GRAPHS

FRANK TSAI

Contents

1. Graphs 1
2. Graph exploration algorithms 4

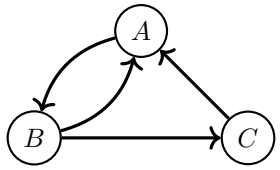
1. Graphs

Definition 1.1. A (directed) *graph* G consists of the following data:

(i) A set V of *vertices*.
(ii) A set $E \subseteq V \times V$ of *edges*.

Remark 1.2. Recall that a binary relation on a set V can be encoded as any subset of $V \times V$. Thus, a graph is a set V equipped with a binary relation E.

Example 1.3. Let $V = \{A, B, C\}$ and $E = \{(A, B), (B, A), (B, C), (C, A)\}$.

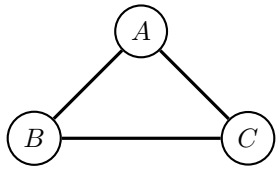


Definition 1.4. A graph (V, E) is *undirected* if the binary relation E is symmetric.

Remark 1.5. Example 1.3 is not an undirected graph since $(B, C) \in E$, but $(C, B) \notin E$. Similarly, $(C, A) \in E$, but $(A, C) \notin E$.

Remark 1.6. In an undirected graph, we drop the arrow tips as they convey no additional information.

Example 1.7. Let $V = \{A, B, C\}$ and $E = V \times V$. Note that E is symmetric.



Definition 1.8. Let $G = (V, E)$ be a graph. A *walk* is a sequence of vertices and edges defined inductively as follows:

(i) The empty sequence is a walk.

Date: November 11, 2023.

(ii) If $(v_1, v_2) \in E$ then the sequence $v_1, (v_1, v_2), v_2$ is a walk.
(iii) If $(v_n, v_m) \in E$ and $v_1, (v_1, v_2), v_2, \ldots, v_n$ is a walk then the sequence

$$v_1, (v_1, v_2), v_2, \ldots, v_n, (v_n, v_m), v_m$$

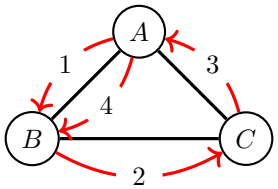
is a walk.

Definition 1.9. The *length* of a walk is the number of edges in that walk.

Example 1.10. Let G be the graph defined in Example 1.7. The sequence

$$A, (A, B), B, (B, C), C, (C, A), A, (A, B), B$$

is a walk.



Each red arrow represents a step of the walk.

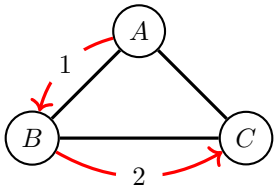
Definition 1.11. Let $G = (V, E)$ be a graph. A *path* is a walk in which all vertices are distinct.

Counterexample 1.12. The walk defined in Example 1.10 is not a path since the vertices A and B are repeated.

Example 1.13. Let G be the graph defined in Example 1.7. The sequence

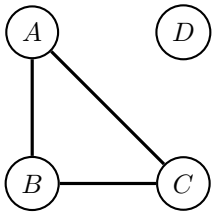
$$A, (A, B), B, (B, C), C$$

is a path.

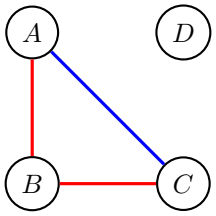


Definition 1.14. Let $G = (V, E)$ be a graph. For any two vertices $u, v \in V$, u and v are said to be *s-t reachable* if there is a path from u to v.

Example 1.15. Consider the following (undirected) graph:



A and C are s-t reachable. There are two paths connecting them (blue and red, respectively):



Counterexample 1.16. Consider the same graph defined in Example 1.15, the vertices A and D are not s-t reachable.

Theorem 1.17. *Let $G = (V, E)$ be an undirected graph. The s-t reachability relation defined in Definition 1.14 is an equivalence relation on the set of vertices V.*

Proof. (Reflexivity): Let $v \in V$. The empty walk is trivially a path. Thus, v is s-t reachable from itself.

(Symmetry): Let $u, v \in V$ be any two vertices. Assume that v is s-t reachable from u, i.e., there is a path p from u to v. We need to prove that u is s-t reachable from v, i.e., there is a path from v to u. To this end, we do induction on the length of p. In the base case, p has length 0, so it is the empty path. Thus, the result follows from reflexivity. In the induction step, p is a path of length $k+1$ and it is of the form $p', (u', v), v$, where u' is some vertex and p' is a path from u to u' of length k. By the induction hypothesis, there is a path $\widetilde{p}$ from u' to u. Since G is undirected, (v, u') is also an edge of G. Thus, the sequence

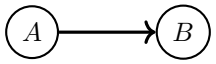
$$v, (v, u'), \widetilde{p}$$

defines a path from v to u.

(Transitivity): Let $u, v, w \in V$ be any three vertices. Assume that there is a path p from u to v and a path p' from v to w. The concatenation of p and p' defines a path from u to w. You can prove this rigorously by induction on the length of p. The detail is left as an exercise. □

Remark 1.18. In general, Theorem 1.17 fails to hold for directed graphs because the s-t reachability relation is, in general, not symmetric.

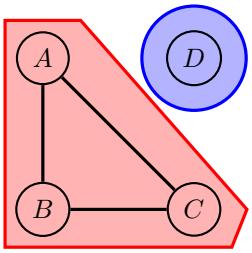
Counterexample 1.19. Consider the following graph.



There is a path from A to B, but there is no path from B to A.

Definition 1.20. Let G be an undirected graph. A *(path) connected component* of G is an equivalence class with respect to the s-t reachability relation.

Example 1.21. The graph defined in Example 1.15 has exactly 2 connected components (red and blue, respectively).



Definition 1.22. An undirected graph G is said to be *(path) connected* if it has exactly one connected component.

Example 1.23. The graph defined in Example 1.7 is connected because it has exactly one connected component.

Counterexample 1.24. The graph defined in Example 1.15 is not connected because it has two connected components.

2. Graph exploration algorithms

Remark 2.1. To explore every vertex of an undirected graph (V, E), we divide V into two piles: frontier and unexplored.

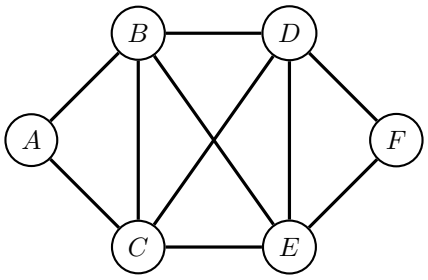
Remark 2.2. If the unexplored pile is empty, then that means every vertex has been explored. We can remove everything from the frontier pile.

Remark 2.3. If the frontier pile is empty, move an arbitrary vertex in the unexplored pile to the frontier pile.

Remark 2.4. If the frontier pile is not empty, pick the next vertex v in the frontier pile and move every vertex adjacent to v in the unexplored pile to the frontier pile. At this point, v has been fully explored, so we can remove v from the frontier pile.

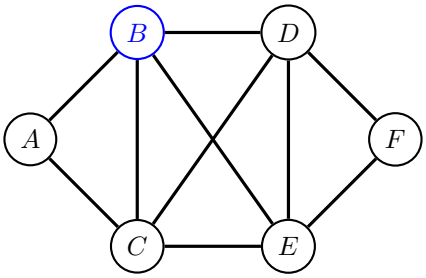
Remark 2.5. The "next vertex" in Remark 2.4 is intentionally abstract. There are two natural ways to pick the "next vertex:" pick the first vertex or the last vertex added to the frontier pile. The former choice corresponds to breadth-first search (BFS) and the latter choice corresponds to depth-first choice (DFS).

Example 2.6. Consider the following graph:

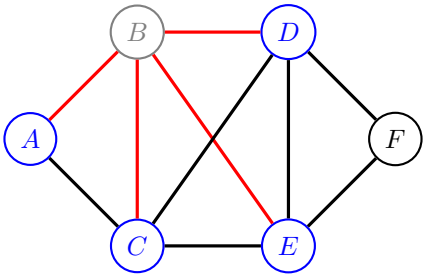


We run BFS on this graph.

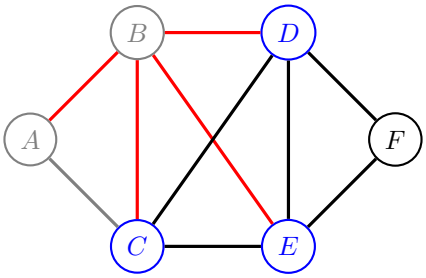
(Step 1): The frontier pile is empty and the unexplored pile consists of every vertex in the graph. By Remark 2.3 we move an arbitrary vertex, e.g, B, to the frontier pile.



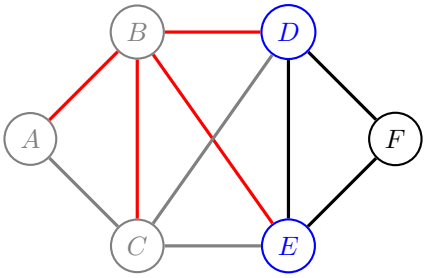
(Step 2): By Remark 2.4, pick the next vertex B. We have to move A, C, D, and E to the frontier pile and remove B from the frontier pile.



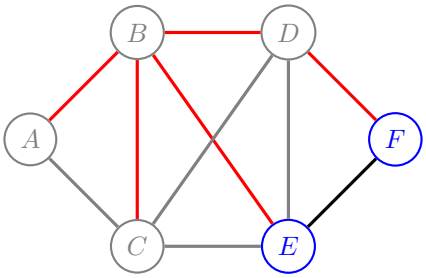
(Step 3): By Remark 2.4, pick the next vertex A. We do not have to move anything to the frontier pile because it is not adjacent to any unexplored vertex. Finally, we need to remove A from the frontier pile.



(Step 4): By Remark 2.4, pick the next vertex C. Again, we do not have to move anything to the frontier pile because it is not adjacent to any unexplored vertex. Finally, we need to remove C from the frontier pile.



(Step 5): By Remark 2.4, pick the next vertex D. We need to move F to the frontier pile and remove D from the frontier pile.



(Step 6): The unexplored pile is empty. By Remark 2.2, we can remove everything from the frontier pile.