path from s to v has length d.

BFS to find Shortest Path

Theorem

For any vertex v, v is at depth d of the BFS tree rooted at s if and only if the shortest path from s to v has length d.

Proof: By strong induction on d. Base case: for $d = 0$, the only vertex with a shortest path of length 0 from s is s; we have that $L_0 = \{s\}$ by definition of BFS.

Now, assume that for some d, for all $1 \leq k \leq d$, L_k consists of all vertices whose shortest path from s has length k. (Goal: show that L_{d+1} consists of all vertices w/ shortest path length $d + 1$.)

First, we show that if a vertex v is in L_{d+1}, its shortest path from s has length $d + 1$. We break this into two parts: first we show that there exists a path of length $d + 1$; then we show that no path has length $< d + 1$.

BFS to find Shortest Path

Proof: Now, assume that for some d, for all $1 \leq k \leq d$, L_k consists of all vertices whose shortest path from s has length k. (Goal: show that L_{d+1} consists of all vertices w/ shortest path length $d+1$.)

First, we show that if a vertex v is in L_{d+1}, its shortest path from s has length $d+1$. We break this into two parts: first we show that there exists a path of length $d+1$; then we show that no path has length $< d+1$.

Since $v \in L_{d+1}$, v has a neighbor $v' \in L_d$. By the I.H., the shortest path from s to v' has length d. Therefore, there is a path from s to v of length $d+1$, so the shortest path from s to v has length *at most* $d+1$.

Now, we show that no path from s to v has length $< d+1$. Consider a path of length k, $p = s, v_1, \ldots, v_{k-1}, v$ for $k < d+1$. By the I.H., v_{k-1} is in level L_{k-1}; but since there is an edge from v_{k-1} to v, v must be in L_k or earlier, contradicting our assumption that $v \in L_{d+1}$.

BFS to find Shortest Path

Proof: *Recall:* Proof By strong induction on d.

Now, assume that for some d, for all $1 \leq k \leq d$, L_k consists of all vertices whose shortest path from s has length k. (Goal: show that L_{d+1} consists of all vertices w/ shortest path length $d+1$.)

Now, we show that if the shortest path from s to v has length $d+1$, then $v \in L_{d+1}$. By I.H., $v \notin L_j$ for $j < d+1$.

Let $p = s, v_1, \ldots, v_d, v$ be a path of length $d+1$ from s to v. By the I.H., $v_d \in L_d$. When we explore the neighbors of v_d, we cannot have already explored v since $v \notin L_j$ for $j < d+1$; thus $v \in L_{d+1}$

BFS to find Shortest Path (wrapup)

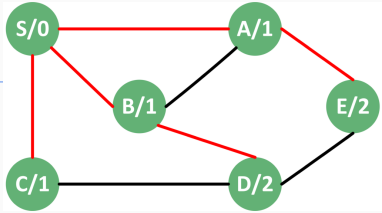
Theorem

For any vertex v, v is at depth d of the BFS tree rooted at s if and only if the shortest path from s to v has length d.

Proof: By strong induction on d. Base case: for $d = 0$, the only vertex with a shortest path of length 0 from s is s; we have that $L_0 = \{s\}$ by definition of BFS.

Summary: We have shown that assuming the I.H. for all $1 \leq k \leq d$, if $v \in L_{d+1}$, then the shortest path from s to v has length $d + 1$; furthermore, if the shortest path from s to v has length $d + 1$, then $v \in L_{d+1}$. Therefore the inductive step is complete.

BFS Properties Summary



- Starts at some node s
- Partitions vertices into levels $L_0, L_1, \ldots$
- Gives a BFS tree T; a vertex at height h in the tree is in L_h
- If $(x, y) \in E$, the level of x and y differ by ≤ 1
- A vertex is at height h in T if and only if its shortest path from s has distance h

Implementing BFS

Implementing BFS

- Can we be more specific about how BFS works?
- Maybe give pseudocode?
- Do we need to store the levels explicitly? How should we store them?
- Key insight: we can explore the nodes in level L_{i+1} in the same order they were added to L_{i+1}. (And note that they were added before any node in L_{i+2}
- So: explore nodes in the same order they were visited!

BFS Pseudocode

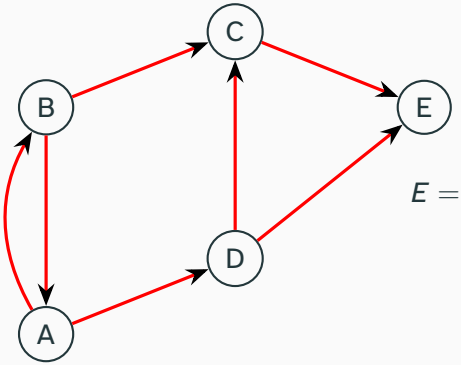
```
1  BFS(G, s):
2      Put s in a queue Q
3      while Q is not empty:
4          v = Q.dequeue() # take the first vertex from Q
5          if v is not marked as visited:
6              mark v as visited
7              for each edge (v,w):
8                  Q.enqueue(w) # add w to Q
```

Note: this algorithm only works if all vertices in *G* are not marked as visited!

Question: How can we calculate the BFS tree *T*? How can we alter this algorithm to give the levels of each vertex?

Directed Graphs

Directed Graphs



$$V = \{A, B, C, D, E\}$$

$$E = \{(A,B), (B,A), (A,D), (B,C), (D,C), (C,E), (D,E)\}$$

- In a directed graph, edges have an ordering: an edge (u, v) is *from* u *to* v.
- Called *directed edges* (some call them arcs; I won't however)
- Good for capturing some kinds of data (website links, etc.)
- Notion of a path, etc., is the same
- We'll discuss connectivity of directed graphs next lecture

Storing a Graph

How to store a graph?

Goal: Use data structures we know to store a graph to allow things like traversals

- *Adjacency List* representation
- *Adjacency Matrix* representation

Adjacency List

- For each vertex, store all neighbor edges/vertices in a linked list
- Works well for:
  - Can find all d_v neighbors of v in $O(1 + d_v)$ time
  - Only requires $O(n + m)$ space (why?)
- Does not work well for:
  - Given an edge $e = (u, v)$, is $e \in E$?
  - Must scan through neighbors of u; requires $\Omega(d_u)$ time.

Example [On Board #5]

Adjacency Matrix

- Store an $n \times n$ matrix
- Store a 1 in entry (i, j) if there is an edge from the ith to the jth vertex
- Works well for:
  - Given an edge $e = (u, v)$, is $e \in E$ in $O(1)$ time.
- Does not work well for:
  - Requires $\Omega(n^2)$ space
  - Finding all d_v neighbors of v takes $\Omega(n)$ time
- Used much less often

Example [On Board #6]