CS331: Algorithms and Complexity
Part V: Graph Algorithms

Kevin Tian

1 Introduction

In this part of the notes, we focus on algorithms for problems on graphs. Graphs are some of the most fundamental and important structures in computer science. As a result, graph algorithms have many applications, both in their own right, and as subroutines in other algorithms.

Throughout, we follow notation in our introduction to graphs in Section 4, Part I, letting $G = (V, E, \mathbf{w})$ be a weighted graph with vertices V and edges $E \in V \times V$. If the weights $\mathbf{w}$ are unspecified, they are assumed to all be 1, in which case G is unweighted. We specify in context whether G is directed or undirected. We always assume G is simple (i.e., has no parallel edges or self-loops), and is given as an input in adjacency list format (see Section 4.3, Part I). Unless otherwise specified, $n := |V|$ and $m := |E|$ refer to the vertex and edge counts of G.

We have already seen several graph algorithms, e.g., single-source shortest paths on DAGs and all-pairs shortest paths (Sections 5.2 and 5.3, Part III), and minimum spanning trees (Section 4.1, Part IV). One recurring theme in these examples is the interplay between general algorithmic paradigms (e.g., data structures, and principles of recursion, DP, and greedy) with graph structure.

The MST algorithm in Section 4.1, Part IV is particularly illustrative of this theme. First of all, we designed and analyzed it by adopting a "greedy stays ahead" perspective. This perspective was fueled by an exchange lemma (Lemma 3, Part IV), that crucially used a characterization of the number of connected components in a forest being inversely related to its number of edges (Lemma 16, Part I). Secondly, we gave an efficient implementation by developing a data structure representation of the forest built by the algorithm. This data structure (stored in an Array) maintained the connected component each vertex belonged to throughout the course of the algorithm.

Let us give another example where implementation details can make a big difference in performance. The goal of Algorithm 1, a generic graph search algorithm, is to determine all vertices t that are *reachable* from a specified source vertex s, i.e., such that $s = t$ or there is a path from s to t in G.

We give a brief proof of correctness for Algorithm 1.

Lemma 1. *Let R be the output of Algorithm 1 on directed or undirected graph $G = (V, E)$ and $s \in V$. Then for all $v \in V$, we have $R[v] =$ **True** iff v is reachable from s.*

Proof. Call a vertex $v \in V$ marked if $R[v] =$ **True** at the end of the algorithm. We first prove that all marked vertices are reachable from s. Observe that each $R[v]$ is set to **True** on Line 11 at most once, because it is set to **True** only if it was previously **False** on Line 10. Therefore, we can prove our claim for all marked v by induction over the order where $R[v]$ was set to **True**.

The first time a vertex v has $R[v]$ set to **True**, v is the source vertex s, which is indeed reachable from s. For any other marked vertex u, u was at some point added to S in Line 13, as the neighbor of another marked vertex v. However, $R[v]$ was set to **True** by this point, so by induction, v is reachable from s. Finally, the existence of the edge (v, u) implies u is also reachable from s.

Finally, we prove that if u is reachable from s, then it is marked. We induct over the shortest path distance from s to u; this distance is at most $n - 1$, as shortest paths cannot include any cycles, else we could remove the cycle and obtain a shorter path. If this distance is 0, then $v = s$ and $R[s] =$ **True**. Otherwise, let v be the second-to-last vertex on the shortest path from s to u, so that the path includes (v, u). There is a shorter path from s to v, so by induction, we have

Algorithm 1: GraphSearch(G, s)

```
1  Input: G = (V, E), a graph, s ∈ V, a source vertex
2  S ← {s}
3  R ← Array.Init(n)                          // Reachable vertices from s.
4  for v ∈ V do
5  |  R[v] ← False
6  end
7  while S ≠ ∅ do
8  |  v ← element of S
9  |  S ← S \ v
10 |  if R[v] == False then
11 |  |  R[v] ← True                          // Mark v as reachable.
12 |  |  for (v, u) ∈ E do
13 |  |  |  S ← S ∪ {u}
14 |  |  end
15 |  end
16 end
17 return R
```

$R[v] =$ **True**. When $R[v]$ is marked **True** on Line 11, u is then immediately added to S on Line 13. Thus, u will later be removed from S, and $R[u]$ will be set to **True**, completing the induction. □

Notice that in proving Lemma 1, we made no assumptions about how Lines 8, 9, and 13 in the algorithm were implemented. In particular, even if the vertices in S can be arbitrarily ordered upon addition and removal, Lemma 1 remains true. In Section 2, we explore two implementation choices by means of using different data structures, and the consquences of these choices.

2 Search

In this section we consider details of implementing Algorithm 1, as well as their consequences on its performance guarantees. To begin, we claim that using a Queue or Stack to maintain the set S throughout the loop on Lines 7 to 16 results in an $O(m)$ runtime, where $m := |E|$.

First, recall that both a Stack and a Queue maintain a set, supporting both insertion and removal of an element from the set in $O(1)$ time. Moreover, we claim that at most $2m$ elements ever enter S throughout the algorithm. This is because each edge (v, u) causes a vertex u to be added on Line 13 exactly once, when v is marked **True** for the first time on Line 11. This causes one element addition per directed edge, and two element additions per undirected edge.

Thus the total runtime cost of the $O(m)$ runs of Lines 10 to 15 using a Stack or a Queue to support element insertion is $O(m)$. Similarly, the total cost of Lines 8 to 9 using a Stack or Queue is $O(m)$, since they can only happen as many times as there are elements added to S. In Sections 2.1 and 2.2, we respectively implement Algorithm 1 with a Queue and a Stack. This decision has significant implications, yielding the *breadth-first search* (BFS) and *depth-first search* (DFS) algorithms.

2.1 Breadth-first search

Breadth-first search is perhaps the most intuitive variant of graph search. We explore several applications that augment the basic variant of BFS which runs Algorithm 1 using a Queue.

Unweighted shortest paths. Beyond reachability, the most famous application of BFS is computing shortest paths in an unweighted graph. Concretely, suppose an (undirected or directed graph) $G = (V, E)$ is given as input, along with a source vertex s. For every vertex $t \in V$ that is reachable from s, we wish to determine the *shortest path distance* between s and t, i.e., the length of the shortest s-t path. If t is unreachable from s, we define the shortest path distance to be ∞.

Earlier, we proved in Lemma 1 that Algorithm 1 marks each vertex $v \in V$ at most once, and it marks precisely the reachable vertices. We claim that a stronger fact holds when Algorithm 1 is

implemented with a Queue, as summarized by the pseudocode in Algorithm 2.

Algorithm 2: BFS(G, s)

```
1  Input: G = (V, E), a graph, s ∈ V, a source vertex
2  S ← Queue.Init()
3  D ← Array.Init(n)                          // Shortest path distances from s.
4  for v ∈ V do
5  |  D[v] ← ∞
6  end
7  S.Enqueue((None, s))
8  while |S| > 0 do
9  |  (p, v) ← S.Dequeue()
10 |  if D[v] == ∞ then
11 |  |  if p ≠ None then
12 |  |  |  D[v] ← D[p] + 1                   // Mark shortest path distance of v.
13 |  |  end
14 |  |  else
15 |  |  |  D[v] ← 0                          // Only applies to the source s.
16 |  |  end
17 |  |  for (v, u) ∈ E do
18 |  |  |  S.Enqueue((v, u))
19 |  |  end
20 |  end
21 end
22 return D
```

Lemma 2. *Let D be the output of Algorithm 2 on directed or undirected graph $G = (V, E)$ and $s \in V$. Then for all $v \in V$, $D[v]$ is the length of the shortest path distance between s and v.*

Proof. We first briefly summarize the differences between Algorithms 1 and 2. Every time a vertex u would be added to S on Line 13 of Algorithm 1, we instead add the entire edge (v, u) (including the information of the parent vertex v) to the Queue in Algorithm 2. Otherwise, the only difference is that rather than maintaining the reachability status of vertices, we maintain an Array D which stores shortest path distances. We now argue that these shortest path distances are indeed correct.

If $v \in V$ is unreachable from s, $D[v]$ is never updated on Line 12, for the same reason that $R[v]$ was not updated in Line 11 as argued in Lemma 1. As $D[v] \leftarrow \infty$ initially, the final value is correct.

Otherwise, for all $i \in [n-1]$, let R_i be the set of vertices whose shortest path distance from s is exactly i. We claim that vertices are added to the queue for the first time in the following order: all of R_0 is added first, then all of R_1, and so on. We prove this by strong induction. The base case is that $R_0 = \{s\}$ is the first vertex added, which is true. Now suppose that vertices in $R_0, R_1, \ldots, R_i$ have been added to the queue in that order. Consider the vertices $u \in V$ enqueued with $v \in R_i$ as (v, u) on Line 18 after each $v \in R_i$ is dequeued for the first time. These vertices have a shortest path distance of at most $i + 1$, because they are reachable from $v \in R_i$. If u's shortest path distance was $\leq i$, it would have already been added by the inductive hypothesis. In the other case, $u \in R_{i+1}$ is indeed added to the queue for the first time, as claimed.

We can thus inductively prove that $D[u]$ is the shortest path distance in the order vertices are queued. The base case is $D[s] = 0$, since s is the first vertex queued. Our earlier argument shows that the first time any other $u \in R_{i+1}$ is queued, it is added as a pair (v, u) where $v \in R_i$. By induction we had correctly memoized $D[v] = i$, and thus we will correctly compute $D[u] = i+1$. □

Intuitively, Lemma 2 shows that BFS maintains a "frontier" of reachable vertices that it slowly grows. This frontier begins with s (the only vertex with shortest path distance 0), then adds all neighbors of s (at distance 1), then adds all their neighbors, and so on. Each frontier may include some vertices that we have already visited, but by performing the check in Line 10, we make sure that we only update distances the first time a vertex is visited (i.e., on the frontier).

Connected components. Another notable application of BFS is its use in computing the *connected components* of an undirected graph, defined in Section 4.2, Part I. Briefly, s and t belong to the same connected component if they are connected, i.e., there exists an s-t path. For undirected graphs, connectivity is an equivalence relation: it is reflexive, symmetric, and transitive. It is reflexive because every vertex is reachable from itself, it is symmetric because undirected s-t paths are also undirected t-s paths, and it is transitive because if t is reachable from s and u from t, then concatenating the paths shows that u is reachable from s. It is a standard fact that any equivalence relation partitions a set into *equivalence classes*. When connectivity is the relation, the set is V, the vertices of an undirected graph $G = (V, E)$, and the equivalence classes are G's connected components. Note that connectivity is not an equivalence relation in directed graphs, due to the lack of symmetry: we cannot reverse a directed path in general.

We have already argued BFS runs in $O(m)$ time, but we can say something stronger: it runs in $O(m_{C_s})$ time, where $C_s \subseteq V$ is the connected component of G that s belongs to, and m_{C_s} is the number of edges with both endpoints in C_s. This is because the only edges encountered in Algorithm 2 are between vertices in C_s, so we do not need to account for edges in other components.

This fact implies a simple algorithm for computing the connected components of G. First, take an arbitrary vertex $s \in V$ and run Algorithm 2 with this source vertex. Next, mark all vertices v that Algorithm 2 computes as being reachable from s as belonging to the connected component of s. We can then recurse with an arbitrary unmarked vertex.[1] The total runtime of this process is proportional to the sum of edge counts across all connected components, i.e., the total number of edges. Thus, we can compute all connected components in $O(m + n)$ time.

2.2 Depth-first search

Depth-first search is a graph search alternative implemented using a Stack rather than a Queue. This variant can be considered the more "aggressive" search algorithm: rather than waiting to complete each reachability frontier before moving onto the next, DFS searches through entire subgraphs at a time before backtracking. We will show that this makes several graph primitives we have discussed previously implementable, such as topologically sorting a DAG.

Unlike BFS, there is a natural way to write DFS recursively that performs identically to implementing Algorithm 1 with a Stack. However, as we will soon see, this recursive implementation of DFS has a major benefit: it lets us explicitly mark when the recursive DFS call at a given vertex has completed. We give both the iterative and recursive DFS variants here as Algorithms 3 and 4.

Algorithm 3: DFSIterative(G, s)

```
Input: G = (V, E), a graph, s ∈ V, a source vertex
S ← Stack.Init()
R ← Array.Init(n)                          // Reachable vertices from s.
for v ∈ V do
    R[v] ← False
end
while |S| > 0 do
    v ← S.Pop()
    if R[v] == False then
        R[v] ← True                        // Mark v as reachable.
        for (v, u) ∈ E do
            S.Push(u)
        end
    end
end
return R
```

We can check that Algorithms 3 and 4 (with the input R set to the all-**False** Array) have exactly

[1] Formally, we can index the vertices, start with $s \leftarrow 1$, and maintain a pointer to the smallest unmarked index to initialize the next call to Algorithm 2. The pointer makes one pass through the vertices, so this adds $O(n)$ time.

Algorithm 4: DFSRecursive(G, s, R)

```
1  Input: G = (V, E), a graph, s ∈ V, a source vertex, R, a length-n Array
2  if R[s] == False then
3  |   R[s] ← True                                  // Mark v as reachable.
4  |   for (s, u) ∈ E do
5  |   |   R ← DFSRecursive(G, u, R)                // Update reachable vertices upon recursing.
6  |   end
7  end
8  return R
```

the same performance. Rather than maintaining S iteratively, Algorithm 4 directly peels off each added vertex and calls DFSRecursive recursively on it, if it has not been explored previously.

Postordering. A key concept related to DFS is the *preordering* and *postordering* of vertices. Informally, DFS iteratively pushes vertices on top of a stack, so that the next vertex explored is the one most recently pushed to the top. We can also imagine that vertices leave the DFS stack when their subtree in the DFS search is fully explored. This happens when either a vertex has no unsearched neighbors, or its last remaining neighbor added to the stack has completed its search.

The preordering of vertices is simply the order in which vertices are added to the stack for the first time (i.e., sorting by starting times). Conversely, the postordering is the order in which vertices leave the stack (i.e., sorting by finish times). Because the postordering is more useful to our exposition, we only explain how to compute it in this section via Algorithm 5, a modification of Algorithm 4, but it is similarly straightforward to modify Algorithm 4 to compute a preordering.

Algorithm 5: Postorder(G, s, R, T, i)

```
1  Input: G = (V, E), a graph, s ∈ V, a source vertex, R, T, length-n Arrays, i ∈ ℕ
2  if R[s] == False then
3  |   R[s] ← True
4  |   for (s, u) ∈ E do
5  |   |   (R, T, i) ← DFSRecursive(G, u, R, T, i)
6  |   end
7  |   i ← i + 1                                    // Increment time counter.
8  |   T[s] ← i                                     // Record finish time of s.
9  end
10 return (R, T, i)
```

We now explain our modifications in Algorithm 5, compared to Algorithm 4. We include two additional parameters: T, which records the finish times of all vertices which have left the recursion stack, and i, a time counter. Whenever a vertex completes its recursive call (Lines 4 to 6), we increment the time counter i on Line 7 and record that the vertex has left the stack at the new time on Line 8. Thus, if Algorithm 5 is called with R set to the all-**False** Array, T set to an empty Array, and $i \leftarrow 1$ as inputs, it will return the finish times of all vertices in T.

It is straightforward to check that these augmentations do not change the runtime of DFS asymptotically; namely, Algorithm 5 still runs in time $O(m_{R_s} + n_{R_s})$, where n_{R_s} is the number of vertices reachable from s, and m_{R_s} is the number of edges that lie on a path from s. This is because we only record a finish time and increment the time counter at most once per explored vertex.

We also remark that a preordering of reachable vertices from s can be computed by marking the order in which Line 3 completes for each vertex, also at no runtime overhead to Algorithm 5.

We now record one important lemma about the postordering computed by Algorithm 5.

Lemma 3. *Let T be the output of Algorithm 5 on directed graph $G = (V, E)$ and $s \in V$. Then for $(u, v) \in E$ such that u, v are both reachable from s, if $T[u] < T[v]$, then there is a path from v to u.*

Proof. Consider the status of the recursion when $R[u] \leftarrow$ **True** is set on Line 3, which happens

on a recursive call of the form $\mathsf{Postorder}(G, u, R, T, i)$. There are three cases: the recursive call to $\mathsf{Postorder}(G, v, R, T, i)$ has not yet started, it has completed, or it is in the middle of executing.

If $\mathsf{Postorder}(G, v, R, T, i)$ has not yet started, then $R[v] =$ **False** at this time. Thus v will be added to the stack immediately after u is visited. Moreover, as a subroutine of u's recursive call to $\mathsf{Postorder}$, v will leave the stack before u leaves the stack. Thus, $T[u] > T[v]$ as v finishes first. This case hence does not apply, because the premise $T[u] < T[v]$ is false.

If $\mathsf{Postorder}(G, v, R, T, i)$ has completed, i.e., v has exited the stack, then again $T[u] > T[v]$, because the loop through u's neighbors has just begun executing and thus u has not finished. Thus we can also not be in this case given the premise of the lemma.

This leaves only the third case, where $\mathsf{Postorder}(G, v, R, T, i)$ has been called but has not completed. This means that u belongs to the recursive exploration process starting from v, as v has not yet exited the stack. Thus, whenever $T[u] < T[v]$, there is a path from v to u as claimed. □

Incidentally, the three types of edges in Lemma 3's proof are sometimes referred to as *forward edges*, *cross edges*, and *back edges* respectively, due to how they appear in the DFS traversal.

Topological sort. Recall that a directed graph is called a DAG if it has no cycles. We claimed in Section 5.2, Part III that the vertices of a directed graph $G = (V, E)$ can be topologically ordered, i.e., so that every edge $(i, j) \in E$ has $i < j$, iff G is a DAG. However, we only proved one direction at the time (that a topological ordering of G means that G is a DAG).

We use the following lemma with Lemma 3 to topologically sort a DAG in linear time.

Lemma 4. *Every DAG $G = (V, E)$ has at least one vertex with no incoming edges and one vertex with no outgoing edges.*

Proof. Suppose for contradiction that every vertex of G has an incoming edge. Continuing to follow incoming edges eventually causes a cycle, since there are only finitely many vertices. This contradicts G being a DAG. A similar contradiction occurs if all vertices have outgoing edges. □

Thus, let $s \in V$ be a vertex with no incoming edges, which we can find in $O(n)$ time. We claim that if G is a DAG, running Algorithm 5 and reversing the postordering T yields a topological order of all vertices reachable from s. To see this, suppose that $(u, v) \in E$ and u, v are both reachable from s. Then we claim u appears before v in the topological order. Suppose for contradiction this was not the case. Then $T[u] < T[v]$ (because the topological order reverses T), so Lemma 3 shows there is a path from v to u. This creates a cycle, contradicting that G is a DAG, so we are done.

If not all of V is reachable from s, then we can simply iterate on any unreached vertices. The total runtime of all calls to Algorithm 5 is $O(m + n)$, analogous to our argument in Section 2.1.

Finally, we remark that DFS can be used to check whether a given directed graph G is a DAG.[2] Indeed, we can simply attempt to run the topological ordering algorithm described earlier. If G is a DAG, then the topological ordering will be valid as we proved. Conversely, if the graph is not acyclic, then no topological ordering will be valid, by Lemma 2, Part III. We can thus verify whether $i < j$ for each $(i, j) \in E$ actually holds with respect to the computed ordering.

Strongly connected components. In Section 2.1, we described the connectivity structure of undirected graphs: a partition of the vertices into equivalence classes (connected components), such that each two vertices in the same component are connected, and there are no edges going between components. We now describe the connectivity structure of directed graphs.

The key issue with our previous notion of connectivity is that it is not an equivalence relation anymore for directed graphs, because it fails to satisfy symmetry. We circumvent this by simply defining a stronger equivalence notion, *strong connectivity*. We say that vertices s and t of directed graph G are *strongly connected* if t is reachable from s and s is reachable from t. We call all of the vertices that are strongly connected to a vertex s the *strongly connected component* (SCC) of s. As before, vertices of any directed graph can be partitioned into SCCs.

[2] The undirected variant of this problem amounts to checking whether a graph is a forest, which can be checked by computing all connected components as in Section 2.1, and counting their edges to ensure components are trees.

However, unlike undirected graphs, there could be edges between SCCs. It is thus helpful to introduce a new representation of a graph to visualize the overall structure. We can associate with each directed graph $G = (V, E)$ a "SCC graph" SCC(G) as follows. Let the SCCs of G be $\{C_i\}_{i \in [k]}$ for $k \geq 1$. Then, SCC(G) has k vertices, corresponding to the k SCCs of G. We add a directed edge from vertex i to vertex j in SCC(G) if there are *any* edges from C_i to C_j in G.

We claim SCC(G) is always a DAG. To see this, suppose there was a cycle between vertices $\{i_1, i_2, \ldots, i_j\}$ in SCC(G). Then there are edges between each consecutive pair of the SCCs $C_{i_1}, C_{i_2}, \ldots, C_{i_j}$. We claim this means that all of these SCCs are actually part of a single bigger SCC, a contradiction. This is because every vertex u in some SCC C_{i_a} can reach any other vertex v in a different SCC C_{i_b} by following cycle edges from any vertex in C_{i_a} (all of which are strongly connected to u) to any vertex in C_{i_b} (all of which are strongly connected to v).

Thus, every directed graph G is a DAG over SCCs. By Lemma 4, there exists a vertex in SCC(G) with no incoming edges. We call the SCC corresponding to this vertex a *source component*, and in particular no vertex in this SCC is reachable from any other SCC. Similarly, if a SCC has no outgoing edges to any other SCC, we call it a *sink component*. The set of vertices that is reachable from a vertex $s \in V$ has a concise description: it is all vertices in C_i, the SCC of s, as well as all vertices in every SCC C_j such that j is reachable from i in the graph SCC(G).

This gives a simple algorithm for computing SCCs. We can identify a vertex s in a sink component C_s, and run Algorithm 3 (or Algorithm 4) to find all reachable vertices from s, which is just C_s and nothing more. This takes time $O(m_{C_s})$ because no other edges outside C_s are used. We can then recurse on any vertex in a sink component in SCC(G) with the vertex corresponding to C_s removed, peeling off one SCC at a time. Assuming that we can repeatedly find a vertex in a sink component, the total time needed to run Algorithm 3 across all SCCs is $O(m)$.

How do we find a vertex in a sink component? The following algorithm, due to Kosaraju and Sharir [Sha81], works. Let rev(G) be G with all edges reversed, i.e., for every edge (i, j) in G, there is an edge (j, i) in rev(G). Then repeatedly run Algorithm 5 on vertices in rev(G) until all vertices are discovered. This forms a total postordering T of all vertices based on when their call to Algorithm 5 returns. We claim v, the last vertex in T, is in a sink component of G.

Why is this? An equivalent claim is that v lies in a source component of rev(G). Because v is the last vertex to have its call to Algorithm 5 complete, the last "top-layer" call to Algorithm 5 is to v. No call on a vertex in any other SCC can reach a source component, so the last top-layer call must be on a vertex in a source component. More generally, the last vertex in any postordering always lies in a source component by a similar argument, even after deleting some SCCs.

Now we can describe the Kosaraju-Sharir algorithm in full. First repeatedly run Algorithm 5 on rev(G) to form T, a postordering of all vertices. Then repeatedly remove the last unvisited vertex according to T, and call DFS on it, marking all newly visited vertices. By our earlier argument, every time we repeat this process, we discover a sink component of G and explore no other SCCs. The overall algorithm for finding all SCCs takes $O(m + n)$ time.

3 Shortest paths

In Section 2, we focused on search algorithms and their applications. One application in Section 2.1 was computing single-source shortest paths (SSSP) on unweighted graphs in $O(m + n)$ time. In fact, we have already seen several other shortest path algorithms in Part III of the notes. There, we gave an SSSP algorithm for (possibly weighted) DAGs, also requiring $O(m + n)$ time.

We also gave several algorithms for the APSP (all-pairs shortest paths) problem in Part III, concluding with the Floyd-Warshall algorithm, which has an $O(n^3)$ runtime. In fact, Floyd-Warshall is essentially still the best-known APSP algorithm, and as we will discuss in Part VIII, this is believed to be unimprovable by polynomial factors, under a popular conjecture.[3]

In this section, we return to the SSSP problem and develop algorithms for it in full generality, i.e., on graphs that may be weighted and contain cycles. We begin by discussing graphs with only positive weight edges in Section 3.1, and handle the most general case in Section 3.2.

[3]However, mild gains are possible: a breakthrough by [Wil21] gave a $2^{\sqrt{\log(n)}} = n^{o(1)}$ factor improvement.

3.1 Dijkstra

In this section, let $G = (V, E, \mathbf{w})$, such that all edge weights $\mathbf{w}$ are positive, i.e., $\mathbf{w} \in \mathbb{R}^E_{>0}$.[4] We give an algorithm that solves SSSP on G: given a source vertex $s \in V$, compute the shortest path distance from s to every $t \in V$. As in Section 5, Part III, we define the shortest path distance by

$$d(s,t) := \min_{\substack{P \text{ is a path from } s \to t \\ P \subseteq E}} \sum_{e \in P} \mathbf{w}_e. \tag{1}$$

If t is not reachable from s, we define $d(s,t) = \infty$. Because we can always run, e.g., Algorithm 3 to solve reachability first and label unreachable vertices with a distance of ∞, let us assume for simplicity in this discussion that all vertices in V are reachable from s.

We present Dijkstra's algorithm for solving SSSP on graphs with positive edge weights, which combines two main ideas: *relaxing tense edges*, and *priority queues*. Let us briefly explain each.

The first idea, relaxing tense edges, assumes that we maintain labels $D[t]$ for every vertex $t \in V$, such that $D[t]$ is an overestimate of $d(s,t)$. An easy way to guarantee this is to always maintain that $D[t]$ is the total weight of some s-t path, so that it is always larger than the shortest s-t path weight, i.e., $d(s,t)$. Now consider some edge (u,t) with t as the tail. We say that the edge is *tense* if the following condition holds: $D[t] > D[u] + \mathbf{w}_{(u,t)}$. If this is the case, we claim that $D[t]$ is "obviously" improvable: by the assumption that $D[u] \geq d(s,u)$,

$$d(s,t) \leq d(s,u) + \mathbf{w}_{(u,t)} \leq D[u] + \mathbf{w}_{(u,t)}, \tag{2}$$

because appending the edge (u,t) to the shortest s-u path gives a valid s-t path, and $d(s,t)$ can only be smaller. Thus, we can improve our overestimate $D[t]$ by performing the following update:

$$D[t] \leftarrow \min(D[t], D[u] + \mathbf{w}_{(u,t)}). \tag{3}$$

The update (3) is called *relaxing* the edge (u,t), and is valid to perform with any overestimates D. In fact, all of our SSSP algorithms simply repeatedly call (3) on specific edges.[5]

The other main idea in Dijkstra's algorithm is to use a priority queue to implement a variant of GraphSearch (Algorithm 1). A priority queue is a data structure that stores a set S of objects from a universe Ω, with associated *values* in $\mathbb{R}$. We only require that a priority queue has three operations: Insert, Delete, and ExtractMin. The first two operations work as you would expect: they add or remove a specified object from the set (we will only call Delete on elements already present). The third operation specifically deletes the lowest-value object from the set.

There are various priority queue implementations with different tradeoffs. We use a Heap, which implements all of Insert, Delete, and ExtractMin in $O(\log(n))$ time, as reviewed in Section 7.2, Part I of the notes. We use the following notation, slightly different than Section 7.2, Part I: Insert(x, val) inserts $x \in \Omega$ with value val $\in \mathbb{R}$, Delete(x) removes the object x (and its value) from the Heap, and ExtractMin deletes the minimum-value object. It is straightforward to modify the implementation in Section 7.2, Part I to provide this functionality, e.g., by storing addresses of all vertices explicitly and only manipulating values. We also denote the value of an object x by x.val, which we can query in $O(1)$ time for a given x. We can now describe Dijkstra's algorithm in Algorithm 6.

Algorithm 6 is very similar in spirit to Algorithm 2; in fact, it is arguably simpler. One main difference is that we do not explicitly need to write down our distance labels in D until a vertex u has already exited the Heap, at which point its current value u.val is permanently recorded as its distance from s in Line 10. Until that point, we keep track of distance labels directly using the Heap, i.e., the distance label of u is u.val. Otherwise, the main distinctions between Algorithms 6 and 2 are that Algorithm 6 starts by adding all vertices to its maintained set, that it uses a Heap rather than a Queue to remove vertices, and that it handles weighted edges.

We claim that Algorithm 6 runs in $O((m+n)\log(n))$ time. This is because we only extract vertices from the Heap and never add them back, so Line 9 only runs n times, requiring $O(n\log(n))$ time in total. Similarly, a loop of Lines 11 to 16 only happens once for each edge (u,t), because it runs when u is extracted from the Heap. Thus, all loops take $O(m\log(n))$ time in total. We remark

[4] Our algorithm extends easily to zero-weight edges too, by just removing them before running the algorithm.

[5] This is even the case for the unweighted and DAG SSSP algorithms we have already seen.

Algorithm 6: SSSPPositive(G, s)

```
1  Input: G = (V, E, w), a graph with w ∈ R^E_{>0}, s ∈ V, a source vertex that can reach all of V
2  S ← Heap.Init({}, n)
3  D ← Array.Init(n)
4  for v ∈ V \ {s} do
5  |  S.Insert(v, ∞)
6  end
7  S.Insert(s, 0)
8  while |S| > 0 do
9  |  u ← S.ExtractMin()
10 |  D[u] ← u.val
11 |  for (u, t) ∈ E do
   |  |     // Equivalently, this loop relaxes the edge (u, t), i.e., it updates t.val ← min(t.val, u.val + w_(u,t)).
12 |  |  if u.val + w_(u,t) < t.val then
13 |  |  |  S.Delete(t)
14 |  |  |  S.Insert(t, u.val + w_(u,t))
15 |  |  end
16 |  end
17 end
18 return D
```

that for graphs where all vertices are reachable from the source, $O((m+n)\log(n)) = O(m\log(n))$, because $m \geq n-1$ (the undirectification of the graph must at least be connected).

It remains to prove correctness. To do so, we use the notion of relaxing tense edges (3).

Lemma 5. *Let D be the output of Algorithm 6 on directed graph $G = (V, E)$ and $s \in V$. Then for all $t \in V$, $D[t] = d(s, t)$ is the shortest path distance between s and t, as defined in (1).*

Proof. We proceed by induction on the number of vertices that have been extracted from the heap. The first time Line 9 is called, the only non-infinity value is $s.\text{val} = 0$, so s is extracted and $D[s]$ is updated to 0. Thus the base case is correct because $d(s, s) = 0$, as all weights are positive.

Next, suppose we have extracted k vertices so far, denoted $V_k := \{v_1 = s, v_2, \ldots, v_k\}$ (so that v_i is the i^{th} vertex extracted), and that all of their distance labels are correct, i.e., $D[v_i] = d(s, v_i)$ for all $i \in [k]$. We want to claim that v_{k+1}, the Heap vertex with the lowest current value, has $v_{k+1}.\text{val} = d(s, v_{k+1})$, as this would complete the induction. For simplicity, let $t := v_{k+1}$.

At the time when all of V_k has been extracted, the value of every vertex v in the Heap is

$$v.\text{val} = \min_{\substack{i \in [k] \\ (v_i, v) \in E}} \left\{ d(s, v_i) + \mathbf{w}_{(v_i, v)} \right\}. \tag{4}$$

Here the above value is defined as ∞ if there are no $(v_i, v) \in E$ for $i \in [k]$. The formula (4) is true by construction: if there are no incoming edges from V_k, then $v.\text{val}$ is never updated on (14), and otherwise, we repeatedly add $v_i.\text{val} + \mathbf{w}_{(v_i,v)}$ to the running minimum comparison on Line 14, each time a v_i is extracted that has an edge to v. By our inductive hypothesis, each $v_i.\text{val} = d(s, v_i)$.

Finally we can complete the proof. Consider some s-t path, and suppose for contradiction that it has strictly less total weight than $t.\text{val}$ as given in the formula (4). This path must cross from the set V_k to the set $V \setminus V_k$ at some point, because $s \in V_k$ and $t \in V \setminus V_k$. Let us suppose the first time this happens, it crosses from v_i to $u \in V \setminus V_k$ by taking the edge (v_i, u). Then the path must go from s to v_i, from v_i to u using an edge, and then from u to t. Thus the total length of the path is

$$\geq d(s, v_i) + \mathbf{w}_{(v_i,u)} + d(u, t) \geq u.\text{val} + d(u, t) \geq t.\text{val},$$

a contradiction. The second inequality used (4) to upper bound $u.\text{val}$, and the third inequality used $d(u, t) \geq 0$ and that $t.\text{val}$ is minimal among all $v \in V \setminus V_k$, because t was extracted. We remark that the third inequality is the only place that positivity of edge weights is used.

This shows that no s-t path can have smaller total weight than t.val, so $d(s,t) \geq t$.val. However, because we obtained t.val by relaxing edges, our earlier argument (2) shows t.val is an overestimate of $d(s,t)$ throughout the algorithm, so $d(s,t) \leq t$.val. In conclusion, we have established $d(s,t) = t$.val, i.e., the $(k+1)^{\text{th}}$ recorded distance is correct, and the induction is complete. □

In conclusion, Dijkstra's algorithm runs in $O((m+n)\log(n))$ time and solves the SSSP problem on graphs with positive edge weights. This runtime turns out to be improvable by using other data structures to implement the priority queue needed by Algorithm 6. In particular, the *Fibonacci heap* developed in [FT87] supports an extra operation called DecreaseKey, that runs in $O(1)$ time and can implement the entire loop of Lines 11 to 16. By using a Fibonacci heap to implement Dijkstra's algorithm, we can obtain an improved runtime of $O(m + n\log(n))$.

3.2 Bellman-Ford

In this section, we finally tackle the SSSP problem in its full generality. As we discussed in Section 5.3, Part III, there is one caveat: we must assume that the input graph $G = (V, E, \mathbf{w})$ has no *negative-weight cycles*, i.e., a directed cycle $C \subseteq E$ such that $\sum_{e \in C} \mathbf{w}_e < 0$. At the end of the section, we will show how to use our algorithm to verify that this assumption actually holds.

Let $G = (V, E, \mathbf{w})$ have no negative-weight cycles, and let $s \in V$ be a source vertex. As usual, we assume we have already ran a reachability algorithm, e.g., Algorithm 3, to remove all vertices not reachable from the source vertex s. Thus, assume for simplicity that all of V is reachable from s. We claim that the following simple-to-describe algorithm, Algorithm 7, due to Bellman and Ford (and in fact earlier proposed by Shimbel) [Shi55, For56, Bel58], optimally solves SSSP on G. In short, Algorithm 7 simply relaxes every edge $n-1$ times sequentially.

Algorithm 7: SSSP(G, s)

```
1  Input: G = (V, E, w), a graph with no negative-weight cycles, s ∈ V, a source vertex that
   can reach all of V
2  D ← Array.Init(n)
3  for v ∈ V \ {s} do
4  |  D[v] ← ∞
5  end
6  D[s] ← 0
7  for ℓ ∈ [n − 1] do
8  |  for (u, v) ∈ E do
9  |  |  D[v] ← min(D[v], D[u] + w_(u,v))
10 |  end
11 end
12 return D
```

It is clear that Algorithm 7 runs in $O(mn)$ time, because each of the $n-1$ loops of Lines 7 to 11 requires $O(m)$ time. We now give a short proof of correctness. Consider making a two-dimensional array $S[v][\ell]$, indexed by $v \in V$ and $\ell \in [n-1]$, such that $S[v][\ell]$ equals $D[v]$ at the end of the ℓ^{th} loop of Lines 7 to 11. Then the update in Lines 8 to 10 can be equivalently rewritten as

$$S[v][\ell] = \min\left(S[v][\ell-1], \min_{(u,v)\in E} S[u][\ell-1] + \mathbf{w}_{(u,v)}\right). \tag{5}$$

We claim $S[v][\ell]$ is the shortest path distance between s and v, restricted to paths of length $\leq \ell$. We essentially gave this argument already in Section 5.3, Part III, but we briefly reproduce it here. The shortest s-v path of length $\leq \ell$ either has length $\leq \ell - 1$, or exactly k. In the latter case, if we let u be the second-to-last vertex on the shortest path, then the path concatenates the edge (u, v) to the shortest s-u path of length $\leq \ell - 1$. Both cases are handled correctly by (5).

Finally, when Algorithm 7 terminates, it returns all of the values $S[v][n-1]$. If G has no negative-weight cycles, this is indeed the shortest path distance between s and v for all $v \in V$, because we proved in Lemma 3, Part III that there always exists a shortest path with $\leq n-1$ edges.

One useful corollary of Algorithm 7's correctness in the absence of negative-weight cycles is that it can be used to detect the presence of negative-weight cycles, as explained in the following.

Lemma 6. *Let D be the output of Algorithm 7 on directed graph $G = (V, E, \mathbf{w})$ (which may have negative-weight cycles), and $s \in V$ that can reach all of V. Then if any $(u, v) \in E$ has*

$$D[v] > D[u] + \mathbf{w}_{(u,v)}, \tag{6}$$

G has a negative-weight cycle, and otherwise, G has no negative-weight cycle.

Proof. We begin with the easier direction. Suppose G has no negative-weight cycle. Then we claim (6) cannot happen at termination. This is because $D[v]$ remains an overestimate of $d(s, v)$ throughout the algorithm, an invariant maintained by (2). Moreover, it is exactly equal to $d(s, v)$ at the end of the algorithm, as we argued earlier. However, (6) would mean that $D[v]$ gets updated to a strictly smaller value, contradicting that it stays an overestimate of $d(s, v)$, i.e., (2).

Now suppose that G has a negative-weight cycle, C, which involves the vertices $\{v_1, \ldots, v_k\}$ in that order. We claim that (6) must occur for some edge. Indeed, suppose (6) was false for every edge, so in particular, it is false for all the cycle edges (v_{i-1}, v_i) for $i \in [k]$, where we let $v_0 \equiv v_k$ for notational convenience. Then summing $D[v_i] \leq D[v_{i-1}] + \mathbf{w}_{(v_{i-1},v_i)}$ for all $i \in [k]$ yields

$$\sum_{i\in[k]} D[v_i] \leq \sum_{i\in[k]} D[v_{i-1}] + \sum_{i\in[k]} \mathbf{w}_{(v_{i-1},v_i)} \implies 0 \leq \sum_{i\in[k]} \mathbf{w}_{(v_{i-1},v_i)}.$$

Here we used that the vertices showing up on both sides of the inequality are exactly the same, albeit in a different order, so we can cancel their summations. Finally, the above inequality contradicts that C is a negative-weight cycle, so (6) must have been true for some edge in C as claimed. □

We thus have obtained an $O(mn)$-time algorithm to detect the presence of negative-weight cycles, building off Algorithm 7. In particular, we can simply check (6) for every edge using the output. This gives us an overhead of $O(m)$ in the runtime, which does not dominate.

We remark that, as of very recently, Bellman-Ford is no longer the state-of-the-art algorithm for SSSP. For graphs with polynomially-bounded edge weights (i.e., integer weights in the range $[-\text{poly}(n), \text{poly}(n)]$), recent breakthroughs by [BNW22, BCF23] gave an $O(m \log^3(n) \log\log(n))$-time algorithm for SSSP. In the real RAM model where weights are allowed to be arbitrary, another breakthrough improved upon Bellman-Ford for the first time in over 60 years: [Fin24] solves SSSP in $\approx O(mn^{\frac{8}{9}})$ time up to logarithmic factors, now improved to $\approx O(mn^{\frac{4}{5}})$ time [HJQ24].

4 Flows and cuts

In this section, we introduce the *maximum flow* (maxflow) and *minimum cut* (mincut) problems. Algorithms for these problems are some of the most powerful tools on graphs, and have many downstream applications due to the variety of problems that can be reformulated as maxflow or mincut instances. Moreover, the close relationship between these two problems serves as an excellent introduction to concepts in continuous algorithms, the subject of our next unit.

4.1 Definitions

We begin by defining the maximum flow and minimum cut problems.

Flows. Let $G = (V, E, \mathbf{c})$ be a directed graph, such that every edge $e \in E$ has a positive *capacity* $\mathbf{c}_e > 0$. We say $\mathbf{f} \in \mathbb{R}^E_{\geq 0}$ is a *flow* if it assigns nonnegative values to each edge; the flow on $e = (u, v) \in E$ intuitively represents an amount of material to be sent from vertex u to v.

We say that a flow $\mathbf{f} \in \mathbb{R}^E_{\geq 0}$ is *feasible* if it respects the capacity constraints, i.e., for all $e \in E$, we have $0 \leq \mathbf{f}_e \leq \mathbf{c}_e$. We denote the total amount of flow that a vertex $v \in V$ produces by

$$\partial\mathbf{f}(v) := \sum_{(v,u)\in E} \mathbf{f}_{(v,u)} - \sum_{(u,v)\in E} \mathbf{f}_{(u,v)}.$$

That is, $\partial \mathbf{f}(v)$ sums up all flows that v produces, and subtracts all flows consumed by v. We call $\partial \mathbf{f}(v)$ the *net flow* at v. One simple observation is that the sum of all net flows always vanishes:

$$\begin{aligned}\sum_{v \in V} \partial \mathbf{f}(v) &= \sum_{v \in V} \sum_{(v,u) \in E} \mathbf{f}_{(v,u)} - \sum_{v \in V} \sum_{(u,v) \in V} \mathbf{f}_{(u,v)} \\ &= \sum_{e \in E} \mathbf{f}_e - \sum_{e \in E} \mathbf{f}_e = 0.\end{aligned} \tag{7}$$

Here we used that every edge $e \in E$ enters and leaves exactly one vertex each.

Fix two vertices $s, t \in V$ of interest, where s is a *source vertex* and $t \neq s$ is a *sink vertex*. We say that a flow $\mathbf{f}$ is an s-t flow if $\partial \mathbf{f}(v) = 0$ for all $v \in V \setminus \{s, t\}$. In other words, in an s-t flow, the net flow at every vertex except the source s and sink t is 0. In light of (7), we additionally have that $\partial \mathbf{f}(s) = -\partial \mathbf{f}(t)$. Thus, we can think of an s-t flow as transporting a $\partial \mathbf{f}(s) = -\partial \mathbf{f}(t)$ total amount of material from s to t, while every other vertex produces (and receives) no net material.

In the *s-t maximum flow* problem, or s-t maxflow for short, we are given as input $G = (V, E, \mathbf{c})$, and $s, t \in V$ with $s \neq t$. Our goal is to compute the largest $\partial \mathbf{f}(s)$ achievable by a feasible s-t flow:

$$\max_{\mathbf{f} \in \mathbb{R}^E_{\geq 0}} \partial \mathbf{f}(s) \text{ subject to } \mathbf{f}_e \leq \mathbf{c}_e \text{ for all } e \in E, \text{ and } \partial \mathbf{f}(v) = 0 \text{ for all } v \in V \setminus \{s, t\}. \tag{8}$$

Throughout this section, when discussing s-t maxflows or s-t mincuts (which we will define shortly), we always assume that t is reachable from s, or else the problem is not well-defined.

Cuts. Again, let $G = (V, E, \mathbf{c})$ be a directed graph with specified edge capacities $\mathbf{c} \in \mathbb{R}^E_{\geq 0}$. For any subset of vertices $S \subseteq V$, we let $\text{cut}(S)$ denote the total amount of edge capacity crossing from S over to its complement, $V \setminus S$. That is, we define

$$\text{cut}(S) := \sum_{\substack{(u,v) \in E \\ u \in S, v \notin S}} \mathbf{c}_{(u,v)}. \tag{9}$$

If $S = \emptyset$ or $S = V$, we let (9) (which is the empty sum) evaluate to 0.

In the *s-t minimum cut* problem, or s-t mincut for short, we are again given $G = (V, E, \mathbf{c})$ and $s, t \in V$ with $s \neq t$. Our goal is to compute the minimum possible value of $\text{cut}(S)$ for an S containing s but not t, i.e., following the definition (9),

$$\min_{S \subseteq V} \text{cut}(S) \text{ subject to } s \in S,\ t \notin S. \tag{10}$$

Intuitively, (10) asks for the cheapest way to cut edges (with costs $\mathbf{c}$) that separates s from t.

Historical note. The s-t maximum flow problem was first formulated by [HR54] in a classified report, where they used a 44-vertex directed graph to model a Soviet road network during the Cold War. Each edge on the network had an associated capacity, representing the rate at which material could travel across a road. This report was only declassified in 1999 at the request of Alexander Schrijver, and is discussed in a broader historical context by the subsequent article [Sch05].

In fact, Harris and Ross [HR54] were interested in the maximum flow problem for darker reasons: their aim was to determine the minimum amount of damage that needed to be done to the network in order to disconnect it. This is a mincut problem. As we will see in the following Section 4.2, s-t maxflow and s-t mincut are essentially the same problem, a fact which Harris and Ross exploited to devise an algorithm to solve their mincut problem on their instance.

4.2 Maxflow-mincut theorems

The punchline of this section is that for any graph $G = (V, E, \mathbf{c})$, and any vertices $s, t \in V$ with $s \neq t$, (8) and (10) are equal. This fact, Theorem 1, is the (strong) maxflow-mincut theorem.

As a warmup, we prove the weak maxflow-mincut theorem: that the s-t maxflow is at most the s-t mincut. This has a clear intuition: let S be the set witnessing (10). If you want to send some amount F of material from s to t, it has to at least cross over from S to $V \setminus S$. The total amount of capacity available for material to cross over is exactly $\text{cut}(S)$, so $F \leq \text{cut}(S)$.

More formally, we have that for any feasible s-t flow $\mathbf{f}$,

$$\begin{aligned}\partial\mathbf{f}(s) &= \sum_{u\in S}\partial\mathbf{f}(u) = \sum_{u\in S}\sum_{(u,v)\in E}\mathbf{f}_{(u,v)} - \sum_{u\in S}\sum_{(v,u)\in E}\mathbf{f}_{(v,u)} \\ &= \sum_{\substack{(u,v)\in E\\ u\in S, v\notin S}}\mathbf{f}_{(u,v)} - \sum_{\substack{(v,u)\in E\\ u\in S, v\notin S}}\mathbf{f}_{(v,u)} \leq \sum_{\substack{(u,v)\in E\\ u\in S, v\notin S}}\mathbf{c}_{(u,v)} = \text{cut}(S).\end{aligned} \tag{11}$$

The second line used that any edge with both endpoints in S shows up twice and thus cancels, by an argument similar to (7). The only inequality used the constraints $0 \leq \mathbf{f}_e \leq \mathbf{c}_e$ for all $e \in E$. By applying the above to the $\mathbf{f}$ achieving the s-t maxflow, we have the weak maxflow-mincut theorem.

What is remarkable is that every inequality used in the above proof must be tight for the strong maxflow-mincut theorem to be true. That is, it must be the case that for the $\mathbf{f}$ achieving the maxflow, and some set $S \subseteq V$, both of the following criteria are true.

- For all $(u, v) \in E$ with $u \in S$ and $v \notin S$, the edge is saturated with flow: $\mathbf{f}_{(u,v)} = \mathbf{c}_{(u,v)}$.
- For all $(v, u) \in E$ with $u \in S$ and $v \notin S$, the edge has no flow: $\mathbf{f}_{(v,u)} = 0$.

Indeed, we will produce an $\mathbf{f}$ and S with these properties in the proof of the strong maxflow-mincut theorem, Theorem 1. We do so by giving a precise characterization of the relationship between the s-t maxflow and the s-t mincut, through the language of *residual graphs*.

In particular, let $\mathbf{f}$ be a feasible flow in $G = (V, E, \mathbf{c})$. We define the residual graph $G_\mathbf{f}$ as follows.

- For every edge $e = (u, v) \in E$ with $0 < \mathbf{f}_e < \mathbf{c}_e$, we add two edges (u, v) and (v, u) in $G_\mathbf{f}$, with capacities $\mathbf{c}_e - \mathbf{f}_e$ and $\mathbf{f}_e$ respectively. Both of these values lie in the range $(0, \mathbf{c}_e)$.
- For every edge $e = (u, v) \in E$ with $\mathbf{f}_e = 0$, we only include an edge (u, v) with capacity $\mathbf{c}_e$.
- For every edge $e = (u, v) \in E$ with $\mathbf{f}_e = \mathbf{c}_e$, we only include an edge (v, u) with capacity $\mathbf{c}_e$.

Intuitively, the residual graph represents the fact that we can continue to send $\mathbf{c}_e - \mathbf{f}_e$ units of flow forward, or $\mathbf{f}_e$ units of flow backward, along each edge $e \in E$ to remain feasible. It thus represents the sorts of flows we can add to a current flow $\mathbf{f}$ to potentially improve its flow value.

Theorem 1. *For any $G = (V, E, \mathbf{c})$, and $s, t \in V$ with $s \neq t$, (8) and (10) have the same value.*

Proof. We already know from (11) that the s-t maxflow is at most the s-t mincut. It thus suffices to provide a feasible flow $\mathbf{f}$ and a set $S \subseteq V$ with $s \in S$, $t \notin S$, such that

$$\partial\mathbf{f}(s) = \text{cut}(S). \tag{12}$$

This implies the s-t maxflow is at least the s-t mincut, which concludes the proof.

We now show (12) holds when $\mathbf{f}$ achieves the s-t maxflow. We claim that t is not reachable from s in $G_\mathbf{f}$. If this is true, let S be the component reachable from s, so that $t \notin S$.

For every edge (u, v) with $u \in S$, $v \notin S$, i.e., from S to its complement, it must be the case that the forward edge (u, v) does not exist in the residual graph $G_\mathbf{f}$. By our earlier definition, this can only happen if $\mathbf{f}_{(u,v)} = \mathbf{c}_{(u,v)}$, as in the other two cases the edge (u, v) exists with a positive capacity.

We follow a similar argument for every edge (v, u) with $u \in S$, $v \notin S$. i.e., from the complement of S to S. It must be that these edges have zero flow, $\mathbf{f}_{(v,u)} = 0$, or else the edge (u, v) would exist in $G_\mathbf{f}$, which connects S to its complement, a contradiction to how we defined S.

We now have a flow $\mathbf{f}$ such that the two criteria we outlined after (11) both hold. Thus every inequality in (11) is tight. We thus have a feasible $\mathbf{f}$ and S that separates s from t, such that $\partial\mathbf{f}(s) = \text{cut}(S)$. This satisfies the criteria for (12) to hold, which was our goal.

It remains to establish our claim that t is not reachable from s in the residual graph $G_\mathbf{f}$. Suppose otherwise for contradiction. Let P be an s-t path in $G_\mathbf{f}$, and let $w > 0$ be the *width* of P, i.e., the minimum capacity along the path. Intuitively, we can think of w as capturing the bottleneck capacity of the path, because w is the most flow we can send along P without becoming infeasible.

It turns out that merely the fact that $w > 0$ is enough. Consider sending w additional units of flow from s to t along the path P. Formally, this modifies our maxflow $\mathbf{f}$ edgewise to a new flow

$\mathbf{f}'$ as follows: for each forward edge $(u,v) \in P$ such that $(u,v) \in E$, we let $\mathbf{f}'_{(u,v)} \leftarrow \mathbf{f}_{(u,v)} + w$, and for each backward edge $(v,u) \in P$ such that $(u,v) \in E$, we let $\mathbf{f}'_{(u,v)} \leftarrow \mathbf{f}_{(u,v)} - w$.

By construction, $\mathbf{f}'$ is still a feasible s-t flow, and it has larger flow value: $\partial\mathbf{f}'(s) = \partial\mathbf{f}(s) + w > \partial\mathbf{f}(s)$. This contradicts that $\mathbf{f}$ achieves the s-t maxflow. Hence, t is not reachable from s as claimed. □

4.3 Maxflow algorithms

One amazing property of our proof of Theorem 1 is that it proceeds by running an algorithm. In particular, define P to be an *s-t augmenting path* if it is an s-t path in a residual graph $G_{\mathbf{f}}$. By examining the proof of Theorem 1, we can conclude that it shows the following.

Corollary 1. *Let $\mathbf{f}$ be a feasible flow in $G = (V, E, \mathbf{c})$, and let P be an s-t augmenting path in $G_{\mathbf{f}}$ with width $w > 0$. Then, we can construct a feasible flow $\mathbf{f}'$ with $\partial\mathbf{f}'(s) = \partial\mathbf{f}(s) + w$ in $O(m)$ time.*

This gives rise to the following generic Algorithm 8 for solving s-t maxflow. We focus on maxflow algorithms in this section because given an s-t maxflow $\mathbf{f}$, it is simple to recover an s-t mincut S: by Theorem 1, we should let S be the component of the residual graph $G_{\mathbf{f}}$ reachable from s.

Algorithm 8: Maxflow(G, s, t)

```
1  Input: G = (V, E), a graph, s ∈ V, a source vertex, t ∈ V, a sink vertex
2  f ← all-zeroes vector in ℝ^E
3  while t is reachable from s in G_f do
4  |   P ← s-t path in G_f
5  |   w ← width of P
6  |   for (u, v) ∈ P do
7  |   |   if (u, v) ∈ E then
8  |   |   |   f_(u,v) ← f_(u,v) + w
9  |   |   end
10 |   |   else
11 |   |   |   f_(v,u) ← f_(v,u) − w
12 |   |   end
13 |   end
14 end
15 return f
```

Just as BFS and DFS are applications of our generic graph search algorithm that repeatedly explores an unexplored vertex, and Dijkstra and Bellman-Ford are applications of our generic SSSP algorithm that repeatedly relaxes a tense edge, Algorithm 8 presents a generic way of solving maxflow problems. The only degree of freedom is in how to choose the augmenting path P on Line 4. We present several different algorithms in this section that make different choices of P.

For simplicity, in the remainder of this section we assume that we have already eliminated any vertices in G not reachable from s, as they will not be used in any s-t path. These vertices can be found via a graph search algorithm. The remaining graph has at least $m \geq n - 1 = \Omega(n)$ edges. We use this assumption to simplify expressions, e.g., we have $O(m + n) = O(m)$.

Any path. The original Ford-Fulkerson algorithm [For56, FF56] simply chooses P in each iteration of Line 4 to be any augmenting path. This algorithm has a provable convergence rate when all of the edge capacities $\mathbf{c}$ are positive integers; thus, let us make the assumption for now that $\mathbf{c} \in \mathbb{N}^E$. We also denote the s-t maxflow value by $F^\star$. Clearly $F^\star$ is finite, as it equals the s-t mincut value, which is bounded by the sum of edge capacities $\sum_{e \in E} \mathbf{c}_e$ (and can be much smaller).

We first claim that $F^\star \in \mathbb{N}$. Consider running Algorithm 8 with any choice of P on Line 4. As long as we preserve the invariant that all flow values $\mathbf{f}_e$ are integers, all edge capacities in $G_{\mathbf{f}}$ also remain integers, so the width w will be a positive integer. Thus, inductively $\mathbf{f} \in \mathbb{Z}^E$ always holds. Each augmenting path adds at least $w \geq 1$ to the flow value, and hence we must terminate in $F^\star$ iterations. At termination, $\mathbf{f}$ achieves the s-t maxflow value and hence $F^\star = \partial\mathbf{f}(s) \in \mathbb{N}$.

The same argument bounds the number of iterations of Algorithm 8 by $F^\star$, so the runtime of Algorithm 8 is $F^\star$ times the cost of running Lines 3 to 14. It is straightforward to implement these

lines in $O(m+n) = O(m)$ time, by running BFS or DFS in Line 4 and then manually passing over the $\leq m$ edges of the path P. Thus the overall algorithm runs in time $O(mF^\star)$.

How satisfied should we be with this runtime? In some sense, it is not truly a polynomial in the input length. If the edge capacities are specified as, say, b-bit integers, each capacity is upper bounded by $U = 2^b$, so a naïve bound on the s-t maxflow value is $F^\star \leq mU$. Thus in this case the Ford-Fulkerson algorithm runs in $O(m^2U)$. However, the input length is $\approx mb = m\log_2(U)$, so if m is small and U is large, the runtime is actually exponential in the input length. In practice, $F^\star$ could be $\ll mU$, and hence can be treated as a parameter to the problem (similar to how the target value V was a parameter in our subset sum runtime, in Section 3.3, Part III). For this reason, we say that the Ford-Fulkerson algorithm runs in *pseudopolynomial* time.

Widest path. Can we improve upon this bound for the runtime of Ford-Fulkerson, by choosing smarter paths in Line 4? One strategy is to set P to the widest s-t path in $G_{\mathbf{f}}$. The intuition is that we want to add the most flow value w in each iteration of Algorithm 8. In Homework III, we show how to compute widest s-t paths in $O((m+n)\log(n)) = O(m\log(n))$ time.

Suppose that the s-t maxflow value is $F^\star$, and at some loop of Lines 3 to 14, the current flow value is $F = \partial\mathbf{f}(s)$. We claim that the s-t maxflow value in the residual graph $G_{\mathbf{f}}$ is $F^\star - F$. This is because $G_{\mathbf{f}}$ captures all ways we can add a flow to $\mathbf{f}$ while remaining feasible, and to produce an s-t maxflow, we need to make up for the fact that $\partial\mathbf{f}(s) = F$ by adding an s-t flow in $G_{\mathbf{f}}$ with value $F^\star - F$. Conversely any larger flow in $G_{\mathbf{f}}$ would lead to a larger $F^\star$, contradicting its definition.

Now, we claim that the widest path P in $G_{\mathbf{f}}$ has width $w \geq \frac{F^\star - F}{m}$. In other words, it contributes at least $\frac{1}{m}$ of the s-t maxflow value in $G_{\mathbf{f}}$. To see why this is useful, it means that we can always decrease the s-t maxflow value in $G_{\mathbf{f}}$ by a multplicative $(1 - \frac{1}{m})$ factor, via augmenting by P. Therefore, after $m\log(F^\star)$ loops of Lines 8 to 10, $G_{\mathbf{f}}$ has s-t maxflow value bounded by

$$\left(1 - \frac{1}{m}\right)^{m\log(F^\star)} \cdot F^\star \leq \exp\left(-\log(F^\star)\right) \cdot F^\star = 1.$$

Here we used the approximation $(1 - \frac{1}{x})^x \leq \exp(-1)$ for $x \geq 1$, where exp is the exponential function. At this point we can just run one more iteration of Ford-Fulkerson to disconnect the residual graph. Thus the runtime of augmenting by the widest path is bounded by

$$O(m\log(F^\star) \cdot m\log(n)) = O(m^2\log(F^\star)\log(n)),$$

because we compute a widest path $O(m\log(F^\star))$ times. We remark that this "final augmentation" argument still relies on the assumption that capacities are integers. We call the above runtime, which depends polynomially on the bit complexity of the input, a *weakly polynomial* runtime.

We conclude by proving our claim that the widest path contributes a $\frac{1}{m}$ fraction of the maxflow value. Our key tool is *flow decomposition*, which says any s-t flow can be decomposed into $\leq m$ flows, each of which pushes flow along a single s-t path, or creates no net flow at any vertex.

Fact 1 (Flow decomposition)**.** *Every s-t flow $\mathbf{f}$ in $G = (V, E, \mathbf{c})$ can be decomposed as $\mathbf{f} = \sum_{i\in[k]} \mathbf{f}_i$ for flows $\{\mathbf{f}_i\}_{i\in[k]} \subset \mathbb{R}^E_{\geq 0}$, where $k \leq m$, and each $\mathbf{f}_i$ satisfies one of the following.*

- *$\mathbf{f}_i$ is a multiple of an s-t path P, i.e., for some $w > 0$, $[\mathbf{f}_i]_e = w$ for all $e \in P$.*
- *$\mathbf{f}_i$ is a* circulation*, i.e., it has $\partial\mathbf{f}_i(v) = 0$ for all $v \in V$ (including $v \in \{s, t\}$).*

We defer a proof of Fact 1 to Chapter 10, [Eri24], but briefly sketch the intuition here. The idea is to begin by making a graph where every edge has capacity $\mathbf{f}_e$. We then repeatedly find paths P from s to t in the graph, and "peel off" one more s-t flow along P, with value set to the width of the path. Such a flow is always feasible, and removing it eliminates an edge in the graph completely. Recursing until $\partial\mathbf{f}(s) = 0$ yields a circulation, which we peel off as the last component. This process can only last m iterations, because each flow we peel off eliminates an edge.

Now consider the s-t maxflow in some residual graph $G_{\mathbf{f}}$, and call it $\mathbf{f}'$. Fact 1 decomposes $\mathbf{f}' = \sum_{i\in[k]} \mathbf{f}'_i$; each $\mathbf{f}'_i$ is either a s-t path or a circulation, and without loss, we can assume they are all s-t paths, because subtracting circulations does not affect flow value. At least one of the $\mathbf{f}'_i$ must contain a $\frac{1}{k} \geq \frac{1}{m}$ fraction of the s-t maxflow value, so it has width $\geq \frac{1}{m}$ of the value. The widest path (which may not be one of the $\mathbf{f}'_i$) can only have greater width, proving our claim.

Modern developments. The s-t maxflow problem and its variants have undergone an algorithmic revolution in recent years, fueled by an interplay between combinatorial techniques, e.g., graph decomposition, and continuous techniques that are better able to exploit the geometry of feasible flow constraints. In particular, the s-t maxflow problem is now solvable in $O(m^{1+o(1)}\log(U))$ time [CKL+22], if all edge capacities are integers in the range $[1, U]$. This runtime is almost-optimal among weakly polynomial time algorithms. The [CKL+22] result was the culmination of a long line of developments that led to gradually faster flow algorithms over the past decade.

One could further insist on *strongly polynomial* runtimes. A strongly polynomial time algorithm works in the *real RAM* model, where there are no bit complexity bounds, and all inputs are treated as arbitrary real numbers. For example, in the real RAM model, there is no guarantee that the widest path algorithm will terminate, because the residual graph could contain arbitrarily small edge weights. The runtime of a strongly polynomial algorithm then counts the number of arithmetic operations used on its real number inputs. The state-of-the-art strongly polynomial s-t maxflow algorithm runs in time $O(mn)$, by combining work of [KRT94, Orl13].

Finally, it is worth mentioning that these new s-t maxflow algorithms have since led to other downstream breakthroughs. For example, it is now known how to compute all-pairs maxflows in $O(m^{1+o(1)}\log(U))$ time, i.e., just as efficiently as a single s-t maxflow [ALPS23].

5 Applications

In this section we outline several creative applications and extensions of the graph algorithmic toolkit that we have developed thus far, as a demonstration of its flexibility.

5.1 Arbitrage

Let $G = (V, E, \mathbf{w})$ model an exchange network, where every vertex $v \in V$ corresponds to a currency (or, e.g., some type of good). Taking an edge from vertex u to v corresponds to exchanging currency u for v, and $\mathbf{w}_{(u,v)}$ represents the *exchange rate*: each unit of currency u will yield $\mathbf{w}_{(u,v)}$ units of currency v if traded. We assume that all exchange rates satisfy $\mathbf{w}_{(u,v)} > 0$.

We wish to detect whether a given exchange network is susceptible to *arbitrage*. In particular, arbitrage is a way to start with 1 unit of some currency, make a sequence of exchanges, and end up with more than 1 unit of the same currency. Due to rapid fluctuations in exchange rates, it is important to detect the presence of potential arbitrage opportunities to either improve predictions on how the market will act, or to make appropriate adjustments to rates.

To model this problem mathematically, what we are looking for is a cycle C, such that $\prod_{e\in C}\mathbf{w}_e > 1$. To see this, if we follow the cycle edges around starting at a vertex v, then $\prod_{e\in C}\mathbf{w}_e$ is exactly the amount of currency v we are left with after making all trades. This seems somewhat close to a primitive we developed in Section 3.2, i.e., the ability to detect negative-weight cycles in $O(mn)$ time, save two differences: it concerns products (rather than sums), and thresholds at 1 (rather than 0). It turns out that we can nonetheless use the Bellman-Ford negative-weight cycle detection method to detect arbitrage in exchange networks, thanks to the following fact.

Fact 2. *Let* $\{w_i\}_{i\in[k]} \subset \mathbb{R}_{>0}$. *Then* $\prod_{i\in[k]} w_i > 1$ *iff* $\sum_{i\in[k]}(-\log(w_i)) < 0$.

Fact 2 shows that there is an arbitrage instance in G iff there is a negative-weight cycle in a modified graph $G' = (V, E, \mathbf{w}')$, defined as follows: for every edge $e \in E$ with edge weight $\mathbf{w}_e$, we give the same edge in G' a weight of $-\log(\mathbf{w}_e)$. Thus, we can detect arbitrage in $O(mn)$ time.

5.2 Heuristics for s-t shortest path

We next give a more practical variant of Dijkstra's SSSP algorithm (Algorithm 6). While Algorithm 6 is quite fast, it does not terminate (as stated) until it has discovered all vertices. If we are only interested in the shortest s-t path for some known target vertex t, rather than all shortest paths from s, this could be inefficient; we only need to wait until Algorithm 6 labels t to terminate. Thus, our goal is to make Algorithm 6 pull the vertex t out of its priority queue as soon as possible.

Consider for instance the behavior of Algorithm 6 in a two-dimensional grid, e.g., if a robot is searching for the shortest path to a target object t in a room. In this case, Algorithm 6 is

essentially performing a breadth-first search in the grid, because it pulls vertices out of the priority queue in the order of their shortest distance from s. That is, its behavior is completely oblivious to t, exploring vertices even if we know they are on the opposite side of s from t.

The A* algorithm [HNR68] is a modification of Dijkstra's algorithm which uses a *heuristic* $h : V \to \mathbb{R}$ to help guide the search. Given a graph $G = (V, E, \mathbf{w})$ and a heuristic h, we can define a new graph $G^{(h)} = (V, E, \mathbf{w}^{(h)})$ on the same vertices and edges, with modified edge weights

$$\mathbf{w}^{(h)}_{(u,v)} := \mathbf{w}_{(u,v)} - h(u) + h(v). \tag{13}$$

We call the heuristic h *consistent* if it ensures all modified weights $\mathbf{w}^{(h)}_{(u,v)}$ remain nonnegative.

Why is this a good idea? We claim that no shortest paths change under the modification (13). However, the shortest path *distances* can change, in a way that lets us discover the target t faster.

Lemma 7. *For any $v \in V \setminus \{s\}$, P is an s-v shortest path in $G^{(h)}$ iff it is also an s-v shortest path in G. In this case, the shortest path distance in $G^{(h)}$ is $\sum_{e \in P} \mathbf{w}_e - h(s) + h(v)$.*

Proof. Fix an arbitrary s-v path $P = \{(v_1 = s, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k = v)\}$. Its total weight in G is $\sum_{e \in P} \mathbf{w}_e$. On the other hand, its total weight in $G^{(h)}$ is, following (13),

$$\begin{aligned}\sum_{e \in P} \mathbf{w}^{(h)}_e &= \sum_{e \in P} \mathbf{w}_e + (h(v_2) - h(v_1)) + (h(v_3) - h(v_2)) + \ldots + (h(v_k) - h(v_{k-1})) \\ &= \sum_{e \in P} \mathbf{w}_e + h(v_k) - h(v_1) = \sum_{e \in P} \mathbf{w}_e + h(v) - h(s),\end{aligned}$$

where we telescoped the edge weight modifications, and used that $v_1 = s$, $v_k = v$ start and end the path. Thus, the total weight of *every* s-v path changes by exactly the same amount, i.e., $h(v) - h(s)$. This proves that shortest s-v paths in G and $G^{(h)}$ are equivalent. The second conclusion follows because the shortest s-v path, P, has its weight changed by $h(v) - h(s)$ as argued earlier. □

To provide some intuition on Lemma 7, we can view the heuristic h as a "price function," modifying edge weights such that leaving a vertex u saves $h(u)$ in some currency, and entering a vertex v costs $h(v)$ of the currency. Lemma 7 restates the fact that in a path, every vertex that is entered is also exited, except for the first and last. Thus, the total cost of every s-v path changes by $h(v) - h(s)$.

If h is consistent (so all edge weights in $G^{(h)}$ are positive), Lemma 7 shows that running Algorithm 6 on $G^{(h)}$ will still discover all of the shortest s-v paths in G. The order in which they are discovered depends on our choice of heuristic h, i.e., Lemma 7 shows that vertices v with smaller $h(v)$ are favored. To take advantage of this flexibility in the vertex discovery order, the idea is to design our heuristic such that $h(v)$ is large for vertices $v \in V$ that we do not want to visit. However, we want to do so in a way that does not violate consistency.

One way to design a consistent heuristic is to set $h(t) = 0$, and to set $h(v) = \mathrm{m}(v, t)$ for any *metric* m, i.e., a distance function that satisfying the triangle inequality. Here we assume $\mathrm{m}(u, v) = \mathbf{w}_{(u,v)}$ for any edge $(u, v) \in E$. The intuition is that this penalizes vertices far from t with a large h label, so they are discovered later. For example, if all vertices are identified with vectors in $\mathbb{R}^d$, then we can set $\mathrm{m}(v, t) = \|\mathbf{v} - \mathbf{t}\|_2$, where $\|\cdot\|_2$ is the Euclidean norm (see Section 5.1, Part I for a review), and $\mathbf{v}, \mathbf{t}$ are the vectors associated with v, t respectively. Any metric yields a consistent heuristic:

$$\mathbf{w}^{(h)}_{(u,v)} = \mathbf{w}_{(u,v)} - h(u) + h(v) = \mathrm{m}(u, v) - \mathrm{m}(u, t) + \mathrm{m}(v, t) \geq 0.$$

In summary, the main idea of A* is that if there is a consistent heuristic h available, penalizing vertices far from a target t, then the modification (13) can speed up how quickly Dijkstra's algorithm discovers t. We note that the worst-case runtime of Dijkstra's algorithm remains unchanged.

APSP. It turns out that A* can slightly improve the APSP runtime achieved by Floyd-Warshall (Section 5.3, Part III), as long as the graph in question is not too dense. Let $G = (V, E, \mathbf{w})$ be a graph free of negative-weight cycles, that we want to solve APSP on. If G had only positive-weight edges, we can do so in $O(mn \log(n))$ time, by running Algorithm 6 from all n source vertices. On the other hand, with negative-weight edges the best algorithm we have given is Floyd-Warshall, which uses $O(n^3)$ time. This is worse if $m = o(\frac{n^2}{\log(n)})$, i.e., G is moderately sparse.

We claim that we can apply a heuristic modification of the form (13) such that all edge weights become nonnegative. If we can compute a suitable heuristic h, the rest of the algorithm takes $O(mn\log(n))$ time, because we can then run Dijkstra n times on $G^{(h)}$. Our strategy is to use $h(v) = -d(s,v)$ for all $v \in V$, where d is the shortest path distance, and s is a vertex that can reach all of G.[6] We can compute all $d(s,v)$ in $O(mn)$ time, using Algorithm 7.

Now consider an edge $(u,v) \in E$. Its heuristically-modified edge weight (13) is

$$\mathbf{w}^{(h)}_{(u,v)} = \mathbf{w}_{(u,v)} + d(s,u) - d(s,v) \geq 0,$$

since we can get from s to v by taking the shortest s-u path and then (u,v), so $d(s,v)$ cannot exceed $d(s,u) + \mathbf{w}_{(u,v)}$. Thus our heuristic creates nonnegative weights, and yields an $O(mn\log(n))$-time APSP algorithm. This method for computing APSP is called Johnson's algorithm [Joh77].

5.3 Flow reductions

The s-t maxflow problem is one of the most powerful *reduction* tools in graph algorithm design. The idea of a flow reduction is to take some problem on a graph G, and transform G into a different graph G', such that solving s-t maxflow on G' yields a solution to our original problem on G.

Here, we outline some basic uses of this technique. However, flow reductions are quite general and apply to many different graph problems, including maxflow with vertex capacities, tuple selection, variants of scheduling, image segmentation, and more. We recommend Chapter 11, [Eri24] and Chapters 7.5 to 7.13, [KT05], for an introduction to a (much) broader range of flow reductions.

Disjoint paths. One of the simplest flow reductions does not even require modifying the graph G. Suppose we are given a directed graph $G = (V,E)$, and we wish to compute the maximum number of disjoint s-t paths we can find in G, where $s \neq t$ are vertices in V. Here we say two paths are disjoint if they do not share any edges in common.

It turns out that the disjoint paths problem is simply an instance of s-t maxflow. Let us give every edge a capacity of 1, and let $\mathbf{f}$ be the s-t maxflow. As argued in Section 4.3, we can assume $\mathbf{f}$ places an integer amount of flow on every edge, because the original capacities are integers. Thus, $\mathbf{f}$ either puts 0 or 1 flow on every edge. We can then use flow decomposition (Fact 1) to repeatedly peel off s-t paths from $\mathbf{f}$; all paths are disjoint, because each edge contains only one unit of flow. The number of s-t paths returned by this process is thus the s-t maxflow value.

Bipartite matching. Another famous flow reduction is bipartite matching. Let $G = (V,E)$ be an unweighted bipartite graph. Here, G being bipartite means V can be partitioned into two sets, $V = L \cup R$, such that every edge $e = (u,v) \in E$ points from L to R, i.e., $u \in L$ and $v \in R$.

A matching M is a subset of E such that each vertex participates in at most one edge. That is, for each $v \in R$, there is at most one edge $(u,v) \in M$, and likewise for each $u \in L$. Our goal is to compute the maximum matching size in G, i.e., the largest $|M|$ achievable. In the stable matching problem in Section 5, Part IV, the maximum matching size was $\frac{n}{2}$ because we could arbitrarily pair up the $\frac{n}{2}$ vertices on the left with the $\frac{n}{2}$ vertices on the right. However, in bipartite graphs where E does not contain every possible edge, such a large matching may not always exist.

We can solve bipartite matching using a flow reduction. At first, this seems surprising: bipartite matching is about vertex capacities, not edge capacities, and there is not really a particular source or sink vertex. Nonetheless, it turns out we can convert bipartite matching into a flow problem on an augmented graph. In particular, consider forming a modified graph

$$G' = (\underbrace{L \cup R \cup \{s,t\}}_{:=V'}, \underbrace{E \cup \{(s,u)\}_{u\in L} \cup \{(v,t)\}_{v\in R}}_{:=E'}),$$

such that every edge in G' is assigned a capacity of 1. To explain our construction a bit more, G' has two additional vertices compared to V: a "super-source" vertex s and a "super-sink" vertex t. We also add directed edges going from s to every vertex in L, the left half of the bipartition, as well as from every vertex in R, the right half of the bipartition, to t.

[6] If SCC(G) is not disconnected, then by Lemma 4, this is always achievable; choose s in a source component of SCC(G). Otherwise, we can recurse on each connected piece of SCC(G) separately.

We claim that the maximum matching size in G is the same as the s-t maxflow value in G'. Given a matching M in G with $|M| = k$, we may extend it to a flow $\mathbf{f}$ on G' achieving $\partial\mathbf{f}(s) = k$: for each edge $(u, v) \in M$, place one unit of flow on each of the three edges in the s-t path $\{(s, u), (u, v), (v, t)\}$. It is straightforward to check this is an s-t flow with value k. Further, $\mathbf{f}$ is a feasible flow in G' because M is a matching; each edge of the form (s, u) and (v, t) is used at most once.

Conversely, suppose $\mathbf{f}$, the s-t maxflow in G', attains value $\partial\mathbf{f}(s) = k$. As argued in Section 4.3, we again may assume $\mathbf{f} \in \mathbb{Z}^{E'}$, i.e., $\mathbf{f}$ puts an integer amount of flow on each edge. Because edge capacities in G' are all 1, this means $\mathbf{f}$ only puts 0 or 1 units of flow edgewise.

We claim we can recover a matching of size k from $\mathbf{f}$. To do so, we again repeatedly peel off s-t paths; each removal of a path subtracts one unit from $\partial\mathbf{f}(s)$, so this process terminates after k paths are removed. Each path is of the form $\{(s, u), (u, v), (v, t)\}$ for some $u \in L$ and $v \in R$, as these are the only s-t paths in G'. Moreover each edge of the form (s, u) or (v, t) can only be used by one path. Thus, if we only keep the edges in these paths that exist in the original graph G, their union forms a matching, as each vertex in $L \cup R$ participates in a single path. This concludes our proof that the maximum matching size in G is the s-t maxflow value in G'.

Further reading

For more on Section 2, see Chapter 20, [CLRS22], or Chapters 5 to 6, [Eri24], or Chapter 3, [KT05], or Chapter 8, [Rou22].

For more on Section 3, see Chapter 22, [CLRS22], or Chapter 8, [Eri24], or Chapter 4.4 and Chapters 6.8 to 6.10, [KT05], or Chapters 9 and 18, [Rou22].

For more on Section 4, see Chapter 24, [CLRS22], or Chapter 10, [Eri24], or Chapters 7.1 to 7.4, [KT05].

For more on Section 5, see Chapters 9 and 11, [Eri24], or Chapters 7.5 to 7.13, [KT05].

References

[ALPS23] Amir Abboud, Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. All-pairs max-flow is no harder than single-pair max-flow: Gomory-hu trees in almost-linear time. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023*, pages 2204–2212. IEEE, 2023.

[BCF23] Karl Bringmann, Alejandro Cassis, and Nick Fischer. Negative-weight single-source shortest paths in near-linear time: Now faster! In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023*, pages 515–538. IEEE, 2023.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BNW22] Aaron Bernstein, Danupon Nanongkai, and Christian Wulff-Nilsen. Negative-weight single-source shortest paths in near-linear time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022*, pages 600–611. IEEE, 2022.

[CKL+22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022*, pages 612–623. IEEE, 2022.

[CLRS22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition*. The MIT Press, 2022.

[Eri24] Jeff Erickson. *Algorithms*. 2024.

[FF56] Lester R. Ford and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[Fin24] Jeremy T. Fineman. Single-source shortest paths with negative real weights in $\widetilde{O}(mn^{8/9})$ time. In Bojan Mohar, Igor Shinkar, and Ryan O'Donnell, editors, *Proceedings of the 56th Annual ACM Symposium on Theory of Computing, STOC 2024, Vancouver, BC, Canada, June 24-28, 2024*, pages 3–14. ACM, 2024.

[For56] Lester R. Ford. Network flow theory. *RAND Corporation, Paper P-923*, 1956.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987.

[HJQ24] Yufan Huang, Peter Jin, and Kent Quanrud. Faster single-source shortest paths with negative real weights via proper hop distance. *CoRR*, abs/2407.04872, 2024.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[HR54] Theodore E. Harris and Frank S. Ross. Fundamentals of a method for evaluating rail net capacities. *RAND Corporation*, 1954.

[Joh77] Donald B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[KRT94] Valerie King, S. Rao, and Robert Endre Tarjan. A faster deterministic maximum flow algorithm. *J. Algorithms*, 17(3):447–474, 1994.

[KT05] Jon Kleinberg and Éva Tardos. *Algorithm Design*. 2005.

[Orl13] James B. Orlin. Max flows in o(nm) time, or better. In *Symposium on Theory of Computing Conference, STOC'13*, pages 765–774. ACM, 2013.

[Rou22] Tim Roughgarden. *Algorithms Illuminated*. Soundlikeyourself Publishing, 2022.

[Sch05] Alexander Schrijver. On the history of combinatorial optimization (till 1960). *Handbooks in Operations Research and Management Science*, 12:1–68, 2005.

[Sha81] Micha Sharir. A strong-connectivity algorithm and its applications to data flow analysis. *Computers and Mathematics with Applications*, 7(1):67–72, 1981.

[Shi55] Alfonso Shimbel. Structure in communication nets. In *Proceedings of the Symposium on Information Networks*, pages 199–203. Polytechnic Press of the Polytechnic Institute of Brooklyn, 1955.

[Wil21] R. Ryan Williams. From circuit complexity to faster all-pairs shortest paths. *SIAM Rev.*, 63(3):559–582, 2021.