Chapter 13

Graph Search

The term *graph search* or *graph traversal* refers to a class of algorithms that systematically explore the vertices and edges of a graph. Graph-search algorithms can be used to compute many interesting properties of graphs; they are indeed at the heart of many graph algorithms. In this chapter, we introduce the concept of a graph search, describe several relatively general algorithms for it, including priority-first search. In the next several chapters, we will consider further specializations of the general graph-search algorithm, specifically the breadth-first-search and the depth-first-search algorithms, each of which is particularly useful in computing certain properties of graphs.

As an example of a graph property that we may wish to compute using graph search, consider reachability queries: we say that a vertex u is ***reachable*** from v if there is a path from v to u. In a digraph, the path from u to v should be directed; in a graph, it should be undirected. As a different kind of query, we might want to determine the shortest path from a vertex to another.

As the properties above suggest, a graph searches usally starts at a specific vertex, and also sometimes at a set of vertices. We refer to such a vertex (vertices) as the ***source*** vertex (vertices). To ensure efficiency, graph search algorithms usually keep track of the vertices that they have already visited so as to avoid visiting a vertex for a second time.

Vertex Hopping. As a warm-up, let start with a technique that we call *vertex hopping*. As we will see vertex hopping is not helpful in many cases but it can offer a good starting point in appreciating some intricacies of graph search. The idea behind vertex hopping is repeatedly select some set of vertices to visit, visit them, and continue until all vertices are visited. More precisely, vertex hoping can be expressed as follows

Algorithm 13.1 (Vertex Hopping). *function vertexHopper (G = (V,E), s) =*

```
visit source s

let X =
  start {s} and
  while $X \neq V$ do
    pick U such that $U \subseteq V \setminus X$
    visit the vertices in U
    visit the edges coming out of vertices in U
    $X = X \cup U$
```

The algorithm starts by visiting the source vertex and uses the set X to keep track of all the vertices that it has thus far visited. At each step (iteration of the while loop), the algorithm non-deterministically selects a set of unvisited vertices and visits them. Note that the algorithm as expressed is naturally parallel: we can visit many vertices at the same time.

Question 13.2. *When the algorithm terminates, does it visit all the edges?*

Since the algorithm visits all the out-edges of a vertex when it visits the vertex and since it visits all the vertices, the algorithm also visits all the edges.

As an example of how we might use vertex hopping to solve a graph problem, let's consider reachability.

Question 13.3. *Can you solve the reachability problem using vertex hopping?*

We can solve the reachability problem by keeping a table mapping each visited vertex to the set of vertices reachable from it. When we visit a vertex u, we extend the table by finding all the vertices that can reach u and extending their reach by adding v for every edge (u, v).

Question 13.4. *Is this algorithm efficient?*

Unfortunately, this algorithm is not efficient. In reachability, we are interested in finding out whether a particular vertex is reachable from another. This algorithm is building the full the reachability information for all vertices.

Question 13.5. *Can you see why vertex hopping is not effective for reachability computations?*

The problem is that the algorithm is not taking advantage of the structure of the graph: it nondeterministically jumps around the graph to visit vertices without following paths in the graph. As we will see in later, vertex hopping, especially its parallel version, can be useful in some applications but for problems such as reachability, the algorithm is not able to recover efficiently the path information.

Since many interesting properties of graphs involves paths, graph-search algorithms that we will consider here will visit (explore) the vertices in the graph by following paths.

Question 13.6. *Can you see how we can build on vertex hopping to visits paths?*

To do this efficiently, graph search algorithms usually maintain a frontier of vertices that are not yet visited but are adjacent to a visited vertex and visit a vertex only when it is in the frontier.

Definition 13.7. *For a graph $G = (V, E)$ and a visited set $X \subset V$, the the* frontier *is the set of un-visited out neighbors of the set of visited vertices X.*

As a special case, graph search algorithms usually start by placing the source vertex in the frontier at the beginning of a graph search.

Put things together, we can write a graph search algorithm as follows.

Algorithm 13.8 (Graph Search)**.**

```
function graphSearch (G, s) =
  let
    (X, F) =
      start ({}, {s}) and
      while (|F| > 0) do
        pick U ⊆ F
        visit U
        X = X ∪ U          % Update X with the newly visited veritces
        F = $N_G^+(X) \setminus X$          % The next frontier
  in X end
```

The algorithm start by initializing the set of visited vertices to the empty set and the frontier to the source. It then proceeds in a number of *rounds*, each of which correspond to an iteration of the while loop. At each round, the algorithm selects a set U of vertices in the frontier, visits them, and updates the visited and the frontier sets. Recall that $N_G^+(v)$ are the outgoing neighbors of the vertex v in the graph G, and $N_G^+(U) = \bigcup_{v \in U} N_G^+(v)$. Note that when visiting a group of vertices U, we can usually visit them in parallel.

The function `graphSearch` terminates when there are no more vertices to visit—i.e., the frontier is empty.

Question 13.9. *Does the algorithm visit all the vertices in the graph?*

This does not mean that it visits all the vertices: vertices that are not reachable from the source will never be visited by the algorithm. In fact, the function *graphSearch* returns the set of vertices that are reachable from the source vertex s. In many applications of graph search, we keep track of other information in addition to or in place of just reachability.

Since the function *graphSearch* as specified is not specific about the set of vertices to be visited next, it can be used to describe many different graph-search techniques. In this book we will consider three methods for selecting the subset. Each such method leads to a specific graph-search technique. Selecting all of the vertices in the frontier leads to breadth-first search, selecting a neighbor of the most recently visited vertex leads to depth-first search, and selecting the highest-priority vertex (or vertices) in the frontier, by some definition of priority, leads to priority-first search.

13.1 Priority-First Search

The graph search algorithm that we described does not specify the vertices to visit next (the set U). This is intentional, because graph search algorithms such as breadth-first search and depth-first search, differ exactly on which vertices they visit next. While graph search algorithms differ on this point, many are also quite similar: there is often a method to their madness, which we will describe now.

Consider a graph search algorithm that assigns a priority to every vertex in the frontier. You can imagine such an algorithm giving a priority to a vertex v when it inserts v into the frontier. Now instead of picking some unspecified subset of the frontier to visit next, the algorithm picks the highest (or the lowest) priority vertices. Effectively we change line 6 in the *graphSearch* algorithm to:

```
Pick $U$ as he highest priority vertices in $F$
```

We refer to such an algorithm as a *priority-first search (PFS)* or a *best-first search*. The priority order can either be determined statically (a priori) or it can be generated on the fly by the algorithm.

Priority-first search is a greedy technique since it greedily selects among the choices available (the vertices in the frontier) based on some cost function (the priorities) and never backs up. Algorithms based on PFS are hence often referred to as greedy algorithms. As you will see soon, several famous graph algorithms are instances of priority-first search, e.g., Dijkstra's algorithm for finding single-source shortest paths and Prim's algorithm for finding Minimum Spanning Trees.

Example 13.10. *How can you use a priority search to explore the web from a start page so that the web sites that are of higher importance to you are visited first?*
You can use priority search to explore the web in a way that gives priority to the sites that you are interested in. The idea would to implement the frontier as a priority queue data structure containing the unvisited outgoing links based on your interest on the link (which we assume to be known). You can then choose what page to visit next by simply selecting the link with the highest priority. After visiting the page, you would remove it from the priority queue and add all its unvisited outgoing links to the list with their corresponding priorities.

.