CHAPTER II
PRELIMINARIES

In this chapter, we provide some background on the mobility model in a two-dimensional grid, the three related decision problem definitions, breadth-first search, growth of functions, plus relevant complexity theory and graph theory. All of these items will be used in later chapters.

2.1 Preliminaries

- Let $N = \{1,2,3,...\}$.
- Let S be the set. The set S^k, for $k \in N$, is the k-fold Cartesian product of the set S.

2.2 Definition of the Mobility Model

The following definition of the mobility model is taken from [1] with permission from the authors, as is part of the ensuing discussion. We define the model here to operate on a 2-dimensional grid. A mobility model is an 8-tuple (S, D, U, L, R, V, C, O), where

1. The set $S = \{s_1, s_2, ..., s_m\}$ is a finite collection of **sources**, where $m \in N$. The value m is the **number of sources**. Corresponding to each source s_i, for $1 \le i \le m$, an **initial location** (x_i, y_i) is specified, where $x_i, y_i \in N$.
2. The set $D = \{000,001,010,101,110\}$ is called the **directions**, and these values correspond to no movement, east, west, south, and north, respectively.
3. The set $U = \{u_1, u_2, ..., u_p\}$ is a finite collection of *mobile devices*, where $p \in N$. The set U is called the set of **users**. The value p is called the **number of users**.

Corresponding to each user u_i, for $1 \le i \le p$, an initial location (x_i, y_i) is specified, where $x_i, y_i \in N$.

4. The set $L = \{l_1, l_2, ..., l_t\}$ is a finite collection of "bit strings," where $t \in N$ and $l_i \in D^t$ for $1 \le i \le t$. Each group of three bits in l_i beginning with the first three defines a step in a given direction for the user u_i's movement or no movement at all if the string is 000. The value *t* is called the **duration of the model.**
5. Let $t(i) \in N$ for $1 \le i \le m$. The set $R = \{r_1, r_2, ..., r_m\}$ is a finite collection of "bit strings," where $r_i \in D^{t(i)}$ for $1 \le i \le m$. Each group of three bits in r_i beginning with the first three defines a step in a given direction for the source s_i's movement or no movement at all if the string is 000. The set *R* is called the **random walks** of the mobility model.
6. The set $V = \{v_1, v_2, ..., v_m\}$ is a finite collection of numbers, where $v_i \in N$. The value v_i is the corresponding number of steps from r_i per unit time that s_i will take. This set is called the **velocities**.
7. The set $C = \{c_1, c_2, ..., c_m\}$ is a finite collection of lengths, where $c_i \in N$. The value c_i is the corresponding diameter of the circular coverage of source s_i. This set is called the **coverages**.
8. The set $O = \{(x_1, y_1, x_2, y_2) \mid x_1, y_1, x_2, y_2 \in N, x_2 > x_1 \text{ and } y_2 > y_1\}$ is a finite collection of rectangles in the plane. This set is called the **obstacles**.

Several remarks are in order about the definition. We have based the model on a 2-dimensional grid for simplicity, but it would certainly be interesting to extend the model to the 3-dimensional case. The sources in *S* correspond to wireless access points. They are broadcasting and receiving signals. Although real mobile sources do not move in discrete steps, by using a fine enough grid, we lose little information by assuming that the sources are always at grid point locations.

The set *D* represents the usual four possible directions for movement in the grid, plus no movement at all. The set *U* represents users with mobile devices. We have modeled the movement of the users by random walks contained in the set *L*. Although we have assumed that all the walks have the same length, this convention is not really a restriction since we can simply pad out shorter walks using the no movement bit string 000 from *D*. For the sake of simplicity, we have assumed that all users travel at the same velocity. The users move to their new locations in unit steps instantaneously.

We have modeled the movement of the sources by random walks contained in the set *R*. To accommodate for different velocities, the walks in *R* have different lengths. In real-life situations mobile-access points move around at different speeds, for example, a hummer may be traveling at speeds in excess of 100 kilometers per hour, whereas an elephant working his way through dense brush may be moving at 1 kilometer per hour. We represent the relative speeds of the sources by natural numbers contained in the set *V*. Of course, a given source may not always travel at a constant velocity. It would be worth examining an extension of the model where any source's speed can change over time.

Different sources will broadcast at different signal strengths depending on a variety of factors, the main one being the amount of power available. We have represented the various signal strengths by specifying the diameter of a circle c_i for each source indicating where its signal can be received. This region is called the **coverage area**. Since buildings and other obstacles may interfere with signal transmission, the model incorporates a set of obstacles *O*. To simplify matters, we only permit rectangular obstacles.

We now turn our attention to the communication protocol which will allow us to illustrate how the model is used. The following communication protocol is needed so that the model works as intended. The sources are always on; they are always broadcasting and listening. Users with mobile devices are moving in and out of the range of each other and various sources. Mobile devices would like to communicate (send and receive messages) with one another. We specify the manner in which they may communicate in what follows. Let $k > 2$ and $k \in N$.

- At a given instance in time any two sources with overlapping-coverage areas may communicate with each other in full-duplex fashion as long as the intersection of their overlapping-coverage area is not completely contained inside obstacles. We say that these two sources are **currently in range**. A series $s_1, s_2, ..., s_k$ of sources are said to be currently in range if s_i and s_{i+1} are **currently in range** for $1 \le i \le k-1$.
- Two mobile devices cannot communicate directly with one another.
- A mobile device D_1 always communicates with another mobile device D_2 through a source or series of sources as defined next. The mobile devices D_1 at location (x_1, y_1) and D_2 at location (x_2, y_2) **communicate through a single source** *s* located at (x_3, y_3) if at a given instance in time the lines between points (x_1, y_1) and (x_3, y_3) and points (x_2, y_2) and (x_3, y_3) are within the area of coverage of *s*, and do not intersect with any obstacle from *O*. The mobile devices D_1 at location (x_1, y_1) and D_2 at location (x_2, y_2) **communicate through a series of sources** s_1 at location (a_1, b_1), s_2 at location (a_2, b_2),…, and s_k at location (a_k, b_k) that are currently in range if the line between points (x_1, y_1) and (a_1, b_1) is inside s_1's coverage area and does not intersect any obstacle from *O* and the line between points (x_2, y_2) and (a_k, b_k) is inside s_k's coverage area and does not intersect any obstacle from *O*.

2.3 A Sample Instance of the Model

To illustrate the mobility model, we provide a specific instance next; see Figure 2.1. Let $M = (S, D, U, L, R, V, C, O)$ be defined as follows:

1. Let $S = \{s_1, s_2, s_3, s_4\}$ with initial locations (2, 5), (5, 5), (6, 4), and (5, 2), respectively.
2. Let $D = \{000, 001, 010, 101, 110\}$.

3. Let $U = \{u_1, u_2, u_3\}$ with initial locations (3, 4), (2, 1), and (6, 2), respectively.
4. Let $t = 3$ and $L = \{l_1, l_2, l_3\}$, where $l_i = \{000,000,000\}$ for $1 \le i \le 3$.
5. Let $R = \{r_1, r_2, r_3, r_4\}$. For clarity Figure 2.1 only illustrates $r_1 = \{101,001,101\}$ and omits the other r_i's, which we assume are all (000,000,000), except for r_2 which is twice as long.
6. Let $V = \{1,2,1,1\}$.
7. Let $C = \{2,2,2,4\}$.
8. Let $O = \{(2,1,4,2)\}$.

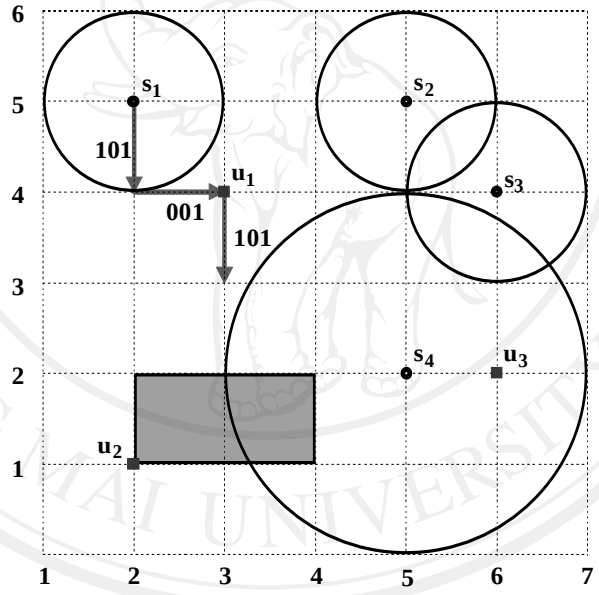

Figure 2.1 Sample instance of the mobility model.

Figure 2.1 illustrates this instance of the model *M* graphically. In this case there are three stationary users. There are four sources s_1, s_2, s_3, and s_4 located at (2, 5), (5, 5), (6, 4), and (5, 2), respectively. An obstacle in this figure is the rectangle defined by the lower-left coordinate (3, 2) and the upper-right coordinate (5, 3). Sources s_1, s_2, and s_3 each have a coverage with a diameter 2, and s_4 has a coverage with a diameter 4. The steps of s_1 at initial location (2, 5) are defined by r_1. In this case, s_1 moves south in the

first step, east in the second step, and south in the third step. The moves are made with a velocity of $v_1 = 1$, or one step per unit of time.

Note that initially, for example, sources s_2 and s_3 are currently in range, sources s_2, s_3, and s_4 are a series of sources currently in range, and sources s_1 and s_2 are not currently in range. Initially, users u_1 and u_3 cannot communicate either by a source or a series of sources. After three steps, u_1 can communicate with u_3 through the series of sources s_1 and s_4.

2.4 Problem Definitions

In this section three interesting problems related to the mobility model are defined. The definitions are from [1].

User Communication Problem (UCP)

INSTANCE: A mobility model (*S*, *D*, *U*, *L*, *R*, *V*, *C*, *O*), two designated users u_a and u_b from *U*, a time $k \le \delta_1$, and $|O| \le \delta_2$, where $\delta_1, \delta_1 \in N$.

QUESTION: Can users u_a and u_b communicate at time *k*?

Sources Reachability Problem (SRP)

INSTANCE: A mobility model (*S*, *D*, *U*, *L*, *R*, *V*, *C*, *O*), two designated sources s_a and s_b from *S*, a time $k \le \delta_1$, and $|O| \le \delta_2$, where $\delta_1, \delta_1 \in N$.

QUESTION: Are sources s_a and s_b in range at time *k*?

Access Point Location Problem (APLP)

INSTANCE: A mobility model (*S*, *D*, *U*, *L*, *R*, *V*, *C*, *O*), two designated users u_a and u_b from *U*, a source diameter $d \le \delta_1$, and a natural number $k \le \delta_2$ where *S* is an empty set and $\delta_1, \delta_1 \in N$.

QUESTION: Can users u_a and u_b communicate throughout the duration of the model if k nonredundant sources of diameter d are placed appropriately in the grid and each source is accessed exactly once?

2.5 Breadth-First Search

Breadth-first search (BFS) is one of the simplest algorithms for searching a graph and the archetype for many important graph algorithms [14]. Prim's minimum-spanning-tree algorithm and Dijkstra's single-source shortest-paths algorithm use ideas similar to those used in standard breadth-first search.

Given a graph $G = (V, E)$ and a distinguished **source** vertex s, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from s. BFS computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a "breadth-first tree" with root s that contains all vertices reachable from s. For any vertex v reachable from s, the path in the breadth-first tree from s to v corresponds to a "shortest path" from s to v in G, that is, a path containing the fewest number of edges possible. The algorithm works on both directed and undirected graphs.

Breadth-first search is so named because it expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from s before discovering any vertices at distance $k + 1$.

To keep track of progress, breadth-first search colors each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is **discovered** the first time it is encountered during the search, at which time it becomes nonwhite. Gray and black vertices, therefore, have been discovered, but breadth-first search distinguishes between them to ensure that the search proceeds in a breadth-first manner. If $(u,v) \in E$ and vertex u is black, then vertex v is either gray or black; that is, all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.

Breadth-first search constructs a breadth-first tree, initially containing only its root, which is the source vertex *s*. Whenever a white vertex *v* is discovered in the course of scanning the adjacency list of an already discovered vertex *u*, the vertex *v* and the edge (*u*, *v*) are added to the tree. We say that *u* is the **predecessor** or **parent** of *v* in the breadth-first tree. Since a vertex is discovered at most once, it has at most one parent. Ancestor and descendant relationships in the breadth-first tree are defined relative to the root *s* as usual: if *u* is on a path in the tree from the root *s* to vertex *v*, then *u* is an ancestor of *v* and *v* is a descendant of *u*.

```
        BFS(G, s)
1.      for each vertex u ∈ V[G] − {s}
2.            do color[u] ← WHITE
3.               d[u] ← ∞
4.               π[u] ← NIL
5.      color[s] ← GRAY
6.      d[s] ← 0
7.      π[s] ← NIL
8.      Q ← φ
9.      ENQUEUE(Q, s)
10.     while Q ≠ φ
11.           do u ← DEQUEUE(Q)
12.              for each v ∈ Adj[u]
13.                    do if color[v] = WHITE
14.                          then color[v] ← GRAY
15.                               d[v] ← d[u] + 1
16.                               π[v] ← u
17.                               ENQUEUE(Q, v)
18.              color[u] ← BLACK
```

Figure 2.2 BFS algorithm.

The breadth-first-search procedure BFS presented in Figure 2.2 assumes that the input graph $G = (V, E)$ is represented using adjacency lists. The algorithm maintains several additional data structures with each vertex in the graph. The color of each vertex

$u \in V$ is stored in the variable *color*[u], and the predecessor of u is stored in the variable $\pi[u]$. If u has no predecessor (for example, if $u = s$ or u has not been discovered), then $\pi[u]$ = NIL. The distance from the source s to vertex u computed by the algorithm is stored in $d[u]$. The algorithm also uses a first-in, first-out queue Q to manage the set of gray vertices.

The procedure BFS works as follows. Lines 1–4 color every vertex white, set $d[u]$ to be infinity for each vertex u, and set the parent of every vertex to be NIL. Line 5 colors the source vertex s gray, since it is considered to be discovered when the procedure begins. Line 6 initializes $d[s]$ to 0, and line 7 sets the predecessor of the source to be NIL. Lines 8–9 initialize Q to the queue containing just the vertex s.

The **while** loop of lines 10–18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had their adjacency lists fully examined. This **while** loop maintains the following invariant: At the test in line 10, the queue Q consists of the set of gray vertices.

Although we will not use this loop invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s. Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q. The **for** loop of lines 12–17 considers each vertex v in the adjacency list of u. If v is white, then it has not yet been discovered, and the algorithm discovers it by executing lines 14–17. It is first grayed, and its distance $d[v]$ is set to $d[u]+1$. Then, u is recorded as its parent. Finally, it is placed at the tail of the queue Q. When all the vertices on u's adjacency list have been examined, u is blackened in lines 11–18. The loop invariant is maintained because whenever a vertex is colored gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also colored black (in line 18).

The results of breadth-first search may depend upon the order in which the neighbors of a given vertex are visited in line 12: the breadth-first tree may vary, but the distances d computed by the algorithm will not.

Analysis

Here we provide an analysis of the BFS algorithm as done in [14]. For analyzing the running time of the BFS algorithm on an input graph $G = (V, E)$. We use aggregate analysis. After initialization, no vertex is ever whitened, and thus the test in line 13 ensures that each vertex is enqueued at most once, and hence dequeued at most once. The operations of enqueuing and dequeuing take $O(1)$ time, so the total time devoted to queue operations is $O(V)$. Because the adjacency list of each vertex is scanned only when the vertex is dequeued, each adjacency list is scanned at most once. Since the sum of the lengths of all the adjacency lists is $\theta(E)$, the total time spent in scanning adjacency lists is $O(E)$. The overhead for initialization is $O(V)$, and thus the total running time of BFS is $O(V + E)$. Thus, breadth-first search runs in time linear in the size of the adjacency-list representation of G.

2.6 Growth of Functions

In this section we review some material on the growth of functions. Much of this review is taken from [14]. The order of growth of the running time of an algorithm gives a simple characterization of the algorithm's efficiency and also allows us to compare the relative performance of alternative algorithms. Once the input size n becomes large enough, merge sort, with its $\theta(n \lg n)$ worst-case running time, beats insertion sort, whose worst-case running time is $\theta(n^2)$. Although we can sometimes determine the exact running time of an algorithm the extra precision is not usually worth the effort of computing it. For large enough inputs, the multiplicative constants and lower-order terms of an exact running time are dominated by the effects of the input size itself.

When we look at input sizes large enough to make only the order of growth of the running time relevant, we are studying the **asymptotic** efficiency of algorithms. That is, we are concerned with how the running time of an algorithm increases with the size of the input *in the limit*, as the size of the input increases without bound. Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs.

Asymptotic Notation

The notations we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers $N = \{0,1,2,...\}$. Such notations are convenient for describing the worst-case running-time function $T(n)$, which is usually defined only on integer input sizes. It is sometimes convenient, however, to *abuse* asymptotic notation in a variety of ways. For example, the notation is easily extended to the domain of real numbers or, alternatively, restricted to a subset of the natural numbers. It is important, however, to understand the precise meaning of the notation so that when it is abused, it is not *misused*. This section defines the basic asymptotic notations and also introduces some common abuses.

1. $\theta - notation$

The $\theta - notation$ asymptotically bounds a function from above and below. Let us define what this notation means. For a given function $g(n)$, we denote by $\theta(g(n))$ the set of functions

$$\theta(g(n)) = \left\{ \begin{array}{c} f(n) : \exists \ positive\ constants\ c_1, c_2\ and\ n_o \\ such\ that\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n)\ for\ n \ge n_o \end{array} \right\}$$

A function $f(n)$ belongs to the set $\theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n. Because $\theta(g(n))$ is a set, we could write "$f(n) \in \theta(g(n))$" to indicate that $f(n)$ is a member of $\theta(g(n))$. Instead, we will usually write "$f(n) = \theta(g(n))$" to express the same notion.

2. $O - notation$

The $O - notation$ used when we have only an asymptotic upper bound. For a given function $g(n)$, we denote by $O(g(n))$ (pronounced "big-Oh of g of n" or sometimes just "Oh of g of n") the set of functions.

$$O(g(n)) = \left\{ \begin{array}{c} f(n) : \exists \ positive\ constants\ c\ and\ n_o \\ such\ that\ 0 \le f(n) \le cg(n)\ for\ n \ge n_o \end{array} \right\}$$

We use *O*-notation to give an upper bound on a function, to within a constant factor. For all values *n* to the right of n_0, the value of the function *f*(*n*) is on or below *g*(*n*).

3. $\Omega - notation$

The $\Omega - notation$ used when we have only an asymptotic lower bound. For a given function *g*(*n*), we denote by Ω(*g*(*n*)) (pronounced "big-omega of *g* of *n*" or sometimes just "omega of *g* of *n*") the set of functions

$$\Omega(g(n)) = \left\{ \begin{array}{l} f(n) : \exists \ positive\ constants\ c\ and\ n_o \\ such\ that\ 0 \le cg(n) \le f(n)\ \ for\ n \ge n_o \end{array} \right\}$$

For all values *n* to the right of n_0, the value of *f*(*n*) is on or above *cg*(*n*).

2.7 Complexity Theory

The following relevant complexity theory is taken from [12–14] and will be used in classifying class and proofs of the problems we are interested. Almost all the algorithms we have studied thus far have been *polynomial-time algorithms*: on inputs of size n, their worst-case running time is $O(n^k)$ for some constant *k*. It is natural to wonder whether all problems can be solved in polynomial time. The answer is no. For example, there are problems, such as Turing's famous "Halting Problem," that cannot be solved by any computer, no matter how much time is provided. There are also problems that can be solved, but not in time $O(n^k)$ for any constant *k*. Generally, we think of problems that are solvable by polynomial-time algorithms as being tractable, or easy, and problems that require superpolynomial time as being intractable, or hard.

NP-complete problems is an interesting class of problems, whose status is unknown. No polynomial-time algorithm has yet been discovered for an *NP*-complete problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them. This so-called *P* versus *NP* question has been one of the deepest, most perplexing open research problems in theoretical computer science since it was first posed in 1971.

2.8 Graph Theory

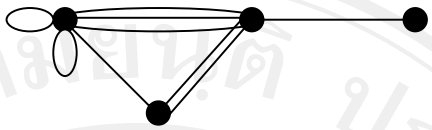

Figure 2.3 Sample graph.

In this section we discuss the basis concepts from graph theory needed in this thesis [11, 17]. Intuitively, a diagram that can be represented by means of points (vertices) and lines (edges) is called a **graph**. The **degree** of a vertex is the number of edges which have the vertex as an endpoint. The point we are trying to make is that a graph is a representative of a set of points and of the way they are join up, and that for our purposes any metrical properties are irrelevant. From this point of view, any two graphs which represent the same situation will be regarded as essential the same graph. More precisely, we shall say that two graphs are **isomorphic** if there is a one-one correspondence between their vertices which as the property that two vertices are joined by an edge in one graph if and only if the corresponding vertices are joined by an edge in the other.

It is worth pointing out that the graph we have been discussing so far is a particular 'simple' graph. A simple graph is a graph that exactly one edge is allowed for a given pair of vertices. If there is more than one edge joining a point pair, the edges are called **multiple edges.** If we want to build a graph by drawing an edge from a point to itself, this is usually called a **loop**. Graphs containing no loops or multiple edges will be referred to as **simple graphs.**

The study of **directed graphs** (or **digraphs**, as we shall usually abbreviate them) arises out of the question, 'what happens if all of the roads have one-way streets?' The directions of the one-way streets being indicated by arrows. Note if not all of the streets are one-way, then we can obtain a digraph by drawing for each two-way road two directed edges, one in each directions.

Much of graph theory involves the study of walks of various kinds, a **walk** being essentially a sequence of edges, one follow one another. A walk in which no vertex appears more than once is a **path**. A path that first and last points are the same is called a **circuit.** A graph in which any two vertices are connected by a path is called a **connected graph.** We shall also be interested in connected graph in which there is only one path connecting each pair of vertices; such graphs are called **tree**. Any graph which can be redrawn without crossing is called **planar graph**.