GRAPHS

FRANK TSAI

Contents

1. Graphs 1
2. Graph exploration algorithms 4

1. Graphs

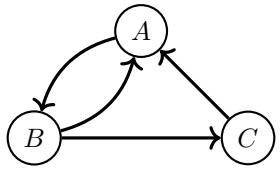
Definition 1.1. A (directed) *graph* G consists of the following data:

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

(1.1) 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.

Notation 1.2. The data described in Definition 1.1 can be formalized as a pair that consists of a set of vertices V and a set of edges E. We write (V, E) to denote a graph with a vertex set V and an edge set E.

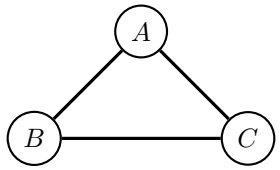
Example 1.3. Let $V = \{A, B, C\}$ and $E = \{(A, B), (B, A), (B, C), (C, A)\}$.



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

(1.2) Example 1.3 is not an undirected graph since $(B, C) \in E$, but $(C, B) \notin E$. Similarly, $(C, A) \in E$, but $(A, C) \notin E$.
(1.3) In an undirected graph, we drop the arrow tips as they convey no additional information.

Example 1.5. Let $V = \{A, B, C\}$ and $E = V \times V$. Note that E is symmetric.



Date: November 12, 2023.

Definition 1.6. 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.

(ii) 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

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

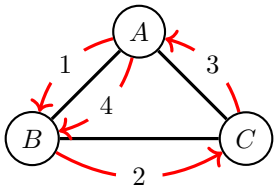
is a walk.

Definition 1.7. The *length* of a walk is the number of edges in that walk.

Example 1.8. Let G be the graph defined in Example 1.5. 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.

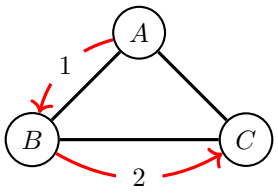
Definition 1.9. Let $G = (V, E)$ be a graph. A *path* is a walk in which all vertices are distinct.

Counterexample 1.10. The walk defined in Example 1.8 is not a path since the vertices A and B are repeated.

Example 1.11. Let G be the graph defined in Example 1.5. The sequence

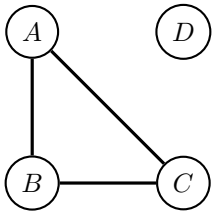
$$A, (A, B), B, (B, C), C$$

is a path.

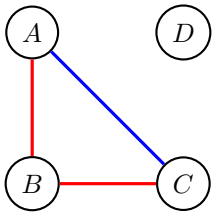


Definition 1.12. Let $G = (V, E)$ be a graph. For any two vertices $u, v \in V$, u and v are said to be *s-t reachable* if there is a path from u to v.

Example 1.13. Consider the following (undirected) graph:



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



Counterexample 1.14. Consider the same graph defined in Example 1.13, the vertices A and D are not s-t reachable.

Theorem 1.15. *Let $G = (V, E)$ be an undirected graph. The s-t reachability relation defined in Definition 1.12 is an equivalence relation on the set of vertices V.*

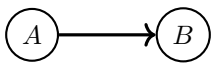
Proof.
(1.4) Reflexivity: Let $v \in V$. The empty walk is trivially a path. Thus, v is s-t reachable from itself.
(1.5) Symmetry: Let $u, v \in V$ be any two vertices. Assume that v is s-t reachable from u, i.e., there is a path p from u to v. We need to prove that u is s-t reachable from v, i.e., there 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.
(1.6) 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 1.16. In general, Theorem 1.15 fails to hold for directed graphs because the s-t reachability relation is, in general, not symmetric.

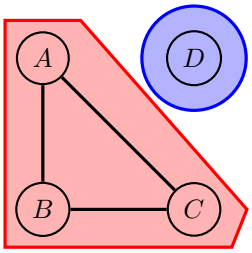
Counterexample 1.17. Consider the following graph.



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

Definition 1.18. 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 1.19. The graph defined in Example 1.13 has exactly 2 connected components (red and blue, respectively).



Definition 1.20. An undirected graph G is said to be *(path) connected* if it has exactly one connected component.

Example 1.21. The graph defined in Example 1.5 is connected because it has exactly one connected component.

Counterexample 1.22. The graph defined in Example 1.13 is not connected because it has two connected components.

2. Graph exploration algorithms

(2.1) To explore every vertex of an undirected graph (V, E), we divide V into three piles: explored, frontier, and unexplored. Initially, the unexplored pile contains every vertex and the explored and frontier piles are empty.

(2.2) We can explore the graph recursively. We match the current piles against the conditions stated in **(2.3)**, **(2.4)**, and **(2.5)** in this order.

(2.3) If the unexplored pile is empty, we can move every vertex in the frontier pile to the explored pile. This is the base case of the recursion.

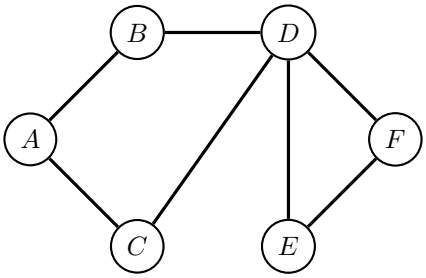
(2.4) If the frontier pile is empty, move an arbitrary vertex in the unexplored pile to the frontier pile. Make a recursive call on the current piles (i.e., go back to **(2.2)**).

(2.5) If the frontier pile is not empty, pick the next vertex v in the frontier pile. If v is not adjacent to any unexplored vertex, then v has been fully explored, so we can move it to the explore pile. Otherwise, we move an arbitrary vertex adjacent to v in the unexplored pile to the frontier pile. Finally, we make a recursive call on the current piles (i.e., go back to **(2.2)**).

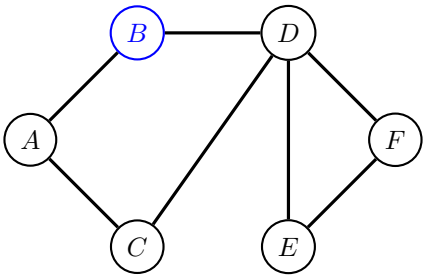
Remark 2.1. The "next vertex" in **(2.5)** is kept abstract intentionally. There are two natural ways to pick the "next vertex:" pick the first vertex or the last vertex added to the frontier pile. The former choice corresponds to breadth-first search (BFS) and the latter choice corresponds to depth-first search (DFS).

Remark 2.2. Each recursive call in **(2.4)** and **(2.5)** is made on a smaller unexplored pile. We can prove theorems about this algorithm by strong induction on the size of the unexplored pile.

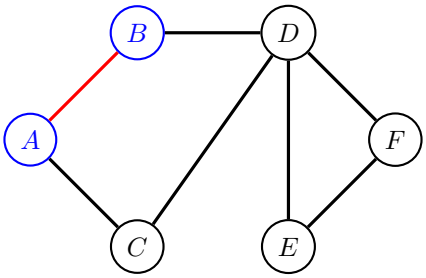
Example 2.3. Run BFS on the following graph:



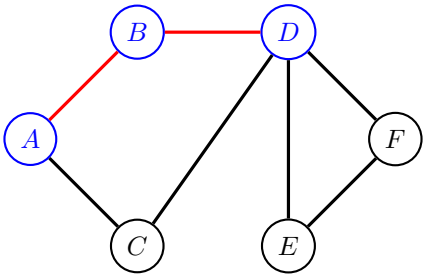
(2.6) Initially, the frontier pile is empty. This matches **(2.4)**, so we move an arbitrary vertex in the unexplored pile to the frontier pile.



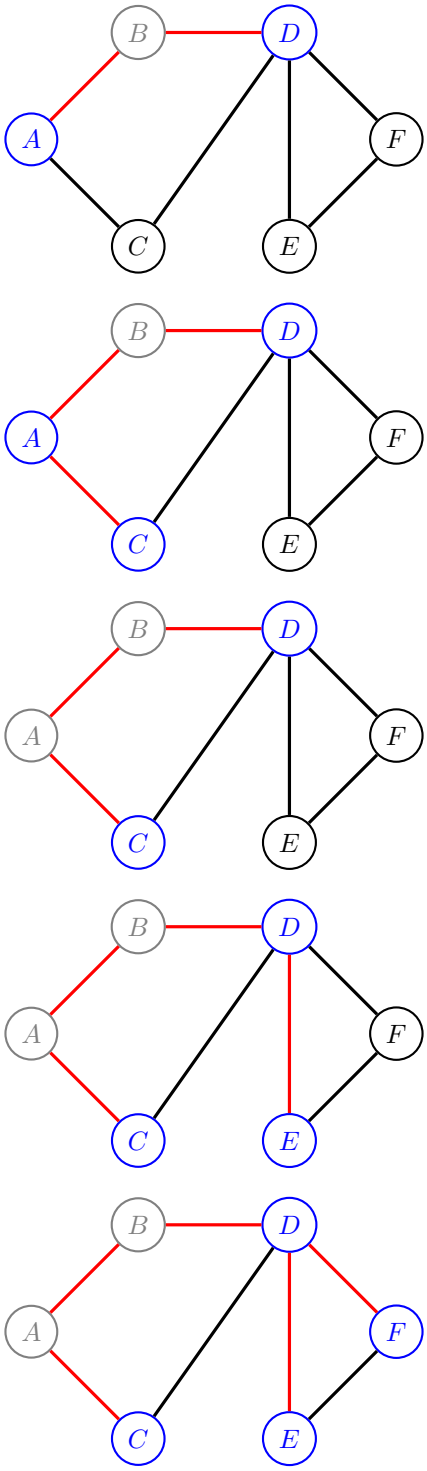
(2.7) The frontier pile is not empty. This matches **(2.5)**. B is adjacent to an explored vertex (e.g., A) so we move A to the frontier pile.



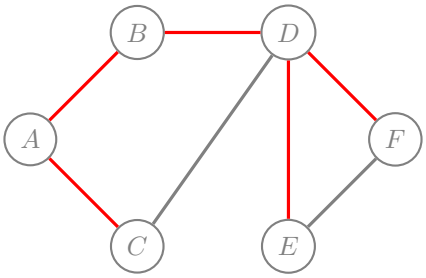
(2.8) The frontier pile is not empty. This matches **(2.5)**. B is added to the frontier pile first, so we choose to explore it.



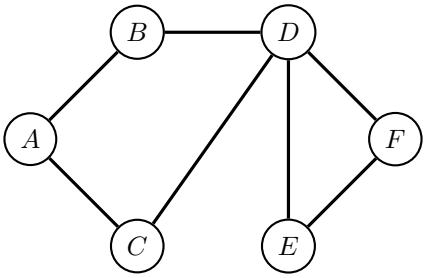
(2.9) The frontier pile is not empty. This matches **(2.5)**. Since B is not adjacent to any unexplored vertex, we can move it to the explored pile.



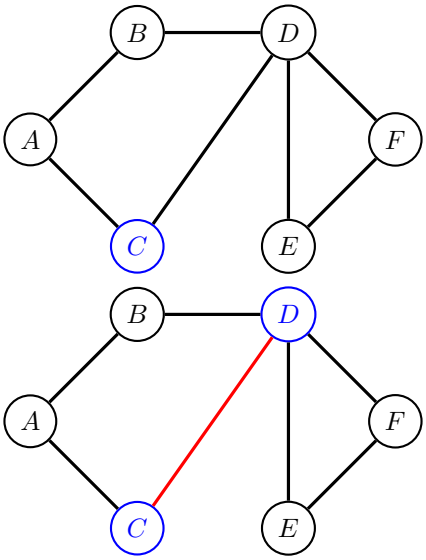
(2.10) At this point, the unexplored pile is empty, so we have reached the base case **(2.3)**.



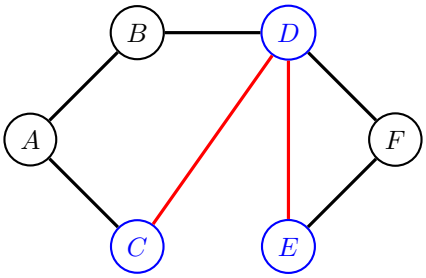
Example 2.4. Run DFS on the same graph defined in Example 2.3.

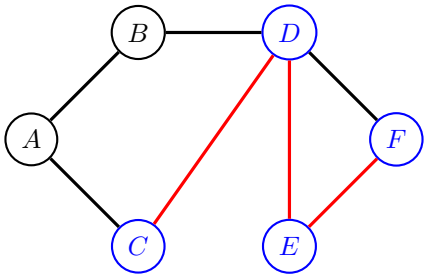


(2.11) This time, let us arbitrarily pick the vertex C.

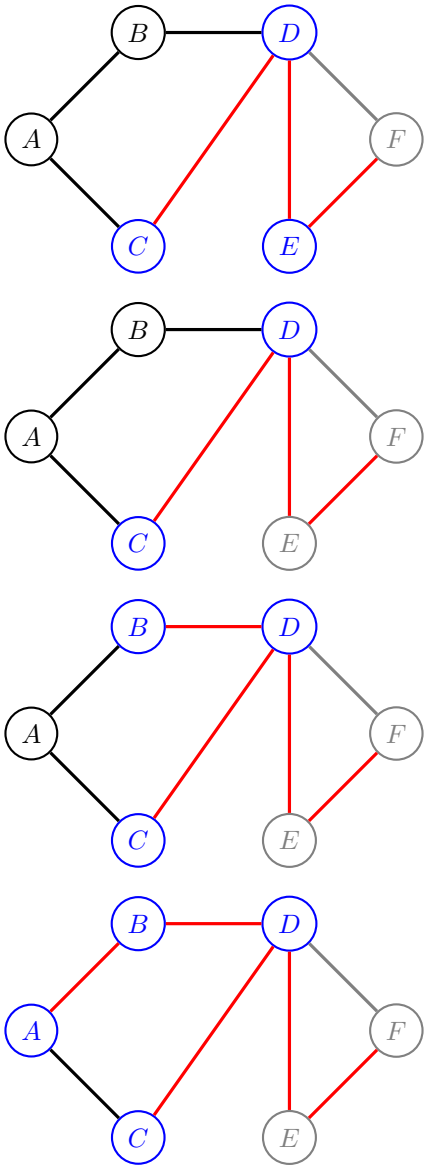


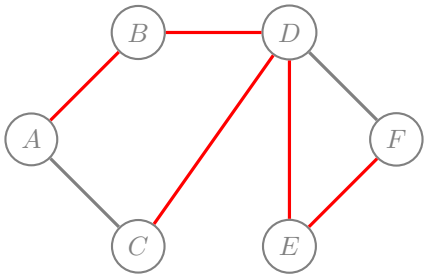
(2.12) Since D is added after C, we need to explore D first.





(2.13) At this point, F has been fully explored. We backtrack until we find a vertex that has not been fully explored.




B
D
A
F
C
E