CME 305: Discrete Mathematics and Algorithms

Instructor: Professor Aaron Sidford (sidford@stanford.edu)

January 11, 2018

Lecture 2 - Graph Theory Fundamentals - Reachability and Exploration[1]

In this lecture we take a step back from the previous lecture and review some basic graph theory and graph exploration algorithms for exploring and decomposing graphs. As we argued in the last lecture, graphs are fundamental and pervasive combinatorial objects. They are one of the simplest types of structured of finite sets: if you have a set of n items, $i_1, ..., i_n$ and pairwise relationships between these items (one of the simplest types of relationships you would imagine) then the natural object to model these relationship is a graph. We will be design algorithms for graph extensively over the next few lectures and they will appear repeatedly throughout the course. Consequently, the material in this lecture will be used extensively throughout the course.

1 Basic Graph Definitions

We denote a graph by $G = (V, E)$. Throughout the course V will denote a finite set we call the *nodes* or *vertices* and E is a finite set of pairs of vertices we call the edges. We think of each edge as encoding a relationship between a pair of the nodes in V.

There are two main cases of graphs we consider. In the first case, the pairs are un-ordered, i.e. $E = \{\{u_1, v_1\}, \{u_2, v_2\}, ..., \{u_m, v_m\}\}$. In this case, we call the graph *undirected*. Here we think that the relationship that edges are encoding is symmetric, there is an edge from u to v, i.e. $\{u, v\} \in E$, i.e. u and v are related, if and only if there is an edge from v to u, since $\{u, v\} = \{v, u\}$.

In the second case, the pairs are ordered, $E = \{(u_1, v_1), (u_2, v_2), ..., (u_m, v_m)\}$. In this case we call the graph *directed*. Here the graph is possibly encoding asymmetric relationships as u related to v does not necessarily imply v related to u. We will say a directed edge $(u, v) \in E$ is pointed from u to v, we occasional call u the *tail* of the edge and v the *head* of the edge.

A quick warning is warranted here. Throughout these notes (and the literature in general) we may be a little informal with the notation and use (u, v) in undirected graphs. We do this both for convenience and for when we have results that apply to both directed and undirected graphs. The meaning should be clear from context, however if you would ever like clarification here or in lecture, please do not hesitate to ask.

We say a graph has *self loops* if (u, u) is an edge for some u and we say a graph has *multi-edges* if edges may repeat, and may refer to a graph where multi-edges are allowed as a *multi-graph*. We call a graph *simple* if it has no self-loops or multi-edges. Note that a simple undirected graph can have at most $\binom{n}{2}$ edges, a simple directed graph may have $2\binom{n}{2}$ edges, and a multi-graph can have an arbitrary number of edges. Sometimes we may look at the *undirected graph associated with a directed graph* and by this we simply mean the graph resulting from ignoring the orientation of edges. Sometimes we may look at the *directed graph associated with undirected graph* which is when take every undirected edge and replace with a directed edge in each direction.

Finally, we may consider weighted graphs where we simply have some vector $w \in \mathbb{R}^E$ that assigns a *weight* or *length* to every edge. These are context specific and we will talk about them when they come up. Occasionally we will talk about unweighted graphs when we wish to emphasize that they have no weights.

[1] These lecture notes are a work in progress and there may be typos, awkward language, omitted proof details, etc. Moreover, content from lectures might be missing. These notes are intended to converge to a polished superset of the material covered in class so if you would like anything clarified, please do not hesitate to post to Piazza and ask.

2 Paths and Cycles

A common operation we perform on graphs is following a sequence of edges from one vertex to the next.

Definition 1 (Path)**.** For a graph $G = (V, E)$ a set of edges $(u_1, v_1), ..., (u_k, v_k) \in E$ is called a *path* if $u_{i+1} = v_i$ for all i. The *path* is called *simple* if the v_i are distinct from each other and u_i.

Note in the definition above, if the graph was undirected we should have written $\{u_1, v_1\}, ..., \{u_k, v_k\}$if we wished to be more formal. We are overloading the ordered pair notation so that we can give a definition that is simultaneously correct for directed and undirected graphs and can write things a little more compactly. We will do this frequently in these notes and the course more broadly.

One common class of paths that will occur repeatedly are called cycles.

Definition 2 (Cycles)**.** A set of edges $(u_1, v_1), ..., (u_k, v_k) \in E$ is called a *cycle* if $(u_1, v_1), ..., (u_{k-1}, v_{k-1})$ is a simple path and $v_k = u_1$.

We define the *length* of a path or *cycle* to be the number of edges in it. If the graph is weighted, then we define the *length* to be the sum of the weights of the edges.

3 Reachability and Connectivity

We use paths to extend edge relations to a relation on arbitrary pairs of nodes in the graph. We use them to define things like reachability and distance.

Definition 3 (Reachability)**.** We say the v is *reachable* from u in graph G if and only if there is a path from u to v.

Definition 4 (Distance)**.** Furthermore, we define the *distance* from u to v in a graph G, denoted $d_G(u, v)$ or just $d(u, v)$ when the graph is clear from context, to be the length of the shortest path from u to v or ∞ if v is not reachable from u.

Note that if we think of think of edges as encoding a relation R on pairs of vertices, i.e. uRv if and only if $(u, v) \in E$ then this relation is not transitive, i.e. uRv and vRw does not necessarily imply that uRw. However, it is the case that if v is *reachable* from u and w is *reachable* from v then w is *reachable* from v (just concatenate the paths). In fact, it is not too hard to show, that reachability as a relation (the pairs that are reachable from each other) is the transitive closure of the standard relation induced by the edges, i.e. it is the minimum transitive relation that contains the edge relation.

Claim 5. Reachability is the transitive closure of the relation imposed by the edges of a graph, i.e. if given a graph G we add an edge (u, v) whenever u ca reach v this new graph is the edge minimal graph containing the original graph where a (u, v) and (v, w) edge in the new graph imply a (u, w) edge in the new graph.

Now it is natural to reason about the structure of the reachability relation. If we make a graph where for every pair of vertices we add an edge from u to v if and only if u can reach v. What do these graphs look like? For undirected graphs, it is easy to say that this will simply make a variety of disjoint cliques or complete graphs. This motivates the following definitions of connectivity for undirected graphs.

Definition 6 (Connectivity)**.** An undirected graph is *connected* if and only if every distinct pair of vertices are reachable from each other. A subgraph of this graph is a called connected component of this graph if it is a vertex maximal connected subgraph (i.e. it is a connected subgraph and all vertices not in the subgraph are not reachable in the original graph).

Now when we reason about connectivity, we are reasoning about equivalence classes in this transitive closure. We say a subgraph is connected if all the vertices in it can reach each other and in a directed graph we say it is (strongly) connected, emphasizing the asymmetry that needs to be fixed. These induce equivalence classes and a natural way to partition a graph. The connected components of an undirected graph are the maximal sets of vertices that are all connected to each other and the same goes for strongly connected components. It is easy to see that undirected graphs can be partitioned into them.

Claim 7. Every undirected graph has a unique vertex partition into connected components.[2]

The same definition applies to directed graphs, but is often called strong connectivity to emphasize its slightly different structure do to its asymmetric nature.

Definition 8 ((Strong) Connectivity)**.** A directed graph is *strongly connected* if and only if every distinct pair of vertices are reachable from each other. A subgraph of this graph is a called a *strongly connected component* of this graph if it is a vertex maximal strongly connected subgraph (i.e. it is a strongly connected subgraph and all vertices not in the subgraph are not both reachable in the original graph and can reach the subgraph).

Note that directed graphs can also be partitioned into strongly connected component, however it is no longer the case that having many edges implies larger connected components. There can be fairly large graphs (edge-wise) that have interesting connectivity structure and no connected components. In fact such a graph can have $\frac{n(n-1)}{2}$ edges and no cycles (called *tournament graphs*). These graphs where all strongly connected components are of size 1 are called DAGs.

Definition 9 (DAG)**.** A directed graph which contains no cycles (and therefore no strongly connected components) is a called a *directed acyclic graph* (DAG).

It can be shown that after computing all strongly connected components in a directed graph and contracting them, the resulting graph is a DAG. Consequently, all directed graphs are DAGs of strongly connected components in this sense.

4 Trees, Forests, and Arborescences

So how do we show that a graph is connected? What is the minimum subgraph that proves a graph an undirected graph is connected? These are called trees. We start with a definition that is fairly intuitive for a tree and then show it has a number of desirable properties.

Definition 10 ((Spanning) Tree)**.** A *(spanning) tree* is an undirected graph where for every pair of vertices there is a unique simple path that connects them.

More generally, any undirected graph containing no cycles we call a *forest*. The word *spanning* in spanning tree is used above when wish to emphasize that all the vertices are in it. Note that forest are simply vertex-disjoint unions of trees. Sometimes that distinction between forests and trees will not always be made abundantly clear, but often the algorithms extend naturally by considering each tree separately.

There are many equivalent characterizations of trees and we give a few below.

Theorem 11 (Tree Characterizations)**.** *If a n-node undirected graph $G = (V, E)$ has any two of the following properties, then it implies the third and is a tree.*

[2] Note that isolated vertices are trivially connected components as they have no distinct pairs of vertices.

1. *G is connected*
2. *G does not contain a cycle*
3. *G has $n-1$ edges*

Proof. First we show that (1) and (2) hold if and only if G is a tree. Suppose that G is a tree, then it is connected by assumption and cannot contain a cycle since this would imply there are two disjoint paths that can be used to reach the vertices on the cycle. On the other hand if G is a connected and does not contain a cycle, then it contains at least one path between every pair of nodes by connectivity and cannot contain more than 1 as this would imply the graph contains a cycle. To see this last claim formally, let $(u_1, v_1), ..., (u_k, v_k)$ and $(\bar{u}_1, \bar{v}_1), .., (\bar{u}_{\bar{k}}, \bar{v}_{\bar{k}})$ denote two distinct disjoint paths from u to v, i.e. $u_1 = \bar{u}_1 = u$ and $v_k = \bar{v}_{\bar{k}} = v$. Let j be the minimum index for which $v_j \neq \bar{v}_j$ and let $l, \bar{l} > j$ be the smallest indices larger than j such that $v_l = \bar{v}_{\bar{l}}$ (note they must exist since $v_k = \bar{v}_{\bar{k}}$). Note that $(u_j, v_{j+1}), ..., (u_{l-1}, v_l)(\bar{v}_{\bar{l}}, \bar{u}_{\bar{l}-1}), ..., (\bar{v}_{j+1}, \bar{u}_j)$, i.e. following the first path from u_j to v_l and then the second path back, is a cycle.

Next we show that (1) and (2) imply (3). Consider adding the edges of G satisfying (1) and (2) to the empty graph one at a time. We start with the empty graph with n-isolated vertices and no-connected components. Every time we add edge it cannot be within a connected component as this would create a cycle. Consequently, the number of connected components must increase by 1. Since the number of connected components in G is exactly 1 exactly $n-1$ edges must be added and G therefore must have $n-1$ edges.

Next we show that (1) and (3) imply (2). Again, consider adding the edges of a graph satisfying (1) and (3) one at a time. A cycle forms only if we add the edge betwen two vertices in the same connected component. However, if we do this then the graph will have more than one connected component at the end. Consequently, edges are only added between different connected components and the resulting graph doesn't contain a cycle.

Finally we show that (2) and (3) imply (1). Again, consider adding the edges of the graph one at a time. As we have argued since cycles are not formed connected components increase until the graph is connected. □

This gives a number of other characterizations. For example, from this theorem it is immediate that trees are edge minimal connected subgraphs, since if we remove edges from cycles until none are left the resulting graph is a tree. Consequently, when we wish to prove that an undirected graph is connected, it suffices to compute a spanning tree. We will look at these a lot.

Trees are a nice structure to think about hierarchical data. When we do this, it is often useful to think of the tress as being rooted at a single vertex and the rest of the vertices being arranged in increasing distance or depth from the root. We typically draw such trees with the root at the top and the nodes reachable from it drawn downward with increasing distance. The following definitions and notations are useful for reasoning about such trees.

Definition 12 (Rooted Tree)**.** A *rooted tree* is a tree $T = (V, E)$ together with a special vertex $r \in V$, called the *root*. We have the following additional definitions associated with rooted trees:

- We call $d_T(r, v)$ the *depth* of node v in the tree.
- For any vertex $v \in V$ we call the vertices on the simple path from v to r (including v and r) the *ancestors* of v, denoted $\text{anc}_T(v)$. We call the $\text{anc}_T(v) \setminus v$ the proper ancestors of v.
- We call the vertices that v is an ancestor of the *descendents* of v, denoted $\text{desc}_T(v)$ and similarly define the proper descendents of v as $\text{desc}_T(v) \setminus v$.

- We call the first node on the path from v to the root the parent of v and we call the nodes that v is a parent of the *children* of v.

Note that characterizing the edge minimal subgraph of a directed graph that preserves strong connectivity is not quite as simple. The number of edges in an edge-minimal strongly connected graph can vary between n and $2(n-1)$. However, characterizing the edge minimal subgraph that preserves reachability from a node does have only $n-1$ edges and is a type of tree called an arborescence.

Definition 13 (Arborescence)**.** An directed arborescence is a directed graph along whose associated undirected graph is a a tree and for which every edge is oriented away from some root r.

Equivalently, we could have defined an arborescence as a graph where for some vertex $r \in V$ there is a unique path from r to v for every vertex v. Such arborescences has similar properties as trees.

Finally, when we wish to talk about a tree that preserves reachability from some node r I will call such a tree a *reachability tree* to avoid specifying whether or not the graph is directed and the tree is truly an undirected tree or a directed arborescence.

5 Graph Exploration

So how do we actually explore a graph efficiently and reason about its basic connectivity structure? More formally, how do we actually compute reachability trees, connected components, and strongly connected graphs?

There are two fundamental popular methods, known as breadth first search and DFS. Starting from a vertex these procedures construct a reachability tree (or in a directed graph an arbrorescense) but in fairly different ways.

We begin with the breadth first search algorithm, or BFS, algorithm. This algorithm, starts at a vertex s and then looks at every vertex reachable from s using a single edge and for each such vertex adds one such edge to the tree. The algorithm then repeats this, for all the vertices that were just added adding one new edge for every new vertex that can be reached. This is repeated until no new vertices can be reached. This algorithm essentially starts from s and grows the tree one layer at a time.

Algorithm 1 Breadth First Search

Input: graph $G = (V, E)$ and start vertex s
Mark all vertices as `unexplored` and let $T := \emptyset$
Let $i = 0$ and $S_i = \{s\}$
while $S_i \neq \emptyset$ **do**
  Set $S_{i+1} := \emptyset$
  for each edge $(u, v) \in E$ with $u \in S_i$ **do**
    if v is `unexplored` **then**
      add (u, v) to T and add v to S_{i+1}
      mark v as `explored`
    end if
  end for
  Let $i := i + 1$
end while
Return T

On the other hand the depth first search starts from s follows an edge from s follows an edge from that vertex and repeats so long as the edge points to a new vertex. Whenever the algorithm cannot follow an edge to a new vertex it then goes back to the previous vertex trying to do the same thing. The algorithm has the following simple specification.

Algorithm 2 Depth First Search

Input: graph $G = (V, E)$ and start vertex s
Mark all vertices as `unexplored` and let $T := \emptyset$
Invoke `DFS`(s) defined below.
Return T

`DFS`(u):
Mark u as `explored`
for each edge $(u, v) \in E$ **do**
  if v is `unexplored` **then**
    add (u, v) to T
    invoke $DFS(v)$
  end if
end for

In the remainder of this section we analyze properties of these algorithms and show how they can be used to compute a variety of interesting properties of a graph. We focus on what these procedures do to compute reachability from s. If we wish to find all the connected components of the graph we could simply apply this procedure repatedly, removing all vertices discovered or present in the tree after an execution until all the vertices are explored.

5.1 Breadth First Search

Now it is easy to see that breadth first search always returns a reachability tree from S. However, one nice property of breadth first search is that it actually explores the vertices in increasing distance from s as we show below.

Lemma 14. *In an execution of BFS the set S_i contains all vertices at distance i from s, i.e. $S_i = \{v \in V | d(s, v) = i\}$.*

Proof. We prove by induction. $S = \{s\}$ and clearly s is the only vertex at distance 0 from s. Now suppose true for all $i \leq k$ we prove true for $k + 1$. Let v be an arbitrary vertex. We add v to S_{k+1} if and only if there is an edge (u, v) with $u \in S_k$ and if $v \notin S_0 \cup S_1 \cup ... \cup S_k$. Thus trivially $d(s, v) > k$. However since $(u, v) \in E$ the length k path to u concatenated with (u, v) is a length $k + 1$ path and thus $d(s, v) \leq k + 1$. Thus the inductive hypothesis holds for $k + 1$. □

As an immediate corollary of this lemma we have that BFS returns a reachability tree for S.

Corollary 15. *BFS always returns a reachability tree for S.*

Proof. Note that for each $v \in S_i$ for $i \geq 1$ there is exactly one (u, v) edge with $u \in S_{i-1}$. Consequently there is a unique path from u to every vertex in every S_i by following the layers of the tree. □

Another nice property of BFS trees, i.e. a tree returned by BFS, is that these lemmas immediately imply that $\text{depth}_T(s, v) = d_T(s, v) = d_G(s, v)$ for all vertices v, in other words tree depth, distance from s in the

tree, and distance from s in the graph all coincide. Consequently, in the case when the graph is undirected in a BFS tree we have that for all edges $(u, v) \in E$ it is the case that $|\text{depth}_T(u) - \text{depth}_T(v)| \leq 1$.

One nice application of this fact and BFS trees is that we can use them to determine whether or not a graph is bipartite. We define bipartite-ness in terms of graph colorability.

Definition 16 (Colorable)**.** We say an undirected graph is k-colorable if and only if is a way to assign one of k-distinct *labels* or *colors* to each vertex of the graph so that for every edge in the graph its endpoints are assigned different colors. We call a graph *bipartite* if is 2-colorable.

Intuitively, a graph is bipartite if we we can partition the vertices into 2 sets so that all edges go between the sets and there is no edge with both endpoints in one sets. Bipartite graphs occur frequently as we will see in Lecture 5. In the following lemma we characterize bipartite graphs in terms of odd length cycles and the depth of vertices in a tree.

Lemma 17. *Each of the following conditions on an undirected connected graph $G = (V, E)$ are equivalent*

1. *$G = (V, E)$ is bipartite*
2. *G contains no cycles of odd length*
3. *All edges in any BFS tree on G occur between vertices of differing depth, i.e.* $\text{depth}(u) \neq \text{depth}(v)$ *for all $\{u, v\} \in E$*

Proof. If G is bipartite, then any cycle has to alternate between the two colors and is therefore is of even length and thus (1) implies (2). If G had an edge between vertices at the same depth then looking at the cycle formed by that edge and the paths from the endpoints to the root (removing repeated edges) forms a cycle of odd length and thus (2) implies (3). Finally, if (3) holds, then since edges occur only between one depth and the next depth we have that coloring all vertices of odd depth one color and all vertices of even depth the other color is a valid 2 coloring of the graph. □

5.2 Depth First Search

Breadth first search seems intrinsically useful in that it computes distances from s in a graph. Why is depth first search (DFS) useful? Here we show that DFS trees, the trees returned by DFS, have a number of nice properties. First we show that in undirected graphs DFS trees allow us to structure the edges so that they only go between ancestors and descendents of vertices, i.e. there are no cross-edges or edges between different branches of the tree. Consequently they may be better for reasoning about certain connectivity structure of undirected graphs.

Lemma 18. *Let (T, r) be a rooted DFS tree of undirected graph G starting from r, then all edges in the the graph are between a vertex and its ancestor or descendent.*

Proof. Let $\{u, v\}$ be an edge in the graph and suppose without loss of generality that u is marked first during the DFS tree construction. Either v is marked during a recurse invocation of DFS from u or $\{u, v\}$ is added to the tree. In either case, v is a descendent of u in the tree. □

One nice application of this fact is that it can be used to find the set of vertices whose removal disconnects the graph, called articulation points.

Definition 19 (Articulation Point)**.** A vertex in an undirected graph is called an *articulation point* if removing the vertex, and all edges incident to it, makes the graph disconnected.

We prove this result in steps. First, to reason about it we will introduce another measure of connectivity defined as follows.

Definition 20 (Biconnectedness)**.** We call a graph *biconnected* if every pair of vertices in the graph are contained in a cycle. We call a subgraph a *biconnected component* if it is a vertex maximal biconnected subgraph, i.e. it is biconnected and there is no biconnected subgraph containing it an more vertices.

Note that unlike connected components, which partition the vertices of the graph, biconnected components can overlap. To see this, take a cycle of length 8 and contract any two vertices at distance 4 from each other. This will make two biconnected components corresponding to the 4 cycles, however the biconnected components will both contain the contracted vertex. Note this example also shows that being biconnected is not the same thing as being 2-edge connected or having minimum cut value at least 2. Note, that this example also suggests a relationship between biconnected components and articulation points, indeed we can show that a vertex is an articulation point if and only if it is contained in at least two biconnected components.

Lemma 21. *A vertex u is an articulation point of an undirected graph $G = (V, E)$ if and only if it is incident to at least two edges not in the same biconnected components.*

Proof. Suppose u is an articulation point, then there must be vertices v, w with $\{u, v\} \in E$ and $\{u, w\} \in E$ such that after removing u there is no path from $\{v, w\}$, otherwise removing u didn't disconnect G since all its neighbors are still connected. Furthermore, there cannot be a cycle containing both v and w since otherwise they would still be connected after removing u. Consequently v and w are in different biconnected components.

On the other hand, if u is in two different biconnected components. Then there must be v, w with $\{u, v\} \in E$ and $\{u, w\} \in E$ simply because u must be in a cycle with a vertex in each component. Since v and w are in different biconnected components there must be no cycle passing through them and consequently any path between them must go through uv (since otherwise we would have a cycle). Consequently, u is an articulation point. □

This lemma says that if we could compute biconnected components we could compute articulation points, however we have also argued that the structure of biconnected components is not as nice as connected components in that they do not partition the vertices. However, in the following lemma we show that biconnected components do partition the edges of the graph.

Lemma 22. *If edges $\{u_1, v_1\}$ and $\{u_2, v_2\}$ are in the same biconnected component and $\{u_2, v_2\}$ and $\{u_3, v_3\}$ are in the same biconnected component then $\{u_1, v_1\}$ and $\{u_3, v_3\}$ are in the same biconnected component.*

Proof. By definition without loss of generality there is a path p_1 from u_1 to u_2 and a path p_2 from v_1 to v_2 with disjoint vertices a cycle containing $\{u_1, v_1\}$ and $\{u_3, v_3\}$. Now follow the path p_1 until it first intersects the cycle at a vertex and follow the path p_2 until it first intersects the cycle at a vertex (this must happen since the cycle contains $\{u_2, v_2\}$). Consider these paths together with the part of the cycle that connects them through $\{u_3, v_3\}$ since we looked for least part of the path before it intersected no vertex appears twice and this is a cycle. □

Consequently, to identify the biconnected components we simply need to identify which edges of the DFS tree are in different biconnected components. In the following lemma we show this admits a nice characterization.

Lemma 23. *Edges $\{u, v\}$ and $\{v, w\}$ with w a descendent of u are in the same biconnected component if and only if there is an edge from a descendent of w to an ancestor of u.*

Proof. Clearly if there is an edge from a descendent of w to an ancestor of u then there is a cycle containing $\{u, v\}$ and $\{v, w\}$ and thus the edges are in the same biconnected component. however, if there is not such an edge, then since all edges from descendents of v are to ancestors of v we see that the only way to reach u from w is through v and consequently there is no cycle containing $\{u, v\}$ and $\{v, w\}$. □

With this we can easily characterize the articulation points by the edges not in the tree.

Lemma 24. *A vertex u is an articulation point if and only if in a DFS tree it either is the root and has more than one child or one of its children does not have an edge to an ancestor of its parent.*

Proof. If u is a root then clearly it is an articulation point if and only if it has more than one child, since no edges go between its children and if it has one child then removing it doesn't disconnect the graph. Furthermore, if u is not the root then it is an articulation point if and only if the edge to its parent isn't in the same biconnected component as the edge to one of its children. However, this happens if and only if one of its children does not have an edge to an ancestor of its parent.

In the next lecture we show how this gives us a linear time algorithm for computing all the articulation points and biconnected components of a graph. □

Note that similar reasoning can be used to show that DFS can be used to yield all the strongly connected components of a graph. It can be shown that all edges in a graph are either between descendent and ancestor in a DFS tree or are to a vertex that was explored earlier (i.e. cross edges, edges between different tree branches are always pointed in the same direction). Furthermore, similar reasoning as here can be used to show that a vertex is in the same strongly connected component as its child in the DFS tree if and only if there is an edge from a descendent of the child to an ancestor of the vertex. Using this, again all strongly connected components can be computed in linear time as we will discuss in the next lecture.