Summary

This project looks, in depth, at the complexity of spanning trees with inner vertex degree constraints. The project attempts to address the problem:

Given a graph G, does G contain a spanning tree with inner vertex degrees equal to $k$?

In this project I aim firstly to give the reader an introduction to the subject of Graph Theory. Secondly, the project will cover the theory of NP-completeness and give the reader an overview of the subject. Thirdly, I will go on to prove that the problem described above is NP-complete. Finally, I will investigate the boundary between the inner vertices constraints, $2 \leq d(u) \leq n-1$ for a spanning tree, and try to establish a bound on where the problem goes from being able to be solved in polynomial time to being NP-complete.

**Subjects:** Graph Theory, NP-Theory.

**Keywords:** Spanning tree, NP-complete, polynomial time complexity.
I would like to thank S. B. Cooper for his notes on the maths module entitled “Introduction to Graph Theory” and also for advising me on which books were most suitable for reference. I would also like to thank my supervisor Haiko Müller for his help when I was struggling to understand aspects of NP-Theory and for pointing me in the right direction.
### Contents

**Section 0 – Introduction**
- 0.0: Aims and Objectives 1
- 0.1: Introduction to Project 2

**Section 1 – Graph Theory**
- 1.0: Introduction to Graph Theory 3
- 1.1: Formal Definition of the Minimum Requirement 7
- 1.2: Linking Graph Theory & Computer Science 8

**Section 2 – NP-Theory**
- 2.0: Efficiency of Algorithms 9
- 2.1: The Theory of NP-completeness 11
- 2.2: Proving NP-completeness 13

**Section 3 – NP-completeness Proofs**
- 3.0: NP-completeness Proof for 2HP 16
- 3.1: NP-completeness Proof for $SP_3$ & $SP_4$ 18
- 3.2: NP-completeness Proof for $SP_k$ 21

**Section 4 – The Polynomial Boundary**
- 4.0: Investigating the Polynomial Boundary 24
- 4.1: The Polynomial Lower Bound 25

**Section 5 – Evaluation**
- 5.0: Conclusion 29

References 30
Appendix A 31
0.0: Aim & Objectives

Project Aim:

To find the boundary between the constraints $2 \leq d(u) \leq n - 1$ and $2 \leq d(u) \leq 2$ on the degrees of the inner vertices of a spanning tree for a graph, where the spanning tree problem goes from being able to be solved in polynomial time to being NP-complete.

Objectives:

The objectives of this project are:

- To gain a good understanding of graph theory
- To understand and be able to prove NP-completeness

Minimum Requirement:

The minimum requirement of this project is to produce a proof to show that for a spanning tree, $T$, with inner vertices degree constraints $d_T(u) = k$, $\forall u \in U$, for integer values of $k \geq 2$, is NP-complete.

Further Extension:

To try to determine a “boundary” between those cases, for values of $k$, of the $SP_k$ problem that are polynomially solvable and those that are NP-complete.

Deliverables:

- This project report
- An NP-complete proof for the minimum requirement
- An investigation into the polynomial/intractable boundary.

Since my mid-project report there have been several changes to the aims and objectives of my project, I have clarified my minimum requirement and also updated my further extension. However, the main aim of the project has remained the same.
0.1 Introduction

In this project I aim firstly to give the reader an introduction to the subject of Graph Theory. I will give examples of how graph theory and computer science are related along with some practical applications of graph theory in the world of computing.

Secondly, the project will cover the theory of NP-completeness and give the reader an overview of the subject.

Thirdly, I will go on to prove that the problem as described in my minimum requirement is NP-complete.

Finally, I will investigate the boundary between the inner vertices constraints, \(2 \leq d(u) \leq n-1\) for a spanning tree, and try to establish a polynomial lower bound where the problem goes from being able to be solved in polynomial time to being NP-complete.

The “open” problem:

In [6] a similar problem to the one I am looking at is identified as being NP-complete. This problem is as follows:

**DEGREE CONSTRAINED SPANNING TREE**

**INSTANCE:** Graph \(G = (V, E)\), positive integer \(K \leq |V|\).

**QUESTION:** Is there a spanning tree for \(G\) in which no vertex has degree larger than \(K\)?

Garey and Johnson use a transformation from HAMILTONIAN PATH and conclude that the problem remains NP-complete for any \(K \geq 2\).

This problem is very similar to mine however in my problem the inner vertices have degree exactly equal to \(K\), as opposed to all the vertices having degree less than or equal to \(K\), as in Garey and Johnson’s problem. I have not come across this problem in my research and as far as I am aware it is an open problem. I have also been unable to find a proof of the Garey and Johnson problem described above, and as far as I know no one has published a paper actually proving that the problem is NP-complete.
1.0: Introduction to Graph Theory

In order to investigate spanning trees with degree constraints it is first necessary to understand the basic concepts behind graph theory. This chapter should familiarise the reader with terminology and notation that is to be used in this project report and give some background knowledge regarding graph theory. Any other notation or relevant terminology not defined here can be found in [10] or [11].

A graph \( G = (V, E) \) consists of a set of \( V \) elements called vertices together with a set \( E \) of unordered pairs of the form \( \{u, v\} \), with \( u, v \in V \), called the edges of \( G \). Notation: write \( uv \) for \( \{u, v\} \). If \( G \) contains no edges, then \( G \) is an empty graph.

The edge \( uv \) is said to join the vertices \( u, v \). We also say that, \( u, v \) are incident with \( uv \), and that \( uv \) is incident with \( u \) and \( v \). Two vertices which are incident with a common edge are adjacent – so are the two edges which are incident with a common vertex.

\[
G_1 : \quad V = \{1, 2, 3, 4, 5\} \\
E = \{\{1, 3\}, \{1, 2\}, \{2, 3\}, \{3, 4\}\}
\]

Note that all graphs that are considered in this project are simple graphs by definition, that is they do not contain so called loops or parallel edges.

If a vertex is not incident with any edge, we call it an isolated vertex (vertex 5 on \( G_1 \)). A graph consisting of just one isolated vertex is a trivial graph – all other graphs are non-trivial.

\( n \) denotes the number of vertices of \( G \). \( m \) denotes the number of edges of \( G \). For \( G_1 \) we have \( n = 5 \) and \( m = 4 \).

The degree of a vertex \( v \) of a graph, written \( d(v) \) is the number of edges incident with the vertex \( v \). E.g. vertex 1 on \( G_1 \) has \( d(v) = 2 \).
Two graphs are isomorphic, written \( G_1 \cong G_2 \), if there is a 1-1 correspondence between the edges, and between the vertices, such that the corresponding edges are incident with corresponding vertices.

If each pair of distinct vertices of a simple graph, \( G \), are joined by an edge (that is, \( u, v \in V \Rightarrow uv \in E \)) then \( G \) is complete. \( K_n \) is the complete graph on \( n \) vertices.

\[ K_5: \]

\begin{center}
Each vertex of \( K_5 \) has \( d(v) = 4 \)
\end{center}

A graph \( G_s(V_s, E_s) \) is a subgraph of \( G(V, E) \) if \( V_s \subseteq V \) and \( E_s \subseteq E \). If \( V_s \neq V \) or \( E_s \neq E \), \( G_s \) is a proper subgraph. If \( V_s = V \), \( G_s \) is called a spanning subgraph of \( G \).

An edge sequence in \( G \) is a finite sequence of edges of the form \( v_0v_1, v_1v_2, v_2v_3, \ldots, v_{n-1}v_n \). \( v_0 \) and \( v_n \) are the initial and final vertices of the edge sequence from \( v_0 \) to \( v_n \).

The length \( l \) of the edge sequence is the number of edges in the edge sequence. If all the edges of the edge sequence are distinct, it is called a chain. If all the vertices of the edge sequence are distinct, it is called a path. The edge sequence, chain or path is closed if \( v_0 = v_n \). A closed path is called a circuit.

\( G \) is connected if given any pair of vertices \( v, w \) of \( G \), there is a path from \( v \) to \( w \). Otherwise \( G \) is disconnected. A component of \( G \) is a maximal connected subgraph. Denoted \( c(G) \) for the number of distinct components of \( G \).

A forest is a graph with no circuits. A tree is a connected forest.

The forest containing all possible (non-isomorphic) trees on 5 vertices:
A **disconnecting set** of a graph $G$ is set of edges of $G$ whose removal disconnects $G$. A **cutset** is a disconnecting set, no proper subset of which is a disconnecting set. A **cut edge** is an edge $e$ such that $\{e\}$ is a cutset.

A **spanning tree** of $G$ is a spanning subgraph of $G$, which is a tree. A spanning tree of $G$ is denoted as $T = (V, F)$, where $F \subseteq E$.

Let $U \subseteq V$ be the set of **inner vertices** of a spanning tree $T$, e.g. $U = \{v : d(v) > 1\}$. $d_T(u)$ is the degree of the inner vertex, $u$, of a spanning tree, $T$. The **leaf** vertices are those vertices that are not inner vertices and have degree $\leq 1$.

**Example:**

**$G_2$:**

![Graph $G_2$](image)

Two spanning trees of $G_2$:

**$T_1$:**

![Spanning tree $T_1$](image)

**$T_2$:**

![Spanning tree $T_2$](image)

- $\bullet = $ Inner vertices.
- $\circ = $ Leaf vertices.

For $T_1$, there are 6 inner vertices $U = \{2, 3, 4, 5, 6, 7\}$ with $d_{T_1}(u)$ as follows:

$d_{T_1}(2) = 2, d_{T_1}(3) = 2, d_{T_1}(4) = 2, d_{T_1}(5) = 2, d_{T_1}(6) = 2, d_{T_1}(7) = 2$.

For $T_2$, there are 3 inner vertices $U = \{2, 6, 7\}$ with $d_{T_2}(u)$ as follows:

$d_{T_2}(2) = 4, d_{T_2}(6) = 3, d_{T_2}(7) = 2$. 
A Hamiltonian circuit of a graph is a path beginning at a vertex \( v \) visiting every other vertex once and then returning to \( v \). Every vertex in a Hamiltonian cycle has degree 2. A Hamiltonian path is similar but the path begins at a vertex \( v \) but ends at a different vertex, \( u \), say. In a Hamiltonian path all inner vertices have degree \( = 2 \), but initial and terminal vertices, \( u \) and \( v \) say, have degree \( \leq 1 \).

The example shown below is a graph of the dodecahedron:

![Graph of the dodecahedron](image)

A corresponding Hamiltonian circuit for the dodecahedron is:

![Hamiltonian circuit](image)

Formally a Hamiltonian Circuit (HC) and a Hamiltonian Path (HP) can be written:

\[
\text{HC} = \{ G : G \text{ has a Hamiltonian circuit} \}
\]

\[
\text{HP} = \{ G : G \text{ has a Hamiltonian path} \}
\]

The completion of this chapter was the first milestone in my schedule and was achieved before the desired date of completion.
Now the main aspects of graph theory have been covered we can give a formal definition of the problem as described in the minimum requirement.

Begin with a graph $G$, with vertex set $V$ and edge set $E$, denoted $G = (V, E)$. Let a spanning tree of this graph, $T$, be denoted by $T = (V, F)$, where $F \subseteq E$.

Let $U \subseteq V$ be the set of inner vertices of $T$, $U = \{ v : d(v) > 1 \}$. In this project a spanning tree which has all inner vertices degrees equal to a positive integer, $k$, i.e. $d_T(u) = k$, will be called a spanning $k$-tree. The corresponding decision problem is denoted as:

$$SP_k = \{ G : G \text{ has a spanning } k\text{-tree} \}$$

An example of this problem is as follows:

$G$:

$T$ has $d_T(u) = 3$, $\forall \ u \in U$, so in this case $G \in SP_3$.

An example of when $k = 2$, $SP_2$, is the Hamiltonian Path problem as described previously. For the minimum requirement I am going to first prove that $SP_3$ and $SP_4$ are NP-complete and then prove that $SP_k$ is NP-complete with arbitrary $k > 1$. 
1.2: Linking Graph Theory to Computer Science

This chapter gives the reader an idea of why graph theory is important in computer science. It acts as a link from graph theory to NP-completeness and prepares the reader for the next section of the project – NP-Theory.

Many problems in mathematics and its applications are of a discrete nature. Graph theory can involve problems with many vertices and the most well known problem is the Travelling Salesman Problem (TSP). So, when trying to research graph theory in depth mathematicians require the use of computer programs. Usually the number of feasible solutions of the TSP increase rapidly with the size of the problem and therefore, it is impossible to find a required solution by searching all alternatives, even with the fastest computers. Accordingly, it is necessary to seek efficient algorithms. Unfortunately for most graph theory problems, efficient algorithms are not known, and it is not even clear whether such algorithms exist. This is where NP-theory comes in to play.

NP-theory provides an approach for classifying problems according to their complexity (i.e. the degree of their intractability to computers). The theory considers a very rich class of problems (the so called class NP) containing most “real” problems. One of the main results of the theory is the fact that there exists a “most complicated” problem in this class. The theory provides a long list of “real” problems that are most complicated. Moreover, most of the problems arising in applications turn out to be “most complicated”. In fact it is often difficult to find a nontrivial problem which can be solved efficiently. Actually the finding of such a “good” problem is a kind of discovery. Examples of problems that can be solved efficiently include: shortest path, minimum spanning tree, planarity testing, and linear programming.

Some of the practical applications of graph theory play a role in the world of computer science. The connecting vertices and edges of graphs make the ideas in graph theory particularly useful to people whom design computer systems. On the smallest end of the scale, a graph can be used to model the way that tiny pulses of electricity flow through the silicon chips that are built into electronic devices. In the big picture, a graph can model the ways that computing systems can be interconnected, even when the computers are located all over the world and connected by telephone wires and satellites.
2.0 Efficiency of Algorithms

Different algorithms possess a variety of different time complexity functions. The efficiency of an algorithm is difficult to analyse, some algorithms can be found to be ‘efficient enough’ and others ‘too inefficient’, depending on the problem at hand. Computer scientists go about analysing them by separating the algorithms in to two distinct types, those that are polynomial time algorithms and those that are super-polynomial (often exponential) time algorithms.

Garey & Johnson define polynomial time algorithms as follows:

“Let us say that a function f(n) is O(g(n)) when there exists a constant c such that \(|f(n)| \leq c \cdot |g(n)|\) for all, but a finite number, values \(n \geq 0\). A polynomial time algorithm is defined to be one whose time complexity function is \(O(p(n))\) for some polynomial function \(p\), where \(n\) is used to denote the input length. Any algorithm whose time complexity function cannot be so bounded is called an non-polynomial time algorithm.”

The distinction between these two types of algorithms at first appears unnoticeable, however, when considering large problem instances there is a clear, significant difference. The table below, taken and updated from [6], looks at several typical complexity functions of each type and their execution times using present day processor speeds:

<table>
<thead>
<tr>
<th>Time complexity function</th>
<th>Size (n)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
</tr>
<tr>
<td>(n)</td>
<td>0.0000001 second</td>
</tr>
<tr>
<td>(n^2)</td>
<td>0.000001 second</td>
</tr>
<tr>
<td>(n^3)</td>
<td>0.00001 second</td>
</tr>
<tr>
<td>(2^n)</td>
<td>0.00001 second</td>
</tr>
<tr>
<td>(3^n)</td>
<td>0.00059 second</td>
</tr>
</tbody>
</table>

Looking at the table it is apparent that there are much more explosive growth rates for the two exponential complexity functions in comparison with the polynomial ones. Garey & Johnson go further and investigate the effects of improved computer technology on algorithms having the above time complexity functions, as shown in the following table:
They observe that with the $2^n$ algorithm, a thousand-fold increase in computing speed only adds 10 to the size of the largest problem instance that can be solved in an hour, whereas with the $n^3$ algorithm this size is multiplied by a factor of ten. Cobham ([2]) and Edmonds ([5]) first discussed the nature of the difference between these two types of algorithms. Edmonds found that polynomial time algorithms could be solved with “good” algorithms, but certain exponential time algorithms were not solvable by such “good” algorithms. In general a problem has not been “well solved” if a polynomial time algorithm has not been found for it. Such problems can be defined as intractable if they are so hard that there is no polynomial time algorithm that can possibly solve them.

However, despite being classed inefficient, not all exponential time algorithms are useless, some of them, such as the simplex and branch and bound methods, run quickly and have been very useful in practice, but examples like these are rare.

Both of the above tables highlight the reasons why polynomial time algorithms are much more efficient and hence more desirable than exponential ones. These reasons bring to light the idea of inherent intractability and lead on to the theory of NP-completeness.
2.1 The Theory of NP-completeness

This chapter aims to give the reader some background behind the theory of NP-completeness and an insight to where the idea originated. For more information on the theory of NP-completeness see [6].

In order to understand the theory of NP-completeness it is first necessary to define some of the terminology used, beginning with the definitions of the two types of classes:

• Definition of the Polynomial class:
  The class P contains all of the decision problems that are solvable in polynomial time by deterministic algorithms.

• Definition of the Non-deterministic Polynomial class:
  The NP class contains all of the decision problems that are solvable in polynomial time by non-deterministic algorithms.

The theory of NP-completeness came about due to the fact that theoreticians needed a more powerful way for proving intractable problems. To do this they concentrated mainly on how various problems were interrelated with respect to their difficulty. The principle technique used for demonstrating that two problems are related is that of “reducing” one to the other.

This is done by giving a transformation that maps any instance of a problem into an equivalent instance of another problem. This transformation provides the means for converting any algorithm that solves the second problem into a corresponding algorithm for solving the first problem. The idea of reduction proved to be a predecessor for NP-completeness.

Stephen Cook, was the first person to identify the theory of NP-completeness. In [3] he highlighted several main aspects of NP theory. Cook began by emphasising the significance of polynomial time reducibility. Secondly he concentrated on the class of NP decision problems that could be solved in polynomial time by a non-deterministic computer.

He then went on to prove that one particular problem in NP, called the "satisfiability" problem, has the property that every other problem in NP can be polynomially reduced to it -
suggesting that the satisfiability (SAT) problem is the "hardest" problem in the NP class. The SAT problem is special because it was the first known NP-complete problem. A proof showing that the SAT problem is NP-complete is complicated and extensive, so I will not give it here, but it can be found, for instance, in [6].

Finally, Cook suggested that other problems in NP might share the same property as the SAT problem of being the "hardest" member of NP. Since then a collection of problems have been proved equivalently "hard" as the satisfiability problem and this collection of problems are known as the class of **NP-complete problems**.

We can now give formal definitions of NP-completeness and NP-hardness:

A decision problem $X$ is called ‘NP-hard’ if for any problem $Y$ existing in the class NP, problem $Y$ polynomially reduces to $X$. A problem is NP-complete if it is NP and NP-hard.

A consequence of this is:

Any decision problem, $X$, whether a member of NP or not, to which we can transform an NP-complete problem will have the property that it cannot be solved in polynomial time unless $P = NP$. In this case we say that problem $X$ is ‘NP-hard’, since it is at least as hard as the NP-complete problems.

The relationship between the $P$ and NP classes is fundamental to the theory of NP-completeness. It is generally believed that the polynomial class, $P$ is a subset of the NP class, but $P \neq NP$, however this has never been proven.

After discovering the NP class, and the idea of NP-completeness, the next step was to find out which problems were NP-complete and which were not. This next step consisted of proving that a problem is NP-complete.
2.2 Proving NP-completeness

Proving NP-completeness can be very complicated, as in the fundamental case of the SAT problem, however, once a single NP-complete problem has been proved the procedure for proving additional problems is greatly simplified. Garey & Johnson, [6], provide a straightforward general technique for proving NP-completeness:

“Given a problem \( \Pi \in \text{NP} \), all we need to do is show that some already known NP-complete problem \( \Pi' \) can be transformed to \( \Pi \). This is done in four steps:

1. Show that \( \Pi \) is in \( \text{NP} \),
2. select a known NP-complete problem \( \Pi' \),
3. then construct a transformation \( f \) from \( \Pi' \) to \( \Pi \), and
4. prove that \( f \) is a (polynomial) transformation.”

In this project step 3 will consist of proving that \( x \in \Pi' \Leftrightarrow f(x) \in \Pi \) for all \( x \).

The transformation will be a reduction and will be denoted: \( \Pi \leq^p \Pi' \).

The above technique is designed to be applied only to decision problems, where the problem has only two possible solutions – yes or no. However, most problems have to be modified to be applied to the standard yes or no format. This is done by splitting the problem into two parts, the first part is a generic instance of the problem in terms of various components, such as, sets, graphs, functions etc. and the second part is a yes-no question asked in terms of the generic instance.

For example:

**Subgraph Isomorphism**

INSTANCE: Two graphs \( G_1 = (V_1, E_1) \) & \( G_2 = (V_2, E_2) \).

QUESTION: Does \( G_1 \) contain a subgraph isomorphic to \( G_2 \), that is, a subset \( V' \subseteq V_1 \) and a subset \( E' \subseteq E_1 \) such that \( |V'| = |V_2|, |E'| = |E_2| \), and there exists a one-to-one function \( f: V_2 \rightarrow V' \) satisfying \( \{u, v\} \in E_2 \) if and only if \( \{f(u), f(v)\} \in E' \)?

When proving NP-completeness there are six basic NP-complete problems that are commonly used as the “known NP-complete problem”, as described in step 2 above.
They are as follows:
3-SATISFIABILITY, 3-DIMENSIONAL MATCHING, VERTEX COVER, CLIQUE, HAMILTONIAN CIRCUIT, & PARTITION.

Proofs for all six of these ‘known NP-complete problems’ can be found in [6].
In relation to the problem of proving the NP-completeness of spanning trees with degree constraints, the most appropriate ‘known NP-complete problem’ to look at in depth would be the Hamiltonian Circuit (HC) problem. As part of my research I looked at several NP-complete proofs involving Hamiltonian problems, [6] and [7], and spanning tree problems.

The most useful papers, involving degree constrained spanning trees, that I came across were [4] and [8]. Paper [4] looks at ‘bounded degree spanning trees’, but mainly concentrates on the connectivity of graphs and not the constraints on the degree of the internal vertices. The authors cover the proofs for Hamiltonian circuits and paths, and talk about the NP-completeness of the problem involving a spanning tree with a maximum degree constraint.

Due to the fact that I am looking at spanning trees, which are not circuits, the Hamiltonian Path (HP) problem is more appropriate to use for reduction than the HC problem. The HP problem is very similar to that of the HC problem but it is slightly different as it doesn’t have the constraint that the first and last vertices in a sequence have to be joined by an edge.

Methods of proving NP-completeness:

There are many different ways of going about proving a problem is NP-complete, and there is no exact method to follow. However, there are several general types of proofs that can be used to provide a structure for proving a new problem is NP-complete. These are commonly known as:
- restriction
- local replacement
- component design

The proof in [6] for the HC problem utilises the component design approach. I think that an adaptation of this method would be the most suitable method to use for my NP-complete proof as my problem is closely related to the HC problem.

The idea behind the component design method is to design a ‘component’ that can be interlinked with similar components to form the constructed instance. Each component
involved performs a function in terms of the given instance. In my proof I will use vertex sets as components and link the components by edges. I will go about proving my problem by using a reduction from HP to SP\textsubscript{k}. I will need to prove that HP reduces to SP\textsubscript{k}. To do this I will take an instance of Π′ which I will denote as $G = (V, E)$ and use an ellipse to denote all vertices and edges in the set $G$. As follows:

![Graph G](image)

The smaller ellipses denote vertices, $u$ & $v$, and the line between the vertices denotes the edge $uv$. Then I will create another graph, denoted $G′ = (V′, E′)$, which contains a set of new vertices in union with $V$. I will then link the vertices in $G$ to the new vertices in $G′$ via edges from $G$ to the new vertices, as shown below, this will create a combined edge and vertex set that will give the required instance of Π. In my proofs I will omit the ellipse denoting the boundary of $G′$.

![Graph G'](image)

Formally this is written:

$G′ = (V′, E′)$ with $V′ = V \cup \{w\}$ and $E′ = E \cup \{wu, wv : w uv, w \in E\}$

Covering the main aspects of the theory of NP-completeness was my second milestone and was completed according to the specified date in my schedule. I have looked at some relative NP proofs, in [4], [6], and [8], through which I have gained a good understanding of how to go about proving my NP-complete problem and now have an idea of which approach to use. The next step is to attempt the NP-complete proof outlined in the minimum requirement.
A graph may have many Hamiltonian Paths within it, with many possible start and finish vertices. It would help when trying to prove the NP-completeness of the SP\(_k\) problem if it was possible to specify the start and finish vertices of the Hamiltonian path contained within SP\(_k\). To do this we create a sub problem of HP called 2HP, which is defined as follows:

\[
2HP = \{ \langle G, u, v \rangle : G \text{ has a HP with start and end vertices } u \text{ and } v \}\]

In order to reduce from 2HP to SP\(_k\), we first need to prove that 2HP is NP-complete. We do this by reducing from HC, which is NP-complete, to 2HP, i.e HC \(\leq^p\) 2HP.

We first construct a graph \(G'\) for each graph, \(G\) as follows:

Let \(G = (V, E)\), and let \(x \in V\). Let \(G\) have a HC (dashed edges give an example of a possible HC) which includes a vertex \(x\), which has degree \(d\).

Then go from HC to 2HP by creating a new graph \(G'\). Let the vertices adjacent to \(x\), \((y_1, \ldots, y_d)\) be connected to a vertex, \(w\), in \(G'\).
Next we show that $G \in \text{HC} \Rightarrow <G', u, v> \in \text{2HP}$.

We first need to create a HP from the given HC. We do this by omitting the edge $y_1x$ from the HC thus creating a Hamiltonian path. We now need to specify the start and finish vertices, in order to form 2HP. Let the vertex $x$ be connected to a vertex $v$ in $G'$, by the edge $xv$. Then declare the vertex $v$ in $G'$ as the end vertex of 2HP. Now deal with the opposing end of the edge $y_1x$. The vertex $y_1$ needs to have 2 edges contained in the HP, so use the previously created edge from $y_1$ to $w$ to make it part of the HP. Then add an extra edge in $G'$ from $w$ to $u$ to clearly define the start vertex, $u$, as required in the 2HP problem. An example of 2HP, using dotted/dashed edges, is given in the diagram of the construction of $G'$ above. Formally this can be written:

$$G = (V, E), \text{ where } x, y_1, \ldots, y_d \in V \text{ and } u, v, w \notin V$$

$$G' = (V \cup \{u, v, w\}, E \cup \{xv, y_1w, \ldots, y_dw, wu\})$$

$$G \in \text{HC} \Rightarrow (G', u, v, w) \in \text{2HP}.$$ 

Then we need to show that $G' \in \text{2HP} \Rightarrow (G, u, v) \in \text{HC}$

If $G'$ contains a Hamiltonian path of the form 2HP then there are always two unconnected vertices in the HP, connecting these two vertices ($y_1$ and $x$ in the example above) forms a HC in $G$.

This concludes the proof and shows that 2HP is NP-complete by reduction. 

We now know that 2HP is NP-complete so we can use it to reduce from in the $\text{SP}_k$ proof.
3.1: NP-complete Proof for SP₃ & SP₄

Before trying to establish a proof for the general case of SPₖ, for arbitrary values of \( k \geq 2 \), I am going to attempt to prove the easier cases of SP₃ & SP₄. Firstly SP₃, denoted by:

\[
\text{SP₃} = \{ <G, u, v>: G \text{ has a spanning 3-tree } \}
\]

It is easy to see that \( \text{SP₃} \in \text{NP} \). (V, F) must be a tree with all inner vertices having degree \( k \), so a non-deterministic algorithm need only guess a subset, \( F \), of \(|V| - 1\) edges and check in polynomial time that all the required edges belong to the edge set of the given graph.

**Proof of \( \text{SP₃} \in \text{NP} \):** First prove \( 2\text{HP} \leq_{m}^p \text{SP₃} \)

Take an instance of 2HP (dotted/dashed edges), in \( G = (V, E) \), and label all inner vertices \( x₁, x₂, \ldots, xₙ \), where \( n \) is total number of inner vertices of 2HP, as follows:

Then for each inner vertex add an edge which is adjacent to a vertex \( y₁, y₂, \ldots, yₙ \). Place these added vertices in \( G' \).
This shows that \((G, u, v) \in 2\text{HP} \Rightarrow G' \in \text{SP}_3\) because all the inner vertices in the spanning tree have degree equal to three, thus \(\text{SP}_3\) is created. This can be formally written as:

\[
G = (V, E), \text{ where } u, v, x_1, \ldots, x_n \in V \text{ and } y_1, y_2, \ldots, y_n \notin V
\]

\[
G' = (V \cup \{y_1, y_2, \ldots, y_n\}, E \cup \{x_1 y_1, x_2 y_2, \ldots, x_n y_n\})
\]

\((G, u, v) \in 2\text{HP} \Rightarrow G' \in \text{SP}_3.\)

Now we prove \(G' \in \text{SP}_3 \Rightarrow (G, u, v) \in 2\text{HP}:\)

Due to the fact that \(G'\) has a spanning 3-tree, the graph \(G\), with leaf vertex \(u\) and leaf vertex \(v\), within \(G'\), must have the property 2HP, as all spanning trees contain a HP.

This concludes the proof for \(\text{SP}_3\), and shows that it is NP-complete.

Proving the NP-completeness of \(\text{SP}_4\) is similar to the proof of \(\text{SP}_3.\)

Given the definition of \(\text{SP}_4:\)

\[
\text{SP}_4 = \{ <G, u, v>: G \text{ has a spanning 4-tree } \}
\]

Using the same reduction as in the case with \(\text{SP}_3\), \(2\text{HP} \leq^p_m \text{SP}_4\), to show that \((G, u, v) \in 2\text{HP} \Rightarrow G' \in \text{SP}_4\) in the construction of \(G'\) you just add an extra edge coming from the inner vertices, i.e. \(x_1\) will have edges \(u x_1, x_1 y_1, x_1 y_2, x_1 x_2\). The additional edges \(x_1 y_{11}\) and \(x_1 y_{12}\), give the inner vertex of the spanning tree the required degree of 4.

![Diagram showing the relationship between G and G']
The proof that $G' \in \text{SP}_4 \Rightarrow (G, u, v) \in \text{2HP}$ is the same as in the proof for $\text{SP}_3$ above. This concludes the proof for $\text{SP}_4$, and shows by reduction that it is NP-complete.

The two proofs for $\text{SP}_3$ and $\text{SP}_4$ are fairly similar and the proof for the arbitrary case of $\text{SP}_k$ should follow the same format.
3.2: NP-complete Proof for SP\(_k\)

This chapter will contain the NP-complete proof for the problem SP\(_k\), as described in the formal definition of the minimum requirement.

**THEOREM:**

The problem SP\(_k\) = { <G, u, v>: G has a spanning k-tree } is NP-complete.

**PROOF:**

It is easy to see that SP\(_k\) ∈ NP. (V, F) must be a tree with all inner vertices having degree k, so a non-deterministic algorithm need only guess a subset, F, of |V| – 1 edges and check in polynomial time that all the required edges belong to the edge set of the given graph.

To show that SP\(_k\) is NP-complete we use the reduction from 2HP to SP\(_k\), as seen previously with the SP\(_3\) and SP\(_4\) proofs. We need to show that (G, u, v) ∈ 2HP ⇒ G' ∈ SP\(_k\) and that G' ∈ SP\(_k\) ⇒ (G, u, v) ∈ 2HP.

**Proof that 2HP ≤\(_m\) SP\(_k\):**

Firstly show that (G, u, v) ∈ 2HP ⇒ G' ∈ SP\(_k\)

Take an instance of 2HP (dotted/dashed edges) in graph G = (V, E). We need to construct a graph G' = (V', E') such that G' has a spanning k-tree if and only if G' has a Hamiltonian Path with start and finish vertices u and v. Denote G in the following way:

Label all inner vertices x\(_1\), x\(_2\),……, x\(_n\), where n is total number of inner vertices of 2HP, as follows: (dotted line represents all vertices in HP between x\(_2\) and x\(_{n-1}\))
Now for the construction of $G'$:

For each inner vertex in $G$ create additional vertices $y_{n1}$, $y_{n2}$, ..., $y_{n(k-2)}$. Place these additional vertices in $G'$. Then create edges from $x_1$ to $y_{11}$, $y_{12}$, ..., $y_{1(k-2)}$ and from $x_2$ to $y_{21}$, $y_{22}$, ..., $y_{2(k-2)}$ and so on up to and including vertex $x_n$. This gives:

Example of the construction of $G'$, with $SP_5$ and 3 inner vertices: $(k = 5, n = 3)$
The creation of the vertices in $G'$ and the edges connecting $G$ and these vertices gives the inner vertices of 2HP degree 5 and hence create $SP_5$. This general method can be applied to all spanning $k$-trees.

This method can be formally written as:

$$G = (V, E), \text{ where } u, v, x_1, \ldots, x_n \in V \text{ and }$$

$$y_{11}, y_{12}, \ldots, y_{1(k-2)}, y_{21}, y_{22}, \ldots, y_{2(k-2)}, \ldots, y_{n1}, y_{n2}, \ldots, y_{n(k-2)}, \not\in V$$

$$G' = (V \cup \{y_{11}, \ldots, y_{n(k-2)}\}, E \cup \{x_1y_{11}, x_1y_{12}, \ldots, x_ny_{n(k-3)}, x_ny_{n(k-2)}\})$$

$$(G, u, v) \in 2HP \Rightarrow G' \in SP_k.$$ 

Now for the proof of $G' \in SP_k \Rightarrow (G, u, v) \in 2HP$:

$G'$ contains a spanning tree with inner vertices that have degree equal to $k$. Taking the set of vertices $V$ in $G'$, i.e. the vertices in $G$, this set of vertices contains a Hamiltonian path with leaf vertex $u$ and leaf vertex $v$.

This concludes the proof for $2HP \leq^P SP_k$ and shows by reduction, that it is NP-complete. 

The proof of my minimum requirement, which was my third milestone, took me longer than I had expected and was completed a few weeks after the desired schedule date.
4.0 Investigating the Polynomial Boundary

In this section I aim to try to determine, or at least narrow down, the “boundary” between those cases of the SP_k problem that are polynomially solvable and those that are NP-complete.

Now that I have proved that the initial problem of $\text{SP} = \{(G, k) : G \in \text{SP}_k\}$ is NP-complete, I can go on to investigate some sub-problems of SP_k. We view the sub-problems of the NP-complete problem SP_k as lying on different sides of an imaginary “boundary”. On one side of the boundary there are those problems that are polynomial time solvable and on the other side there are those problems that are intractable. In my research I found two useful papers that looked at the complexity of spanning tree problems, [1] and [9]. Paper [9] had a particularly useful table, (pp 287), highlighting certain properties of trees and their relative complexity.

I will look at the constraints placed on k in relation to the total number of vertices, n. I will try and establish a polynomial lower bound in order to decrease the possible range of the values of k in the following constraint:

$$n^{1-\lambda} \leq k \leq n - 1 \quad \text{where} \quad 1 \geq \lambda > 0 \quad \text{and} \quad |V| = n$$

At some point in this range the problem of finding an algorithm deciding whether or not a graph has a spanning tree with inner vertex degrees equaling k goes from being NP-complete to being polynomial time solvable. I aim to reduce this region so that the constraint looks like this, with the question mark representing some function of n:

$$n^{1-\lambda} << k << ?$$

I will begin by looking at the initial case of exactly one inner vertex, then go on to look at two and three inner vertices and finally generate a formula for an arbitrary number of inner vertices.
4.1 The Polynomial Lower Bound

Firstly investigate the case where there is exactly one internal vertex, $|U| = 1$. We denote the total number of inner vertices as $c$ and the total number of edges in the graph as $m$.

This case can be modelled as follows:

With one inner vertex, all vertices in the graph must be connected to the inner vertex. This forms a set of $n - 1$ vertices that are adjacent to $u_1$. So if $n - 1$ vertices are adjacent to $u_1$ then $u_1$ must have degree $k = n - 1$.

To check if a graph has a exactly one inner vertex with degree $k = n - 1$ you need to first check all $n$ vertices in the graph and then check all $m$ edges connecting each pair of vertices. This method of checking gives this problem time complexity of $O(nm)$, i.e. it can be checked in polynomial time. The case with one inner vertex is the simplest of the sub-problems so the restriction $k \leq n - 1$ is the upper most bound on $k$.

Next we look at the case where $|U| = 2$, which can be modelled (e.g.) as follows:
The private sets contain those vertices adjacent to only one inner vertex; the mutual sets contain vertices that are adjacent to both inner vertices. We can denote the sets, using the notation: \( N(u_i) = \text{neighbours of } u_i \), i.e. the vertices adjacent to \( u_i \), as follows:

- Private set for \( u_1 \) is \( N(u_1) \setminus N(u_2) \)
- Private set for \( u_2 \) is \( N(u_2) \setminus N(u_1) \)
- Mutual set for \( u_1 \) and \( u_2 \) is \( N(u_1) \cap N(u_2) \)

To form the problem where both inner vertices, \( u_1 \) and \( u_2 \), have degree equal to \( k \) we first need to divide up the total number of vertices, excluding \( u_1 \) and \( u_2 \), (i.e. \( n - 2 \) vertices) into two separate sets of \( k - 1 \) vertices. So the total number of leaf vertices in terms of \( n \) must equal \( n - 2 \) and the total number of leaf vertices in terms of \( k \) must equal \( (k - 1) + (k - 1) \). This gives:

\[
    n - 2 = (k - 1) + (k - 1) \rightarrow n - 2 = 2k - 2 \rightarrow k = \frac{n}{2}
\]

To calculate the time complexity for checking if a graph has exactly two inner vertices with degree equal to \( k \), you first you check the vertices in the private sets, then check the vertices in the mutual set, and finally check all \( m \) edges adjacent to these vertices. So the time complexity of the problem involving exactly two inner vertices is \( O(n^2m) \).

So we now have narrowed the lower polynomial bound down from \( k \leq n - 1 \) to \( k \leq \frac{1}{2}n \). The range in between these two polynomial bounds, i.e. \( \frac{1}{2}n < k < n - 1 \), can be checked in \( O(n) \) time as you need only check the \( n \) vertices in the graph to see that it does not contain a spanning \( k \)-tree as you cannot have, for example, 1.5 inner vertices.

Now for the case where \( |U| = 3 \).

This case is more complicated than the previous cases as it involves three private sets, for inner vertices \( u_1, u_2, u_3 \):

- Private set for \( u_1 \) is \( N(u_1) \setminus (N(u_2) \cup N(u_3)) \)
- Private set for \( u_2 \) is \( N(u_2) \setminus (N(u_1) \cup N(u_3)) \)
- Private set for \( u_3 \) is \( N(u_3) \setminus (N(u_1) \cup N(u_2)) \)

As well as four mutual sets:

- \( (N(u_1) \cap N(u_2)) \setminus N(u_3) \)
- \( (N(u_1) \cap N(u_3)) \setminus N(u_2) \)
- \( (N(u_2) \cap N(u_3)) \setminus N(u_1) \)
- \( N(u_1) \cap N(u_2) \cap N(u_3) \)
In this case where $|U| = 3$ there are always $n - 3$ vertices in total adjacent to the inner vertices.

In the diagram above, showing an example $u = 3$, $u_1$ and $u_3$ have $k - 1$ adjacent vertices and vertex $u_2$ has $k - 2$ adjacent vertices.

This implies that:

$$k - 1 + k - 2 + k - 1 = 3k - 4 = n - 3$$

Rearranging this gives:

$$k = \frac{n - 3 + 4}{3} = \frac{n + 1}{3}$$

The time complexity of the problem involving exactly three inner vertices is $O(n^3 m)$. First you check which vertices are in the private sets, then check which vertices are in the mutual sets, then check which vertices are in the set $N(u_1) \cap N(u_2) \cap N(u_3)$, these three checks are done in $O(n^3)$ time. Then finally check all $m$ edges adjacent to all of these vertices, done in $O(m)$ time. Combining these checks gives the time complexity of $O(n^3 m)$.

Again, checking the region: $\frac{n + 1}{3} < k < \frac{n}{2}$ can be done in $O(n)$ time. In fact this polynomial time complexity applies to all regions in between two values of $u$.

The method applied to $|U| = 3$ can be applied to all $|U|$. However, the distribution of those vertices in the mutual sets throughout the inner vertices could become complicated for large values of $|U|$. The problem of which inner vertex should receive a vertex from a mutual set first would probably need some sort of distribution algorithm. After the private sets had been taken into account the algorithm would decide which inner vertex has the lowest degree, and thus which inner vertex should have priority when the vertices in the mutual sets are distributed. I would suggest further research into this area in order to design some such algorithm.
Analysis of \(|U|\) to find a formula for when \(u\) equals an arbitrary constant, \(c\):

| Number of inner vertices, \(|U|\) | Total number of leaf vertices in terms of \(n\) | Total number of leaf vertices in terms of \(k\) | Time Complexity |
|-----------------------------------|--------------------------------|--------------------------------|----------------|
| 1                                 | \(n - 1\)                     | \(k\)                          | \(O(nm)\)      |
| 2                                 | \(n - 2\)                     | \(2k - 2\)                     | \(O(n^2m)\)    |
| 3                                 | \(n - 3\)                     | \(3k - 4\)                     | \(O(n^3m)\)    |
| 4                                 | \(n - 4\)                     | \(4k - 6\)                     | \(O(n^4m)\)    |
| \(c\)                             | \(n - c\)                     | \(ck - 2(c - 1)\)              | \(O(n^c m)\)   |

From this table we can see that \(k\) can be calculated by setting the number of leaf vertices in terms of \(n\) equal to the number of leaf vertices in terms of \(k\), as follows:

\[ n - c = ck - 2(c - 1) \]

This rearranges to give:

\[ k = \frac{n + c - 2}{c} \]

Checking that this arbitrary problem occurs has time complexity \(O(n^c m)\) which is polynomial. So we have reduced the possible region and now have a polynomial lower bound on the degree of the inner vertices of a graph with \(c\) inner vertices.

So we can now say that the boundary, where the problem goes from being polynomially solvable to being NP-complete, lies in the region:

\[ n^{1-\lambda} \ll k \ll \frac{n + c - 2}{c} \]

\[ \uparrow \quad \uparrow \]

NP-c → Polynomial

This section of my project was a further extension to my project and hence was the last section to be completed. Although my NP-complete proof took longer than I expected, which meant I undertook the extension after the desired schedule date, I still managed to complete it in time.
5.0 Conclusion

In order to investigate the aim of the project, which was to try to find the boundary where the problem went from being polynomial time solvable to being NP-complete, I first needed to prove that the problem itself was NP-complete. I successfully proved that the problem, $SP_k$, was NP-complete (in section 3.2) by using a reduction from a known NP-complete problem, 2HP.

Once it was established that the problem was NP-complete, I was able to go on to investigating the polynomial/intractable boundary between the constraints. I looked at the polynomial bound and tried to reduce it in order to narrow down the boundary. I managed to find a lower polynomial bound (in section 4.1) and now know that at some point in the region,

$$n^{1-\lambda} \ll k \ll \frac{n+c-2}{c}$$

the problem goes from being polynomial to being NP-complete. In finding a lower polynomial bound I successfully reduced the possible polynomial/intractable region. Due to the fact that the $SP_k$ problem is NP-complete it is impossible to narrow the boundary completely, because one of the properties of NP-completeness is that it is believed that NP does not equal P.

I conclude that the project did achieve its original aim as a ‘better’ boundary was successfully found. The main objectives of this project were firstly to gain a good knowledge of graph theory and secondly to have an understanding of NP-Theory in order to be able to prove NP-completeness. I believe that my previous knowledge of graph theory combined with considerable research has enabled me to achieve a competent understanding of the subject. Also, I feel I have ascertained a good understanding of NP-Theory and the idea of NP-completeness. I think that looking at many NP-completeness proofs, along with studying some chapters from Garey and Johnson’s book, have been crucial in helping me to prove my own NP-complete problem. Overall I believe I have achieved the main objectives that I set out to achieve at the beginning of the project.


Appendix A:  
Reflection on the Project Experience

I found that the project experience was difficult and time consuming but ultimately rewarding. Throughout this experience I came across many problems and on reflection would go about things differently if I went through the experience again. For my project I wanted to try and combine the mathematical side of my degree with the computing side. I found that the subject of graph theory along with the idea of NP-completeness allowed me to do this.

What I learnt:
I learnt that it is essential to try and clarify your objectives early in the project development, as it is vital to know what your problem is and what you are going to do to address the problem. This allows you to plan your schedule and you should try to make the up most effort to stick to it, in order to achieve all your goals within the allotted time span. I also think that it is important not to underestimate the amount of background research that needs to be done and the time it takes to do. I would suggest that clarification of the aims and objectives and the background research should try to be completed in the first semester, so that at the beginning of the next semester the implementation of the solution can begin.

Problems I came across:
After handing in my mid-project report I found out that there was a problem with obtaining a license for the LEDA package, forcing me to change my initial aims and objectives. I found that the idea of NP-completeness can be quite a tricky concept to grasp and that although books are useful it is much easier to understand if you have it explained to you first hand. Once you see some examples of how NP-completeness is proved it is a lot easier to go about proving open NP-complete problems. I found it a lot easier to understand NP proofs by drawing diagrams to represent the symbols in the proof. I feel that my NP-complete proof is clear and easy to follow because of the diagrams I used.

Suggestions for further research:
In order to take my project further I think it would be worthwhile to look at additional degree restrictions perhaps involving the connectivity and the maximum degree of a spanning tree. Also I think that utilisation of the LEDA package would enable the analysis of various types of spanning trees with many vertices and also provide the facility to test some of my results.