1 Introduction

Abstract

Graphs are discrete structures that are frequently used to model many real-world problems such as communication networks, social networks, and biological networks. We present sequential, parallel, and distributed graph algorithm concepts, challenges in graph algorithms, and the outline of the book in this chapter.

1.1 Graphs

Graphs are discrete structures that are frequently used to model many real-world problems such as communication networks, social networks, and biological networks. A graph consists of *vertices* and *edges* connecting these vertices. A graph is shown as $G = (V, E)$ where V is the set of vertices and E is the set of edges it has. Figure 1.1 shows an example graph with vertices and edges between them with $V = \{a, b, c, d\}$ and $E = \{(a, b), (a, e), (a, d), (b, c), (b, d), (b, e), (c, d), (d, e)\}$, (a, b) denoting the edge between vertices a and b for example.

Graphs have numerous applications including computer science, scientific computing, chemistry, and sociology since they are simple yet effective to model real-life phenomenon. A vertex of a graph represents some entity such as a person in a social network or a protein in a biological network. An edge in such a network corresponds to a social interaction such as friendship in a social network or a biochemical interaction between two proteins in the cell.

Study of graphs has both theoretical and practical implications. In this chapter, we describe the main goal of the book which is to provide a unified view of graph algorithms in terms of sequential, parallel, and distributed graph algorithms with

K. Erciyes, *Guide to Graph Algorithms*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-73235-0_1

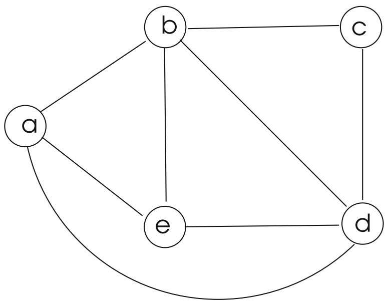

Fig. 1.1 An example graph consisting of vertices $\{a, b, \ldots, h\}$

emphasis on sequential graph algorithms. We describe a simple graph problem from these three views and then review challenges in graph algorithms by finally outlining the contents of the book.

1.2 Graph Algorithms

An *algorithm* consists of a sequence of instructions processed by a computer to solve a problem. A *graph algorithm* works on graphs to find a solution to a problem represented by the graphs. We can classify the graph problems as sequential, parallel, and distributed, based on the mode of running these algorithms on the computing environment.

1.2.1 Sequential Graph Algorithms

A *sequential algorithm* has a single flow of control and is executed sequentially. It accepts an input, works on the input, and provides an output. For example, reading two integers, adding them and printing the sum is a sequential algorithm that consists of three steps.

Let us assume the graph of Fig. 1.2 represents a small network with vertices labeled $\{v_1, v_2, \ldots, v_8\}$ and having integers 1,...,13 associated with each edge. The

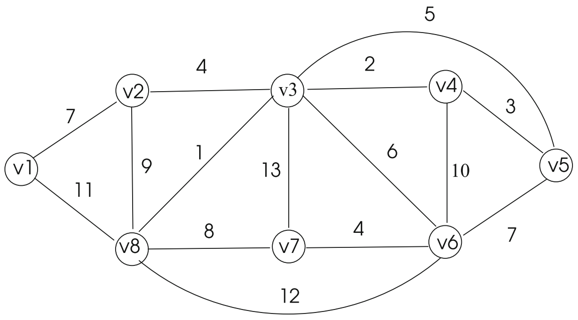

Fig. 1.2 An example graph

integer value for each edge may denote the cost of sending a message over that edge. Commonly, the edge values are called the *weights* of edges. Let us assume our aim is to design a sequential algorithm to find the edge with the maximum value. This may have some practical usage, as we may need to find the highest cost edge in the network to avoid that link while some communication or transportation is done.

Let us form a distance matrix D for this graph, which has entries $d(i, j)$ showing the weights of edges between vertices v_i and v_j as below:

$$D = \begin{bmatrix} 0 & 7 & 0 & 0 & 0 & 0 & 0 & 1 \\ 7 & 0 & 4 & 0 & 0 & 0 & 0 & 9 \\ 0 & 4 & 0 & 2 & 5 & 6 & 14 & 1 \\ 0 & 0 & 2 & 0 & 3 & 10 & 0 & 0 \\ 0 & 0 & 5 & 3 & 0 & 13 & 0 & 0 \\ 0 & 7 & 6 & 10 & 13 & 0 & 4 & 12 \\ 0 & 7 & 14 & 0 & 0 & 4 & 0 & 8 \\ 11 & 9 & 1 & 0 & 0 & 12 & 8 & 0 \end{bmatrix}$$

If two vertices v_i and v_j are not connected, we insert a zero for that entry in D. We can have a simple sequential algorithm that finds the largest value in each row of this matrix first and then the maximum value of all these largest values in the second step as shown in Algorithm 1.1.

Algorithm 1.1 *Sequential_graph*

```
1: int $D[n, n] \leftarrow$ edge weights
2: int $max[8], maximum$
3: for $i$=1 to 8 do
4:     $max[i] \leftarrow D[i, 1]$
5:     for $j$=2 to 8 do
6:         if $D[i, j] > max[i]$ then
7:             $max[i] \leftarrow D[i, j]$
8:         end if
9:     end for
10: end for
11: $maximum \leftarrow max[1]$
12: for $i$=2 to 8 do
13:     if $max[i] > maximum$ then
14:         $maximum \leftarrow max[i]$
15:     end if
16: end for
17: output $maximum$
```

This algorithm requires 8 comparisons for each row for a total of 64 comparisons. It needs n^2 comparisons in general for a graph with n vertices.

1.2.2 Parallel Graph Algorithms

Parallel graph algorithms aim for performance as all parallel algorithms. This way of speeding up programs is needed especially for very large graphs representing complex networks such as biological or social networks which consist of huge number of nodes and edges. We have a number of processors working in parallel on the same problem and the results are commonly gathered at a single processor for output. Parallel algorithms may synchronize and communicate through shared memory or they run as distributed memory algorithms communicating by the transfer of messages only. The latter mode of communication is a more common practice in parallel computing due to its versatility to realize in general network architectures.

We can attempt to parallelize an existing sequential algorithm or design a new parallel algorithm from scratch. A common approach in parallel computing is the partitioning of data to a number of processors so that each computing element works on a particular partition. Another fundamental approach is the partitioning of computation across the processors as we will investigate in Chap. 4. We will see some graph problems are difficult to partition into data or computation.

Let us reconsider the sequential algorithm in the previous section and attempt to parallelize it using data partitioning. Since graph data is represented by the distance matrix, the first thing to consider would be the partitioning of this matrix. Indeed, row-wise or column-wise partitioning of a matrix representing a graph is commonly used in parallel graph algorithms. Let us have a controlling processor we will call the *supervisor* or the *root* and two *worker* processors to do the actual work. This mode of operation, sometimes called *supervisor/worker model*, is also a common practice in the design of parallel algorithms. Processors are commonly called *processes* to mean the actual processor may also be doing some other work. We now have three processes p_0, p_1, and p_2, and p_0 is the supervisor. The process p_0 has the distance matrix initially, and it partitions and sends the first half of the rows from 1 to 4 to p_1 and 5 to 8 to p_2 as shown below:

$$D = \left[\begin{array}{ccccccccc}
0 & 7 & 0 & 0 & 0 & 0 & 0 & 1 & p_1 \\
7 & 0 & 4 & 0 & 0 & 0 & 0 & 9 & \\
0 & 4 & 0 & 2 & 5 & 6 & 14 & 1 & \\
0 & 0 & 2 & 0 & 3 & 10 & 0 & 0 & \\
\hline
0 & 0 & 5 & 3 & 0 & 13 & 0 & 0 & p_2 \\
0 & 7 & 6 & 10 & 13 & 0 & 4 & 12 & \\
0 & 7 & 14 & 0 & 0 & 4 & 0 & 8 & \\
11 & 9 & 1 & 0 & 0 & 12 & 8 & 0 &
\end{array}\right]$$

Each worker now finds the heaviest edge incident to the vertices in the rows it is assigned using the sequential algorithm described and sends this result to the supervisor p_0 which finds the maximum of these two values and outputs it. A more general form of this algorithm with k worker processes is shown in Algorithm 1.2. Since data is partitioned to two processes now, we would expect to have a significant decrease in the runtime of the sequential algorithm. However, we have communication costs between the supervisor and the workers now which may not be trivial for large data transfers.

Algorithm 1.2 *Parallel_graph*

```
1: int $D[n, n] \leftarrow$ edge weights of graph $G$
2: int $Max[n]$, $E[n/k, n] maximum$
3: if $i = root$ then
4:     row-wise partition distance matrix $D$ of graph into $D_1, ..., D_k$
5:     for $i$=1 to $k$ do
6:         send $D_i$ to $p_i$
7:     end for
8:     for $i$=1 to $k$ do
9:         receive $largest_i$ from $p_i$ into $max[i]$
10:    end for
11:    find the maximum value of $max$ using the sequential algorithm
12:    output $maximum$
13: else
14:    receive my rows into $E$
15:    find the maximum value in $E$ using sequential algorithm
16:    send $root$ my maximum value
17: end if
```

Designing parallel graph algorithms may not be trivial as in this example, and in general we need more sophisticated methods. The operation we need to do may depend largely on what was done before which means significant communications and synchronization may be needed between the workers. The inter-process communication across the network connecting the computational nodes is costly and we may end up designing a parallel graph algorithm that is not efficient.

1.2.3 Distributed Graph Algorithms

Distributed graph algorithms are a class of graph algorithms in which we have a computational node represented by a vertex of the graph. The problems to be solved with such algorithms are related to the network they represent; for example, it may be required to find the shortest distance between any two nodes in the network so that whenever a data packet comes to a node in the network, it forwards the packet to one of its neighbors that is on the least cost path to the destination. In such algorithms, each node typically runs the same algorithm but has different neighbors to communicate and transfer its local result. In essence, our aim is to solve an overall problem related to the graph representing the network by the cooperation of the nodes in the network. Note that the nodes in the network can only communicate with their neighbors and this is the reason these algorithms are sometimes referred to as *local* algorithms.

In the distributed version of our sample maximum weight edge finding algorithm, we have computational nodes of a computer network as the vertices of the graph, and our aim is that each node in the network modeled by the graph should receive the largest weight edge of the graph in the end. We will attempt to solve this problem using *rounds* for the synchronization of the nodes. Each node starts the round, performs some function in the round, and does not start the next round until all other nodes have also finished execution of the round. This model is widely used for distributed algorithms as we will describe in Chap. 5 and there is no other central control other than the synchronization of the rounds. Each node starts by broadcasting the largest weight it is incident to all of its neighbors and receiving the largest weight values from neighbors. In the following rounds, a node broadcasts the largest weight it has seen so far and after a certain number of steps, the largest value will be propagated to all nodes of the graph as shown in Algorithm 1.3. The number of steps is the *diameter* of the graph which is the maximum number of edges between any two vertices.

Algorithm 1.3 *Distributed_graph*

```
1: boolean $finished, round\_over \leftarrow false$
2: message type $start, result, stop$
3: while $count \leq diam(G)$ do
4:     receive max(j) from all neighbors
5:     find the maximum of all received values
6:     send the maximum value to all neighbors
7:     $count \leftarrow count + 1$
8: end while
```

We now can see fundamental differences between parallel and distributed graph algorithms using this example as follows.

- Parallel graph algorithms are needed mainly for the speedup they provide. There are a number of processing elements that work in parallel which cooperate to finish an overall task. The main relation between the number of processes and the size of the graph is that we would prefer to use more processes for large graphs. We assume each processing element can communicate with each other in general although there are some special parallel computing architectures such as processors forming a cubic architecture of communication as in the *hypercube*.
- In distributed graph algorithms, computational nodes are the vertices of the graph under consideration and communicate with their neighbors only to solve a problem related to the network represented by the graph. Note that the process number is the number of vertices of the graph for these algorithms.

One important goal of this book is to provide a unified view of graph algorithms from these three different angles. There are cases we may want to solve a network problem on parallel processing environment, for example, all shortest paths between any two nodes in the network may need to be stored in a central server to be transferred to individual nodes or for statistical purposes. In this case, we run a parallel algorithm for the network using a number of processing elements. In a network setting, we need each node to work to know the shortest paths from it to other nodes.

A general approach is to derive parallel and distributed graph algorithms from a sequential one but there are ways of converting a parallel graph algorithm to distributed one or vice versa for some problems. For the example problem we have, we can have each row of the distance matrix D assigned to a single process. This way, each process can be represented by a network node provided that it communicates with its neighbors only. Conversions as such are useful in many cases since we do not design a new algorithm from scratch.

1.2.4 Algorithms for Large Graphs

Recent technical advancements in the last few decades have resulted in the availability of data of very large networks. These networks are commonly called *complex networks* and consist of tens of thousands of nodes and hundreds of thousands of links between the nodes. One such type of networks is the biological networks within the cell of living organisms. A protein–protein interaction (PPI) network is a biological network formed with interacting proteins outside the nucleus in the cell.

A social network consisting of individuals interacting over the Internet may again be a very large network. These complex networks can be modeled by graphs with vertices representing the nodes and edges the interaction between the nodes like any other network. However, these networks are different than a small network modeled by a graph in few respects. First of all, they have very small diameters meaning the shortest distance between any two vertices is small when compared to their sizes. For example, various PPI networks consisting of thousands of nodes are found to have a diameter of only several units. Similarly, social networks and technological networks such as the Internet also have small diameters. This state is known as *small-world* property. Second, empirical studies suggest these networks have very few nodes with very high number of connections; and most of the other nodes have few connections to neighbors. This so-called *scale-free* property is exhibited again in most of the complex networks. Lastly, the size of these networks being large requires efficient algorithms for their analysis. In summary, we need efficient and possibly parallel algorithms that exploit various properties such as small-world and scale-free features of these networks.

1.3 Challenges in Graph Algorithms

There are numerous challenges in graphs to be solved by graph algorithms.

- *Complexity of graph algorithms*: A polynomial time algorithm has a complexity that can be expressed by a polynomial function. There are very few polynomial time algorithms for the majority of problems related to graphs. The algorithms at hand typically have exponential time complexities which means even for moderate size graphs, the execution times are significant. For example, assume an algorithm A to solve a graph problem P has time complexity 2^n, n being the number of vertices in the graph. We can see that A may have poor performance even for graphs with $n > 20$ vertices. We then have the following choices:

– *Approximation algorithms*: Search for an approximation algorithm that finds a suboptimal solution rather than an optimal one. In this case, we need to prove that the approximation algorithm always provides a solution within an *approximation ratio* to the optimal solution. Various proof techniques can be employed and there is no need to experiment the approximation algorithm other than statistical purposes. Finding and proving approximation algorithms are difficult for many graph problems.
– *Randomized algorithms*: These algorithms decide on the course of execution based on some random choice, for example, selection of an edge at random. The output is presented typically as *expected* or *with high probability* meaning there is a chance, if even slightly, that the output may not be correct. However, the randomized algorithms provide polynomial solutions to many difficult graph problems.
– *Heuristics*: In many cases, our only choice is the use of common sense approaches called *heuristics* in search of a solution. Choice of a heuristic is commonly pursued by intuition and we need to experiment the algorithm with the heuristic for a wide range of inputs to show it works *experimentally*.

There are other methods such as *backtracking* and *branch-and-bound* which work only for a subset of the search space and therefore have less time complexities. However, these approaches can be applied to only subset of problems and are not general. Let us exemplify these concepts by an example. A *clique* in a graph is a subgraph such that each vertex in this subgraph has connections to all other vertices in the subgraph as shown in Fig. 1.3. Finding cliques in a graph has many implications as these exhibit dense regions of activity. Finding the largest clique of a graph G with n vertices, which is the clique with the maximum number of vertices in the graph, cannot be performed in polynomial time. A *brute force algorithm*, which is typically the first algorithm that comes to mind, will enumerate all 2^n subgraphs of G and check the clique condition from the largest to the smallest. Instead of searching for an approximation algorithm, we could do the following by intuition: start with the vertex that has the highest number of connections called its degree; check whether all of its neighbors have the same number of connections and if all have, then we have a clique. If this fails, continue with the next highest degree vertex. This heuristic will work fine but in general, we need to show experimentally that a heuristic works for most of the input variations, for 90% for example but an algorithm that works fine for 60 % of the time with diverse inputs would not be a favorable heuristic.

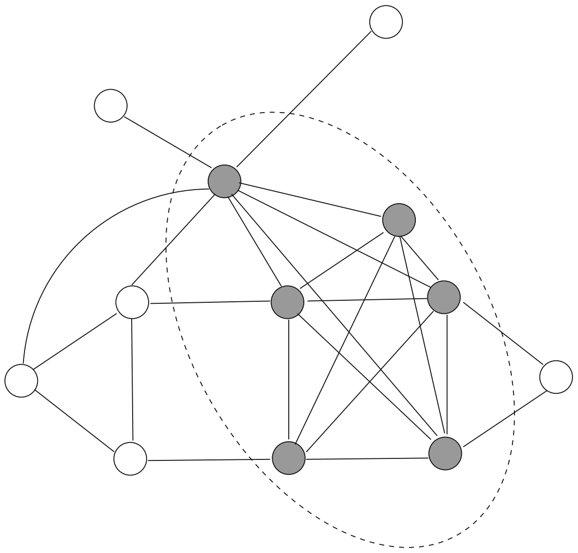

Fig. 1.3 The maximum clique of a sample graph is shown by dark vertices. All vertices in this clique are connected to each other

- *Performance:* Even with polynomial time graph algorithms, the size of the graph may restrict its use for large graphs. Recent interest in large graphs representing large real-life networks demands high-performance algorithms which are commonly realized by parallel computing. Biological networks and social networks are examples of such networks. Therefore, there is a need for efficient parallel algorithms to be implemented in these large graphs. However, some graph problems are difficult to parallelize due to the structure of the procedures used.
- *Distribution*: Several large real networks are distributed in a sense each node of the network is an autonomous computing element. The Internet, the Web, mobile ad hoc networks, and wireless sensor networks are examples of such networks which can be termed as computer networks in general. These networks can again be modeled conveniently by graphs. However, the nodes of the network now actively participate in the execution of the graph algorithm. This type of algorithms is termed *distributed algorithms*.

The main goal of this book is the study of graph algorithms from three angles: sequential, parallel, and distributed algorithms. We think this approach will provide a better understanding of the problem at hand and its solution by also showing its possible application areas. We will be as comprehensive as possible in the study of sequential graph algorithms but will only present representative graph algorithms for parallel and distributed cases. We will see some graph problems have complicated parallel algorithmic solutions reported in research studies and we will provide a contemporary research survey of the topics in these cases.

1.4 Outline of the Book

We have divided the book into three parts as follows.

- *Fundamentals*: This part has four chapters; the first chapter contains a dense review of basic graph theory concepts. Some of these concepts are detailed in individual chapters. We then describe sequential, parallel, and distributed graph algorithms in sequence in three chapters. In each chapter, we first provide the main concepts about the algorithm method and then provide a number of examples on graphs using the method mentioned. For example, in the sequential algorithm methods, we give a greedy graph algorithm while describing greedy algorithms. This part basically forms the background for parts II and III.
- *Basic Graph Algorithms*: This part contains the core material of the book. We look at the main topics in graph theory at each chapter which are trees and graph traversals; weighted graphs; connectivity; matching; subgraphs; and coloring. Here, we leave out some theoretical topics of graph theory which do not have significant algorithms. The topics we investigate in the book allow algorithmic methods conveniently and we start each chapter with brief theoretical background for algorithmic analysis. In other words, our treatment of related graph theoretical concepts is not comprehensive as our main goal is the study of graph algorithms rather than graph theory on its own. In each chapter, we first describe sequential algorithms and this part is one place in the book that we try to be as comprehensive as possible by describing most of the well-established algorithms of the topic. We then provide only sample parallel and distributed algorithms on the topic investigated. These are typically one or two well-known algorithms rather than a comprehensive list. In some cases, the parallel or distributed algorithms at hand are complicated. For such problems, we give a survey of algorithms with short descriptions.
- *Advanced Topics*: We present recent and more advanced topics in graph algorithms than Part II in this section of the book starting with algebraic and dynamic graph algorithms. Algebraic graph algorithms commonly make use of the matrices associated with a graph and operations on them while solving a graph problem. Dynamic graphs represent real networks where edges are inserted and removed from a graph in time. Algorithms for such graphs, called dynamic graph algorithms, aim to provide solutions in shorter time than running the static algorithm from scratch.

  Large graphs representing real-life networks such as biological and social networks tend to have interesting and unexpected properties as we have outlined. Study of such graphs has become a major research direction in network science recently. We therefore considered it to be appropriate to have two chapters of the book dedicated for this purpose. Algorithms for these large graphs have somehow different goals, and community detection which is finding dense regions in these graphs has become one of the main topics of research. We first provide a chapter on general description and analysis of these large graphs along with algorithms to compute some important parameters. We then review basic complex network types with algorithms used to solve fundamental problems in these networks. The final chapter is about describing general guidelines on how to search a graph algorithm for the problem at hand.

We conclude this chapter by emphasizing the main goals of the book once more. First, it would be proper to state what this book is not. This book is not intended as a graph theory book, or a parallel computing book or a distributed algorithms book on graphs. We assume basic familiarity with these areas although we provide a brief and dense review of these topics as related to graph problems in Part I. We describe basic graph theory including the notation and basic theorems related to the topic at the beginning of each chapter. Our emphasis is again on graph theory that is related to the graph algorithm we intend to review. We try to be as comprehensive as possible in the analysis of sequential graph algorithms but we review only exemplary parallel and distributed graph algorithms. Our main focus is guiding the reader to graphs algorithms by investigating and studying the same problem from three different views: a thorough sequential, typical parallel, and distributed algorithmic approaches. Such an approach is effective and beneficial not only because it helps to understand the problem at hand better but also it is possible to convert from one approach to another saving significant amount of time compared to designing a completely new algorithm.