Hamilton Circuit Algorithm

T4468

Developed by

Ghulam Mustafa Babar

Abdul Saeed

Supervised by

Prof. Dr. M. Sikandar Hayat Khiyal

Department of Computer Science

International Islamic University, Islamabad

(2006)

بسم الله الرحمن الرحيم

Department of Computer Science

International Islamic University Islamabad

Date: 16-07-2007

Final Approval

It is certified that we have read the project thesis submitted by **Mr. Ghulam Mustafa Babar** registration no. 105-CS/MS/03 & **Mr. Abdul Saeed** registration no. 235-CS/MS/S05, and it is our judgment that this project is of sufficient standard to warrant its acceptance by the International Islamic University, Islamabad for the degree of MS in Computer Science.

Committee

External Examiner

Dr. Abdul Sattar
Ex. DG
Pakistan Computer Bureau

[illegible]

Internal Examiner

Engr. Dr. Syed Afaq Hussain
Head Dept. of Electronics Engineering
IIUI.

[illegible]

Supervisor

Prof. Dr. M. Sikandar Hayat Khiyal
Head,
Department of Computer Science,
International Islamic University,
Islamabad.

[illegible]

A dissertation submitted to the

Department of Computer Science,

International Islamic University, Islamabad

as a partial fulfillment of the requirements

for the award of the degree of

MS in Computer Science

Dedicated to

The Holy Prophet (SAW)

and

My Parents

DECLARATION

We hereby declare that this thesis, neither as a whole nor as a part thereof has been copied out from any source. It is further declared that we have developed the project entirely on the basis of our personal efforts made under the sincere guidance of our teachers. If any part of this thesis is found to be copied, we shall standby the consequences. No portion of the work presented in this thesis has been submitted in support of any application for any other degree or qualification of this or any other University or Institute of learning.

Ghulam Mustafa Babar
105-CS/MS/03

Abdul Saeed
235-CS/MS/S05

ACKNOWLEDGEMENT

All praise to Almighty Allah, the most merciful and compassionate, who enabled us to complete this project.

We express our gratitude to Dean, Faculty of Applied Sciences, IIUI, Professor Dr. Khalid Rashid and our kind supervisor Prof. Dr. M. Sikandar Hayat Khiyal who kept our morale high by his suggestions and appreciation. Without his precious guidance and help we could never be able to develop this software. We would like to express my gratitude to all of our teachers including Brig. Shiraz Baig, Dr. Muhammad Sher and Mr. Asim Munir in the department for their guidance and great moral support of our efforts.

And last but not the least; we would like to acknowledge the support of our family members. We would like to admit that we owe all our achievements to our truly, sincere and most loving parents, brothers and sisters, who mean the most to us, and whose prayers are a source of determination for us.

Ghulam Mustafa Babar

Abdul Saeed

PROJECT IN BRIEF

Project Title:	Hamilton Circuit Algorithm
Developed For:	General purpose research
Developed By:	**Ghulam Mustafa Babar** **105-CS/MS/03** **Abdul Saeed** **235-CS/MS/S05**
Supervised By:	Prof. Dr. M. Sikandar Hayat Khiyal Head, Department of Computer Science, International Islamic University, Islamabad.
Tool Used:	Microsoft Visual C#.NET, Visio2003.
Operating System:	Windows XP Professional
System Used:	Pentium IV with 256 MB RAM
Date Started:	September, 2004
Date Completed:	November, 2006

ABSTRACT

Hamilton Circuit Algorithm is one of the famous classical computation problems which are considered as NP Complete. There is no complete algorithm exist which can solve it in the polynomial time. The purpose of this research was to develop an algorithm to determine the Hamilton Circuit in a given graph. This Algorithm will find Hamilton Circuit in polynomial steps. First we divided the problem into different special cases e.g. find a Hamilton Circuit in a graph in which every node has degree two then tried to find Hamilton Circuit in a graph in which every node has degree three and so on. During these experiments we found certain basic conditions which were very useful to find the solution. We then combined these conditions and based on them, we designed a new algorithm which finds Hamilton Circuit in polynomial time.

Ghulam Mustafa Babar

105-CS/MS/03

Abdul Saeed

235-CS/MS/S05

TABLE OF CONTENTS

Chapter 1

Introduction

1. Introduction

This Chapter will briefly discuss about the Hamilton Circuit Problem, its history, the existing work and our proposed solution on it.

1.1. Brief History

In 1856, Sir Rowan Hamilton [1] described a certain mathematical game called the *Icosian* played on the surface of a dodecahedron. Starting from a given vertex, the objective was to find a path of consecutive vertices along the edges, visiting every vertex exactly once and returning to the original vertex to complete a circuit. The general problem of trying to find such *Hamiltonian Circuits* in arbitrary graphs turned out to be very difficult to solve.

In 1952, Dirac [2] proposed a condition that guarantees the existence of a Hamiltonian circuit in a simple graph G with $n \geq 3$ vertices: *a lower bound on the minimum degree* $\delta \geq n/2$ *suffices*. This is the best possible lower bound because the graph consisting of cliques of orders $\lfloor (n+1)/2 \rfloor$ and $\lceil (n+1)/2 \rceil$ sharing a common vertex has minimum degree $\delta = \lfloor (n-1)/2 \rfloor$ but has no Hamiltonian circuit. However, Dirac's original proof is a proof by contradiction that does not show how one may actually construct the stipulated Hamiltonian circuit.

In 1972, Karp [3] showed that the problem of finding Hamiltonian circuits (respectively, tours) in graphs is **NP** - complete. Thus, the existence or non-existence of a polynomial-time algorithm for deciding whether a Hamiltonian circuit (respectively, tour) exists in any given graph would resolve one of the most important open problems in mathematics and computer science, the **P** versus **NP** question [4].

1.2. Definition

A path $X_0, X_1, X_2, \ldots, X_{n-1}, X_n$ in the graph $G=(V,E)$ is called Hamilton Path if $V=\{ X_0, X_1, X_2, \ldots, X_{n-1}, X_n \}$ and X_i is not equal to X_j for $0 \leq i < j \leq n$. A circuit $X_0, X_1, X_2, \ldots,$

X_{n-1}, X_n, X_0 (with n>1) in a graph G=(V,E) is called a Hamilton Circuit if X_0, X_1, X_2, …, X_{n-1}, X_n is Hamilton Path [8].

1.3. Existing Solutions

Is there a simple way to determine whether a graph has a Hamilton Circuit or Hamilton Path? At first, it might seem that there should be an easy way to determine this. Surprisingly there are no known simple necessary and sufficient criteria for the existence of Hamilton Circuit [8].

1.3.1. Theorems

Many theorems are known that give sufficient conditions for the existence of Hamilton Circuit. But still these theorems don't cover the major range of problem variations.

- **Dirac's Theorem:**

  If G is a simple graph with n vertices with $n \geq 3$ such that the degree of every vetex in G is at least n/2, then G has a Hamilton Circuit [2].

- **Ore's Theorem:**

  If G is a simple graph with n vertices with $n \geq 3$ such that $deg(u) + deg(v) \geq n$ for every pair of non-adjacent vertices u and v in G, then G has Hamilton Circuit [3].

1.3.2. Algorithms

Although there is no algorithm exists which can find the Hamilton Circuit in a graph in polynomial time. There is the only one algorithm which is meant to determine the Hamilton Circuit in a graph and the technique used is called BACKTRACKING which leads the algorithm to exponential time.

1.3.2.1. Backtracking Algorithm for the Hamilton Circuit Problem

A state space tree for this problem is as follows:

Put the starting vertex at level 0 in the tree; call it the zeroth vertex on the path. At level 1, consider each vertex other than the starting vertex as the first vertex after the starting one. At level 2, consider each of these same vertices as the second vertex, and so on. Finally, at level n-1, consider each of these same vertices as the (n-1)st vertex.

Following considerations enable us to backtrack in this state space tree:

1. The *i*th vertex on the path must be adjacent to the (i - 1)st vertex on the path.
2. The (n - 1)st vertex must be adjacent to the 0th vertex (the starting one).
3. The *i*th vertex cannot be one of the first i – 1 vertices.

The number of nodes in the state space tree for this algorithm is

$1 + (n - 1) + (n - 1)^2 + \ldots + (n - 1)^{n-1} = (n - 1)^n - 1 / (n - 2)$

which is much **worse than the exponential.**

1.3.2.2. Traveling Sales Person Algorithm

Another variant of this problem is Traveling Sales Person problem. People shouldn't get confused with the similarity of domain for these two problems. In both Hamilton Circuit and Traveling Sales Person Problem we require a path or tour which starts from any given node and passes through all other nodes and ends again to the starting node. The main difference between Hamilton Circuit and Traveling Sales Person is, Traveling Sales Person is to find a tour from weighted, directed graph whereas Hamilton Circuit is the tour from un-weighted, un-directed graph. In other words, we can say that Hamilton Circuit is a special case of Traveling Sales Person in which graph is un-weighted and undirected. Thus we can try the known algorithms of Traveling Sales Person for Hamilton Circuit with little adjustment or modification. On the basis of this theory, we now have a look at mathematicians work for the solution of Traveling Sales Person Problem. We have two solutions for Traveling Sales Person:

1. Dynamic Programming Solution:

2. Brute Force Solution

Complexity:

Brute Force Algorithm:	$(n-1)!$	(Worse than Exponential)
Dynamic Programming Algorithm:	$n^2 2^n$	(Exponential)

None of these solutions are Polynomial time.

1.4. Research Overview

The purpose of this research was to develop an algorithm to determine the Hamilton Circuit in a given graph. This Algorithm will find Hamilton Circuit in polynomial steps. First we divided the problem into different special cases e.g. find a Hamilton Circuit in a graph in which every node has degree two then tried to find Hamilton Circuit in a graph in which every node has degree three and so on. During these experiments we found certain basic conditions which were very useful to find the solution. We then combined these conditions and based on them, we designed a new algorithm which finds Hamilton Circuit in polynomial time.

Chapter 2

Basic Concept

2. Basic Concepts

In this chapter, we have discussed the basic concepts needed to understand this research project. First we covered the graph concept, and then we focused on graph representation methods, after that we discussed algorithms and finally we have gone through the computational complexity theory.

2.1. Graph Concepts

Now we discuss the necessary graph theory which can be helpful to understand the Hamilton Circuits and Hamilton Paths the research presented in the following chapters.

2.1.1. Graph

A **graph** or **undirected graph** *G* is an ordered pair *G* := (*V*, *E*) that is subject to the following conditions:

- *V* is a set of **vertices** or **nodes**,
- *E* is a set of unordered pairs of distinct vertices, called **edges** or **lines**.
- The vertices belonging to an edge are called the **ends**, **endpoints**, or **end vertices** of the edge.

V (and hence *E*) are usually taken to be finite sets, and many of the well-known results are not true (or are rather different) for **infinite graphs** because many of the arguments fail in the infinite case. The **order** of a graph is |*V*| (the number of vertices). A graph's **size** is |*E*|, the number of edges. The **degree** of a vertex is the number of other vertices it is connected to by edges.

2.1.1.1. Loops and Links

A **loop** is an edge (directed or undirected) with both ends the same; these may be permitted or not permitted according to the application. In this context, an edge with two different ends is called a **link**.

2.1.1.2. Multiple Edges

Sometimes *E* and *A* are allowed to be **multisets**, so that there can be more than one edge (called multiple edges) between the same two vertices. Another way to allow multiple edges is to make *E* a set, independent of *V*, and to specify the endpoints of an edge by an **incidence relation** between *V* and *E*. The same applies to a directed edge set *A*, except that there must be two incidence relations, one for the head and one for the tail of each edge.

2.1.1.3. Path

In graph theory, a **path** in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence. The first vertex is called the *start vertex* and the last vertex is called the *end vertex*. Both of them are called *end or terminal vertices* of the path. The other vertices in the path are *internal vertices*. A **cycle** is a path such that the start vertex and end vertex are the same. Notice however that unlike with paths, any vertex of a cycle can be chosen as the start, so the start is often not specified.

The same concepts apply in a directed graph, with the edges being directed from each vertex to the following one. Often the terms *directed path* and *directed cycle* are used in this case.

A path with no repeated vertices is called a **simple path**, and cycle with no repeated vertices aside from the start/end vertex is a **simple cycle**. In modern graph theory, most often "simple" is implied; i.e., "cycle" means "simple cycle" and "path" means "simple path", but this convention is not always observed, especially in applied graph theory.

A simple cycle that includes every vertex of the graph is known as a Hamiltonian cycle.

Two paths are *independent* (alternatively, *internally vertex-disjoint*) if they do not have any internal vertex in common.

The *length* of a path is the number of edges that the path uses, counting multiple edges multiple times. In the graph shown in Figure 2.1, (1, 2, 3, 1, 4, 5) is a path of length 5, and (5, 6, 7) is a simple path of length 2.

A weighted graph associates a value (*weight*) with every edge in the graph. The *weight of a path* in a weighted graph is the sum of the weights of the traversed edges. Sometimes the words *cost* or *length* are used instead of weight.

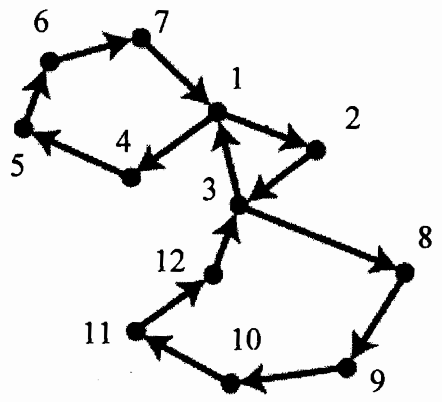


Figure 2.1: Path

2.1.1.4. Cycle

Cycle in graph theory and computer science has several meanings:

- A closed walk, with repeated vertices allowed. (This usage is common in computer science. In graph theory it is more often called a **closed walk.**)
- A closed (simple) path, with no other repeated vertices than the starting and ending vertices. (This usage is common in graph theory.) This may also be called a **simple cycle**, **circuit**, **circle**, or **polygon**.
- A closed directed walk, with repeated vertices allowed. (This usage is common in computer science. In graph theory it is more often called a **closed directed walk.**)

- A closed directed (simple) path, with no repeated vertices other than the starting and ending vertices. (This usage is common in graph theory.) This may also be called a **simple (directed) cycle**.
- The edge set of an undirected closed path without repeated vertices. This may also be called a **circuit**, **circle**, or **polygon**.
- An element of the binary or integral (or real, complex, etc.) cycle space of a graph. (This is the usage closest to that in the rest of mathematics, in particular algebraic topology.) Such a cycle may be called a **binary cycle**, **integral cycle**, etc.
- An edge set which has even degree at every vertex; also called an **even edge set** or, when taken together with its vertices, an **even subgraph**. This is equivalent to a binary cycle, since a binary cycle is the indicator function of an edge set of this type.

2.1.1.5. Distance

In the mathematical subfield of graph theory, the **distance** between two vertices in a graph is the number of edges in a shortest path connecting them. This is also known as the **geodesic distance**.

There are a number of other measurements defined in terms of distance:

- The **eccentricity** ε of a vertex v is the greatest distance between v and any other vertex.
- The **radius** of a graph is the minimum eccentricity of any vertex.
- The **diameter** of a graph is the maximum eccentricity of any vertex in the graph. That is, it is the greatest distance between any two vertices. A **peripheral vertex** in a graph of diameter d is one that is distance d from some other vertex—that is, a vertex that achieves the diameter.
- A **pseudo-peripheral vertex** v has the property that for any vertex u, if v is as far away from u as possible, then u is as far away from v as possible. Formally, if the

distance from u to v equals the eccentricity of u, then it equals the eccentricity of v.

2.1.1.6. Degree

In graph theory, the **degree** (or **valency**) of a vertex is the number of edges incident to the vertex. The degree of a vertex v is denoted deg (v).

For an **undirected graph**, the degree of a vertex is the number of edges incident to the vertex. This means that each loop is counted twice. This is because each edge has two endpoints and each endpoint adds to the degree.

In a **directed graph**, an edge has two distinct ends: a head (the end with an arrow) and a tail. Each end is counted separately. The sum of head endpoints count toward the **indegree** and the sum of tail endpoints count toward the **outdegree**.

The indegree is denoted $\deg^{+}(v)$ and the outdegree as $\deg^{-}(v)$

2.1.2. Simple Graph

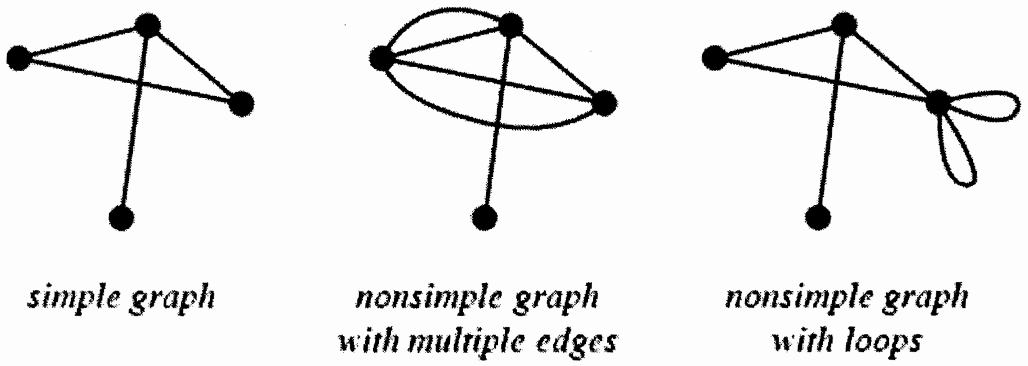

simple graph
nonsimple graph with multiple edges
nonsimple graph with loops

Figure 2.2: Simple and Non-simple graphs

A simple graph, also called a strict graph, is an unweighted, undirected graph containing no graph loops or multiple edges. Unless stated otherwise, the unqualified term "graph" usually refers to a ***simple* graph**. A simple graph with multiple edges is sometimes called a **multi-graph** as shown in Figure 2.2.

2.1.3. Multigraph

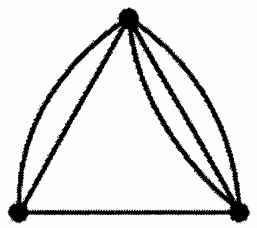


Figure 2.3: Multigraph

A **multigraph** (Figure 2.3) is a graph which is permitted to have multiple edges, (also called "parallel edges") i.e. edges that have the same end nodes. Formally, a **multigraph** G is an ordered pair G:=(V, E) with

- V a set of vertices or nodes,
- E a multiset of unordered pairs of distinct vertices, called edges or lines.

Some authors also allow mutigraphs to have loops, that is, an edge that connects a vertex to itself.

A **multidigraph** is a directed graph which is permitted to have multiple arcs, i.e., arcs with the same source and target nodes. A **multidigraph** G is an ordered pair G:=(V,A) with

- V is a set of vertices or nodes,
- A is a multiset of ordered pairs of vertices called directed edges, arcs or arrows.

A **mixed multigraph** G:=(V,E, A) may be defined in the same way as a mixed graph.

2.1.4. Pseudograph

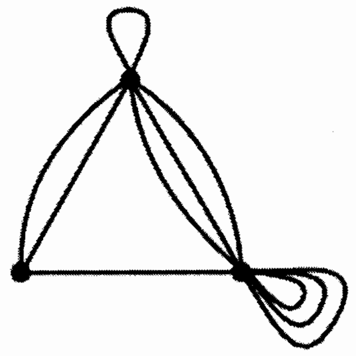


Figure 2.4: Pseudograph

A pseudograph is a non-simple graph in which both graph loops and multiple edges are permitted as shown in Figure 2.4.

2.1.5. Labeled Graph

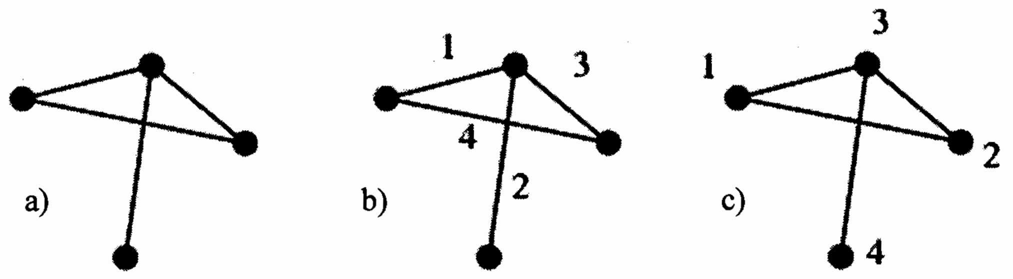


Figure 2.5: a) unlabled graph **b)** edge-labeled graph **c)** vertex-labeled graph

A labeled graph G = (*V*, *E*) is a finite series of graph vertices *V* with a set of graph edges *E* of 2-subsets of *V* shown in Figure 2.5. Given a graph vertex set *Vn* = {1, 2, ..., *n*}, the number of vertex-labeled graphs is given by $2^{n(n-1)/2}$, Two graphs G and H with graph vertices *Vn* = {1, 2, ..., *n*} are said to be isomorphic if there is a permutation *P* of *Vn* such that {*u*, *v*} is in the set of graph edges E(G) iff {*p*(*u*), *p*(*v*)} is in the set of graph edges *E*(*H*).

2.1.6. Null Graph

The empty graph of 0 nodes is called **null graph**.

Use of the null graph is discouraged since it is felt by many in the graph theoretical community that allowing the null graph causes much more trouble than it is worth. For example, the null graph has no automorphism group, it cannot be imbedded on the sphere obeying the polyhedral formula, it is connected and acyclic but has too many edges to be a tree, and so on. It is an exception to so many things that the community (or most of it) has decided that the only good null graph is a dead null graph.

2.1.7. Directed Graph

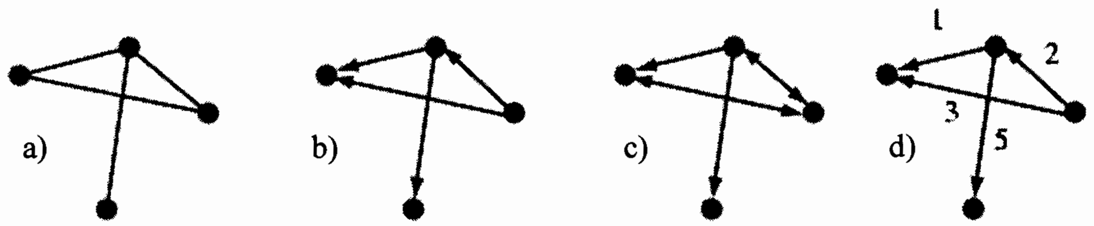


Figure 2.6: a) undirected graph b) oriented graph c) directed graph d) network

A **directed graph** or **digraph** G is an ordered pair G:=(V, A) with

- V, a set of vertices or nodes,
- A, a set of ordered pairs of vertices, called directed edges, arcs, or arrows.

An edge e = (x, y) is considered to be directed from x to y; y is called the head and x is called the tail of the edge; y is said to be a direct successor of x, and x is said to be a direct predecessor of y. If a path leads from x to y, then y is said to be a successor of x, and x is said to be a predecessor of y as shown in Figure 2.6.

A variation on this definition is the oriented graph, which is a graph (or multigraph; see below) with an orientation or direction assigned to each of its edges. A distinction between a directed graph and an oriented simple graph is that if x and y are vertices, a directed graph allows both (x, y) and (y, x) as edges, while only one is permitted in an oriented graph. A more fundamental difference is that, in a directed graph (or multigraph), the directions are fixed, but in an oriented graph (or multigraph), only the underlying graph is fixed, while the orientation may vary.

A directed acyclic graph, also called a dag or DAG, is a directed graph with no directed cycles.

A directed graph having no multiple edges or loops (corresponding to a binary adjacency matrix with 0s on the diagonal) is called a **simple directed graph**.

A complete graph in which each edge is bidirected is called a **complete directed graph**.

A directed graph having no symmetric pair of directed edges (i.e., no bidirected edges) is called an **oriented graph**.

A complete oriented graph (i.e., a directed graph in which each pair of nodes is joined by a single edge having a unique direction) is called a **tournament**.

2.1.8. Function Graph

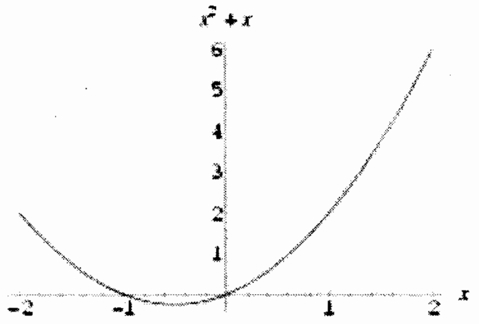


Figure 2.7: Function Graph

Given a function $f(x_1, \ldots, x_n)$ defined on a domain U, the graph of f is defined as the set of points (which often form a curve or surface) showing the values taken by f over U (or some portion of U). A graph is sometimes also called a plot. Unfortunately, the word “graph” is uniformly used by mathematicians to mean a collection of vertices and edges connecting them. In some education circles, the term “vertex-edge graph” is used in an attempt to distinguish the two types of graph. However, as Gardner notes, “The confusion of this term with the ‘graphs’ of analytic geometry is regrettable, but the term has stuck [in the mathematical community].” In this work, the term “graph” will therefore be used to refer to a collection of vertices and edges, while a graph in the sense of a plot of a function will be called a “function graph” as shown in Figure 2.7.

2.1.9. Sub-graph and Super-graph

A graph G′ whose graph vertices and graph edges form subsets of the graph vertices and graph edges of a given graph G. If G′ is a sub-graph of G, then G is said to be a super graph of G′.

2.1.10. Graph Cycles or Circuits

A cycle of a graph *G*, sometimes also called a circuit, is a subset of the edge set of *G* that forms a path such that the first node of the path corresponds to the last.

A cycle that uses each graph vertex of a graph exactly once is called a Hamiltonian circuit. A graph containing no cycles of length three is called a triangle-free graph.

2.1.11. Bipartite Graph

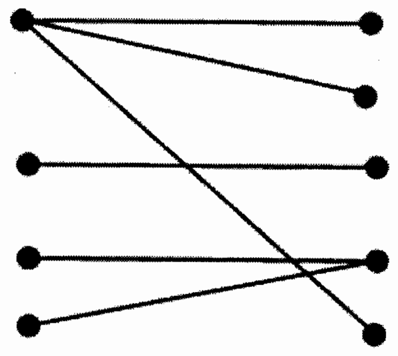


Figure 2.8: Bipartite Graph

A **bipartite graph**, also called a **bigraph** (Figure 2.8), is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. A bipartite graph is a special case of a *k*-partite graph with *k* = 2. Bipartite graphs are equivalent to two-colorable graphs, and a graph is bipartite iff all its cycles are of even length.

2.1.12. Complete Graph

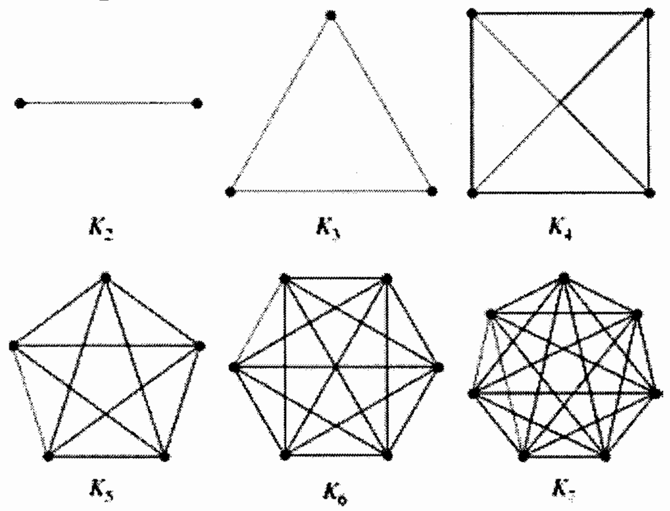


Figure 2.9: Complete Graph

A complete graph is a graph in which each pair of graph vertices is connected by an edge. The complete graph with n graph vertices is denoted Kn and has n(n-1)/2 (the triangular numbers) undirected edges, where nPk is a binomial coefficient. In older literature, complete graphs are sometimes called universal graphs.

The complete graph on 0 nodes is a trivial graph known as the **null graph**, while the complete graph on 1 node is a trivial graph known as the **singleton graph.**

The graph complement of the complete graph K_n is the **empty graph** on 'n' nodes.

A complete graph is a regular graph of degree n − 1. All complete graphs are their own cliques. They are maximally connected as the only vertex cut which disconnects the graph is the complete set of vertices

2.1.13. Connected Graph

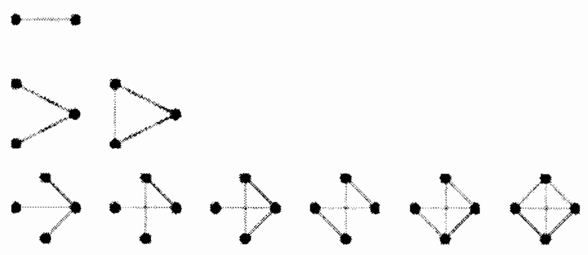


Figure 2.10: Connected Graph

A graph which is connected in the sense of a topological space, i.e., there is a path from any point to any other point in the graph. A graph that is not connected is said to be disconnected. This definition means that the null graph and singleton graph are considered connected, while empty graphs on $n \geq 2$ nodes are disconnected.

2.1.14. Regular Graph

A graph is said to be regular of degree 'r' if all local degrees are the same number 'r'.

A 0-regular graph is an empty graph, a 1-regular graph consists of disconnected edges, and a 2-regular graph consists of disconnected cycles. The first interesting case is therefore 3-regular graphs, which are called cubic graphs.

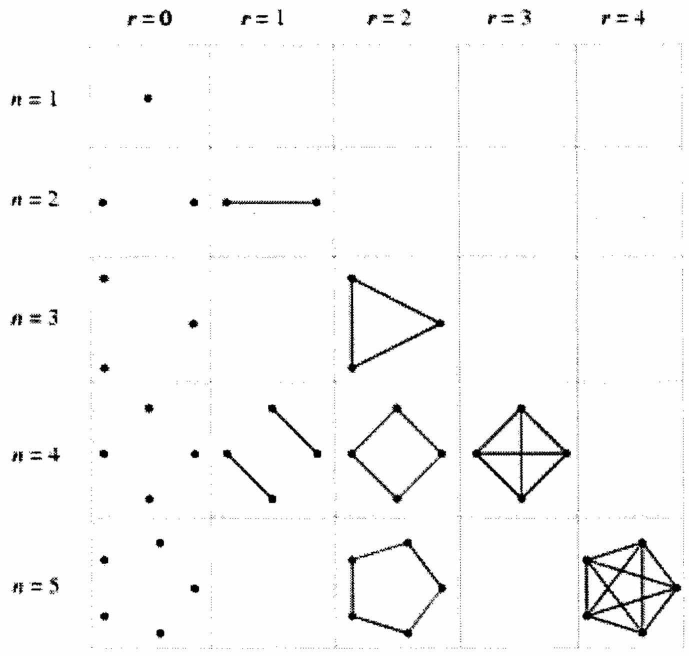


Figure 2.11: Regular Graph

A strongly regular graph is a regular graph where every adjacent pair of vertices has the same number l of neighbors in common, and every non-adjacent pair of vertices has the same number n of neighbors in common. The smallest graphs that are regular but not strongly regular are the cycle graph and the circulant graph on 6 vertices.

The complete graph Km is strongly regular for any m.

2.1.15. Isomorphic Graphs

Two graphs which contain the same number of graph vertices connected in the same way are said to be isomorphic. Formally, two graphs G and H with graph vertices V={1,2,...,n} are said to be isomorphic if there is a permutation *P* of V_n such that $\{u, v\}$ is in the set of graph edges *E*(*G*) iff $\{p(u), p(v)\}$ is in the set of graph edges $E(H)$.

Determining if two graphs are isomorphic is thought to be neither an NP-complete problem nor a P-problem, although this has not been proved. In fact, there is a famous complexity class called graph isomorphism complete which is thought to be entirely disjoint from both NP-complete and from P.

However, a polynomial time algorithm is known when the maximum vertex degree is bounded by a constant.

2.1.16. Planar Graph

A graph is planar if it can be drawn in a plane without graph edges crossing (i.e., it has graph crossing number 0). The number of planar graphs with $n = 1$, 2, ... nodes are 1, 2, 4, 11, 33, 142, 822, 6966, 79853, ...

A planar graph already drawn in the plane is called a plane graph. A plane graph can be defined as a planar graph with a mapping from every node to a position in 2D space, and from every edge to a plane curve, such that each curve has two extreme points, which coincide with the positions of its end nodes, and all curves are disjoint except on their extreme points.

The equivalence class of topologically equivalent drawings on the sphere is called a planar map. Although a plane graph has an external or unbounded face, none of the faces of a planar map has a particular status.

There are a number of efficient algorithms for planarity testing, which are unfortunately all difficult to implement. Most are based on the o(n^3) algorithm of Auslander and Parter.

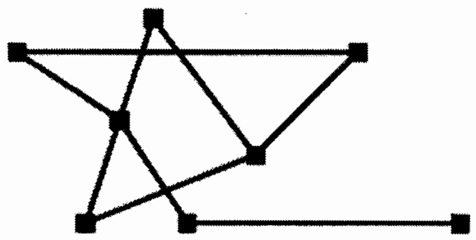
2.1.17. Trees

A **tree** is a connected acyclic simple graph. A vertex of degree 1 is called a **leaf**, or *pendant vertex*. An edge incident to a leaf is a **leaf edge**, or *pendant edge*. (Some people define a leaf edge as a *leaf* and then define a *leaf vertex* on top of it. These two sets of definitions are often used interchangeably.) A non-leaf vertex is an **internal vertex**. Sometimes, one vertex of the tree is distinguished, and called the **root**. A **rooted tree** is a tree with a root. **Rooted trees** are often treated as **directed acyclic graphs** with the edges pointing away from the root.

Trees are commonly used as data structures in computer science.

- A **subtree** of the tree *T* is a subgraph of *T*.
- A **forest** is an acyclic simple graph.
- A **subforest** of the forest *F* is a subgraph of *F*.
- A **spanning tree** is a spanning subgraph that is a tree. Every graph has a spanning forest. But only a connected graph has a spanning tree.
- A special kind of tree called a **star** is $K_{1,k}$. An induced star with 3 edges is a **claw**.
- A ***k*-ary** tree is a rooted tree in which every internal vertex has *k children*. An 1-ary tree is just a path. A 2-ary tree is also called a **binary tree**.

2.1.18. Spanning tree



(**Figure 2.12:** A spanning tree (light) of a graph (dark), superimposed)

In the mathematical field of graph theory, a spanning tree T of a connected, undirected graph G is a tree composed of all the vertices and some (or perhaps all) of the edges of G. Informally, a spanning tree of G is a selection of edges of G that form a tree spanning every vertex. That is, every vertex is connected to the tree, but no cycles (or loops) are formed. On the other hand, every bridge of G must belong to T.

A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle, or as a minimal set of edges that connect all vertices.

More generally, a spanning forest of an arbitrary (possibly unconnected), undirected graph G is a maximal forest that is a subgraph of G. Equivalently, a spanning forest is the union of a spanning tree for every connected component of G.

In certain fields of graph theory it is often useful to find a minimum spanning tree of a weighted graph.

2.1.19. Cliques

The complete graph Kn of order n is a simple graph with n vertices in which every vertex is adjacent to every other. The example graph is not complete. The complete graph on n vertices is often denoted by Kn. It has n(n-1)/2 edges (corresponding to all possible choices of pairs of vertices).

A clique (pronounced "cleek") in a graph is a set of pairwise adjacent vertices. Since any subgraph induced by a clique is a complete subgraph, the two terms and their notations are usually used interchangeably. A k-clique is a clique of order k. In the example graph above, vertices 1, 2 and 5 form a 3-clique, or a triangle.

The clique number $\omega(G)$ of a graph G is the order of a largest clique in G.

2.1.20. Strongly connected component

A related but weaker concept is that of a strongly connected component. Informally, a strongly connected component of a directed graph is a subgraph where all nodes in the subgraph are reachable by all other nodes in the subgraph. Reachability between nodes is established by the existence of a path between the nodes.

A directed graph can be decomposed into strongly connected components by running the Depth-first search (DFS) algorithm twice: first, on the graph itself and next on the transpose of the graph in decreasing order of the finishing times of the first DFS. Given a directed graph G, the transpose GT is the graph G with all the edge directions reversed.

2.1.21. Knots

A knot in a directed graph is a collection of vertices and edges with the property that every vertex in the knot has outgoing edges, and all outgoing edges from vertices in the knot have other vertices in the knot as destinations. Thus it is impossible to leave the knot while following the directions of the edges.

If a general resource graph is expedient, then a knot is a sufficient condition for deadlock occurrence.

2.1.22. Minors

A minor G2 = (V2,E2) of G1 = (V1,E1) is an injection from V2 to V1 such that every edge in E2 corresponds to a path (disjoint from all other such paths) in G1 such that every vertex in V1 is in one or more paths, or is part of the injection from V1 to V2. This can alternatively be phrased in terms of contractions, which are operations which collapse a path and all vertices on it into a single edge.

2.1.23. Weighted graphs and networks

A weighted graph associates a label (weight) with every edge in the graph. Weights are usually real numbers. They may be restricted to rational numbers or integers. Certain algorithms require further restrictions on weights; for instance, the Dijkstra algorithm works properly only for positive weights. The weight of a path or the weight of a tree in a weighted graph is the sum of the weights of the selected edges. Sometimes a non-edge is labeled by a special weight representing infinity. Sometimes the word cost is used instead of weight. When stated without any qualification, a graph is always assumed to be unweighted. In some writings of graph theory the term network is a synonym for a weighted graph. A network may be directed or undirected, it may contain special vertices (nodes), such as source or sink. The classical network problems include:

- minimum cost spanning tree,
- shortest paths,
- maximal flow (and the max-flow min-cut theorem)

2.1.24. Properties of graphs

Two edges of a graph are called adjacent (sometimes coincident) if they share a common vertex. Similarly, two vertices are called adjacent if they share a common edge, in which case the common edge is said to join the two vertices. An edge and a vertex on that edge are called incident.

The graph with only one vertex and no edges is called the trivial graph. A graph with only vertices and no edges is known as an edgeless graph, empty graph, or null graph (there is no consistency in the literature). The graph with no vertices and no edges is sometimes called the null graph or empty graph, but not all mathematicians allow this object.

In a weighted graph or digraph, each edge is associated with some value, variously called its cost, weight, length or other term depending on the application; such graphs arise in many contexts, for example in optimal routing problems such as the traveling salesman problem.

Normally, the vertices of a graph, by their nature as elements of a set, are distinguishable. This kind of graph may be called vertex-labeled. However, for many questions it is better to treat vertices as indistinguishable; then the graph may be called unlabeled. (Of course, the vertices may be still distinguishable by the properties of the graph itself, e.g., by the numbers of incident edges). If vertices are indistinguishable they may be distinguished by giving each vertex a label, hence the name vertex-labeled graph. The same remarks apply to edges, so that graphs which have labeled edges are called edge-labeled graphs. Graphs with labels attached to edges or vertices are more generally designated as labeled. Consequently, graphs in which vertices are indistinguishable and edges are indistinguishable are called unlabelled. (Note that in the literature the term labeled may apply to other kinds of labeling, besides that which serves only to distinguish different vertices or edges.)

2.2. Graph Representation

In this section we shall cover data structures and methods to represent and traverse the graph and how to perform different operations on graphs.

2.2.1. Graph Data structure

In computer science, a **graph** is an abstract data type (ADT) that consists of a set of nodes and a set of edges that establish relationships (connections) between the nodes. The graph ADT follows directly from the graph concept from mathematics.

A graph G is defined as follows: G = (V,E), where V is a finite, non-empty set of vertices and E is a set of edges (links between pairs of vertices). When the edges in a graph have no direction, the graph is called undirected, otherwise called directed. In practice, some information is associated with each node and edge.

2.2.2. Representation

In typical graph implementations, nodes are implemented as structures or objects. There are several ways to represent edges, each with advantages and disadvantages:

- As an adjacency list

An adjacency list associates each node with an array of incident edges. If no information is required to be stored in edges, only in nodes, these arrays can simply be pointers to other nodes and thus represent edges with little memory requirement. An advantage of this approach is that new nodes can be added to the graph easily, and they can be connected with existing nodes simply by adding elements to the appropriate arrays. A disadvantage is that determining whether an edge exists between two nodes requires O(n) time, where n is the average number of incident edges per node.

- As an adjacency matrix

An alternative way is to keep a square matrix (a two-dimensional array) M of boolean values (or integer values, if the edges also have weights or costs associated with them). The entry Mi,j then specifies whether an edge exists that goes from node i to node j. An advantage of this approach is that finding out whether an edge exists between two nodes becomes a trivial constant-time memory look-up. Similarly, adding or removing an edge is a constant-time memory access. The shortest path between any two nodes can be determined using the Floyd-Warshall algorithm. A disadvantage is that adding or removing nodes from the graph requires re-arranging the matrix accordingly, which may be costly depending on its size.

- Other representations

Yet another way is based on keeping a relation (table) of edges, with key (source, target), where source and target are the connected vertices. Known algorithms allow the table to be manipulated and searched in loglinear time. Mneson takes this approach.

In the general case, a graph may consist of many edges between many vertices, and unless the matrix representation for the edges is chosen, there may even be more than one edge connecting the same pair of vertices. Edges can be bidirectional or unidirectional.

Most data structures that are graphs are more structured than the general graph. A graph may, for example, be acyclic. In this case, each edge is unidirectional, and there is no way to traverse the edges in such a way as to ever visit the same node twice. An example of an acyclic graph is a directed acyclic word graph, a method of encoding a word-list for computer versions of word games such as Scrabble. A simple example of an acyclic graph is a non-circular singly linked list.

In most cases, the only information contained by the edge is that there is a relationship between the two nodes connected, and the information is stored in the node itself. However, some graphs have numerical values associated with each edge. These graphs can be used for different problems such as the Traveling salesman problem.

Additionally, there are graph-like structures where information is kept in the edges. One data structure has all of the information in the edges, and none of the information is in the nodes. This data structure can be very useful in modelling things like the pipes in a factory, or the wires in an airplane.

2.2.3. Adjacency list:

In graph theory, an adjacency list is the representation of all edges or arcs in a graph as a list.

If the graph is undirected, every entry is a set of two nodes containing the two ends of the corresponding edge; if it is directed, every entry is a tuple of two nodes, one denoting the source node and the other denoting the destination node of the corresponding arc.

Typically, adjacency lists are unordered.

2.2.4. Application in computer science

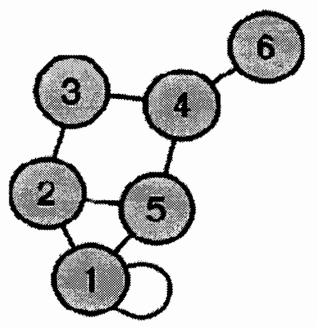
In computer science, an adjacency list is a closely related data structure for representing graphs. In an adjacency list representation, we keep, for each vertex in the graph, all other vertices which it has an edge to (that vertex's "adjacency list"). For instance, the representation suggested by van Rossum, in which a hash table is used to associate each vertex with an array of adjacent vertices, can be seen as an instance of this type of representation, as can the representation in Cormen et al in which an array indexed by vertex numbers points to a singly-linked list of the neighbors of each vertex.

One difficulty with the adjacency list structure is that it has no obvious place to store data associated with the edges of a graph, such as the lengths or costs of the edges. To remedy this, some algorithms texts such as that of Goodrich and Tamassia advocate a more object oriented variant of the adjacency list structure, sometimes called an incidence list, which stores for each vertex a list of objects representing the edges incident to that vertex. To complete the structure, each edge must point back to the two vertices forming its endpoints. The extra edge objects in this version of the adjacency list cause it to use more memory than the version in which adjacent vertices are listed directly, but are a convenient location to store additional information about each edge.

2.2.5. Adjacency matrix

In mathematics and computer science, the adjacency matrix for a finite graph, G, on n vertices is an $n \times n$ matrix where the nondiagonal entry a*ij* is the number of edges joining vertex i and vertex j, and the diagonal entry aii is either twice the number of loops at vertex i or just the number of loops (usages differ; this article follows the former; directed graphs always follow the latter). There exists a unique adjacency matrix for each graph, and it is not the adjacency matrix of any other graph. In the special case of a finite, simple graph, the adjacency matrix is a (0, 1)-matrix with zeros on its diagonal. If the graph is undirected, the adjacency matrix is symmetric.

For sparse graphs, that are graphs with few edges, an adjacency list is often the preferred representation because it uses less space. An alternative matrix representation for a graph is the incidence matrix.



$$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Figure 2.13: Adjacency Matrix

2.2.6. Incidence Matrix

In mathematics, an incidence matrix is a matrix that shows the relationship between two classes of objects. If the first class is X and the second is Y, the matrix has one row for each element of X and one column for each element of Y. The entry in row x and column y is 1 if x and y are related (called incident in this context) and 0 if they are not. There are variations; see below.

In graph theory, the incidence matrix of an **undirected graph** G is a p × q matrix [bij] where p and q are the number of vertices and edges respectively, such that bij = 1 if the vertex vi and edge xj are incident and 0 otherwise. This matrix is also called the unoriented incidence matrix.

The incidence matrix of a **directed graph** D is a p × q matrix [bij] where p and q are the number of vertices and edges respectively, such that bij = − 1 if the edge xj leaves vertex vi, 1 if it enters vertex vi and 0 otherwise. (Many authors use the opposite sign convention!)

An **oriented incidence matrix** of an undirected graph G is the incidence matrix, in the sense of directed graphs, of any orientation of G. That is, in the column of edge e, there is a +1 in the row corresponding to one vertex of e and a -1 in the row corresponding to the other vertex of e, and all other rows have 0. All oriented incidence matrices of G differ only by negating some set of columns. In many uses, this is an insignificant difference, so one can speak of the oriented incidence matrix, even though that is technically incorrect.

2.2.7. Graph Operations

Graph algorithms are a significant field of interest for computer scientists. Typical operations associated with graphs are: finding a path between two nodes, like depth-first search and breadth-first search and finding the shortest path from one node to another, like Dijkstra's algorithm.

2.2.7.1 Depth-first search

Depth-first search (DFS) is an algorithm for traversing or searching a tree, tree structure, or graph. Intuitively, one starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.

Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a LIFO stack for expansion.

Space complexity of DFS is much lower than BFS (breadth-first search). It also lends itself much better to heuristic methods of choosing a likely-looking branch. Time complexity of both algorithms is proportional to the number of vertices plus the number of edges in the graphs they traverse.

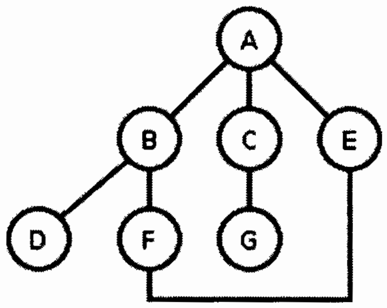


Figure 2.14: Depth First Search

When searching large graphs that can not be fully contained in memory, DFS suffers from non-termination when the length of a path in the search tree is infinite. The simple solution of "remember which nodes I have already seen" doesn't always work because there can be insufficient memory. This can be solved by maintaining an increasing limit on the depth of the tree, which is called iterative deepening depth-first search.

a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G.

Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.

Iterative deepening prevents this loop and will reach the following nodes on the following depths, assuming it proceeds left-to-right as above:

- 0: A
- 1: A (repeated), B, C, E

(Note that iterative deepening has now seen C, when a conventional depth-first search did not.)

- 2: A, B, D, F, C, G, E, F

(Note that it still sees C, but that it came later. Also note that it sees E via a different path, and loops back to F twice.)

- 3: A, B, D, F, E, C, G, E, F, B

For this graph, as more depth is added, the two cycles "ABFE" and "AEFB" will simply get longer before the algorithm gives up and tries another branch.

2.2.7.2. Vertex orderings

It is also possible to use the depth-first search to linearly order the vertices (or nodes) of the original graph (or tree). There are three common ways of doing this:

- A preordering is a list of the vertices in the order that they were first visited by the depth-first search algorithm. This is a compact and natural way of describing the progress of the search, as was done earlier in this article. A preordering of an expression tree is the expression in Polish notation.

- A postordering is a list of the vertices in the order that they were last visited by the algorithm. A postordering of an expression tree is the expression in reverse Polish notation.

- A reverse postordering is the reverse of a postordering, i.e. a list of the vertices in the opposite order of their last visit. When searching a tree, reverse postordering is the same as preordering, but in general they are different when searching a

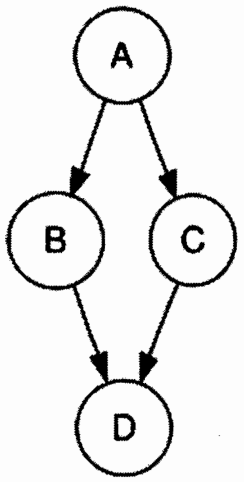


Figure 2.15: Vertex Ordering

graph. For example, when searching the graph beginning at node A, the possible preorderings are A B D C and A C D B (depending upon whether the algorithm chooses to visit B or C first), while the possible reverse postorderings are A B C D and A C B D. Reverse postordering produces a topological sorting of any directed acyclic graph. This ordering is also useful in control flow analysis as it often represents a natural linearization of the control flow. The graph above might represent the flow of control in a code fragment like

```
if (A) then {
     B
    } else {
     C
    }
    D
```

and it is natural to consider this code in the order A B C D or A C B D, but not natural to use the order A B D C or A C D B.

2.2.7.3. Breadth-first search

In graph theory, breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

BFS is an uninformed search method that aims to expand and examine all nodes of a graph systematically in search of a solution. In other words, it exhaustively searches the entire graph without considering the goal until it finds it. It does not use a heuristic.

From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

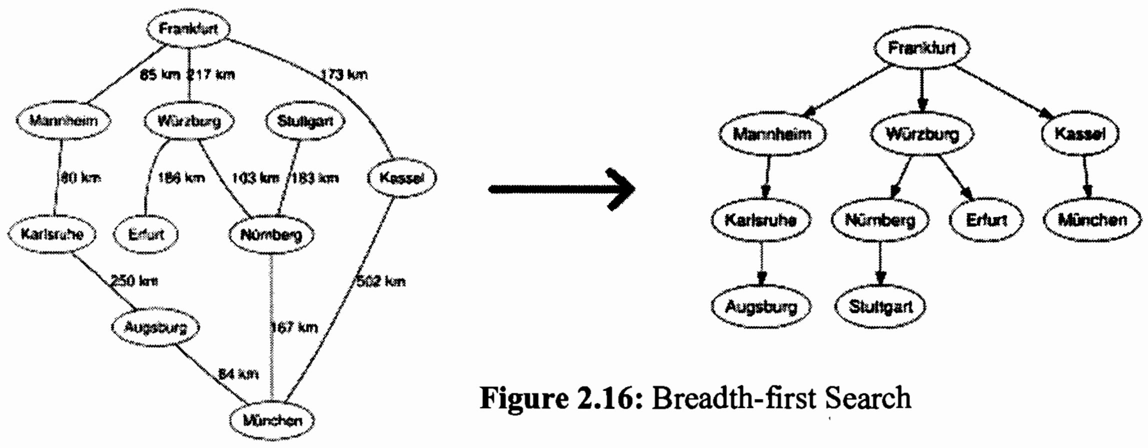


Figure 2.16: Breadth-first Search

2.2.7.3.1. Algorithm:

1. Put the ending node (the root node) in the queue.
2. Pull a node from the beginning of the queue and examine it.

    - If the searched element is found in this node, quit the search and return a result.

- Otherwise push all the (so-far-unexamined) successors of this node into the end of the queue, if there are any.

3. If the queue is empty, every node on the graph has been examined -- quit the search and return "not found".
4. Repeat from step 2.

2.2.7.3.2. Applications of BFS

Breadth-first search can be used to solve many problems in graph theory, for example:

- Finding all connected components in a graph.
- Finding all nodes within one connected component
- Finding the shortest path between two nodes u and v (in an unweighted graph)
- Testing a graph for bipartiteness

2.3. Algorithms

In this section, we will discuss algorithms and related concepts like their efficiency and complexity and importance of writing efficient algorithms. Different approaches to target a particular problem and select the best algorithm for that problem.

2.3.1. Concepts and Definitions

A computer program is composed of individual modules, understandable by a computer, that solve specific tasks (such as sorting). Our concern in this text is not the design of entire programs, but rather the design of the individual modules that accomplish the specific tasks. These specific tasks are called problems. Explicitly, we say that *a problem* is a question to which we seek an answer. Examples of problems follow.

Example 2.1 The following is an example of a problem:

- Sort the list S of n non decreasing order. The answer is the number in sorted sequence

By a list we mean a collection of items arranged in a particular sequence. For example,

$$S = [10, 7, 11, 5, 13, 8]$$

is a list of six number in which the first number is 10, the second is 7 and so on. We say "non decreasing order" in Example 2.1 instead of increasing order to allow for the possibility that the same number may appear more than once in the list.

Example 2.2 The following is an example of a problem.

- Determine whether the number x is in the list S of numbers. The answer is yes if x is in S and no if it is not.

A problem may contain variables that are not assigned specific values in the statement of the problem. The variables are called ***parameter*** to the problem. In Example 2.1 there are two parameters: S (the list) and n (the number of items in S). In Example 2.2 there are three parameters: S, n and the number x. It is not necessary in these two examples to make n one of the parameters because its value is uniquely determined by S. However, making n a parameter facilitates our descriptions of problems.

Because a problem contains parameters, it represents a class of problems one for each assignment of values to the parameters. Each specific assignment of values to the parameters is called an ***instance*** of the problem. A ***solution*** to an instance of a problem is the answer to the question asked by the problem in the instance.

Example 2.3 An instance of the problem in Example 2.1 is

S = [10, 7, 11, 5, 13, 8] and n = 6.

The solution to this instance is [5, 7, 8, 10, 11, 13]

Example 2.4 An instance of the problem in Example 2.2 is

S = [10, 7, 11, 5, 13, 8] n = 6 and x = 5.

The solution to this instance is, "yes, *x* is in S."

We can find the solution to the instance in Example 2.3 by inspecting S and allowing the mind to produce the sorted sequence by cognitive steps that cannot be specifically described. This can be done because S is so small that at a conscious level. The mind seems to scans S rapidly and produces the solution almost immediately (and therefore one cannot describe the steps the mind follows to obtain the solution). However, if the instance had a value of 1000 for n, the mind would not be able to use this method, and it certainly would not be possible to convert such a method of sorting numbers to a computer program. To produce eventually a computer program that can solve all instances of a problem, (we must specify) a general step-by-step procedure for producing the solution to each instance. This step-by-step procedure is called an ***algorithm***. We say that the algorithm solves the problem.

Example 2.5 An algorithm for the problem in Example 2.2 is as follows. Starting with the first item in S, compare *x* with each item in S in sequence until *x* is found or until S is exhausted. If *x* is found, answer yes; if *x* is not found, answer no.

The following algorithm represents the list S by an array and, instead of merely returning yes or no, returns *x*'s location in the array if *x* is in S and returns 0 otherwise. This particular searching algorithm does not require that the items come from an ordered set, but we still use our standard data type.

Algorithm 2.1: Sequential Search

Problem: Is the key *x* in the array S of n keys?
Inputs: positive integer n, array of keys S index form 1 to n, and a key *x*.
Outputs: location, the location of *x* in S (0 if *x* is not in S).

```
procedure   seqsearch  (n:integer;
        S:array[1..n] of keytype;
        x: keytype;
        var location:index)
begin
```

```
    location:= 1;
    while location $\le n$ and S[location] $\ne x$ do
        location:=location+1;
    end;
    if location>n then
        location:=0;
    end;
end;
```

Besides the data type **keytype,** we often use the following, which might not be predefined in a particular language data types;

Data type	Meaning
Index	An integer variable used as an index.
Number	A variable that could be defined as integral (int) or real (float).
Bool	A variable that can take the values "true" or "false"

We use the data type number when it is not important to the algorithm whether the numbers can take any real values or are restricted to the integers.

Sometimes we use the following nonstandard control structure:

- ***Repeat (n times)*** { }

This means repeat the code n times. In programming languages, it would be necessary to introduce an extraneous control variable and write a *for loop*. We only use a for loop when we actually need to refer to the control variable within the loop.

When the name of an algorithm seems appropriate for a value it returns, we write the algorithm as a function. Otherwise, we write the algorithm as a procedure (void function in C++) and use reference parameters (that is, parameters that are passed by address) to return values If the parameter is not an array, it is declared with an ampersand (&) at the end of the data type name. For our purposes this means that the parameter contains a value returned by the algorithm. Because arrays are automatically passed by reference in C++ and the ampersand is not used in C++ when passing arrays, we do not use the ampersand to indicate that and array contains values returned by the algorithm. Instead, since the reversed word const is used in C++ to prevent modification of a passed array, we use const to indicate that the array does not contain values returned by the algorithm.

In general, we avoid features peculiar to C++, so that the pseudocode is accessible to someone who knows only another high-level language. However, we do write instructions like i++ which means increment i by 1.

2.3.2. The Importance of Developing Efficient Algorithms

Previously we mentioned that, regardless of how fast computers become or how cheap memory gets, efficiency will always remain an important consideration. Next we show why this is so by comparing two algorithms for the same problem.

2.3.2.1. Sequential Search versus Binary Search

Earlier we mentioned that the approach used to find a name in the phone book is a modified binary search, and is usually much faster than a sequential search. Next we compare algorithms for the two approaches to show how much faster the binary search is.

We have already written an algorithm that does a sequential search – namely, Algorithm 2.1. An algorithm for doing a binary search of an array that is sorted in non decreasing order is similar to thumbing back and forth in a phone book. That is, given that we are searching for *x*, the algorithm first compares *x* with the middle item of the array. If they are equal, the algorithm is done. If *x* is smaller than the middle item, then *x* must be in the first half of the array (if it is present at all), and the algorithm repeats the searching procedure on the first half of the array. (That is, *x* is compared with the middle item of the first half of the array. If they are equal, the algorithm is done, etc.) If *x* is larger than the middle item of the array, the search is repeated on the second half of the array. This procedure is repeated until *x* is found or it determined that *x* is not in the array. An algorithm for this method follows.

Algorithm 2.2 Binary search

```
Problem: Determine whether x is in the sorted array S of n keys.
Input:  Positive integer, sorted (non decreasing) array of keys S
indexed from 1 to n, a key x.
Out put: location, the location of x in S (0 if x is not in S)
procedure binsearch (n: integer;
            S:  array[1..n] of keytype;
            x: keytype,
            var location: index )
var
        low, high, mid: index;
begin
        low := l; high := n;
        location := 0;
        while low = high and location = 0 do
            mid = [(low+high) div 2;
            if  x = s [mid] then
                location := mid
            else if x < s [mid] then
                high = mid – l;
            else
                low =mid + l;
            end;
        end;
end;
```

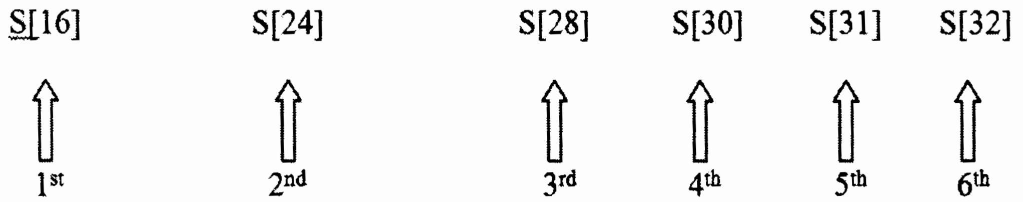
Let's compare the work done by Sequential Search and Binary Search. For focus we will determine the number of comparisons done by each algorithm. If the array S contains 32 items and x is not in the array, Algorithm 2.1 (Sequential Search) compares x with all 32 items before determining that x is not in the array.

In general, Sequential Search does n comparisons to determine that x is not in array of size n. It should be clear that this is the most comparisons Sequential Search ever makes when searching an array of size n. That is, if x is in the array, the number of comparison is not greater than n.

Next consider Algorithm 2.2 (Binary Search). There are two comparisons of x with S[mid] in each pass through the while loop (except when x is found). In an efficient assembler language implementation of the algorithm, x would be compared with S[mid] only once in each pass, the result of that comparison would set the condition code, and the appropriate branch would take place based on the value of the condition code. This means that there would be only one comparison of x with S [mid] in each pass through loop. We will assume the algorithm is implemented in this manner. With this assumption, Figure 2.1 shows array of size 32. Notice that 6=lg32+1. By "lg" we mean $\log_2$. The $\log_2$ is encountered so often in analysis of algorithms that we reserve the special symbol lg for it. You should convince yourself that this is the most comparisons Binary Search ever does. That is, if x is in the array, or if x is smaller than all the array items, or if x is between two array items, the number of comparisons is not greater than when x is larger than all the array items.

Suppose we double the size of the array so that it contains 64 items. Binary Search does only one comparison more because the first comparison more because the first comparison cuts the array in half, resulting in a subarray of size 32 that is searched. Therefore, when x is larger than all the items in an array of size 64, Binary Search does seven comparisons. Notice that $7 = \lg 64+1$. In general, each time we double the size of the array we add only one comparison we add only one comparison. Therefore, if n is a power of 2 and x is larger than all the items in an array of size *n* the number of comparisons done by Binary Search is lg n + 1.

Table 2.1 compares the number of comparison done by Sequential Search and Binary Search for various values of *n*, when *x* is larger than all the items in the



(**Figure**2.17 The array items that Binary search compares with *x* when *x* is larger than all the items in an array of size 32. The items are numbered according to the order in which are compared)

array. When the array contains around 4 billion items (about the number of people in the world), Binary Search does only 33 comparisons whereas Sequential Search compares *x* with all 4 billion items. Even if the computer was capable of completing one pass through the while loop in a nanosecond (one billionth of a second), Sequential Search would take 4 seconds to determine that *x* is not in the

array, whereas Binary Search would make that determination almost instantaneously. This difference would be significant in an on-line application or if we needed to search for many items.

Table 2.1 The numbers of comparisons done by Sequential Search and Binary Search when *x* is larger than all the array items

Array Size	Number of comparisons by Sequential Search	Number of comparisons by Binary Search
128	128	8
1,024	1,024	11
1,048,576	1,048,576	21
4,294,967,296	4,294,967,296	33

For convenience, we considered only arrays whose size were powers of 2 in the previous discussion of Binary Search. It is an example of the divide-and-conquer approach.

As impressive as the searching example is it is not absolutely compelling because sequential search still gets the job done in an amount of time tolerable to a human life span. Next we will look at an inferior algorithm that does not get the job done in a tolerable amount of time.

2.3.3. The Fibonacci Sequence

The algorithm discussed here computes the nth term of the Fibonacci sequence, which is defined recursively as follows:

$$f_0 = 0$$
$$f_1 = 1$$
$$f_n = f_{n-1} + f_{n-2} \quad n \geq 2$$

Computing the first few terms, we have

$$f_2 = f_1 + f_0 = 1+0 = 1$$
$$f_3 = f_2 + f_1 = 1+1 = 2$$
$$f_4 = f_3 + f_2 = 2+1 = 3$$
$$f_5 = f_4 + f_3 = 3+2 = 5 \quad \text{etc.}$$

There are various applications of the Fibonacci sequence in computer science and mathematics. Because the Fibonacci sequence is defined recursively, we use the following recursive algorithm form the definition.

Algorithm 2.3 nth Fibonacci Term (Recursive)

Problem: Determine the nth term in the Fibonacci sequence,

Inputs: a non negative integer n.

Outputs: fib, the nth term of the Fibonacci sequence.

```
function fib (n: integer): integer;
begin
        if  n ≤ 1 then
                fib:=n;
```

```
            else
                    fib:=fib (n-1)+ fib(n-2);
            end;
    end;
```

By "nonnegative integer" we mean an integer that is greater than or equal to 0, whereas by "positive integer" we mean an integer that is strictly greater than 0. We specify the input to the algorithm in this manner to make it clear what values the input can take. However, for the sake of avoiding clutter, we declare *n* simply as an integer in the expression of the algorithm. We will follow this convention throughout the text.

Although the algorithm was easy to create and is understandable. it is extremely inefficient. When computing fib (5), we need fib (4) and fib (3). Then to obtain fib(3) we need fib (2) and fib(1) over and over again. For example, Fib (2) is computed three times.

How inefficient is this algorithm? The algorithm computes the following numbers of terms to determine fib (n) for $0 \leq n \leq 6$;

n	**Number of Terms computed**
0	1
1	1
2	3
3	5
4	9
5	15
6	25

The first six values can be obtained by counting the nodes in the subtree rooted at fib(n) for $1 \leq n \leq 5$. Whereas the number of terms for fib(6) is the sum of the nodes in the tree rooted at fib (5) and fib (4) plus the one node at the root. These numbers do not suggest a simple expression like the one obtained for Binary

Search. Notice, however, that in the case of the first seven values, the number of terms in the tree for Fibonacci series is more than doubles every time n increases by 2. for example, there are nine terms in the tree when n = 4 and 25 terms when n = 6. Let's call T(n) the number of terms in the recursion tree for n. If the number of terms more than doubled every time *n* increased by 2, we would have the following for *n* a positive power of 2:

$$\begin{aligned} T(n) &> 2 \times T(n-2) \\ &> 2 \times 2 \times T(n-4) \\ &> 2 \times 2 \times 2\, T(n-6) \\ &\quad \bullet \\ &\quad \bullet \\ &> \underbrace{2 \times 2 \times 2 \times \ldots \times 2}_{n/2 \text{ terms}} \times T(0) \end{aligned}$$

Because T(0) = 1, this would mean $T(n) > 2^{n/2}$.

Algorithm 2.4 nth Fibonacci Term (iterative)

```
Problem:   Determine the nth term in the Fibonacci Sequence.
Inputs:    a non negative integer n.
Outputs:   fib 2, the nth term in the Fibonacci Sequence.
function fib2 (integer n):integer;
var
    i: index;
    f:array[0..n] of integer;
begin
    f[0] := 0;
    if n>0 then
        f [1] = 1;
        for i := 2 to n do
            f[i] := f [i-1] + f[i-2];
            end;
        end;
        fib2:=f[n];
    end;
```

Algorithm 2.4 can be written without using the array f because only the two most recent terms are needed in each iteration of the loop. However, it is more clearly illustrated using the array.

To determine fib2(n) the previous algorithm computes every one of the first n + 1 terms just once. So it computes n+1 terms to determine the nth Fibonacci term. Recall that Algorithm 2.3 computes more then $2^{n/2}$ terms to determine the *n*th Fibonacci term. Table 2.2 compares these expressions for various values of *n*. The execution times are based on the simplifying assumption that one term can be computed in 10^{-9} second. The table show the time it would take Algorithm 2.4 to computes the nth term on a hypothetical computer that could compute each term in a nanosecond, and it shows a lower bound on the time it would take to execute Algorithm 2.4. By the time *n* is 80, Algorithm 2.3 takes at least 18 minutes. When *n* is 120, it takes more then 36 years, and amount of time intolerable to a human life span. Even if we could build a computer one billion times as fast. Result can be obtained by diving the time for the 200th term by one billion. We see that regardless of how fast computer become, Algorithm 2.3 will still take an intolerable amount of time unless *n* is small. On the other hand, Algorithm 2.4 computes the *n*th Fibonacci term almost instantaneously. This comparison shows why the efficiency of an algorithm remains an important consideration regardless of how fast computer become.

Table 2.2 A comparison of Algorithms 4.3 and 4.4

n	n+1	$2^{n/2}$	Execution Time Using Algorithm 4.4	Lower Bound Execution Time Using Algorithm 4.3
40	41	1,048.576	41 ns*	1048 us*
60	61	1.1×10^9	61 ns	1 s

80	81	1.1×10^{12}	81 ns	18 min
100	101	1.1×10^{15}	101 ns	13 days
120	121	1.2×10^{18}	121 ns	36 days
160	161	1.2×10^{24}	161 ns	3.8×101^{7} years
200	201	1.3×10^{30}	201 ns	4×10^{13} years

*1 ns = 10^{-9} second
*1us = 10^{-6} second

Algorithm 2.3 is a divide-and conquer algorithm. Recall that the divide-and-conquer approach produced a very efficient algorithm (Algorithm 2.2: Binary Search) for the problem of searching a sorted array. The divide-and-conquer approach leads to very efficient algorithms for some problems, but very inefficient algorithms for other problems. Our efficient algorithm for computing the nth Fibonacci term (Algorithm 2.4) is an example of the dynamic programming approach. We see that choosing the best approach can be essential.

We showed that Algorithm 2.3 computes at least an exponentially large number of terms, but could it be even worse? The answer is no. It is possible to obtain an exact formula for the number of terms, for further discussion of the Fibonacci sequence.

Algorithm 2.5 Add array members

Problem: Add all the numbers in the array S of n numbers.
Inputs: positive integer n, array of numbers S indexed from 1 to n.
Outputs: sum, the sum of the numbers in S.

```
function sum (n:integer; S:array[1..n] of number): number;
var
    i: index;
    sum: number;
begin
```

```
        sum := 0;
        **for** i := 1 to n **do**
            sum := sum + S[i] ;
        **end**;
    **end**;
```

2.3.4. Analysis of Algorithms

To determine how efficiently an algorithm solves a problem, we need to analyze the algorithm. We introduced efficiency analysis of algorithms when we compared the algorithms in the preceding section. However, we did those analyses rather informally. We will now discuss terminology used in analyzing algorithms and the standard methods for doing analyses. We will adhere to these standards in the remainder of the text.

2.3.4.1. Time Complexity Analysis

When analyzing the efficiency of an algorithm in terms of time, we do not determine the actual number of CPU cycles, because this depends on the particular computer on which the algorithm is run. Furthermore, we do not even want to count every instruction executed, because the number of instructions depends on the programming language used to implement the algorithm and the way the programmer writes the program. Rather, we want a measure that is independent of the computer, the programming language, the programmer and all the complex details of the algorithm such as incrementing of loop indices, setting of pointers etc. We learned that Algorithm 2.2 is much more efficient than Algorithm 2.1 by comparing the numbers of comparisons done by the two algorithms for various values of n, where n is the number of items in the array. This is a standard technique for analyzing algorithms. In general, the running time of an algorithm increases with the size of the input, and the total running time is roughly proportional to how many times some basic operation (such as comparison instruction) is done. We therefore analyze the algorithm's efficiency by determining the number of times some basic operation is done as a function of the size of the input.

For many algorithms it is easy to find a reasonable measure of the size of the input, which we call the input size. For example, consider Algorithms 2.1 (Sequential Search), 2.2 (Binary Search) and 2.5 (Add Array Members). In all these algorithms, *n*, the number of items in the array, is a simple measure of the size of the input. Therefore, we can call *n* the input size. In some algorithms, it is more appropriate to measure the size of the input using two numbers. For example, **when the input to an algorithm is a graph, we often measure the size of the input in terms of both the number of vertices and the number of edges.** Therefore, we say that the input size consists of both parameters.

Sometimes we must be cautious about calling a parameter the input size. For example, in Algorithms 2.3 (nth Fibonacci Term, Recursive) and 2.4 (nth Fibonacci Term, Iterative), you may think that *n* should be called the input size. However, *n* is the input; it is not the size of the input. For this algorithm, a reasonable measure of the size of the input is the number of symbols used to encode *n*. If we use binary representation, the input size will be the number of bits it takes to encode *n*, which is (floor) [lg n] + 1. For example

$$n= 13 = 1101_2 \text{ (4 bits)}$$

Therefore, the size of the input *n* = 13 is 4. We gained insight into the relative efficiency of the two algorithms by determining the number of terms each computes as a function of *n*, but still *n* does not measure the size of the input. It will usually suffice to use a simple measure, such as the number of times this instruction or group of instructions is done. We call this instruction or group of instructions the **basic operation** in the algorithm. For example, *x* is compared with an item S in each pass through the loops in Algorithms 2.1. Therefore, the compare instruction is a good candidate for the basic operation in each of these algorithms. By determining how many times Algorithm 2.1 do this basic operation for each value of n, we gained insight into the relative efficiency of the algorithm.

In general, at **time complexity analysis** of an algorithm is the determination of how many times the basic operation is done for each value of the input size. Although we do not want to consider the details of how an algorithm is implemented. We will ordinary assume that the basic operation is implemented as efficiently as possible. For example we assume that Algorithm 2.1 is implemented such that the comparison is done just once. In this way, we analyze the most efficient implementation of the basic operation.

There is no hard and fast rule for choosing the basic operation. It is largely a matter of judgment and experience. As already mentioned, we ordinarily do not include the instructions that compose the control structure. For example, in Algorithm 2.1 we don't include the instructions that increment and compare the index in order to control the passes through the while loop. Sometimes it suffices simply to consider one pass through a loop as one execution of the basic operation. At the other extreme, for a very detailed analysis one could consider the execution of each machine instruction as doing the basic operation once. As mentioned earlier, because we want our analyses to remain independent of the computer, we will never do that in this text.

At times we may want to consider two different basic operations. For example, in an algorithm that sorts by comparing keys, we often want to consider the comparison instruction and the assignment instruction each individually as compose the basic operation, but rather that we have two instructions together one being the comparison instruction and the other being the assignment instruction . We do this because ordinarily a sorting algorithm does not do the same insight into the efficiency of the algorithm by determining how many times each is done.

As discussed earlier, a time complexity analysis of an algorithm determines how many times the basic operation is done for each value of the input size. In some cases the number of times it is done depends not only on the input size, but also on the input's values. This is the case in Algorithm 2.1 (Sequential Search). For example, if x is the first item in the array, the basic operation is done once, whereas if x is not in the array, it is done n times. In other cases, such as Algorithm 2.5 (Add Array Members), the basic operation is always done the same number of times for every instance of size n. When this is the case, T(n) is defined as the number of times the algorithm does the basic operation for an instance of size n. T (n) is called the every-case time complexity of the algorithm, and the determination of T (n) is called an every-case time complexity analysis. Examples of every-case time complexity analysis follow.

2.3.4.2. Every-Case Time Complexity Analysis of Algorithm 2.5

Basic operation: the addition of an item in the array to sum.

Input size: n, the number of items in the array.

Regardless of the value of the numbers in the array, there are n passes through the *for* loop. Therefore, the basic operation is always done n times and

$$T(n) = n.$$

As discussed previously, the basic operation in Algorithm 2.1 is not done the same number of times for all instances of size n. So this algorithm does not have an every-case time complexity. This is true for many algorithms. However, this does not mean that we cannot analyze such algorithms, because there are three other analysis techniques that can be tried. The first is to consider the maximum number of times the basic operation is done. For a given algorithm, W(n) is called worse case time complexity of the algorithm, and the determination of W(n) is called a worst case time complexity analysis. If T(n) exists, then clearly W(N) = T(n). The following is an analysis of W(n) in a case where T(n) does not exist.

2.3.4.3. Worst-Case Time Complexity Analysis of Algorithm 2.1

Basic Operation: the comparison of an item in the array with x.

Input size: n, the number of items in the array.

The basic operation is done at most n times, which is the case if x is the last item in the array or if x is not in the array. Therefore,

$$W(n) = n.$$

Although the worst case analysis informs us of the absolute maximum amount of time consumed, in some cases we may be more interested in knowing how the algorithm performs on the average. For a given algorithm, A(n) is defined as the average (expected value) of the number of times the algorithm does the basic operation for an input size of n. A(n) is called the average case time complexity of the algorithm, and the determination of A(n) is called an average case time complexity analysis. As is the case for W(n), if T(n) exists, then A(n) = T(n).

To compute A(n), we need to assign probabilities to all possible inputs of size. It is important to assign probabilities based on all available information. For example, our next analysis will be average case analysis of Algorithm 2.1. We will assume that if x is in the array, it is equally likely to be in any of the array slots. If we know only that x may be somewhere in the array, our information gives us no reason to prefer one array slot over another. Therefore, it is reasonable to assign equal probabilities to all array slots; this means that we are determining the average search time when we search for all items the same number of times. If we have information indication that the inputs will not arrive according to this distribution, we should not use this distribution in our analysis. For example if chosen at random from all people in the United States, an array slot containing the common name "John" will probably be searched more often that one containing the uncommon name "Felix". We should not ignore this information and assume that all slots are equally likely.

As the following analysis illustrates, it is usually harder to analyze the average case than it is to analyze the worst case.

2.3.4.4. Average-Case Time Complexity Analysis of Algorithm 2.1

Basic operation: The comparison of an item in the array with *x*.

Input size: *n*, the number of items in the array,

We first analyze the case where it is known that *x* is in S, where the items in S are all distinct, and where we have no reason to believe that *x* is more likely to be in one array slot than it is to be in another. Based on this information, for $1 \leq k \leq n$, the probability that *x* is in the *k*th array slots is 1/n. If *x* is in the *k*th array slot, the number of times the basic operation is done to locate *x* (and, therefore, to exit the loop) is *k*. This means that the average time complexity is give by

$$A(n) = \sum_{k=1}^{n} (k \times 1/n) = 1/n \times n(n + 1)/2 =(n+ 1)/2.$$

Next we analyze the case where *x* may not be in the array. To analyze this case we must assign some probability *p* to the event that *x* is in the array. If *x* is in the array. We will again assume that it is equally likely to be in any of the slots from 1 to *n*. The probability that *x* is in the *k*th slot is then *p*/*n*, and the probability that it is not in the array is 1 - p. Recall that there are *k* passes through the loop if *x* is found in the *k*th slot, and *n* passes through the loop if *x* is not in the array.

The average time complexity is therefore given by

$$A(n) = \sum_{k=1}^{n} (k \times p/n) + n(1-p)$$
$$= p/n \times n(n+1)/2 + n(1-p) = n(1-p/2) + p/2$$

The last step in this triple equality is derived with algebraic manipulations. If p = 1, A(n) = (n + 1)/2, as before, whereas if p = 1/2 , A(n) = 3n/4 + 1/4. This means that about 3/4 of the array is searched on the average.

Before proceeding, we offer a word of caution about the average. Although an average is often referred to as a typical occurrence, one must be careful in interpreting the average in this manner. For example, a meteorologist may say that a typical January 25 in Chicago has a high of 22F because 22F has been the average high for that date over the past 80 years. The paper may run an article saying that the typical family in Evanston, Illinois, earns $50,000 annually be cause that is the average income. An average can be called "typical" only if the actual case do not deviate much from the average (that is, only if the standard deviation is small). This may be the case for the high temperature on January 25. However, Evanston is a community with wealthy areas and fairly poor areas. It is more typical for a family to make either $20,000 annually or $100,000 annually than to make $50,000. Recall in the previous analysis that A(n) is (n + 1)/2 when it is known that *x* is in the array. This is not the typical search time, because all search times between 1 and *n* are equally typical. Such considerations are important in algorithms that deal with response time. For example, consider a system that monitors a nuclear power plant. If even a single instance has a bad response time, the results could be catastrophic. It is therefore important to know whether the average response time is 3 second because all response times are around 3 seconds or because most are 1 second and some is 60 second.

A final type of time complexity analysis is the determination of the smallest number of times the basic operation is done. For a given algorithm, B(n) is defined as the minimum number of times the algorithm will ever do its basic operation for and input size of n. So B(n) is called the **best case time complexity** of the algorithm, and the determination of B(n) is called the **best case time complexity analysis.** As is the case for W(n) and A(n) if T(n) exists, then B(n)= T(n). Let's determine B(n) for Algorithm 2.1.

2.3.4.5. Best-Case Time Complexity Analysis of Algorithm 2.1

Basic operation: the comparison of an item in the array with *x*.

Input size: n, the number of items in the array.

Because $n \geq 1$, there must be at least one pass through the loop. If x = S[1],there will be one pass through the loop regardless of the size of n. Therefore,

$$B(n) = 1$$

For algorithms that do not have every case time complexities, we do worst case and average case analyses much more often than best case analyses. An average case analysis is valuable because it tells us how much time the algorithm would take when used many times on many different inputs. This would be useful, all possible inputs. Often, a relatively slow sort can occasionally be tolerated if, on the average, the sorting time is good. For most of the applications, a best-case analysis would be of little value.

We have only discussed analysis of the time complexity of an algorithm. All these same considerations also pertain to analysis of memory complexity, which is an analysis of how efficient the algorithm is in terms of memory. Although most of the analysis in this text is time complexity analyses, we will occasionally find it useful to do a memory complexity analysis.

2.3.4.6. Usage of Analysis Theory

When applying the theory of algorithm analysis, one must sometimes be aware of the time it takes to execute the basic operation, the overhead instructions, and the control instructions on the actual computer on which the algorithm is implemented. By "over head instructions" we mean instructions such as initialization instructions before a loop. The number of times these instructions execute does not increase with input size. The basic operation, over head instructions, and control instructions are all properties of an algorithm and the implementation of the algorithm. They are not properties of a problem. This means that they are usually different for two different algorithms for the same problem.

We are assuming that the time it takes to process the overhead instructions is negligible. If this were not the case, these instructions would also have to be considered to determine the efficiency of an algorithm.

2.4. Algorithm Classes

There are various ways to classify algorithms, each with its own merits.

2.4.1. Classification by implementation

One way to classify algorithms is by implementation means.

- Recursion or iteration: A recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation or the other. For example, towers of hanoi is well understood in recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.
- **Logical:** An algorithm may be viewed as controlled logical deduction. This notion may be expressed as:

  Algorithm = logic + control

  The logic component expresses the axioms which may be used in the computation and the control component determines the way in which deduction is applied to the axioms. This is the basis for the logic programming paradigm. In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component. The appeal of this approach is the elegant semantics: a change in the axioms has a well defined change in the algorithm.

- **Serial or parallel:** Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms, which take advantage of computer architectures where several processors can work on a problem at the same time. Parallel algorithms divide the problem into more symmetrical or asymmetrical subproblems and pass them to many processors and put the results back together at one end. The resource consumption in parallel algorithms is both processor cycles on each processor and also the communication overhead between the processors. Sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Iterative algorithms are generally parallelizable. Some problems have no parallel algorithms, and are called inherently serial problems.
- **Deterministic or non-deterministic:** Deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithm solve problems via guessing although typical guesses are made more accurate through the use of heuristics.
- **Exact or approximate:** While many algorithms reach an exact solution, approximation algorithms seek an approximation which is close to the true solution. Approximation may use either a deterministic or a random strategy. Such algorithms have practical value for many hard problems.

2.4.2. Classification by design paradigm

Another way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories will include many different types of algorithms. Some commonly found paradigms include:

- **Divide and conquer:** A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively), until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in conquer phase by merging them. A simpler variant of divide and conquer is called decrease and conquer algorithm, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem. Divide and conquer divides the problem into multiple subproblems and so conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is binary search algorithm.

- **Dynamic programming:** When a problem shows optimal substructure, meaning the optimal solution to a problem can be constructed from optimal solutions to subproblems, and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, we can often solve the problem quickly using dynamic programming, an approach that avoids recomputing solutions that have already been computed. For example, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is, subproblems are more or less independent in divide and conquer, where as the overlap of subproblems occur in dynamic programming. The difference between the dynamic programming and straightforward recursion is in caching or memoization of recursive calls. Where subproblems are independent, there is no chance of repetition and memoization does not help, so dynamic programming is not a solution for all. By using memoization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.
- **The greedy method:** A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the subproblems do not have to be known at each stage; instead a "greedy" choice can be made of what looks best for the moment. The difference between dynamic programming and the greedy method is, it extends the solution with the best possible decision (not all feasible decisions) at an algorithmic stage based on the current local optimum and the best

decision (not all possible decisions) made in previous stage. It is not exhaustive, and does not give accurate answer to many problems. But when it works, it will be the fastest method. The most popular greedy algorithm is finding the minimal spanning tree as given by Kruskal.

- **Linear programming:** When solving a problem using linear programming, the program is put into a number of linear inequalities and then an attempt is made to maximize (or minimize) the inputs. Many problems (such as the maximum flow for directed graphs) can be stated in a linear programming way, and then be solved by a 'generic' algorithm such as the simplex algorithm. A complex variant of linear programming is called integer programming, where the solution space is restricted to all integers.
- **Reduction:** This is another powerful technique in solving many problems by transforming one problem into another problem. For example, one selection algorithm for finding the median in an unsorted list is first translating this problem into sorting problem and finding the middle element in sorted list. The goal of reduction algorithms is finding the simplest transformation such that complexity of reduction algorithm does not dominate the complexity of reduced algorithm. This technique is also called transform and conquer.
- **Search and Enumeration:** Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes the search algorithms and backtracking.
- **The probabilistic and heuristic paradigm:** Algorithms belonging to this class fit the definition of an algorithm more loosely.

1. Probabilistic algorithms are those that make some choices randomly (or pseudo-randomly); for some problems, it can in fact be proven that the fastest solutions must involve some randomness.
2. Genetic algorithms attempt to find solutions to problems by mimicking biological evolutionary processes, with a cycle of random mutations yielding successive generations of "solutions". Thus, they emulate reproduction and "survival of the

fittest". In genetic programming, this approach is extended to algorithms, by regarding the algorithm itself as a "solution" to a problem. Also there are

3. Heuristic algorithms, whose general purpose is not to find an optimal solution, but an approximate solution where the time or resources to find a perfect solution are not practical. An example of this would be local search, taboo search, or simulated annealing algorithms, a class of heuristic probabilistic algorithms that vary the solution of a problem by a random amount. The name "simulated annealing" alludes to the metallurgic term meaning the heating and cooling of metal to achieve freedom from defects. The purpose of the random variance is to find close to globally optimal solutions rather than simply locally optimal ones, the idea being that the random element will be decreased as the algorithm settles down to a solution.

2.4.3. Classification by field of study

Every field of science has its own problems and needs efficient algorithms. Related problems in one field are often studied together. Some example classes are search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, computational geometric algorithms, combinatorial algorithms, machine learning, cryptography, data compression algorithms and parsing techniques.

Fields tend to overlap with each other, and algorithm advances in one field may improve those of other, sometimes completely unrelated, fields. For example, dynamic programming was originally invented for optimisation of resource consumption in industry, but is now used in solving a broad range of problems in many fields.

2.4.4. Classification by complexity

This is actually problem classification in the strict sense. Some algorithms complete in linear time, and some complete in exponential amount of time, and some never complete. One problem may have multiple algorithms, and some problems may have no algorithms. Some problems have no known efficient algorithms. There are also mappings from some problems to other problems. So computer scientists found it is suitable to classify the problems rather than algorithms into equivalence classes based on the complexity.

2.5. Order of Algorithm

We just illustrated that an algorithm with a time complexity of n is more efficient than with a time complexity of n2 for sufficiently large values of n, regardless of how long it takes to process the basic operations in the two algorithms. Suppose now that we have two algorithms for the same problem, and that their every – case time complexities are 100n for the first algorithm and 0.01n2 for the second algorithm. Using an argument such as the one just given, we can show that the first algorithm will eventually be more efficient than the second one. For example, if it takes the same amount of time to process the basic operations in both algorithms and the overhead is about the same, the first algorithm will be more efficient if

$$0.01n2 > 100n$$

Dividing both sides by 0.01n yields

$$n > 10{,}000.$$

If it takes longer to process the basic operation in the first algorithm than in the second, then there is simply some larger value of n at which the first algorithm becomes more efficient.

Algorithm with time complexities such as n and 100n are called linear –time algorithms because their time complexities are linear in the input size n, whereas algorithms with time complexities such as n2 and 0.01n2 are called quadratic-time algorithms because their time complexities are quadratic in the input size n. There is a fundamental principle here. That is, any linear-time algorithm is eventually more efficient than any quadratic-time algorithm. In the theoretical analysis of an algorithm, we are interested in eventual behavior. Next we will show how algorithms can be grouped according to their eventual behavior. In this way we can readily determine whether one algorithm's eventual behavior is better than another's.

2.5.1. Introduction to Order

Function such as $5n^2$ and $5n^2 +100$ are called pure quadratic functions because they contain no linear term, whereas a function such as $0.1n^2 +n+100$ is called a complete quadratic function because it contains a linear term. Table 1.3 shows that eventually the quadratic term dominates this function. That is, the values of the other terms eventually become insignificant compared with the value of the quadratic term. Therefore, although the function is not a pure quadratic function, we can classify it with pure quadratic functions. This means that if some algorithm has this time complexity, we can call the algorithm a quadratic time algorithm. Intuitively, it seems that we should always be able to throw away low order terms when classifying complexity functions. For example, it seems that we should be able to classify $0.1\ n^3 +10n^2+5n+25$ with pure cubic functions. We will soon establish rigorously that we can do this. First let's impart an intuitive feel for how complexity functions are classified.

The set of all complexity functions that can be classified with pure quadratic functions is called $\Theta(n^2)$, where Θ is the Greek capital letter "theta." If a function is a member of the set $\Theta(n^2)$, we say that the function is order of n^2. For example, because we can throw away low order terms,

$$G(n) = 5n^2 + 100n +20\ E\ \Theta(n^2)$$

which means that G(n) is order n^2.

When an algorithm's time complexity is in $\Theta(n^2)$, the algorithm is called a quadratic time algorithm or a $\Theta(n^2)$ algorithm. We also say that the algorithm is $\Theta(n^2)$. Exchange Sort is a quadratic time algorithm.

Table 2.3 The quadratic term eventually dominates

N	**$0.1\ n^2$**	**$0.1\ n^2 +n + 100$**
10	10	120
20	40	160
50	250	400
100	1,000	1,200
1000	100,000	101,100

Similarly, the set of complexity functions that can be classified with pure cubic functions is called $\Theta(n^3)$, and functions in that set are said to be order of n^3, and so on. We will call these sets complexity categories. The following are some of the most common order categories:

Table 2.4 Some common order catagories

Notation	**Name**	**Example**
$O(\log n)$	logarithmic	Finding an item in a sorted list
$O((\log n)^c)$	polylogarithmic	Deciding if a number is prime with the AKS primality test
$O(n)$	Linear	Finding an item in an unsorted list
$O(n \log n)$	Linearithmic, Loglinear, or Quasilinear	Sorting a list with heapsort
$O(n^2)$	Quadratic	Sorting a list with insertion sort
$O(n^c)$, $c > 1$	Polynomial, sometimes called algebraic	Finding the shortest path on a weighted digraph with the Floyd-Warshall algorithm
$O(c^n)$	exponential, sometimes called geometric	Finding the (exact) solution to the traveling salesperson problem
$O(n!)$	Factorial, sometimes called combinatorial	Determining if two logical statements are equivalent

In this ordering, if f(n) is in a category to the left of the category containing g(n) then f(n) eventually lies beneath g (n) on a graph. One might expect that as long as an algorithm is not an exponential time algorithm, it will be adequate. However, even the quadratic time algorithm takes 31.7 years to process an instance with an input size of 1 billion (10^9). On the other hand, the $\Theta(n\lg n)$ algorithm takes only 29.9 seconds to process such an instance. Ordinary an algorithm has to be $\Theta(n\lg n)$ or better for us to assume that it can process extremely large instances in tolerable amounts of time. This is not to say that algorithms whose time complexities are in the higher order categories are not useful. Algorithms with quadratic, cubic, and even higher order time complexities can often handle the actual instances that arise in many applications.

Here we develop theory that enables us to define order. We accomplish this by presenting two other fundamental concepts. The first is " big O"

2.5.1.1. Big O

For a given complexity function f (n), **Big O** written as O(f (n)) is the set of complexity functions g(n) for which there exists some positive real constant c and some nonnegative integer N such that for all $n \geq N$

$$g(n) \leq c \text{ x } f(n)$$

If $g(n) \in O(f(n))$, we say that g(n). g(n) eventually it falls beneath *cf*(*n*) and stays there. Although $n^2 + 10n$ is initially above $2n^2$ in that figure, for $n \geq 10$

$$n^2 + 10n \leq 2\,n^2$$

We can therefore take c = 2 and N = 10 in the definition of "big O" to conclude that

$$n^2 + 10n \in 0(n^2)$$

If for example, g(n) is in $O(n^2)$, then eventually g(n) lies beneath some pure quadratic function cn^2 on a graph. This means that if g(n) is the time complexity for some algorithm, eventually the running time of the algorithm will be at least as fast as quadratic. For the purposes of analysis, we can say that eventually g(n) is at least as good as a pure quadratic function " Big O" (and other concepts that will be introduced soon) are said to describe the asymptotic behavior of a function because they are concerned only with eventual behavior. We say that "big O" puts an **asymptotic upper bound** on a function.

Just as "big O" puts an asymptotic upper bound on a complexity function, the following concept puts an **asymptotic lower bound** on a complexity function.

2.5.1.2. Omega "Ω"

For a given complexity function f(n), Ω(f(n)) is the set of complexity functions g(n) for which there exists some positive real constant c and some nonnegative integer N such that, for all $n \geq N$.

$$g(n) \geq c \times f(n)$$

The symbol Ω is the Greek capital letter "omega" If g (n).

As is the case for "big O" there are no unique constants c and n for which the conditions in the definition of Ω hold. We can choose whichever ones make our manipulations easiest.

If a function is in Ω (n^2) then eventually the function lies above some pure quadratic function on a graph. For the purposes of analysis, this means that eventually it is at least as bad as a pure quadratic function. However, as the following example illustrates, the function need not be a quadratic function.

If a function is in both $O(n^2)$ and Ω (n^2). We can conclude that eventually the function lies beneath some pure quadratic function on a graph and eventually it lies above some pure quadratic function on a graph. That is, eventually it at least as good as some pure quadratic function and eventually it is at least as bad as some pure quadratic function. We can therefore conclude that its growth is similar to that of a pure quadratic function. This is precisely the result we want for our rigorous notion of order. We have the following definition.

2.5.1.3. Theta "Θ"

For a given complexity function f(n)

$$\Theta(f(n) = O(f(n)) \cap \Omega(f(n))$$

This means that Θ(f(n)) is the set of complexity function g(n) for which there exists some positive real constants c and d and some nonnegative integer N such that, for all $n \geq N$

$$c \times f(n) \leq g(n) \leq d \times f(n)$$

If g (n) € Θ(f(n)), we say that g(n) is order of f(n). Assuming that n € $\Omega(n^2)$ means we are assuming that there exits some positive constant c and some nonnegative integer N such that, for n $\geq$ N

$$1/c = cn^2$$

If we divide both sides of this inequality by on, we have for n $\geq$ N

$$1/c \geq n$$

However, for any n > 1/c, this inequality cannot hold, which means that it cannot hold for all n $\geq$ N. this contradiction proves that n is not in Ω (n^2)

2.5.1.4. Small "o"

For a given complexity function f(n), **small o** written as o(f(n)) is the set of all complexity functions g(n) satisfying the following: For every positive real constant c there exists a nonnegative integer N such that, for all n $\geq$ N,

$$g(n) \leq c \text{ x } f(n)$$

If g(n) є o(f(n)), we say that g (n) is **small o** of f (n). Recall that "big 0" means there must be some real positive constant c for which the bound holds. This definition says that the bound must hold for every real positive constant c. Because the bound holds for every positive c, it holds for arbitrarily small c. For example, if g (n) є 0(n), there is an N such that, for n> N

$$g(n) \leq 0.00001 \text{ x } f(n)$$

We see that g(n) becomes insignificant relative to f(n) as n becomes large. For the purposes of analysis, if g(n) is in o(f(n)), then g(n) is eventually much better than functions such as f(n).

2.6. Computational Complexity Theory

In computer science, computational complexity theory is the branch of the theory of computation that studies the complexity, or efficiency, of solving computational problems. Two major aspects are considered: time complexity and space complexity, which are respectively how many steps does it take to perform a computation, and how much memory is required to perform that computation. The complexity of different algorithms is usually compared on their performance on very large instances and asymptotic notation is often used.

To analyze complexity various abstract machines are used to model computation in thought experiments. Most importantly of these are the Turing machines, which are hypothesized by the Church-Turing Thesis to be able to model any reasonable computation. Problems that can be solved on a particular abstract machine under some constraints are grouped into sets called complexity classes.

One of the most important problems, in both computational complexity theory and computer science in general, is whether the P complexity class is equal to the NP complexity class.

2.6.1. Overview

Complexity theory deals with the relative computational difficulty of computable functions. This differs from computability theory, which deals with whether a problem can be solved at all, regardless of the resources required.

A single "problem" is a complete set of related questions, where each question is a finite-length string. For example, the problem FACTORIZE is: given an integer written in binary, return all of the prime factors of that number. A particular question is called an instance. For example, "give the factors of the number 15" is one instance of the FACTORIZE problem.

The time complexity of a problem as discussed earlier is the number of steps that it takes to solve an instance of the problem as a function of the size of the input (usually measured in bits), using the most efficient algorithm. To understand this intuitively, consider the example of an instance that is n bits long that can be solved in n^2 steps. In this example we say the problem has a time complexity of n^2. Of course, the exact number of steps will depend on exactly what machine or language is being used. To avoid that problem, the Big O notation is generally used. If a problem has time complexity $O(n^2)$ on one typical computer, then it will also have complexity $O(n^2p(n))$ on most other computers for some polynomial p(n), so this notation allows us to generalize away from the details of a particular computer.

Example 2.6.1: Mowing grass has linear complexity because it takes double the time to mow double the area. However, looking up something in a dictionary has only logarithmic complexity because a double sized dictionary only has to be opened one time more (e.g. exactly in the middle - then the problem is reduced to the half).

2.6.2. Decision problems

Much of complexity theory deals with decision problems. A decision problem is a problem where the answer is always yes or no. Complexity theory distinguishes between problems verifying yes and problems verifying no answers. A problem that reverse the yes and no answers of another problem is called a complement of that problem.

For example, a well known decision problem IS-PRIME returns a yes answer when a given input is a prime and a no otherwise. While the problem IS-COMPOSITE determines whether a given integer is not a prime number (i.e. a composite number). When IS-PRIME returns a yes, IS-COMPOSITE returns a no, and vice versa. So the IS-COMPOSITE is a complement of IS-PRIME, and similarly IS-PRIME is a complement of IS-COMPOSITE.

Decision problems are often considered because an arbitrary problem can always be reduced to some decision problem. For instance, the problem HAS-FACTOR is: given integers n and k written in binary, return whether n has any prime factors less than k. If we can solve HAS-FACTOR with a certain amount of resources, then we can use that solution to solve FACTORIZE without much more resources. This is accomplished by doing a binary search on k until the smallest factor of n is found, then dividing out that factor and repeating until all the factors are found.

An important result in complexity theory is the fact that no matter how hard a problem can get (i.e. how much time and space resources it requires), there will always be even harder problems. For time complexity, this is determined by the time hierarchy theorem. A similar space hierarchy theorem can also be derived.

2.6.3. Complexity classes

A complexity class is the set of all of the computational problems which can be solved using a certain amount of a certain computational resource.

2.6.3.1. Complexity Class P

The complexity class P is the set of decision problems that can be solved by a deterministic machine in polynomial time. This class corresponds to an intuitive idea of the problems which can be effectively solved in the worst cases.

Polynomial-time algorithms are closed under composition. Intuitively, this says that if I write a function which is polynomial-time assuming that function calls are constant-time, and if those called functions themselves require polynomial time, then the entire algorithm takes polynomial time. One consequence of this is that P is low for itself. This is also one of the main reasons that P is considered to be a machine-independent class; any machine "feature", such as random access, which can be simulated in polynomial time can simply be composed with the main polynomial-time algorithm to reduce it to a polynomial-time algorithm on a more basic machine.

2.6.3.2. Complexity Class NP

The complexity class NP is the set of decision problems that can be solved by a non-deterministic machine in polynomial time. This class contains many problems that people would like to be able to solve effectively, including the Boolean satisfiability problem, the Hamiltonian path problem and the Vertex cover problem. All the problems in this class have the property that their solutions can be checked effectively.

- **Introduction and applications**

The importance of this class of decision problems is that it contains many interesting searching and optimization problems where we want to know if there exists a certain solution for a certain problem.

Thus, the challenge of NP problems is to efficiently find the answer, given an efficient (polynomial-time) way of verifying it once it is found. This challenge was solved for compositeness testing only in 2002; there is still no known polynomial-time way to solve the more general factoring problem of determining whether a number between 1 and m divides n.

- **Why Some NP Problems are Hard**

Because of the many important problems in this class, there have been extensive efforts to find algorithms that decide the problems in NP in time which is polynomial in the input size, which is generally considered efficient. However, there are a large number of problems in NP that defy such attempts, seeming to require super-polynomial time. Whether these problems really aren't solvable in polynomial time is one of the greatest open questions in computer science.

An important notion in this context is the set of NP-complete decision problems, which is a subset of NP and might be informally described as the "hardest" problems in NP. If there is a polynomial-time algorithm for even one of them, then there is a polynomial-time algorithm for all the problems in NP. Because of this, and because dedicated research has failed to find a polynomial algorithm for any NP-complete problem, once a problem has been proven to be NP-complete this is widely regarded as a sign that a polynomial algorithm for this problem is unlikely to exist.

Many complexity classes can be characterized in terms of the mathematical logic needed to express them - this field is called descriptive complexity.

2.6.4. Open questions

- **The P = NP question:** The question of whether *NP* is the same set as *P* (that is whether problems that can be solved in *non-deterministic* polynomial time can be solved in *deterministic* polynomial time) is one of the most important open questions in theoretical computer science. Questions like this motivate the concepts of *hard* and *complete*. A set of problems *X* is hard for a set of problems *Y* if every problem in *Y* can be transformed "easily" into some problem in *X* that produces the same solution. The definition of "easily" is different in different contexts. An important *hard* set in complexity theory is NP-hard - a set of problems, which are not necessarily in *NP* themselves, to which any *NP* problem can be reduced in polynomial time. Set *X* is complete for *Y* if it is hard for *Y*, and is also a subset of *Y*. An important complete set in complexity theory is the NP-complete set. This set contains the most "difficult" problems in NP, in the sense that they are the ones most likely not to be in P. Due to the fact that the problem of $P = NP$ remains unsolved, being able to reduce a problem to a known NP-complete problem would indicate that there is no known polynomial-time solution for it. Similarly, because all *NP* problems can be reduced to the set, finding an *NP-complete* problem that can be solved in polynomial time would mean that $P = NP$.

- **Incomplete problems in NP:** Another open question related to the $P = NP$ problem is whether there exist problems that are in NP, but not in P, that are not NP-complete. In other words problems that has to be solved in non-deterministic polynomial time, but cannot be reduced to in polynomial time from other non-deterministic polynomial time problems. One such problem, that is known to be NP but not known to be NP-complete, is the graph isomorphism problem - a problem that tries to decide wether two graphs are isomorphic (i.e. share the same properties). It has been shown that if $P \neq NP$ then such problems exist.

2.6.5. Intractability

Problems that are solvable in theory, but cannot be solved in practice, are called *intractable*. What can be solved "in practice" is open to debate, but in general only problems that have polynomial-time solutions are solvable for more than the smallest inputs. Problems that are known to be intractable include those that are EXPTIME-complete. If NP is not the same as P, then the NP-complete problems are also intractable.

To see why exponential-time solutions are not usable in practice, consider a problem that requires 2^n operations to solve (n is the size of the input). For a relatively small input size of n=100, and assuming a computer that can perform 10^{10} (10 giga) operations per second, a solution would take about $4*10^{12}$ years, much longer than the current age of the universe.

Chapter 3

Literature Survey

3. Literature Survey

In this chapter we shall cover the literature survey and what benefits we got from different research papers we studied during the process of find the solution for Hamilton Circuit problem. We mainly gone through the digital libraries of ACM/SIAM, IEEE and CiteSeer and found/collected the relevant research paper to get knowledge of existing research on this topic. Some of these research papers were fairly supportive and useful in our work. Here are a few research paper we would like to discuss as these show the existing work and dimension of research on Hamilton Circuit Algorithm.

3.1. Paper 1: A Search Procedure for Hamilton Paths and Circuits [11]

ABSTRACT: A search procedure is given which will determine whether Hamilton paths or circuits exist in a given graph, and will find one or all of them. A combined procedure is given for both directed and undirected graphs. The search consists of creating partial paths and making deductions which determine whether each partial path is a section of any Hamilton path whatever, and which direct the extension of the partial paths.

REVIEW: Techniques for finding Hamilton circuits and paths are fairly numerous. Most involve exhaustive searches, carried out sequentially or in parallel, which eliminate partial paths only when they double back upon themselves.

Hakimi improves upon these methods by the addition of deduction rules which allow earlier termination of partial paths, and elimination of certain edges from consideration. The present paper extends this method by adding some additional deduction rules and extending the method to directed as well as undirected graphs.

This is perhaps the best research towards finding Hamilton Path or Hamilton Circuit from a graph. In this paper the author gone through both the directed and undirected graphs. The author gave a simple search procedure which selects any single node as the initial path. Then path is started from that node to adjacent or successor node and test for admissibility. If test is passed, next success is selected. Also graphs are deduced to simple smaller graphs and solved individually and check for connectivity on each step. Algorithm also applies the concept of deletion of unused edges from the path to simplify the search.

3.2. Paper 2: Hamilton Cycles in Random Subgraphs of Pseudo-Random Graphs [13]

Abstract: Given an r-regular graph G on n vertices with a Hamilton cycle, order its edges randomly and insert them one by one according to the chosen order, starting from the empty graph. We prove that if the eigenvalue of the adjacency matrix of G with the second largest absolute value satisfies $\lambda = o(r^{5/2}/(n^{3/2}(\log n)^{3/2}))$,then for almost all orderings of the edges of G at the very moment τ^* when all degrees of the obtained random subgraph $H\tau^*$ of G become at least two, $H\tau^*$ has a Hamilton cycle. As a consequence we derive the value of the threshold for the appearance of a Hamilton cycle in a random subgraph of a pseudo-random graph G, satisfying the above stated condition.

Review: This paper can be viewed as the first step in studying random sub-graphs of pseudo-random graphs. Questions of a similar kind can be asked with respect to other properties of pseudo-random graphs, like independence and chromatic numbers, existence of perfect matching, factors and many others. Their study should combine existing techniques for the binomial random graphs G (n, p) with known results on the edge distribution of pseudo-random graphs.

This paper mainly focuses on studying random subgraphs and tries to find the number of Hamilton Circuits approximation in these sub-graphs. This paper doesn't discuss any algorithm but tried to find different properties and to mathematically prove these theorems and corollaries. But we can get an idea of sub-graphs study and to solve the sub-graphs as part of our study. We worked on this idea initially and tried to solve the Hamilton Circuit Problem by this approach which is similar to divide and conquer rule. But due to certain failures both in divide and conquer and finding of all Hamilton Circuits in each sub-graph and then regeneration of bigger/complete Hamilton Circuit for whole graph we had to switch our approach to hybrid solution.

3.3. Paper 3: Some Hamilton Paths and a Minimal Change Algorithm [14]

ABSTRACT: A class of graphs whose vertices represent certain combinatorial configurations and whose edges represent minimal changes is defined. A Hamilton path through such a graph indicates the existence of a minimal change algorithm for generating the configurations. Necessary and sufficient conditions for the existence of Hamilton paths are given for this class of graphs.

REVIEW: In this paper authors are concerned with generating all k-subsets of an n-set, that is, all subsets of size k of a set of size n. An algorithm to generate such configurations is presented; which is a minimal change algorithm. Such algorithm exists if n and k are both even, or if $k = 2$ and $n \geq 7$, and that such an algorithm does exist in all other cases where $k \leq 5$.

In this paper, authors tried to give necessary and sufficient conditions for the existence of a Hamilton path in $G_{n,k}$, that is, a path through $G_{n,k}$ which visits every vertex exactly once. Such a path defines a minimal change algorithm, for it is essentially an ordering of the elements of $V_{n,k}$ so that successive elements are adjacent in $G_{n,k}$, that is, they differ by a minimal change which satisfies.

In second section authors proved that if $G_{n,k}$ has a Hamilton path and $1 < k < n - 1$, then n is even and k is odd. They do so by considering $G_{n,k}$ as bipartite and then we calculate the difference in size of the two parts. Denote by $\eta_{n,k}$ and $\omega_{n,k}$ the numbers of vectors in $V_{n,k}$ whose components add to an even and odd number.

In third section authors provided proof of sufficiency in the previous section by induction. On the given algorithm in this section, a recursive minimal change algorithm could be encoded. However, the complex nature of the induction makes the algorithm conceptually difficult to implement. Also there are no deterministic conditions to find its solution in polynomial time. Here the problem of finding a simple minimal change algorithm remains open. The authors admit that a general algorithm on their proposed proof in third section would be inefficient and it would be desirable to have a minimal change algorithm where the average time taken to produce a subset is constant, independent of n.

For small fixed values of *k*, however, simple efficient algorithms exist. For instance, in the procedure SUBSETS of given algorithm, prints out all 3-subsets of an n-set, in minimal change order. The subsets are held in the array a [1…3]. Two procedures INC and DEC are used.

3.4. Paper 4: Approximately counting Hamilton cycles in dense graphs [15]

ABSTRACT: We describe a fully polynomial randomized approximation scheme for the problem of determining the number of Hamiltonian cycles in an n-vertex graph with minimum degree $(\frac{1}{2}+\varepsilon)n$, for any fixed $\varepsilon > 0$. We show that the exact counting problem is NP-complete. We also describe a fully polynomial randomized approximation scheme for counting cycles of all sizes in such graphs.

REVIEW: Due to hardness of most counting problems, authors led to an interest in approximate counting. The most fruitful approach in this respect has been randomized approximation. This is based on the idea of a Fully Polynomial Randomized Approximation Scheme (FPRAS).

In this paper, authors added another entry to the randomly approximable hard counting problems: that of counting the number of Hamiltonian cycles in "dense" graphs. Let G = (V, E) be a graph, where $V = \{v_1, v_2, \ldots, v_n\}$. Denote the degree of vertex v_i by d_i, for i= 1,2,...,*n*. G is called dense if $\min_i d_i \geq (1/2 + \varepsilon)n$, where $0 < \varepsilon < 1/2$ is a fixed constant. Under these circumstances it is known that G must contain a Hamilton cycle. Moreover, the proof of this fact is easily modified to give a simple polynomial-time algorithm for constructing such a Hamilton cycle. This algorithm, which uses edges whose existence is guaranteed by the pigeonhole principle to "patch together" disjoint cycles, provides the required easy decision procedure.

Authors proved following theorems in this regard to qualify their solution.

Theorem 1: If G is dense then there is an FPRAS for approximating its number of Hamilton cycles.

Theorem 2: There exist both a fully polynomial randomized approximation scheme and a fully polynomial almost uniform sampler for the set of 2-factors in a dense graph.

Theorem 3: HC is NP-complete, even when restricted to graphs G of minimum degree at least $(1 - \varepsilon)n$, where n is the number of vertices in G, and $\varepsilon > 0$.

Their approach to constructing an FPRAS for Hamilton cycles in a dense graph G is via a randomized reduction to sampling and estimating 2-factors in G.

It is not difficult to adapt the above methods to the corresponding directed case. We can have both minimum in-degree and out-degree at each vertex guaranteed to be at least $(1/2 + \varepsilon)n$. Also we may similarly count the number of connected k-factors in G for any $k = o(n)$. (Hamilton cycles are, of course, connected P-factors.)

Authors left open the following questions. First, is it possible to count approximately as $\varepsilon \rightarrow 0$ in any fashion? Secondly, is there a random walk on Hamilton cycles and (in some sense) "near-Hamilton-cycles" which is rapidly mixing? In other words, can we avoid the Tutte construction and the need for 2-factors with many cycles?

3.5. Paper 5: A method for finding Hamilton paths and Knight's tours [16]

ABSTRACT: The use of Warnsdorff's rule for finding a knight's tour is generalized and applied to the problem of finding a Hamilton path in a graph. A graph-theoretic justification for the method is given.

REVIEW: A path in a graph is a Hamilton path if and only if it goes through each node of the graph once and only once. These paths were named for Sir William Hamilton who invented and analyzed a game to find these paths through the vertices of a regular dodecahedron. A problem of this type is the classic knight's tour problem on a chessboard. The knight is placed on a square and must cover the whole board, moving to each square once and only once.

Author discussed the solution proposed by H. Warndroff for knight's tour:

"Select the move which connects with the fewest number of further moves, providing this number is not 0. If a tie occurs it may be broken arbitrarily."

This rule proved unusually successful and generally applicable a until a few carefully constructed counterexamples showed that in case of ties some of the options failed to find knight's tours. However, not much was done in analyzing the rule and its failures because of the large computational effort involved in using the rule. This rule was justified on common sense basis.

Author compares combinatoric methods proposed for Hamilton Circuits/Paths and the Warndroff method and concludes that combinatoric methods are suitable for small graph but can't be feasible for large richly connected graphs. Also Warnsdorff's rule is a simple computational rule for finding knight's tours. It presents an attractive method for finding Hamilton paths in richly connected graphs.

Experiments show that Warnsdorff's rule maximizes the number of connections remaining at the current point in the path. A knight's tour is a path on the chessboard which maximizes the number of connections at the 63rd move; i.e., there is one remaining move and there can be at most one at this point. The move tree of the various paths of the knight is **exponential** in nature, and maximization at the 63rd level is not computationally possible. Maximizing the current level propagates down the move tree, and therefore is likely to maximize the later levels. Warnsdorff's rule is just an approximation to the algorithm of searching the complete tree for a connection at the 63rd level.

In implementing Warnsdorff's rule on a B5500 in extended ALGOL, the rule was tried starting from each square of an 8 × 8 board and found to fail at least once for each of five fixed orderings 6 of moves. In fact, the rule failed twelve times with a median of three failures for a given move ordering.

To overcome this high rate of failure, the Warnsdorff's rule was revised for arbitrary selection in case of ties and following tie-breaking solution was proposed and tested. For each tie move, sum the number of moves available to it at the next level and pick the one yielding a minimum. In theory this can be carried through as many levels as necessary for tie-breaking.

The revised solution is powerful to find the knight's tour on chess board and Hamilton Path (not cycle) in a large richly connected graph. The program was also used to find several Hamilton paths in an especially tricky regular graph s of degree 3 with 46 nodes, proposed by W. Tutte.

Author didn't discuss the complexity of the algorithm but just gave a general statement about the complexity that time needed is directly proportional to the number of edges in the graph. Also Reader shouldn't get confused with Hamilton Path finding algorithms with Hamilton Circuit algorithms. It is possible to find a Hamilton Path which passes though all the nodes but to find Hamilton Cycle in a graph is NP-complete.

3.6. Paper 6: Sorting, Minimal Feedback Sets and Hamilton Paths in Tournaments [17]

ABSTRACT: We present a general method for translating sorting by comparisons algorithms to algorithms that compute a Hamilton path in a tournament. The translation is based on the relation between minimal feedback sets and Hamilton paths in tournaments. We prove that there is a one to one correspondence between the set of minimal feedback sets and the set of Hamilton paths. In the comparison model, all the tradeoffs for sorting between the number of processors and the number of rounds hold when a Hamilton path is computed. For the CRCW model, with O(n) processors, we show the following: (i)

Two paths in a tournament can be merged in O(log log n) time (Valiant"s algorithm): (ii) a Hamilton path can be computed in O(log n) time (Cole"s algorithm). This improves a previous algorithm for computing a Hamilton path whose running time was $O(\log^2 n)$ using $O(n^2)$ processors.

REVIEW: A tournament T=(V,D) ($|V| = n$) is a directed graph in which every pair of vertices is joined by a directed edge. It can be viewed as a complete graph whose edges are given an orientation.

This paper investigates the complexity of computing a Hamilton path in a tournament and other problems related to it. The methodology relies on the intimate relationship that exists between Hamilton paths and minimal feedback sets in tournaments and their connection to sorting algorithms. Sorting by comparisons may be viewed as computing a Hamilton path in a transitive (acyclic) tournament. This paper follows the reverse order i.e. how sorting algorithms can be generalized to compute a Hamilton path in an arbitrary tournament.

Parallel algorithms to compute a Hamilton path in a tournament have appeared in different papers. The key idea in computing the Hamilton path in these papers is the following: in every tournament there exists a vertex (separator) whose indegree and outdegree are bounded from below by |V|/4; this gives a recursive formulation of the problem with only a logarithmic number of steps. The difficulty with this approach is that the best bound known for finding a separator requires $O(n^2)$ processors. The lower bound on the number of edges whose orientation must be known in order to find a separator is $\Omega(n^2)$.

A minimal edge feedback set F in a directed graph G = (V, D) is a set of edges such that G` = (V, D - F) is acyclic, and F is minimal with respect to containment. Paper proves that there is a one to one correspondence between the set of minimal feedback sets and the set of Hamilton paths in an arbitrary tournament and shows how a minimal feed back set can generate Hamilton Path and vice versa.

This paper in general gives idea of in-degree and out-degree based approach on directed tournament which finds, sorts and merges paths. Some of these techniques are used in our algorithm with necessary alterations and amendments.

3.7. Paper 7: Hamilton Circuits in Directed Butterfly Networks.

Abstract: In this paper we prove that the wrapped Butterfly digraph WBF(d, n) of degree d and dimension n contains at least d – 1 arc-disjoint Hamilton Circuits, answering a conjecture of D. Barth. We also conjecture that WBF(d, n) can be decomposed into d Hamilton Circuits, except for d = 2 n = 2, d = 2 n = 3 and d = 3 n = 2. We show that it suffices to prove the conjecture for d prime and n = 2. Then we give by a clever computer search such a Hamilton decomposition for all primes less than 12000, and so as corollary we have a Hamilton decomposition of WBF(d, n) for any d divisible by a number q, $4 \leq q \leq 12000$.

REVIEW: The Butterfly digraph of degree d and dimension n, denoted BF(d, n) has as vertices the couples (x, l), where x is an element of $Z^n{}_d$ that is a word $x_{n-1}x_{n-2}...x_1x_0$ where the letters belong to Z_d and $0 \leq l \leq n$ (l is called the level). For $l < n$, a vertex $(x_{n-1}x_{n-2}...x_1x_0, l)$ is joined by an arc to the d vertices $(x_{n-1}x_{n-2}...x_1x_0, l+1)$ where α is any element of Z_d.

The wrapped Butterfly Digraph WBF(d, n) is obtained from BF(d, n) by identifying the vertices of the last and first level namely (x, n) with (x, 0). In other words the vertices are the couples (x, l) where x is an element of $Z^n{}_d$ that is a word $x_{n-1}x_{n-2}...x_1x_0$ where the letters belong to Z_d and l Z_n (l is called the level). For l , a vertex $(x_{n-1}x_{n-2}...x_1x_0, l)$ is joined by an arc to the d vertices $(x_{n-1}x_{n-2}...x_1x_0, l+1)$ where α is any element of Z_d (and where l + 1 has to be taken modulo n).

Authors show that in a lot of cases Butterfly digraphs have a Hamilton decomposition and give strong evidence that the only exceptions should be WBF(2, 2), WBF(2, 3) and WBF(3, 2). The paper furthermore reduces the problem to check if L(Kp, p) has a Hamilton decomposition for p prime (or equivalently that Kp, p has an eulerian compatible decomposition). The interest came from a conjecture of Barth and Raspaud concerning the decomposition of Butterfly networks into unidirected Hamilton cycles. This conjecture is solved by generalizing the technique in section 3.2 of paper.

3.8. Conclusion:

Hamilton Circuit Algorithm is one of the oldest classical algorithm problems. Studies have been made to find the complete and generalized solution of this problem, but there is no such algorithm exist which can find the Hamilton Circuit in polynomial time steps. Many theorems are known to find the sufficient conditions for the existence of Hamilton Circuit but still these theorems don't cover the major range of problem variations. Researchers mostly tried to find solution in a certain set of conditions desired by their problem. Very rare literature is found for generalized solution. Also this problem is common between mathematicians and computer scientists. Mathematicians mostly try to find and prove the mathematical theorems that lead to sufficient conditions while computer scientist mostly try to apply programming logic. We can combine both approaches to find out the solution.

Chapter 4

Research Findings

4. Research Findings

In this chapter we shall briefly cover our research work and our findings regarding our research project. We first give an overview of the research, then we shall cover the formal definitions then we divide the problem into different special cases and finally we shall combining the properties and rules found in each division.

4.1. Overview

The purpose of this research is to develop an algorithm for the determination of Hamilton Circuit in the given graph. First, we divide our problem into different cases and try to solve each case independently. Reader shouldn't confuse it with Divide and Conquer approach. In "Divide and Conquer" approach, the algorithm divides the problem into different sub parts and tries to target each sub part one-by-one. Algorithm will cover it in the same sequence not recursive calls for each sub part of the problem. We shall divide our problem into a few special cases and in each case we shall try to find the conditions and rules to solve it and in the end we shall combine these findings to develop the above mentioned algorithm. We will not emphasize mainly on mathematical proves of these rules/properties found in each special case. These are more or less simple and obvious and need not to prove mathematically.

4.2. Introduction

A path $X_0, X_1, \ldots, X_{n-1}, X_n$ in the graph $G = (V, E)$ is called a Hamiltonian Path if $V = \{X_0, X_1, \ldots, X_{n-1}, X_n\}$ and X_i is not equal to X_j for $0 \leq i < j \leq n$. A circuit $X_0, X_1, \ldots, X_{n-1}, X_n$ (with $n > 1$) in a graph $G = (V, E)$ is called a Hamiltonian Circuit if $X_0, X_1, \ldots, X_{n-1}, X_n$ is a Hamiltonian Path.

Is there a simple way to determine whether a graph has a Hamiltonian Circuit? At first, it might seem that there should be an easy way to determine this. Surprisingly, there is no known simple necessary and sufficient criterion for the existence of Hamiltonian Circuit. However, many theorems are known that give sufficient conditions for the existence of Hamilton Circuit. Also certain properties can be used to show that graph has no Hamiltonian Circuit [1].

4.3. Basic Properties of Hamilton Circuit

1. If a graph has any vertex of degree less than two then the graph doesn't have Hamilton Circuit
2. If the vertex in the graph has degree two, then both edges that are incident with this vertex must be part of any Hamilton Circuit.
3. If there is an edge in the graph which is when removed, divide the graph into two disjoint sub-graphs then the original graph can't have Hamilton Circuit.
4. If there is a vertex in the graph which is when removed (along with the connected edges) divides the graph into two disjoint sub-graphs, then the graph doesn't have any Hamilton Circuit.
5. A Hamilton Circuit can't have a smaller circuit within it.
6. When a Hamilton Circuit is being constructed and this circuit has passed through a vertex, then all remaining edges incident with this vertex other than the two used in the circuit can be removed from consideration.

4.4. Special Cases

Now we consider some special cases and try to solve each case independently and finally combine their results

4.4.1. When Each Vertex Has Degree Two

As mentioned earlier in property-2, if a vertex in the graph has degree two, then both edges that are incident with this vertex must be part of any Hamilton Circuit. Since in our case, all vertices have degree two therefore all edges must be in the path. In this case, nothing is to be done; Hamilton Circuit is present and sorted already. This can be considered the best case for this problem.

Example 4.1

Following are the examples of degree two graphs. No need of finding Hamilton Circuit.

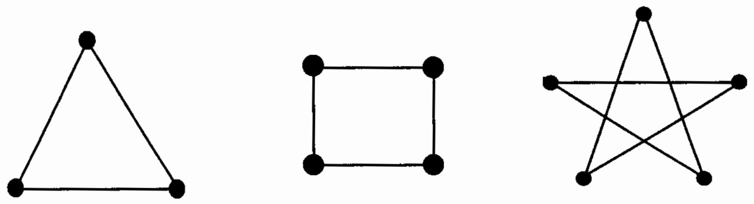


Figure 4.1: Degree Two Graphs

4.4.2. When Each Vertex Has Degree Three

Following are the instructions to find Hamilton Circuit in a graph having each vertex of degree three.

1. Select a starting vertex at random since each vertex has equal probability.
2. Select any two adjacent edges to the vertex. In other words, start path on both directions.
3. Remove the third edge.
4. Since each vertex has degree three, therefore the vertex on the other end of the removed edge is left with only two edges. According to property-2, both edges must be in the graph. So mark both edges as Virtual Single Unit (VSU) on the other end of the removed path.
5. The selected vertex has also left with two edges and both are selected as well, therefore mark them as another Virtual Single Unit (VSU).
6. Since we are proceeding on both sides from starting vertex, we name one endpoint as Active Endpoint and the other end (vertex) of VSU as Passive Endpoint. The Active Endpoint will take decision for further move at each step and Passive Endpoint will only move further when it gets some Virtual Single Unit incident to it or found an edge which can create a small cycle (Property-5).
7. At each extension on Active Endpoint or on either endpoints of Virtual Single Unit, the unused edges are removed and opposite ends of removed

edges are reconsidered for Virtual Single Unit creation, extension or merging with other Virtual Single Unit, Active Endpoint or Passive Endpoint.

8. Both endpoints, Active and Passive, can only be joined when all vertices are traversed. Otherwise, whenever both endpoints share the same edge remove that edge (property-5) by taking it as unused edge and extend both endpoints according to property-2.
9. On each extension, check the concerning Virtual Single Units (VSU) for the edge which can create smaller cycle, remove it and extend VSU.
10. If on joining two Virtual Single Units, we found two unused edges coming out of same vertex, don't join Virtual Single Units rather adopt the alternate path on Active or Passive Endpoint. If no alternate path to adopt on Active Endpoint then Graph doesn't have any Hamilton Circuit.

Example 4.2:

Consider the Icosian Puzzle, invented in 1857 by William Rowan Hamilton. The graph shown is isomorphic to the graph consisting of the vertices and edges of the original dodecahedron as shown in Figure-4.2.

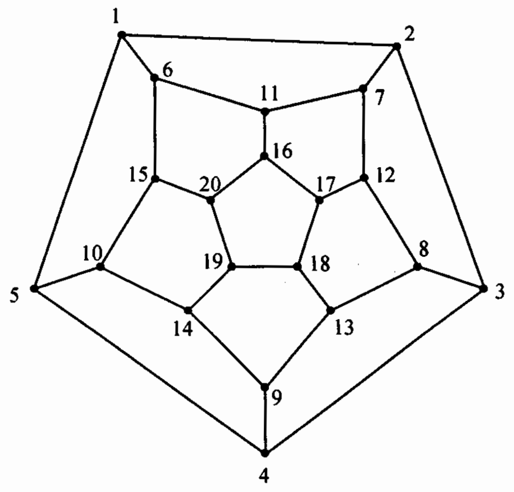


Figure 4.2: Icosian Dodecahedron

Now according to step-1, select any vertex at random, say vertex labeled 5. According to step-2, select any two edges incident, say edges (5, 4) and (5, 10) selected. According to step-3, remove the third edge (5, 1). According to step-5, edge (1, 6) and (1, 2) are marked as Virtual Single Unit on the other endpoint of

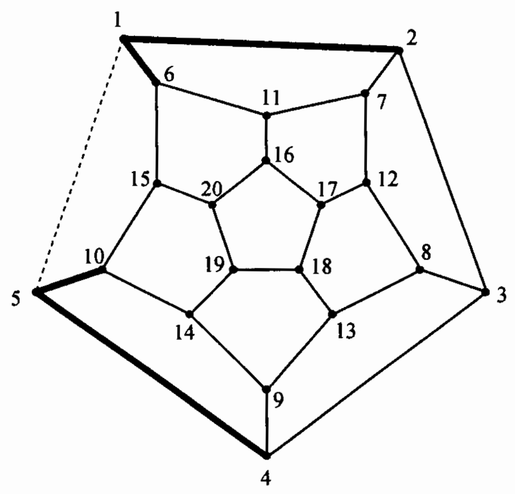


Figure 4.3: First Move

removed edge. According to Step-6, mark vertex 10 as Passive Endpoint and vertex 4 as Active Endpoint (Figure-4.3). Thus vertex 4 or Active Endpoint will proceed.

Next, select any one edge at ACTIVE endpoint to proceed, say (4, 9). Thus the new Active Endpoint will be vertex 9 now. Also edge (3, 4) is unused because both edges at vertex 4 are selected, therefore (3, 4) is to remove now. On removing edge (3, 4) the other vertex of the edge i.e. vertex 3 is left with only two edge and therefore both edges will be mark as Virtual Single Unit (edge (3, 8) and edge (3, 2)). Since vertex-2 is also part of another V.S.U therefore both V.S.U's are combined and marked as single unit i.e. edge (6, 1), edge (1, 2), edge (2, 3), edge (3, 8). Also unused edge (2, 7) at vertex 2 is removed. Again vertex at opposite side of edge (2, 7) i.e. vertex 7 is left with two edges only. Thus both edges will again mark as Virtual Single Unit i.e. edge (11, 7) and edge (7, 12) as shown in Figure-4.4.

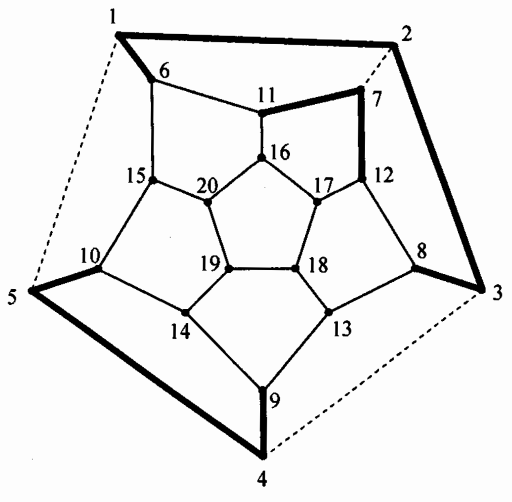


Figure 4.4: Second Move

Now again, select any one edge at Active Endpoint (vertex-9) to proceed, say edge (9, 13). Thus our Active Endpoint will move to vertex-13. Also unused edge (9, 14) at vertex-9 is removed. On removing edge (9, 14), other vertex of edge is left with two edges thus creating Virtual Single Unit "10, 14, 19" but since vertex-10 is Passive Endpoint which is now will move to vertex 19 on combining VSU with Passive Endpoint. Also unused edge at vertex 10 i.e. edge (10, 15) is removed and which leads to the creation of another Virtual Single Unit at

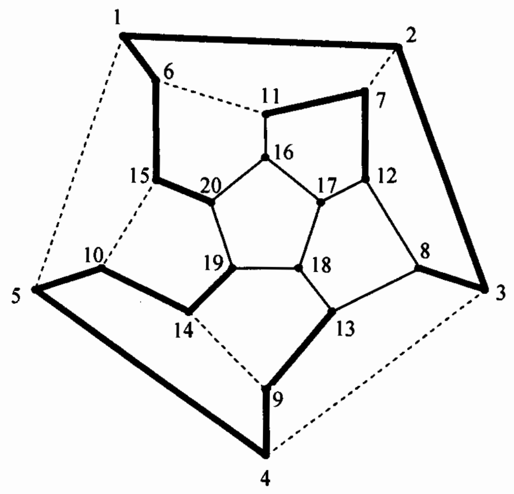


Figure 4.5: Third Move

opposite vertex-15 of removed edge i.e. "6, 15, 20". But vertex-6 is already part of another VSU therefore on combining both VSU's we get the bigger VSU i.e. "20, 15, 6, 1, 2, 3, 8". The unused edge at vertex-6 is removed and opposite vertex (which is already endpoint of another VSU) is extended as VSU "16, 11, 7, 12" as shown in Figure-4.5.

Now again, select any one edge at Active Endpoint (vertex-13) to proceed, say edge (13, 8). Remove the unused edge at vertex-13 i.e. (13, 8). The opposite vertex of removed edge is left with only two edges thus forming VSU "19, 18, 17". Since Passive Endpoint is at vertex-19 therefore on joining of VSU and Passive Endpoint, unused edge (19, 20) is removed and Passive Endpoint is moved to vertex-17. At the same time on the removal of edge (19, 20), vertex-20 is left with only two edges. One of them is already in a VSU therefore the VSU

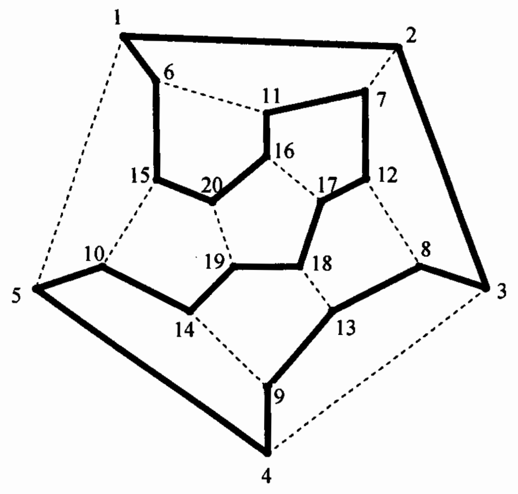


Figure 4.6: Hamilton Circuit

will be just extended one edge without the removal of any edge. Now the extended edge again connects two VSU's therefore unused edge (16, 17) is removed. The other end of removed edge (vertex-17) is left with only two edges but vertex-17 is the Passive Endpoint. Therefore Passive Endpoint is extended by (17, 12). Now vertex-12 is also endpoint of a VSU therefore Passive Endpoint is connected to a VSU whose other end vertex (vertex-8) is already connected with Active Endpoint. Now just remove unused edge (8, 12) to get the Hamilton Circuit as shown in Figure-4.6.

4.4.3. When Each Vertex has Degree Four

Algorithm is almost same as previously discussed for degree three graphs. The only difference is, on selecting two incident edges at a vertex we can remove the remaining two but the opposite ends (vertices) of removed edges are not left with only two incident edges in most cases therefore don't form a VSU.

Following are the instructions to find Hamilton Circuit from the given graph of degree four:

1. There should not be any edge which is when removed, divide the graph into two disjoint sub-graphs (property-3). Remove every edge one by one and check if it is it divides the graph into two disjoint sub-graphs. If so, there is no need to find the Hamilton Circuit, graph doesn't have any Hamilton Circuit.
2. Select a starting vertex at random since each vertex has equal probability.
3. Check for next move as:
   3.1. If no Active endpoint is marked, select any two adjacent edges to the vertex selected in Step-1. In other words, start path on both directions. Mark the selected edges as main Virtual Single Unit (VSU). Since we are proceeding on both sides from starting vertex, we name one endpoint as Active Endpoint and the other end (vertex) of VSU as Passive Endpoint. On each move/step, decision of move (selection of edge for extension) will be taken at Active Endpoint and Passive Endpoint will only move further when it gets some Virtual Single Unit incident to it or found an edge which can create a small cycle (Property-5).
   3.2. If Active endpoint is marked, mark any one edge on Active endpoint as selected edge. Move the Active endpoint to the other end of selected edge.
4. Remove the remaining edges.
5. Check the other end of removed edges. If any edge is left with only two edges, both edges must be in Hamilton Circuit (property-2). So mark both edges as Virtual Single Unit (VSU) on the other end of the removed edge.

5.1. If any endpoint of the new VSU is joined with any existing VSU, existing VSU is extended by copying ID from old VSU to new VSU.

5.2. If no endpoint of new VSU is joined with any existing VSU, new VSU is given new ID.

6. Check all VSU's one by one for merging. If endpoint of any VSU is adjacent to any other VSU i.e. if two VSU's have some common endpoint, merge these VSU's by copying VSU ID one to the other.
7. Whenever both endpoints share some edge

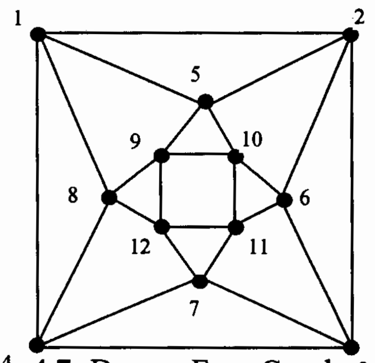
7.1. If main VSU path has all vertices, join Active and Passive endpoint. The Hamilton Circuit is found.

7.2. If main VSU path doesn't have all vertices, remove that edge (property-5) by taking it as unused edge and extend both endpoints according to property-2.

8. On each extension, check the concerning Virtual Single Units (VSU) for the edge which can create smaller cycle, remove it and extend VSU.
9. If on joining two Virtual Single Units, we found two unused edges coming out of same vertex, don't join Virtual Single Units rather adopt the alternate path on Active or Passive Endpoint. If no alternate path to adopt on Active Endpoint then Graph doesn't have any Hamilton Circuit.
10. Repeat from step-3 to step-9 until we find Hamilton Circuit in step 7.1 or we failed to find Hamilton Circuit in step-9.

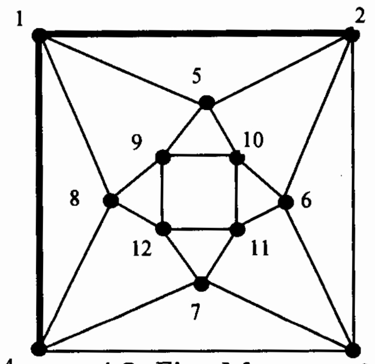
Example 4.3:

Consider the following graph of degree four shown in figure 4.7. Since every vertex has equal degree and every edge on these vertices has equal probability to be selected for the start of HC, we therefore select vertex 1 as our starting vertex. Now select any two edges on vertex 1 say (1,4) and (1,2). Mark one end point



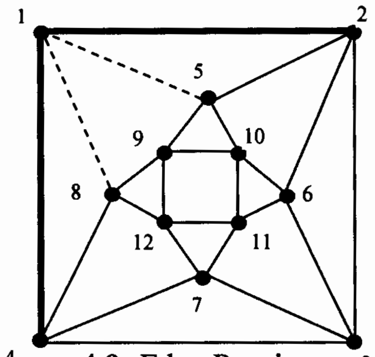
4.7: Degree Four Graph

of this (main) VSU as Active Endpoint say vertex-4 and other endpoint as Passive Endpoint i.e. vertex-2. This is the first move as shown in Figure 4.8.



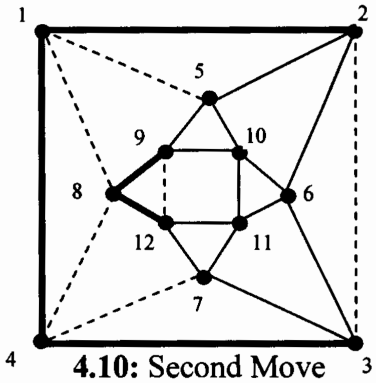
4.8: First Move

Now the next step is to remove/delete the unused edges on vertex-1 i.e. (1, 8) and (1, 5). Now check the other ends of removed edge for VSU, since both vertex-5 and vertex-8 are left with three edges, no new VSU can be created on these vertices. After completing all steps 5 to step 9, there isn't any difference in the graph as shown in Figure 4.9.



4.9: Edge Pruning

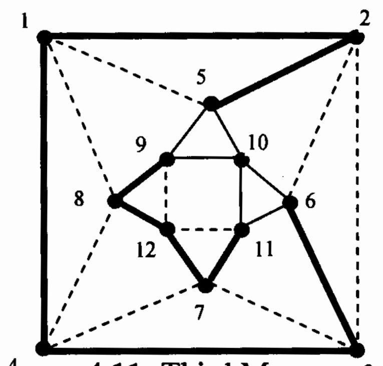
Now for move two, select any edge on Active Endpoint (vertex-4) say edge (4, 3). Remove unused edge on vertex-4 those are edges (4, 7) and (4, 8). Now vertex-8 is left with only two edges which must be part of HC. Therefore a new VSU is created at vertex-8 and VSU path is {12, 8, 9}. Now the Active Endpoint is moved to the new endpoint of main VSU path {2, 1, 4, 3} i.e. vertex-3 as shown in Figure 4.10. According to step 7, remove edge (2, 3) as it is shared by the endpoints of the main VSU path and edge (9, 12) is removed as it is shared by the endpoint of second VSU path (Figure 4.10).



4.10: Second Move

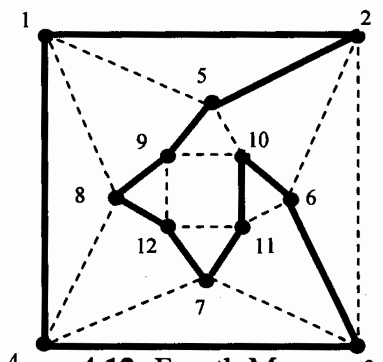
Now for third move, select edge (3, 6) on Active Endpoint. Unused edge (3, 7) is to remove as both edges on vertex-3 are selected. Check the other end vertex of removed edge i.e. vertex-7 which is left with only two edges so create new VSU path {11, 7, 12}. Since VSU path {12, 8, 9} and {11, 7, 12} have common vertex-12 so both are merged and resultant VSU path {11, 7, 12, 8, 9} as shown in

Figure 4.11. While this merging, unused edge (11, 12) is to remove since both edges are selected at vertex-12. Now according to step 7, remove edge (6, 2) which is common in both endpoints of main VSU path. In result, vertex-2 is left with only two edges, so extend VSU on passive endpoint by edge (2, 5). Now the passive endpoint of main VSU path is vertex-5 as shown in Figure 4.11.



4.11: Third Move

For move four, select edge (6, 10) on Active Endpoint. Unused edge (6, 11) is removed which results only two edges left on vertex-11. So VSU path {11, 7, 12, 8, 9} is extend to vertex-10. Since main VSU path has common vertex-10 with this VSU path, both are merged at vertex-10. While merging the unused edge on vertex-10 are also deleted. In result, main VSU path is extended by edge (5, 9) which results in Hamilton Circuit in the given graph as shown in Figure 4.12.



4.12: Fourth Move

4.5. The Algorithm

Now that we have considered different special cases and try to solve each case independently; we can combine these cases and merge these algorithms created for each case with necessary alteration to get the final/generalized algorithm to find the HC in any undirected graph. Here are the steps of the final proposed algorithm for Hamilton Circuit:

1. Check for degree of each vertex. If any vertex has degree less than two, there is no Hamilton Circuit in the graph.
2. There should not be any edge which is when removed, divide the graph into two disjoint sub-graphs (property-3). Remove every edge one by one and check if it is

it divides the graph into two disjoint sub-graphs. If so, there is no need to find the Hamilton Circuit; graph doesn't have any Hamilton Circuit.

3. Check for all vertices of degree two. If any such vertex is found, both of its incident edges must be in HC. Create a VSU containing both of these edges.
4. If any VSU is created in step-3, start from it as main VSU. If there are multiple VSU created in step-3, select the first VSU as main VSU and move to the next step to find the HC. If no VSU is created in step-3, search for degree 3 vertex. If degree 3 vertex is found, select it as starting vertex else just select any vertex as starting vertex at random since each vertex has equal probability.
5. Check for next move as:
   5.1. If no Active endpoint is marked, select any two adjacent edges to the vertex selected in Step-4. In other words, start path on both directions. Mark the selected edges as main Virtual Single Unit (VSU). Since we are proceeding on both sides from starting vertex, we name one endpoint as Active Endpoint and the other end (vertex) of VSU as Passive Endpoint. On each move, decision of move (selection of edge for extension) will be taken at Active Endpoint and Passive Endpoint will only move further when it gets some Virtual Single Unit incident to it or found an edge which can create a small cycle (Property-5).
   5.2. If Active endpoint is marked, mark any one edge on Active endpoint as selected edge. Move the Active endpoint to the other end of selected edge.
6. Remove the remaining edges.
7. Check the other end of removed edges. If any vertex is left with only two edges, both edges must be in Hamilton Circuit (property-2). So mark both edges as Virtual Single Unit (VSU) on the other end of the removed edge.
   7.1. If any endpoint of the new VSU is joined with any existing VSU, existing VSU is extended by copying ID from old VSU to new VSU.
   7.2. If no endpoint of new VSU is joined with any existing VSU, new VSU is given new ID.

8. Check all VSU's one by one for merging. If endpoint of any VSU is adjacent to any other VSU i.e. if two VSU's have some common endpoint, merge these VSU's by copying VSU ID from one to the other.
9. Whenever both endpoints of a VSU share some edge
   9.1. If VSU is main VSU path and it has all vertices, join Active and Passive endpoint. The Hamilton Circuit is found.
   9.2. Else remove that edge (property-5) by taking it as unused edge and extend both endpoints according to property-2.
10. If on joining two Virtual Single Units, we found two unused edges coming out of the same vertex and the vertex has left with only one, don't join Virtual Single Units rather discard the whole move (started in step-5) and adopt the alternate path on Active or Passive Endpoint. If no alternate path to adopt on Active Endpoint then Graph doesn't have any Hamilton Circuit.
11. Repeat from step-5 to step-10 until we find Hamilton Circuit in step 7.1 or we failed to find Hamilton Circuit in step-10.

Now we write the above algorithm in a little bit formal way. We shall write it in pseudo code which is similar to Pascal like form.

```
Procedure FindHC ()
Begin
        If any node has degree < 2, FAILED
        If FindCycle () fails for any edge of graph
            FAILED (No Hamiltonian Circuit)
        End
        ReturnValue: = 0
        While ReturnValue is zero
            ReturnValue: = MoveNext ()
        End
        If ReturnValue = -1
            FAILED
        Else
            SUCCESS
        End
End
```

Function FindCycle (e: edge)
There should not be any edge which is when removed, divide the graph into two disjoint sub-graphs. Any such edge cannot be part of any cycle. So this function tries to find closest/smallest cycle containing the given edge. Function returns TRUE if found some cycle else returns FALSE.

```
Begin
    Mark one endpoint (node) as RED node & Add to RED list
    Mark other endpoint (node) as BLUE node & Add to BLUE list
    Remove the given edge
    LOOP
            Get each node one by one from RED list & Get all connected nodes.
            Add all connected node to temporary list
            Remove all edges incident to nodes in RED list
            Empty the RED list
            Copy all nodes from temporary list to RED list

            Remove all edges connecting one node in RED list to any other RED node

            If RED list is empty
                    Return FALSE (loop is broken here)
            End

            If any node in RED list is connected to BLUE list by some edge in graph
                    Return TRUE (loop is broken here)
            End

            Get each node one by one from BLUE list & Get all connected nodes to
            them. Add all connected nodes to temporary list.
            Remove all edges incident to nodes in BLUE list.
            Empty the BLUE list.
            Copy all nodes from temporary list to BLUE list

            Remove all edges connecting one node in BLUE list to any other BLUE
            node

            If RED list is empty
                    Return FAILURE (loop broken here)
            End

            If any node in RED list is connected to BLUE list by some edge in graph
                    Return SUCCESS (loop broken here)
            End
    End
End
```

```
Function MoveNext ()
Begin
    If any node has degree < 2 Return -1
    If any node has degree = 2 CALL MarkVSU () for that node
    If Main Path/VSU exist, Get First node of path as Active Endpoint
    Else Get first node of Node List as Active Endpoint
    If no incident edge selected
            Mark any two edges as selected
            CALL PruneNode (ActiveEP)
            CALL PrunePath ()
            If FindCycle() fails for any unselected edge
                    Return -1
            End
    Else if one edge is selected
            Select any of unselected edge as selected edge
            If selected edge is connecting two VSUs
                    Get unselected edges on either node of connecting edge
                    If both unselected edges have same other end node
                    Mark selected edge as unselected & select the alternate edge on
                    ActiveEP
            End
            CALL PruneNode (ActiveEP)
            CALL PrunePath ()
            If FindCycle() fails for any unselected edge
                    Mark selected edge as unselected & select the alternate edge on
                    ActiveEP
                    CALL PruneNode (ActiveEP)
                    CALL PrunePath ()
                    If FindCycle() fails for any unselected edge
                            Return -1
                    Else
                            Return 0
                    End
            Else
                    Return 0
            End
    Else IF both incident edges are assigned different VSU ID
            CALL PruneNode (ActiveEP)
            If Main VSU covers all nodes
                    Return 1
            End
    End
End
```

```
Function PruneNode ( n: node )
Begin
    Remove the unselected edge at 'n' & CALL MarkVSUs () on other end node of
    removed edge
    If both incident edges are assigned different VSU ID CALL MarkVSU (n)
    Else If both incident edges are not assigned VSU ID CALL MarkVSU(n)
    Else If one edge is assigned and other is unassigned.
            Copy VSU ID from assigned to unassigned edge
            Call MarkVSU() on other end node of previously unassigned edge
    End
End

Function PrunePath ()
Begin
    If endpoints of any VSU are sharing some edge
            Remove edge
            CALL MarkVSU on either node of removed edge
    End
End

Function MarkVSU (n: node)
Begin
    If Degree of node = 2
            If any edge is not marked as selected, mark it as selected
            If Different VSU ID is set on both edges, Merge VSUs
            Else If VSU ID is set on one edge
                    Copy VSU ID from assigned to unassigned edge
                    CALL MarkVSU () on otherend node of previously unassigned
                    edge
            Else if both edges are unassigned
                    If other end of first selected edge (Say m) is endpoint of some
                    VSU
                            Copy VSU ID from assigned to unassigned edge on 'm'
                            Call PruneNode(m).
                    End

                    If other end of second selected edge (Say k) is endpoint of some
                    VSU
                            Copy VSU ID from assigned to unassigned edge on 'k'
                            Call PruneNode(k)
                    End
                    If both selected edges on given node are assigned different ID,
                    Merge VSUs.
            Else if one incident edge is assigned
                    Copy VSU ID from assigned to unassigned
            Else
```

```
                Set new ID on either edge and add Path to VSUList
            End
        Else If Degree of node > 2 and any two edges are marked selected
            CALL PruneNode(NodeNum)
        End
    End
```

Results:

We have tried graphs up to 30 nodes, having degree 3 and higher degree graphs and successfully solved all these graphs using the above mentioned algorithm. Graphs that do not have Hamiltonian Circuit are mostly rejected in six basic properties given in section 2. There is no back tracking involved in this algorithm therefore we can find Hamiltonian Circuit in a graph of degree three in polynomial steps.

Complexity:

Find HC :-

$$n + n\,(FindCycle) + n(MoveNext)$$

$$= n + n(n\,(2(n(n-1)/2)^2) + n\,(n + 2(\,2(n-3)(20+n) + 2(n-3)(20+n) + n\,(2(n(n-1)/2)^2)))$$

$$Big\ O \approx n^6$$

Find Cycle :-

$$n\,(2e^2)$$

$$n\,(2(n(n-1)/2)^2) \qquad (For\ Complete\ Graphs)$$

Move Next :-

$$n + PruneNode + PruneNode + FindCycle + PruneNode + PruneNode + FindCycle$$

$$= n + 2(\,2(n-3)(20+n) + 2(n-3)(20+n) + n\,(2(n(n-1)/2)^2)\,)$$

Prune Node :-

$$MarkVSU + MarkVSU$$

$$= 2(n-3)(20+n)$$

MarkVSU :-

(n-3)(20+n)

Quantitative Analysis of Complexity:

Nodes	Brute Force $(n-1)!$	Back Tracking $(n-1)^n$	Proposed Algorithm
3	2	8	729
4	6	81	4096
5	24	1024	15625
6	125	15625	46656
7	750	279936	117649
8	5250	5764801	262144
9	42000	134217728	531441
10	378000	3486784401	1000000
11	3780000	100000000000	1771561
12	41580000	3138428376721	2985984
13	498960000	106993205379072	4826809
14	6,486,480,000	3,937,376,385,699,289	7,529,536

Chapter 5

Simulator Design

5. Simulator Design

For the sake of proof of concept, we have created a simulator which works on proposed algorithm to find out the Hamilton Circuit from the given graph. In this chapter, we shall discuss the design of the simulator. Since it's a simulator of an algorithm so we can't give the use cases or sequence diagrams of it as these doesn't cover the main purpose or flow of algorithm rather we shall cover the Flow Diagrams of the simulator to show the flow of algorithm in the simulator and how it comes out with the right solution.

5.1. Level 0 DFD

At abstract level, we can define the function of the system as, "we present a graph to the simulator and it gives the result. Result may be a Hamilton circuit if found, or shows a message describing that no Hamilton circuit exists in the given graph".

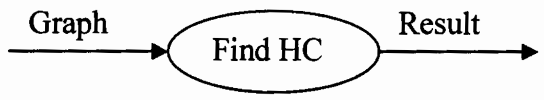


Figure 4.1: Level 0 DFD

5.2. Level 1 DFD

At this level, we divide the DFD into three sub processes as shown in figure 1.2. Process check degree traverses the whole graph and check degree of each node. If degree of any vertex is less than two, HC is not found. Process Find Cycle checks the graph

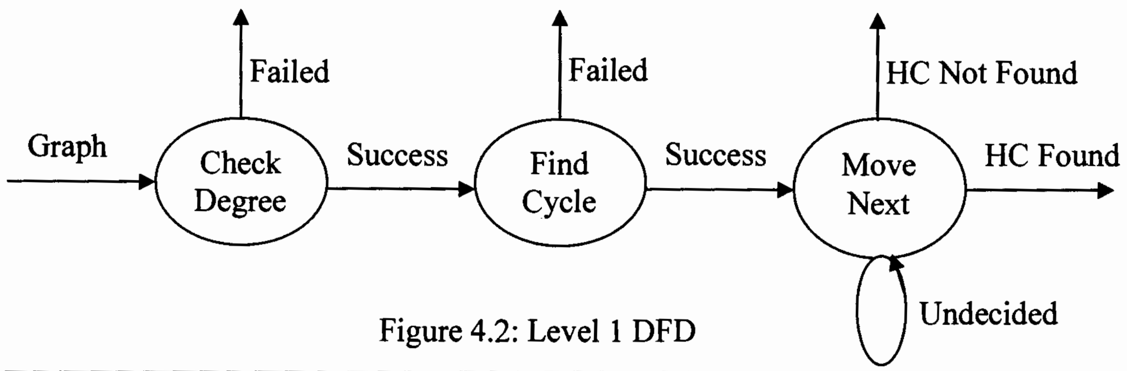


Figure 4.2: Level 1 DFD

connectivity, if failed then graph doesn't have Hamilton Circuit. Process "Move Next" is the main starting point for step by step search for HC. This process will be expanded in the following DFD levels.

5.3. Level 2 DFD

At this level, we expand our data flow diagram to cover more details of processes "Find Cycle" and "Move Next" as shown in the following sections.

5.3.1. Move Next

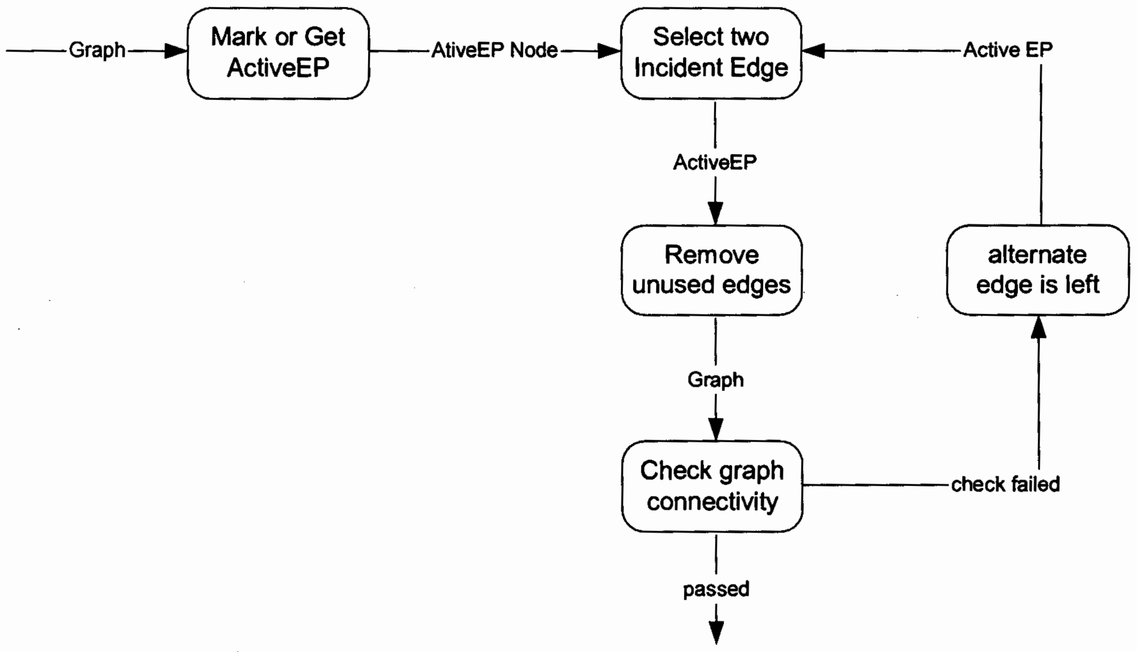


Figure 4.3

5.3.2. Find Cycle

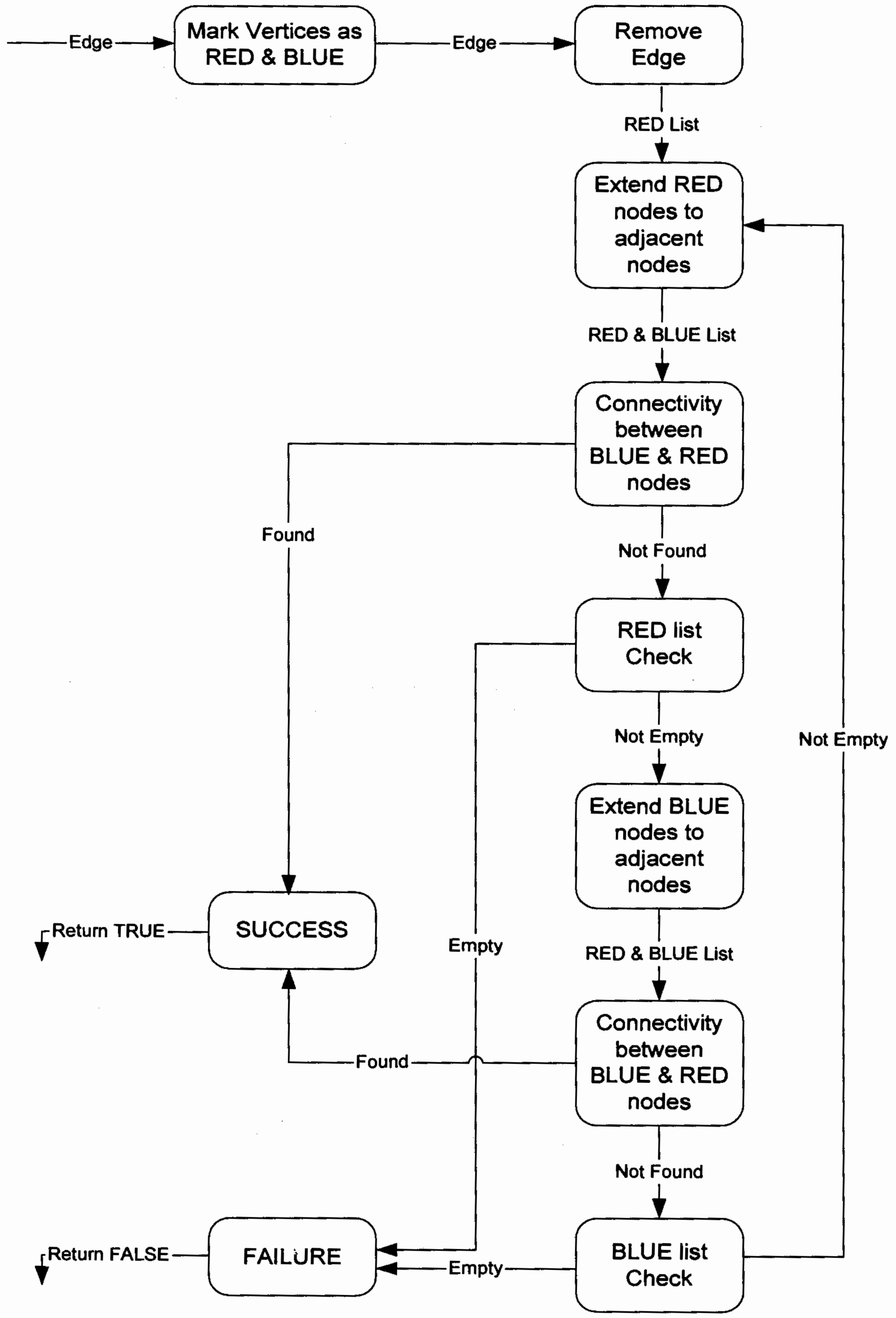


Figure 4.4

5.4. Level 3 DFD

At this level, we shall try to give complete details/flow of process "Move Next" and the processes internally used in process "Move Next". First we shall give the structure of an important process Mark VSU and then we shall complete flow of process "Move Next"

5.4.1. Mark VSU

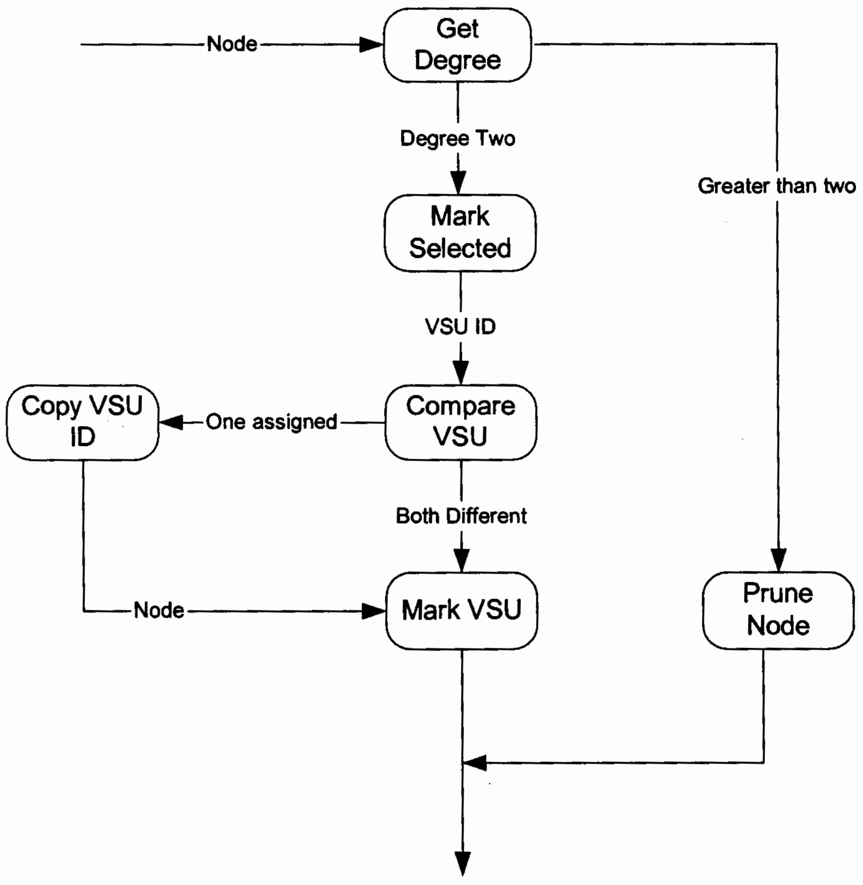


Figure 4.5

5.4.2. Move Next

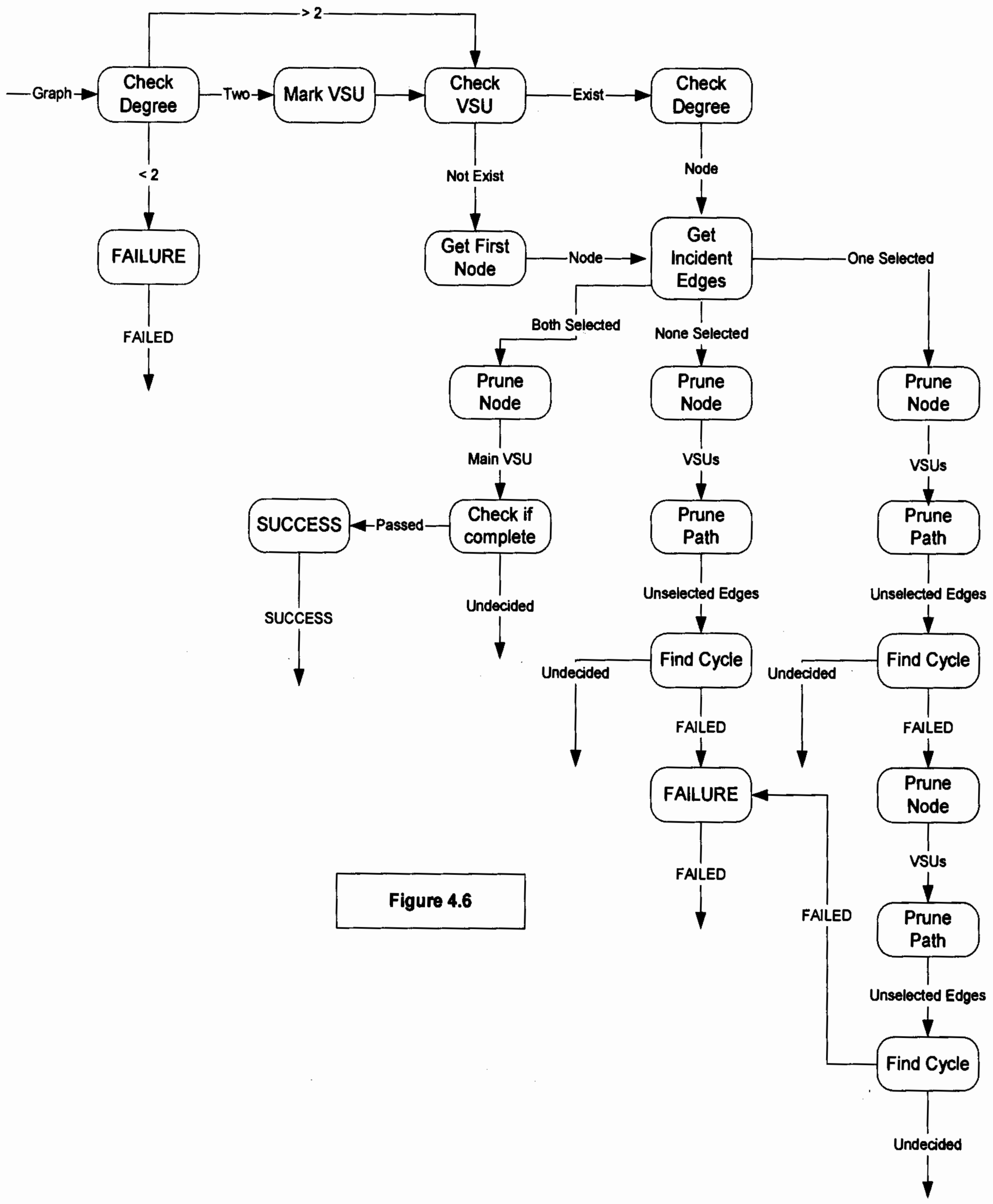


Figure 4.6

5.4.3. Prune Node

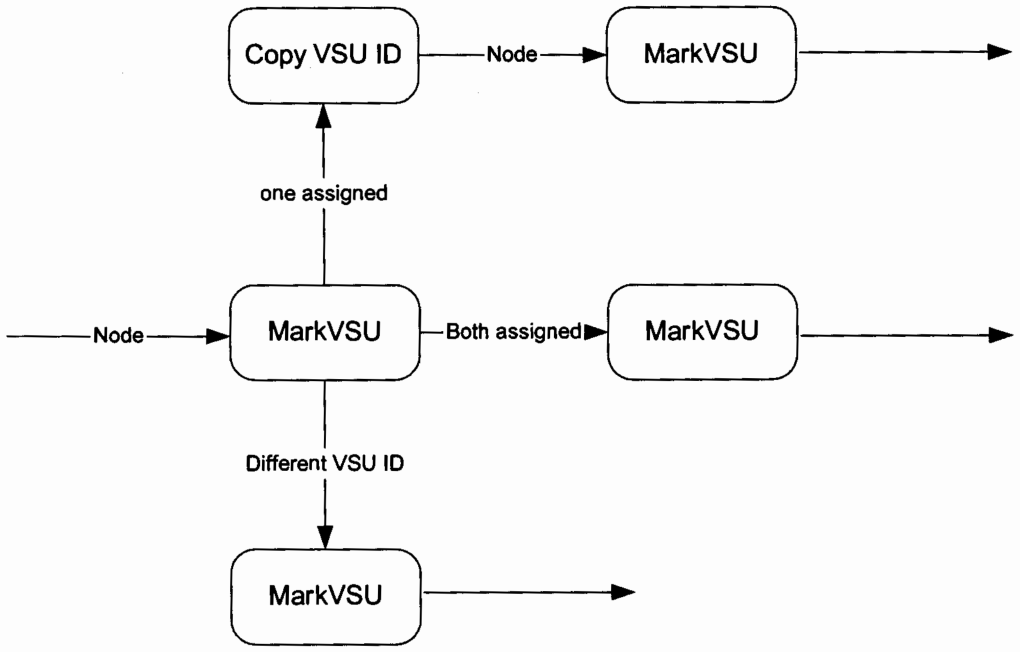


Figure 4.7

5.4.4. Prune Path

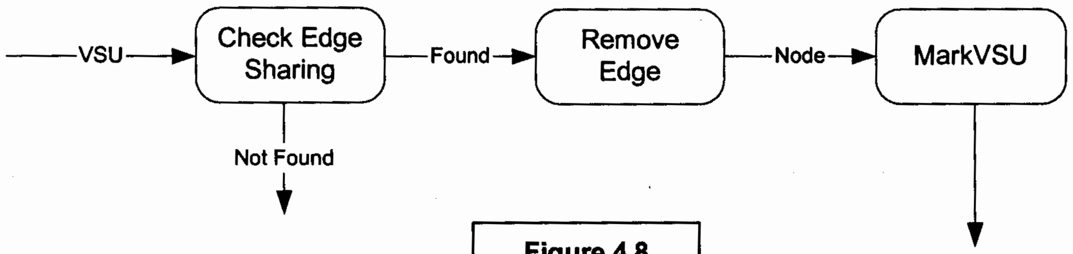


Figure 4.8

This is the last DFD showing the flow of function Prune Path.

Chapter 6

Implementation

6. Implementation

System development starts when system design of proposed system exists. In our case, we have to follow the structured System design techniques since main focus is on algorithm which is not the parallel. Although we used latest programming tools available which are bases on OOP concepts, therefore we shall follow a hybrid approach to show the algorithm and the simulator. System development is the renovation of system design into a genuine functioning computer system. This task is accomplished by coding.

6.1. Programming Tools Used

- Visual C#.Net
- .Net Framework

Because Microsoft Visual C#.NET is considered one of the "hard-rock" programming tools and is ranked one of the best Object Oriented Programming tool. So for good combination we selected Microsoft environment. .Net framework also provides the rich class library which gives a jump start to our simulators.

6.2. Interface Designing

This stage of implementation provides the mean of interaction to the user with the system. The output of this stage will let the user be able to interact with system. The main purpose of this phase is to develop a user-friendly interface for the ease of customer. The quality of good interface is that it must be simple, user-friendly, and attractive. The interface should not be complex.

These points should be considered while designing interface:

- **User-Friendly Interface**

  The interface should be simple. Proper messages should be displayed on the screen to let the user know about the transactions. Transactions should be completed in stepwise manner.

- **Simple Language**

  The language used on the screen should be simple and precise. Language should not be complex or the steps should not be confusing to the user.

6.3. Coding

Coding is the subsequent phase after system design. During this phase detail design representation of the proposed system is translated into a programming language realization. The objective of coding phase is to convert the design into code using programming language.

The assortment of a programming language or tool is critical step as it affects the complexity of testing and maintenance. Furthermore the distinctiveness of selected language has an impact on the quality and efficiency of coding. Each programming language has its own merits and demerits but the choice of language depends on the precise rations of the proposed system.

- Public Class BaseForm

```
public class BaseForm
{
        // HamCircuit Object for Base Form
        HamCircuit hc = new HamCircuit ();

        public BaseForm()
        {
                //
                // Required for Windows Form Designer support
                //
                InitializeComponent();

                //
                // TODO: Add any constructor code after
                // InitializeComponent call
                //
        }

        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
                Application.Run(new BaseForm());
        }
}
```

- Public Class HamCircuit

```
public class HamCircuit
{
        public ArrayList node_list;
        public ArrayList edge_list;
        public ArrayList vsu_list;
        public HamCircuit()
        {
                //
                // TODO: Add constructor logic here
                //
                node_list = new ArrayList ();
                edge_list = new ArrayList ();
                vsu_list  = new ArrayList ();
        }

        public bool FindHC ()
        {
                if ( node_list.Count == 0 )
                {
                        MessageBox.Show ( "No Graph to Find Hamilton Circuit" );
                        return false;
                }

                // if any node has degree < 2, FAILED
                int i;
                for ( i=0; i<node_list.Count; i++ )
                {
                        if ( ((Node)node_list[i]).GetDegree () < 2 )
                        {
                                MessageBox.Show ( "Graph Has No Hamilton Circuit. \nNode
\'" + i + "\' Has Degree Less Than Two" );
                                return false;
                        }
                }

                // if FindCycle() fails for any edge of graph
                //    FAILED
                // end if
                for ( i=0; i<edge_list.Count; i++ )
                {
                        if ( !FindCycle ( i ) )
                        {
                                Edge edge = (Edge)edge_list[i];
                                MessageBox.Show ( "Graph Has No Hamilton Circuit. \nEdge
Linking Nodes \'" + edge.node1 +
                                        "\' and \'" + edge.node2 + "\' is the Only Link
Between Two Sides" );
                                return false;
                        }
                }


                // ReturnValue := 0
                // while ReturnValue is zero
```

```
                    //   ReturnValue := MoveNext ()
                    // end
                    int ReturnValue = 0;
                    while ( ReturnValue == 0 )
                    {
                            ReturnValue = MoveNext ();
                    }

                    // if ReturnValue = -1
                    //   FAILED
                    // else
                    //   SUCCESS
                    // end if
                    if ( ReturnValue == -1 )
                    {
                            MessageBox.Show ( "Graph Has No Hamilton Circuit." );
                            return false;
                    }

                    MessageBox.Show ( "Hamilton Circuit Found. \n Path is " + (string)vsu_list[0]
);

                    return true;
            }
}
```

Chapter 7

System Testing

7. System Testing

Software is never cent percent correct, no matter which developing technique is used. Every software must be verified. Software analysis and testing are important to control the quality of the product. Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding.

Testing is the process of executing a program with the intent of finding errors. There are two different types of testing from execution point of view. They are as follows.

7.1. Black Box Testing

In Black Box Testing, we only check whether the interface is available and check whether it is working properly or not. Incorrect or missing functions, interface errors, errors in data structures or external database access, performance errors, initialization and termination errors can be discovered using black box testing.

7.2. White Box Testing

In White Box Testing, requires the tester to go through each line of code. Flowcharts are used for this type of testing. Logical errors can be discovered and repaired using this type of testing. Using White Box Testing methods, the software engineer can derive test cases that

1) Guarantees that all independent paths within a module have been exercised at least once
2) Exercise all logical decisions on their true and false sides
3) Execute all loops at their boundaries and within their operational bounds
4) Exercise internal data structures to assure their validity

White Box Testing is very time consuming strategy, so, Black Box Testing is used for this project with White Box Testing to cover maximum flow paths. For White box testing, first we tried as many dry runs as possible along with the Flow Charts. Then after implementation of every function, multiple input classes were tried and compared with the outputs obtained in dry runs. This technique gave quite impressive results. Secondly the container functions were tested in bottom-up manner and finally we tried to test as much permutations of graphs as possible but still we can't eliminate the human factor during these test.

7.3. Test Cases

A good test case is one that has a high probability of finding an as-yet undiscovered error. The test cases developed using Black Box Testing is as follows.

7.3.1. Degree Two Permutations

Test Case	Check whether different permutations of degree two graphs give desired results
Functional Area	Simulator Dialog Box
Test Name	***Degree two permutations verification***
Description (Purpose)	To check that all permutations of degree two graphs are valid and after only giving valid data user can get the same graph as if all nodes have degree two graph, it's the best case.
Prerequisite	Application is in runner mode
Input	Different Permutations of degree two graph
Actions to be performed	1. Draw nodes by "Draw Nodes" button 2. Draw edges by "Draw Edges" button such that every node has two edges 3. Click "Next" button

Expected Result(s)	**Status**
Hamilton Circuit is to be shown.	Pass

Comments

7.3.2. Degree Three Permutations

Test Case	Check whether different permutations of degree three graphs give desired results
Functional Area	Simulator Dialog Box
Test Name	***Degree two permutations verification***
Description (Purpose)	To check that all permutations of degree three graphs are valid and after only giving graphs having HC, user can get the HC out of the given graph.
Prerequisite	Application is in runner mode
Input	Different Permutations of degree two graph
Actions to be performed	1. Draw nodes by "Draw Nodes" button 2. Draw edges by "Draw Edges" button such that every node has two edges 3. Click "Next" button

Expected Result(s)	**Status**
Hamilton Circuit should be shown if exist.	Pass

Comments

7.3.3. Degree Four Permutations

Test Case	Check whether different permutations of degree four graphs give desired results
Functional Area	Simulator Dialog Box
Test Name	***Degree four permutations verification***
Description (Purpose)	To check that all permutations of degree four graphs are valid and after only giving graphs having HC, user can get the HC out of the given graph.
Prerequisite	Application is in runner mode
Input	Different Permutations of degree four graph
Actions to be performed	1. Draw nodes by "Draw Nodes" button 2. Draw edges by "Draw Edges" button such that every node has four edges 3. Click "Next" button

Expected Result(s)	**Status**
Hamilton Circuit should be shown if exist.	Pass

Comments

7.3.4. Degree Five Permutations

Test Case	Check whether different permutations of degree five graphs give desired results
Functional Area	Simulator Dialog Box
Test Name	***Degree five permutations verification***
Description (Purpose)	To check that all permutations of degree five graphs are valid and after only giving graphs having HC, user can get the HC out of the given graph.
Prerequisite	Application is in runner mode
Input	Different Permutations of degree five graph
Actions to be performed	1. Draw nodes by "Draw Nodes" button 2. Draw edges by "Draw Edges" button such that every node has five edges 3. Click "Next" button
Expected Result(s)	**Status**
Hamilton Circuit should be shown if exist.	Pass
Comments	

7.3.5. Degree Six Permutations

Test Case	Check whether different permutations of degree six graphs give desired results
Functional Area	Simulator Dialog Box
Test Name	***Degree six permutations verification***
Description (Purpose)	To check that all permutations of degree six graphs are valid and after only giving graphs having HC, user can get the HC out of the given graph.
Prerequisite	Application is in runner mode
Input	Different Permutations of degree six graph
Actions to be performed	1. Draw nodes by "Draw Nodes" button 2. Draw edges by "Draw Edges" button such that every node has six edges 3. Click "Next" button
Expected Result(s)	**Status**
Hamilton Circuit should be shown if exist.	Pass
Comments	

7.3.6. Combination of Degree Two to Six in A Graph

Test Case	Check whether combination of degree two to degree six in a graphs give desired results
Functional Area	Simulator Dialog Box
Test Name	***Combination of degree two to six in a graph verification***
Description (Purpose)	To check that different combinations of degree two to six in a graphs are valid and after only giving graphs having HC, user can get the HC out of the given graph.
Prerequisite	Application is in runner mode
Input	Different Permutations of combination of degree two to degree six in one graph
Actions to be performed	1. Draw nodes by "Draw Nodes" button 2. Draw edges by "Draw Edges" button 3. Click "Next" button

Expected Result(s)	**Status**
Hamilton Circuit should be shown if exist.	Pass

Comments

7.4. System Evaluation

The objective of system evaluation is to determine whether the desired objectives have been accomplished or not. This is necessary because it checks the compatibility of developed system with the existing system. Determining the merits and demerits of the proposed system over the existing system is also covered in the system evaluation. This is concerned with the detailed study of the developed system.

A system is successful if the information produced by it has the properties of accuracy, in due course and fullness. The developed Simulator to find Hamilton Circuit from a given graph has all three properties. Special care has been taken to optimize the information processing speed of the system that results in expected output (HC) in polynomial time.

Chapter 8

Conclusion

8. Conclusion

The algorithm developed in this research project is not a hypothetical idea. The simulator is available for the sake of proof of concepts. We have tried graphs up to 30 nodes, having degree 3 and above and successfully solved all these graphs using the above mentioned algorithm and the software simulator. Graphs that do not have Hamiltonian Circuit are mostly rejected in six basic properties given in section 2. There is no back tracking involved in this algorithm therefore we can find Hamiltonian Circuit in a graph of degree three in polynomial steps.

Due to the time constraints to the submission of this project the system could not be fully evaluated but generally it produces information that posses the properties of accuracy, completeness, timeliness and conciseness. Some of the measurable human factors that are central in evaluation are ease of use, speed of performance and rate of errors.

All the factors mentioned above do not guarantee a unique interface, no matter how carefully designed and implemented has got its respective pros and cons. The ones associated with my simulator are mentioned below:

8.1. The Simulator at its Best

The Pros of my software are as below:

- The simulator is reliable because it produces accurate results in polynomial time.
- The software is user friendly; because its design is made as user friendly as possible, keeping in mind the diversity of its users.
- The software has all of the helping aspects that are covered while developing this software.
- The software also generates proper error messages for the convenience of the user. This enables the users to interact more easily with this software.
- The rate of errors is considerably reduced as maximum possible permutation are tested and data validation checks have been provided at each step of each function to ensure correct flow according the theoretical bounds.

8.2. Software Requirements

The software requirements for this software are:

- Windows XP/2000.
- Microsoft .Net framework

8.3. Hardware Requirements

The hardware requirements of the software developed are:

- Intel Pentium III/IV series computer.
- 128 MB of RAM.

References

References

1. W. R. Hamilton, *Memorandum respecting a new System of Roots of Unity*, Philosophical Magazine, volume 12 (4th series), 1856.
2. G. A. Dirac, *Some theorems on abstract graphs*, Proc. London. Math. Soc. 2, 1952.
3. R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, Plenum Press, 1972.
4. Stephen Cook, *The P versus NP Problem*, Official Problem Description, Millennium Problems, Clay Mathematics Institute, 2000.
5. J. A. Bondy and U.S.R. Murty , *Graph Theory with Applications*, North-Holland, 1976.
6. Beinecke, L. W. and Wilson, R. J. (Eds.). Graph Connections: Relationships Between Graph Theory and Other Areas of Mathematics. Oxford, England: Oxford University Press, 1997.
7. Björner, A. and Wachs, M. "Bruhat Order of Coxeter Groups and Shellability." *Adv. Math.* **43**, 87-100, 1982.
8. Kenneth H. Rosen, "*Discrete Mathematics and Its Application*", 4th Edition, p. 481-483.
9. Eric W. Weisstein. "Dirac's Theorem." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/DiracsTheorem.html
10. Ore, O. "A Note on Hamiltonian Circuits." *Amer. Math. Monthly* **67**, 55, 1960
11. Rubin, F. "A Search Procedure for Hamilton Paths and Circuits." J. ACM 21, 576-580, 1974
12. Bermond, Darrot *"Hamilton Circuits in Directed Butterfly Networks" CiteSeer Digital Library, Discrete Applied Mathematics (1996)*
13. A. Frieze, Michael Krivelevich *"Hamilton Cycles in Random Subgraphs of Pseudo-Random Graphs" Discrete Mathematics, Volume 256, Issue 1-2 (September 2002), P:137-150*
14. Eades, Hickey *"Some Hamilton Paths and a Minimal Change Algorithm"* Journal of the ACM (JACM), Volume 31, Issue 1 (January 1984), Pages: 19–29
15. Dyer, Martin *"Approximately counting Hamilton cycles in dense graphs" Symposium of Discrete Algorithm, Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms(1994), Arlington, Virginia, US. P:336-343*
16. *Ira Pohl, "A method for finding Hamilton paths and Knight's tours" Communications of the ACM, Volume 10, Issue 7 (1967), p: 446-449*
17. *Bar-Noy, "Sorting, Minimal Feedback sets and Hamilton Paths in Tournaments" CS-TR-88-1239 (1988)*

Publication

PROCEEDINGS OF
THE 2006 INTERNATIONAL CONFERENCE ON SCIENTIFIC COMPUTING

CSC'06

Editor

Hamid R. Arabnia

Associate Editors

George A. Gravvanis
Ashu M. G. Solo

Las Vegas, Nevada, USA
June 26-29, 2006
©CSREA Press

Copyright © 2006 CSREA Press
ISBN: 1-60132-007-8
Printed in the United States of America

Contents

Finding Hamilton Circuit in a Graph

Ghulam Mustafa Babar
EnterpriseDB Corporation,
Software Technology Park-1,
Islamabad, Pakistan

Dr. Sikandar Hayat Khiyal
Dept. of Computer Science
Intl. Islamic University,
Islamabad, Pakistan

Abdul Saeed
Dept. of Computer Science
Intl. Islamic University,
Islamabad, Pakistan

Abstract - The purpose of this paper is to develop an algorithm to determine the Hamilton Circuit in a given graph of degree three. This algorithm will find Hamilton Circuit in polynomial steps. We got some properties; these properties are combined to develop the above mentioned algorithm. These principles and their use will be explained by different examples. More or less these principles are simple and obvious.

Keywords: Hamilton Circuits, Degree of a node, Virtual Single Unit, Active End Point, Passive End Point.

1. Introduction:

A path X0, X1, ..., Xn-1, Xn in the graph G = (V, E) is called a Hamilton Path if V = { X0, X1, ..., Xn-1, Xn } and Xi ≠ Xj for $0 \le i < j \le n$. A circuit X0, X1, ..., Xn-1, Xn (with $n > 1$) in a graph G = (V, E) is called a Hamilton Circuit if X0, X1, ..., Xn-1, Xn is a Hamilton Path.

Is there a simple way to determine whether a graph has a Hamilton Circuit? At first, it might seem that there should be an easy way to determine this. Surprisingly, there are no known simple necessary and sufficient criteria for the existence of Hamilton Circuit. However, many theorems are known that give sufficient conditions for the existence of Hamilton Circuit. Also certain properties can be used to show that graph has no Hamilton Circuit [1].

2. Terminology:

Here are the special terms used in the following discussion.

1. **Active and Passive End Points:**
   While finding the Hamilton Circuit, we extend the path on both directions from starting node. One end point is referred as Active Endpoint and other is referred as Passive Endpoint. On each move, we extend Active Endpoint. Passive Endpoint is only extended when some Virtual Single Unit (VSU) is connected with it or it fulfills some special condition described in some rule.
2. **Virtual Single Unit (VSU):**
   During the process of finding Hamilton Circuit, multiple independent paths are created. Each independent path is referred as Virtual Single Unit. In other words, VSU is simply a path which has been decided for Hamilton Circuit. These VSUs are connected and finally form the Hamilton Circuit.
3. **Assigned Edge:**
   Any edge which is assigned some VSU ID is referred as Assigned Edge.
4. **Selected Edge:**
   Any edge which is selected for Main Path or some VSU but not assigned ID until now is referred as Selected Edge.
5. **Unselected Edge:**
   Any edge which is undecided for any VSU/Main Path is referred as Unselected Edge.

3. Basic Properties for Hamilton Circuit:

Following are some of the basic properties for Hamilton Circuit

1) If a graph has any vertex of degree one then the graph cannot have Hamilton Circuit.
2) If a vertex in the graph has degree two, then both edges that are incident with this vertex must be part of any Hamilton Circuit.
3) If there is an edge in the graph which is when removed, divide the graph into two disjoint sub-graphs then the original graph cannot have Hamilton Circuit.
4) If there is a vertex in the graph which is when removed (connected/incident edges will also be removed) divide the graph into two disjoint sub-graphs, then the original graph cannot have Hamilton Circuit. (This property is not necessary for degree-3 graphs but useful for higher degree graphs therefore we are not including checks for this property in the following algorithm).
5) If edges of degree two vertices create a cycle and that cycle does not contain all vertices of graph, the graph cannot have any Hamilton Circuit.
6) When a Hamilton Circuit is being constructed this circuit passes through a vertex, then all remaining edges incident with this vertex, other than the two used in the circuit, can be removed [5].

4. Useful Theorems:

There are two famous theorems regarding the existence of Hamilton Circuit.

1) **Dirac's Theorem:** If G is a simple graph with n vertices with $n \geq 3$ such that the degree of every vertex in G is at least n/2 then G has a Hamilton Circuit.[2]
2) **Ore's Theorm:** If G is a simple graph with n vertices with $n \geq 3$ such that $\deg(u) + \deg(v) \geq n$ for every pair of non-adjacent vertices u and v in G, then G has Hamilton Circuit. [3]

But still these don't give enough information to find out Hamilton Circuit or we can't produce an algorithm just on the basis of these algorithms.

5. Special Cases:

Now we consider some special cases and try to solve each case independently and finally combine their results to produce the algorithm. Right now we shall stick to the graphs up to degree three. The graphs having degree higher than three have more probability to contain Hamilton Circuit.

5.1. When each vertex in a graph has degree two:

As mentioned earlier in property-2, if a vertex in the graph has degree two, then both edges that are incident with this vertex must be part of any Hamilton Circuit. Since in this case, all vertices have degree two therefore all edges must be in the path. In this case, nothing is to be done, Hamilton Circuit is already present and this can be considered the best case.

5.2. When each vertex in a graph has degree three:

Following are the rules to find Hamilton Circuit in a graph having degree three:

(i) Select a starting node at random (since every vertex has same probability).
(ii) Select any two incident edges to the vertex. In other words, start extending path on both directions from starting node.
(iii) Remove the third edge.
(iv) Since each vertex has degree three therefore the vertex on the opposite side of removed edge is left with two edges and according to the property-2, both edges must be in the path, therefore mark both edges as Virtual Single Unit that must be in the path/circuit.

(v) In this algorithm, we will proceed on both sides from starting vertex, we name one endpoint (vertex) as ACTIVE ENDPOINT, and the other endpoint (vertex) as PASSIVE ENDPOINT. The ACTIVE ENDPOINT will take decision for further move at each step and PASSIVE ENDPOINT will only move/extended further when it gets some VIRTUAL SINGLE UNIT (V.S.U) incident to it or found an edge which can create a smaller cycle (property-5). In this case the other end of VIRTUAL SINGLE UNIT is marked as PASSIVE ENDPOINT after joining of original path with V.S.U.

(vi) On every move on ACTIVE ENDPOINT, the last vertex in the extended path is marked as ACTIVE ENDPOINT and same with PASSIVE ENDPOINT.

(vii) At each extension on either endpoint of main path or some V.S.U, the unused edges are removed and opposite ends (vertices) of removed edges are reconsidered for V.S.U creation, extension or joining with another V.S.U or ACTIVE/PASSIVE ENDPOINT'S. [This step has highest priority after every move/extension]

(viii) On each extension, check the concerning V.S.U'S for the edge which can create smaller cycle, removed it & extend V.S.U. Cycle can only be created if endpoints of a V.S.U or ACTIVE/PASSIVE ENPOINTS share the same edge and we adopt that particular edge. Thus simply check whether endpoints of V.S.U or ACTIVE/ PASSIVE ENDPOINTS are sharing the same edge, if so, remove that edge to avoid smaller cycle. [This step is second highest priority item after step-(vii) on every move/extension]

(ix) On joining two V.S.U'S, always check the unused edges, if pair of unused edges, on either node of connecting edge, is leading to same vertex, adopt the alternate path other than joining VSUs, as we cannot remove two edges from any vertex. (it leads to property-1 of section-3)

(x) On each extension, check if the ACTIVE ENDPOINT reach at certain vertex where remaining two edges are connecting it with two (or endpoints of a single) VSU(s) then the move is wrong. Adopt the alternate path right. If we don't have any choice or the other path isn't possible or result in wrong move then the graph doesn't have Hamilton Circuit.

(xi) Search an edge (unselected i.e. not part of any VSU) in the graph which is when removed, divide the graph into two disjoint sub-graphs. If found such edge then the original graph does not have Hamilton Circuit.

(xii) For extension on ACTIVE ENDPOINT, select either of two edges incident to ACTIVE ENDPOINT for next move.

(xiii) Whenever two VSUs are combined, both result one larger VSU, containing all the vertices of both the smaller VSUs. If one of the VSUs is main path, move the ACTIVE/PASSIVE ENDPOINTS to the new endpoint of main path.

(xiv) Both endpoints, ACTIVE & PASSIVE, can only be joined when all vertices are traversed otherwise, whenever both endpoints share the same edge that will be removed (property-5) considering it unused and extend both endpoints according to property-2.

6. The Algorithm:

Here is the actual algorithm to find out the Hamilton Circuit in a degree three graph.

Procedure FindHC ()

```
Begin
    If any node has degree < 2, FAILED

    If FindCycle () fails for any edge of graph
        FAILED (No Hamilton Circuit)
    End

    ReturnValue: = 0
    While ReturnValue is zero
        ReturnValue: = MoveNext ()
    End

    If ReturnValue = -1
```

```
        FAILED
    Else
        SUCCESS
    End
End
```

Function FindCycle (e: edge)

There should not be any edge which is when removed, divide the graph into two disjoint sub-graphs. Any such edge cannot be part of any cycle. So this function tries to find closest/smallest cycle containing the given edge. Function returns TRUE if found some cycle else returns FALSE.

```
Begin
    Mark one endpoint (node) as RED node & Add to RED list
    Mark other endpoint (node) as BLUE node & Add to BLUE list
    Remove the given edge
    LOOP
        Get each node one by one from RED list & Get all connected nodes.
        Add all connected node to temporary list
        Remove all edges incident to nodes in RED list & Empty the RED list
        Copy all nodes from temporary list to RED list
        Remove all edges connecting one node in RED list to any other RED node

        If RED list is empty Return FALSE        (loop is broken here)

        If any node in RED list is connected to BLUE list by some edge in graph
            Return TRUE                          (loop is broken here)
        End

        Get each node one by one from BLUE list & Get all connected nodes to them. Add all
        connected nodes to temporary list.
        Remove all edges incident to nodes in BLUE list.
        Empty the BLUE list.
        Copy all nodes from temporary list to BLUE list

        Remove all edges connecting one node in BLUE list to any other BLUE node

        If RED list is empty Return FAILURE          (loop is broken here)

        If any node in RED list is connected to BLUE list by some edge in graph
            Return SUCCESS                       (loop is broken here)
        End
    End
End
```

Function MoveNext ()

```
Begin
    If any node has degree < 2 Return -1
    If any node has degree = 2 Call MarkVSU () for that node
    If Main Path/VSU exist, Active Endpoint:= First node of (first) path
    Else Active Endpoint := First node of Node List
    If no incident edge selected
        Mark any two edges as selected
        Call PruneNode (ActiveEP)
        Call PrunePath ()
        If FindCycle() fails for any unselected edge
            Return -1
        End
    Else if one edge is selected
        Select any of unselected edge as selected edge
        If selected edge is connecting two VSUs
                Get unselected edges on either node of connecting edge
                If both unselected edges have same other end node
                Mark selected edge as unselected & select the alternate edge on ActiveEP
        End
        Call PruneNode (ActiveEP) and then PrunePath ()
        If VSULinking() fails for any VSU
                Mark selected edge as unselected & select the alternate edge on ActiveEP
                Call PruneNode (ActiveEP) and then PrunePath ()
                If VSULinking() fails for any VSU
                        Return -1
                Else
                        Return 0
                End
        Else
                Return 0
        End
    Else If both incident edges are assigned different VSU ID
        Call PruneNode (ActiveEP)
        If Main VSU covers all nodes
                Return 1        ( SUCCESS )
        End
    End
End
```

Function VSULinking (id: VSUID)

Since in each step, we prune the unused edges, this may create two subgraphs connected by less than two edges, which is failure case for Hamilton circuit.

```
Begin
        Add Endpoints of main VSU to RED list & Endpoints of provided (argument) VSU to BLUE list.
        Remove all edges of both VSU's to simplify calculations
        Counter:=0
        Loop
                Extend RED side to adjacent nodes & remove old edges from RED list
                If any common edge found between two sides, increment 'Counter'
                If Counter = 2
                        Return SUCCESS
                Else if any side is left with only one node
                        Return FAILED
                End
                Extend BLUE side to adjacent nodes & remove old edges from BLUE list
                If any common edge found between two sides, increment 'Counter'
                If Counter = 2
                        Return SUCCESS
                Else if any side is left with only one node
                        Return FAILED
                End
        End
End
```

Function PruneNode (n: node)

```
Begin
        Remove the unselected edge at 'n' & Call MarkVSUs () on other end node of removed edge
        If both incident edges are assigned different VSU ID Call MarkVSU (n)
        Else If both incident edges are not assigned VSU ID Call MarkVSU(n)
        Else If one edge is assigned and other is unassigned.
                Copy VSU ID from assigned to unassigned edge
                Call MarkVSU() on other end node of previously unassigned edge
        End
End
```

Function PrunePath ()

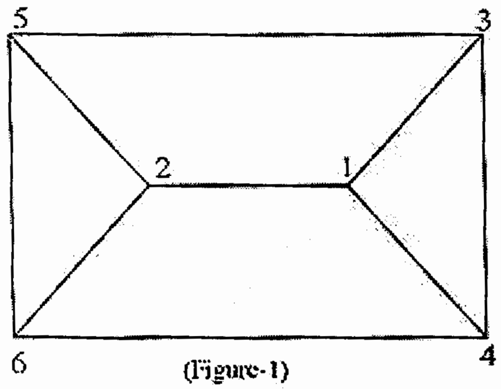
```
Begin
        If endpoints of any VSU are sharing some edge
                Remove edge & Call MarkVSU() on either nodes of removed edge
        End
End
```

Function MarkVSU (n: node)

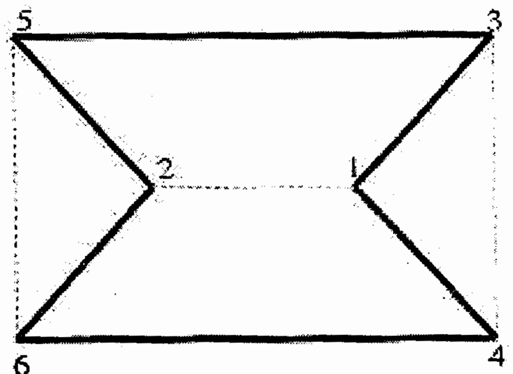
```
Begin
  If Degree of node = 2
        If any edge is not marked as selected, mark it as selected
        If Different VSU ID is set on both edges, Merge VSUs
        Else If VSU ID is set on one edge
                Copy VSU ID from assigned to unassigned edge
                Call MarkVSU () on otherend node of previously unassigned edge
        Else If both edges are unassigned
                If other end of first selected edge (Say m) is endpoint of some VSU
                        Copy VSU ID from assigned to unassigned edge on 'm'. Call PruneNode(m).
                End

                If other end of second selected edge (Say k) is endpoint of some VSU
                        Copy VSU ID from assigned to unassigned edge on 'k'. Call PruneNode(k)
                End
                If both selected edges on given node are assigned different ID,
                        Merge VSUs.
                Else If one incident edge is assigned
                        Copy VSU ID from assigned to unassigned
                Else
                        Set new ID on either edge and add Path to VSUList
                End
   If Degree of node > 2 and any two edges are marked selected
                Call PruneNode(NodeNum)
   End
 End
```

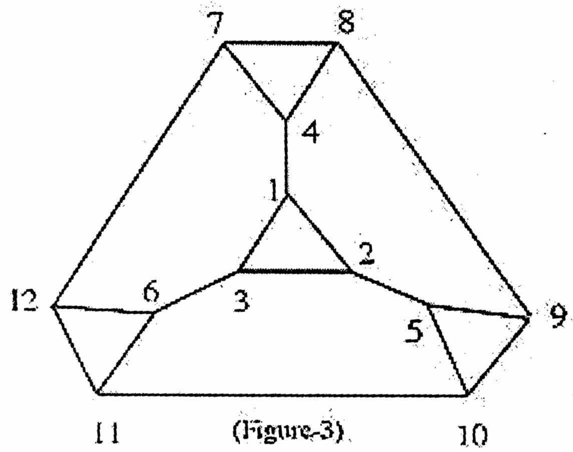
7. Experiments;



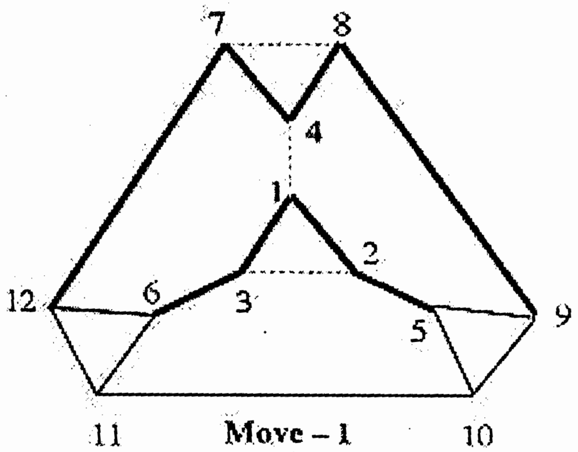
(Figure-1)



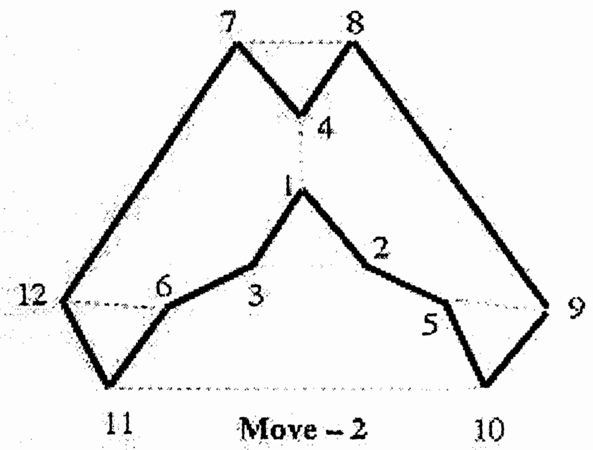
Move – I
(Figure-2)



(Figure-3)



Move – 1
(Figure-4)



Move – 2
(Figure-5)

8. Conclusion:

We have tried graphs up to 30 nodes, having degree 3 and successfully solved all these graphs using the above mentioned algorithm. Graphs that do not have Hamilton Circuit are mostly rejected in six basic 'roperties given in section 2. There is no back tracking involved in this algorithm therefore Hamilton cuit in a graph of degree three can be found in polynomial steps.

9. References:

Kenneth H. Rosen, "*Discrete Mathematics and Its Application*", 4th Edition, p. 481-483

Eric W. Weisstein. "Dirac's Theorem." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/DiracsTheorem.html

Ore, O. "A Note on Hamilton Circuits." *Amer. Math. Monthly* 67, 55, 1960

Rubin, F. "A Search Procedure for Hamilton Paths and Circuits." J. ACM 21, 576-580, 1974

Ralph P. Grimaldi, "Discrete and Combinatorial Mathematics", 4th Edition, p.523-528