Computational Basis for Phylogenetics and Artificial Intelligence in Fossil Evaluation

The Current and Future Role of Computer Sciences to Paleontology and Evolutionary Sciences

by
Ruben Safir
April, 2017

A Master's Thesis submitted to the Graduate Faculty of
Long Island University, Brooklyn Campus
In Partial Fulfillment of the Requirements for the Degree of
Masters of Sciences

MAJOR DEPARTMENT:
Technology, Innovation,
and Computer Science

Advisor:
Professor Christopher League

Chair:
Professor Mohammed Ghriga

Table of Contents

The Digital Revolution in Biology and Evolutionary Sciences and the problems it presents for Computer Scientists

Like most areas of civilization, Paleontology has been undergoing a quiet revolution, not just with the exciting discoveries in the field, but with the analysis of evolutionary processes by mathematical algorithms. The discoveries of the evolutionary link between Dinosaurs and Birds has been confirmed and argued through the application of cladistics and phylogenetics. In fact, one can argue that the adaptation of computational analysis within the area of phylogeny makes analysis of cladistics possible, and that makes the understanding of such discoveries within the context of known evolution theory even more revolutionary than the fossil discoveries themselves. Computer algorithms, scanning technology, and the application of artificial intelligence in the area of evolutionary sciences is at its infancy. And yet, the use of computers has remarkably changed the field from the basic education of Paleontologists, to the end analysis of fossils, and soft tissue. Computational Sciences has become the language of paleontological discourse.

Through the use of computational mathematics, the field of Paleontology has been more tightly integrated with genetics, and molecular biology. This cocktail has been added to the traditional fields of geology, comparative anatomy, and biology. Paleontologist are far greater generalists than perhaps any other area in the sciences. It also means that there is an opportunity for serious students of computer sciences to make contributions to the field which previously had been unavailable.

The current crop of paleontologists come out of their primary education with familiarity with a group of tools that have been developed over the last 2 decades in order to analyze fossils and their placement within the evolutionary tree. While they have a deep understanding of the principles of cladistics and phylogeny, along with statistics, their study of the mathematics and algorithms is short of what computer science students experiences. Software used for their analysis includes AWTY, PAUP, TNT and the R programming environment. Students are trained to use these applications, and discussions revolve about how they are applied. There are long conversations about studies within mailing lists and forums that attack and support the application of software.[1] But within the textbooks and references being used to learn the principles of the mathematics of Phylogeny, these text don't provide the exacting rational proofs, nor do they provide the coding examples that exist in standard texts for Computer Sciences. As an algorithms text, which essentially these Phylogeny books are, they fail in communicating an organized body of knowledge, which for mathematics textbooks is a clear necessity in order to enable students to comprehend the material as a whole. The difference in approach can be traced to an antiquated idea that Biology is less exacting mathematically than other sciences. The

1 See the Dinosaur Mailing List out of USC. For example on March 28th, 2017, it was noted “Ah, so they didn't know how to use TNT effectively. I'm not surprised based on what I found when adding Daemonosaurus and Chilesaurus to their matrix. Stay tuned to my blog for more details...”

math, however, doesn't care about such notions. It's logic is formalized and its application has to be made in all cases with understanding of the root logic. When compared to standard texts in Computer Sciences, such as Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein's “Introduction to Algorithms”, one can quickly see where parallel works in Paleontology are lacking in clear explanation, detail and logic.

With Phylogeny texts clear definitions are lacking and the texts vary in nomenclature from section to section. While these textbooks are encyclopedic on the topic, and are thick in its annotation, for students of computer sciences, they lack the hierarchical organization of presentation that is customary, and they fail to tie topics together into a formal continuum of knowledge. Worse that this, these text books fail to present the material as a subset of a larger and well established body of computational sciences, and graph theory. There is a lack of explicit proofs One might have to treck through original papers in order to fully grasp the concepts which are laid out in the text. If one is to understand the principles presented well enough to solve problems and code example algorithms, then they need to dig deeper than what is presented in the materials.

This thesis hopes to clarify much of the fundamental mystery that exists in paleontology by laying out some of the founding algorithms to the math and sciences that it uses and present it in a comprehensive fashion in a single location. It can not cover the entirety of the field, as that would take years to write, but it can put the computer scientist on the correct road and open up much of the black box that currently is being used by students in this area. As such, we will break down the exploration of paleontology into three themes:

1. Graph Theory and Phylogenetics/Cladistics
2. Pattern Recognition Algorithms and Theory
3. Artificial Intelligence and Expert Systems

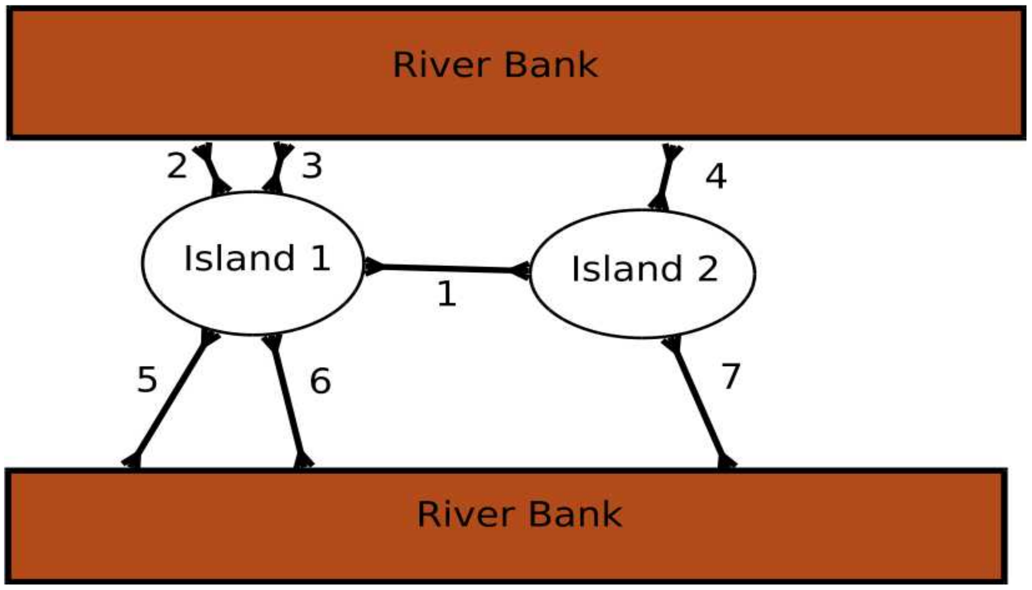
Terminology

- Map: A geological representation of the world
- Graph: An diagram that represents an abstraction of states and processes that join states
- Vertex: A state in a graph
- Node: synonym for a vertex
- Edge: A line in a graph that represents a relationship between vertexes.
- Arches: Euler and Lugar's synonym for an Edge
- Cost: An abstract quantified measure of the processes needed to travel between states

Graph Theory and Phylogenetics

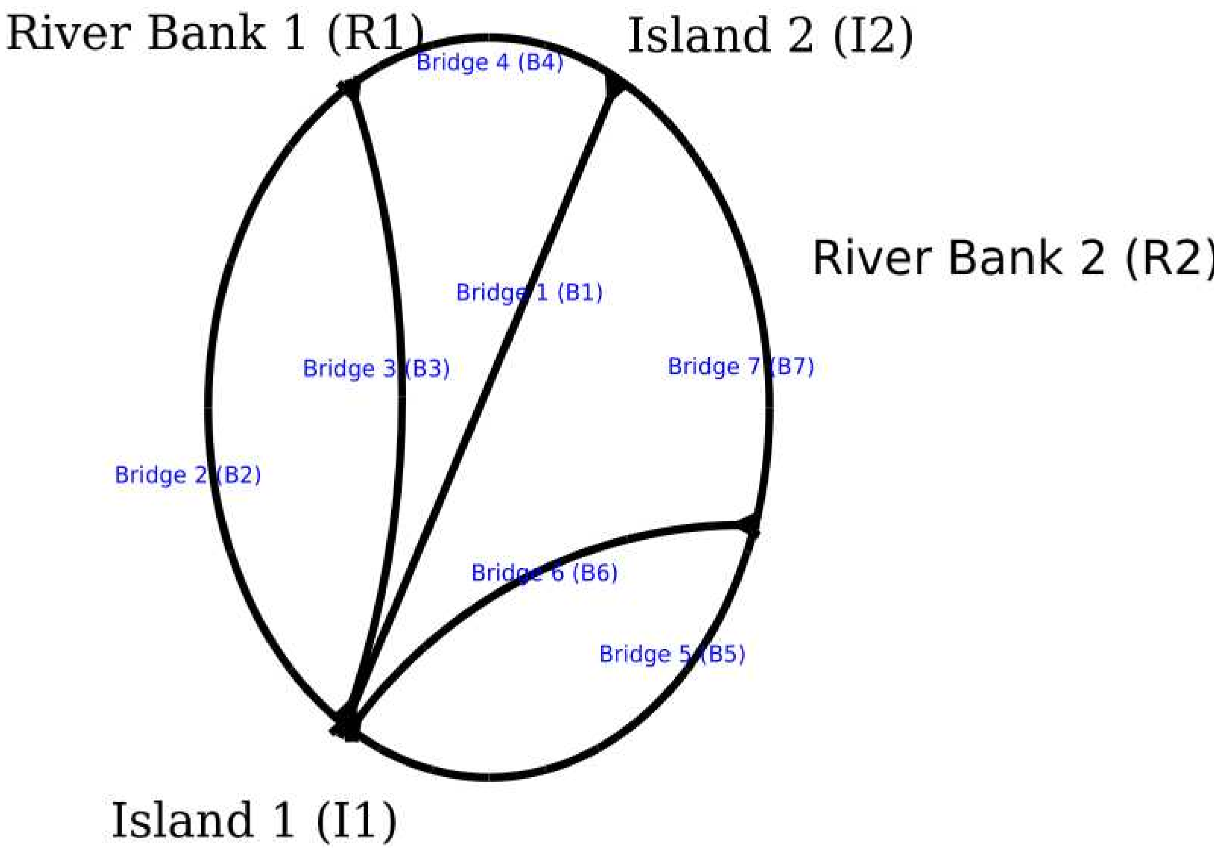
Classical Graph Theory: Euler and the Königsberg Problem

Classical graph theory in mathematics starts with Euler's "Seven Bridges of Königsberg" problem which can be described as follows: Given the following geography of the city of Königsberg, see below, the question arises, "Is it possible to travel through the city by crossing every bridge, never crossing a bridge twice".



As covered in George Luger's standard text on artificial intelligence[2] these kinds of problems are **graph problems** and they perform s*tate space searches*, which are of critical import to paleontology. We create what is called a s*tate space graph* in order to apply graph theory to the problem and try to describe the graph in terms of *predicates*. A predicate is a type of question which follows specific mathematical rules that can be described and applied in a mechanical fashion to draw conclusions. According to Lugar, our graph is composed of nodes and arches. Nodes are states, and in this case landing points. Arches (also called links or edges) are transitions. Normally we call "arches", "edges". When we look at the city of Königsberg through this kind of filter, striping the nodes and arches of any meaning of length or size, we can create an abstract map that looks as follows:

2 George Luger, Artificial Intelligence – Structures and Strategies for Complex Problem Solving, Pearson – Addison Wesley, 6th Edition pg. 80



With this graph in mind we can note that for our problem it matters not if we cross a bridge one way or the other. If we cross any bridge twice, then we know that such a proposed solution fails, that is we can not walk the city by that path, to each node, without crossing a bridge twice. If we can not find any such path, then we have answered our problem negatively. If we do find such a path, then we have answered our query positively.

So what do we know? We know the following predicates:

First: **crossing(Node1, Node2, Arch1) == crossing(Node2, Node1, Arch1)**

This implies that no matter how we cross Node1 and Node2 its the same "bridge".

Therefore we have the following sets of crossings:

```
1. crossing(I1,I2,B1)
2. crossing(I1,R1,B2)
3. crossing(I1, R1, B3)
4. crossing(I1,R2,B6)
```

5. `crossing(I1,R2, B5)`
6. `crossing(R1,I2, B4)`
7. `crossing(I2,R2,B7)`

Euler noted the following predicate as well, For any node that one enters on the path, you need to exit through a unique arch. *If you enter a node, you have to exit through a new and unused arch. Therefore, each node must have an even number of arches.* There are two exceptions to this rule, at the beginning and at the end. *If the beginning and the end are different nodes, then they must have an odd number of attached arches.*

This classical theory, originally published in 1736 in Latin, is the starting point to understand graph theory. Since all of Paleontology involves evolutionary graphs, basic understanding of graph theory is a prerequisite to understanding the fundamental theories behind cladistics and phylogeny. Let’s examine a coding example for determining the Königsberg problem. For this paper, with will start with C++ coding examples using GNU make and gcc.

Our makefile is as follows:

```
XX:=gcc
CXXFLAGS:=-Wall -ggdb -pg
LDFLAGS:= -pg -pthread

konigsburg: euler.o nodes.o
	$(CXX) $(CXXFLAGS) $(LDFLAGS) -o konisgburg.exe euler.o nodes.o

euler: euler.cpp
	$(CXX) $(CXXFLAGS) $(LDFLAGS) -c euler.cpp

nodes: nodes.cpp nodes.h
	$(CXX) $(CXXFLAGS) $(LDFLAGS) -c nodes.cpp

include make.deps
make.deps: *.cpp; ${CXX} ${CXXFLAGS} -M *.cpp >$@

```

Graphs have nodes (or vertices) and edges. Graphs can be directional, where traveling between points can only go in a single direction, or bidirectional, such as our example. There can also be assigned costs between vertices's. We can describe nodes within a class where each node object is a class instance. And we can describe that class in nodes.h. For maximum flexibility, we can add C++ templates to nodes, which allow us to describe node instances according to any type of data we chose.

The simplest data structure can be a class of node with just two data members within it, name and data. We drop this in the .h file, which when we use templates also includes most of the class code.

```cpp
/*
 * =====================================================================================
 *
 *       Filename:  nodes.h
 *
 *    Description:  Basic Node for Euler Graph
 *
 *        Version:  1.0
 *        Created:  02/11/2017 06:14:42 PM
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Ruben Safir (mn), ruben@mrbrklyn.com
 *        Company:  NYLXS Inc - LIU Thesis
 *
 * =====================================================================================
 */

#ifndef NODES_H
#define NODES_H
#include<iostream>
#include<string>
#include<vector>
namespace tree {

   /*
    * =====================================================================================
    *        Class:  Node
    *  Description:  This class describes a single node and its data
    * =====================================================================================
    */
   template < class T >
      class Node
      {
         public:

            /* ====================  LIFECYCLE     ======================================= */
            Node (std::string n, T d): data{d}, name{n} {

            };                             /* constructor */

            /* ====================  ACCESSORS     ======================================= */
            std::string get_name()const{
               return name;
            }

            /* ====================  MUTATORS      ======================================= */
            /* ====================  OPERATORS     ======================================= */
            /* ====================  DATA MEMBERS  ======================================= */
            T data;
         protected:

         private:
            std::string name;

      }; /* ----------  end of template class Node  ---------- */

   /*
    * =====================================================================================
    *        Class:  Edges
    *  Description:  Bridges - usually I wouldn't make edges but we are being consistant
    *  with the Luger description of the problem.
    * =====================================================================================
    */
   template < class T >
      class Edge
      {
         public:

            /* ====================  LIFECYCLE     ======================================= */
            Edge (std::string n, T d): data{d}, name{n}{};

            /* ====================  ACCESSORS     ======================================= */
            std::string get_name(){
               return name;
            }

            /* ====================  MUTATORS      ======================================= */

            /* ====================  OPERATORS     ======================================= */

            /* ====================  DATA MEMBERS  ======================================= */
            T data;
         protected:

         private:
            std::string name;

      }; /* ----------  end of template class Edges  ---------- */
   template< class node_type, class bridge_type>
   class Predicate{
      public:
         Predicate(Node<node_type>& in_a, Node<node_type>& in_b, Edge<bridge_type>& in_c)
            : a{in_a}, b{in_b}, c{in_c}{
               a.data++;
               b.data++;
            };
         Node<node_type>& a;
         Node<node_type>& b;
         Edge<bridge_type>& c;
   };

}//close of namespace tree

#endif
```

```
/*

 * =====================================================================================
 *
 *       Filename:  euler.cpp
 *
 *    Description:  Demonstration Program for Eulers Konisgburg problem
 *
 *        Version:  1.0
 *        Created:  02/12/2017 10:21:23 AM
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Ruben Safir (mn), ruben@mrbrklyn.com
 *        Company:  NYLXS Inc
 *
 * =====================================================================================
 */

#include "nodes.h"

int is_euler(tree::Node<int> ** city )
{
   int odd = 0;
   int size = 4;
   for(int i=0;i<size;i++){
      if( (city[i]->data % 2) == 1 )
      {
         odd++;
         std::cout << odd << std::endl;
      }
   }
   if(odd == 0 || odd == 2){
      return true;
   }
   return false;
}
```

```

int main(int argc, char * argv[])
{
    /*Create all the nodes */
    /* data can be anything useful to describe a node.  In this case
    ¦* it will be an int to represent degree (or number of connections */

    tree::Node<int> I1 {"Island 1", 0};
    tree::Node<int> I2 {"Island 2", 0};
    tree::Node<int> R1 {"Riverbank 1", 0};
    tree::Node<int> R2 {"Riverbank 2", 0};

    tree::Node<int> * city[] = {&I1,&I2,&R1,&R2};
    /* For Edges, we use an open/close integer.  zero is open.  Once it is crossed
    ¦* we can close it setting it to one */

    tree::Edge<int> B1 {"Bridge 1", 0};
    tree::Edge<int> B2 {"Bridge 2", 0};
    tree::Edge<int> B3 {"Bridge 3", 0};
    tree::Edge<int> B4 {"Bridge 4", 0};
    tree::Edge<int> B5 {"Bridge 5", 0};
    tree::Edge<int> B6 {"Bridge 6", 0};
    tree::Edge<int> B7 {"Bridge 7", 0};

/* Creating all the predicates */
    tree::Predicate<int, int> crossing_1 {I1, I2, B1};
    tree::Predicate<int, int> crossing_2 {I1, R1, B2};
    tree::Predicate<int, int> crossing_3 {I1, R1, B3};
    tree::Predicate<int, int> crossing_4 {I2, R1, B4};
    tree::Predicate<int, int> crossing_5 {I1, R2, B5};
    tree::Predicate<int, int> crossing_6 {I1, R2, B6};
    tree::Predicate<int, int> crossing_7 {I2, R2, B7};

    std::cout << I1.get_name() << " " << I1.data << std::endl;
    std::cout << I2.get_name() << " " << I2.data << std::endl;
    std::cout << R1.get_name() << " " << R1.data << std::endl;
    std::cout << R2.get_name() << " " << R2.data << std::endl;
    std::cout << "Is it walkable ==> "  << is_euler(city) << std::endl;



    return EXIT_SUCCESS;
}
```

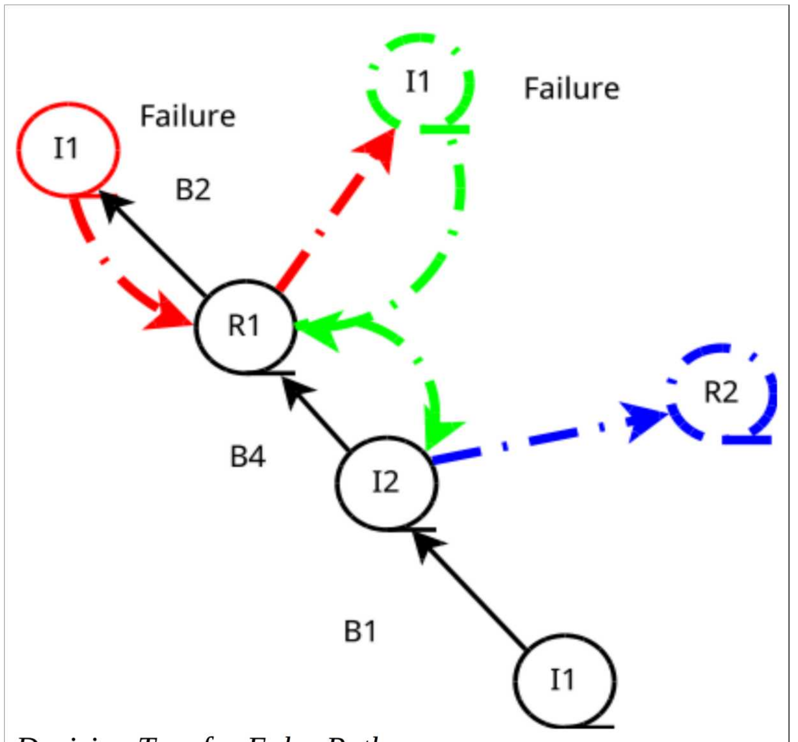
A few notes on this code are worth mentioning. Leveraging C++ templates is useful in the header files in order to lay down the ground work for future graphs. This is particularly useful for work in the biological sciences and paleontology specifically. Graphs that involve species, each species can be viewed as a state within the process of evolution are described in edges. The mechanisms that allow for transformation between species, that is the events that push evolution. The molecular biology involved with genetic problems can vary, depending on the   specifics of the problem being solved. There can be a great variety of the types of events that are being described in these graphs. Graphs can be laid out using parsimony, and within that space there can be different parsimony rules for different problems or different investigators. Further, we can evaluated our graph with statistical analysis. We will be looking at a means of developing such constructions. This specific code example, however, assumes that the data type for nodes are of a generic integer type, and would need to be further adapted, perhaps through C++ heredity, through leveraging the templates.

Understanding and Coding Decision Trees

The above code will swiftly tell you if any topographical map can be successfully traversed according the the Königsberg postulate. The next logical step is to evaluate how one might transverse the graph assuming that we have a solution, which in this case we don't. Several mechanism exist in the literature. The first step involves identification of an odd-degreed node. We can build that fact directly into our nodes so that every time we find a crossing that emulates from a node (each **crossing is a Predicate** in our graph with one edge and two nodes), it increments the count of crossed edges for that node. This count is an integer sum termed a ***degree*** stored within a node under this algorithm. Since we are storing degrees within our node code objects, we can query nodes directly, in a round robin fashion until we find an appropriate root (one with an odd numbered degree).

The next step finds all the predicates that include that node. We choose one, mark it as crossed, and then examine the associated second node within that crossing. Repeat with the associated node until either we run out of nodes, or when we reach a node with has no more associated uncrossed crossings. We can then examine all the predicates and if there are any with are any that are uncrossed, then we have a failure, and we need to back track and search the previous node for a different predicate. If there is none that hasn't been crossed, then we back up again, until we either find a successful path or reach our starting point with no further paths. If we exhaust all paths and none are successful, at that point, we know the graph has no solution.

This is a basic algorithm for walking through a decision tree. The tree is yet another type of a graph, and we can see its form in the figure captioned “Decision Tree for Euler Path”. Volumes of important papers have been written about means of optimizing such trees, searching them, reorganizing them, analyzing their properties and more. With each node the complexity of the problem grows, minimally, geometrically. Trees allow us to form relationships between states, and since they grow geometrically, we can store a large number of states and reach individual states quickly as we transverse through the tree. Each node in our case, can have two or more choices of crossings. For some problems, reaching an exact decision path might be beyond our ability to calculate within reasonable time. After all, the trees grow geometrically. Under those circumstances, heuristic methods then have to be introduced to bring approximate solutions.

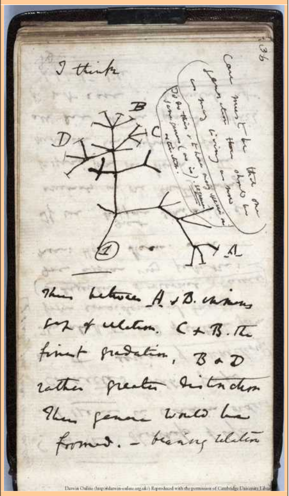


Decision Tree for Euler Path

With phylogeny we are focused on a particular kind of a tree, the tree of life. Current evolutionary theory is that life on earth began once, and all other life evolved from that single source. [3] It is indeed an astonishing hypothesis which to this point has been never been successfully challenged, although displacing these axioms have been tried by quacks and scientists for over 100 years. This describes a

3 Darwin, Charles, “The Origins of Species” , 24 November 1859

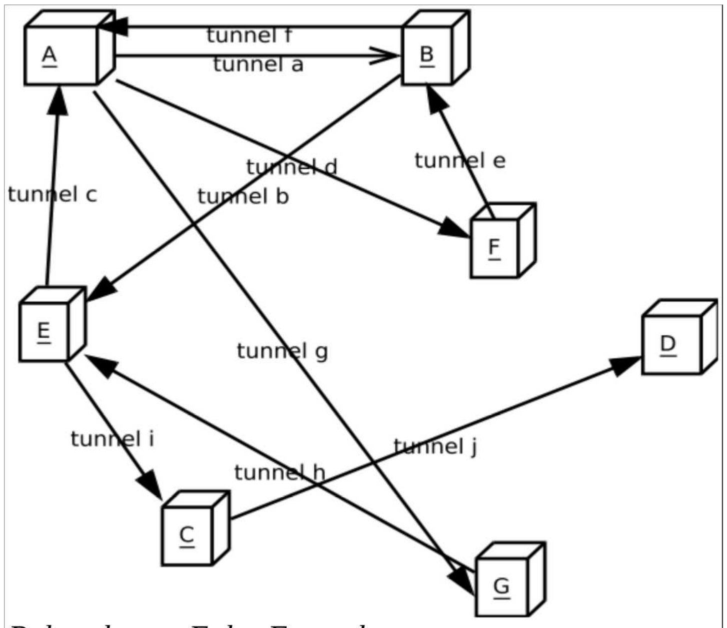
rooted tree. It’s updated version states that all of life is related, and the mechanism for these relations are primarily based on the sequence of DNA which records the instructions for the development, grown and function which is passed from one generation to the next. Over time, recording or transcription errors occur which is essential for the adaptation of species to their environment. This process has been under development for about 4 billion years. [4]


I think
A
B
C
D
Thus between A & B immense gap of relation. C & B the finest gradation, B & D rather greater distinction. Thus genera would be formed. — bearing relation

Darwin - 1836

Rarely will we examine the entire tree of life. The idea is almost absurd. The fossil record is just a crack in the window of time. If one thinks about the huge number of life forms on the planet right this moment, including millions of ever changing bacterial forms, fungi, alga, plants, insects, over 10,000 species of living dinosauria, and synapsid (Mammals and like forms), it should become clear that even the most well represented of species, say like Tyrannosaurus Rex, is just the smallest number of individuals within a species, of only a sliver of species from any given era or epoch.[5][6]

One needs to understand the nature of a decision tree in order to understand the complexity of evolutionary calculations. It is necessary to examine multiple possible trees and try to choose the most likely proper relationships between species. For a Euler's Königsberg problem, let’s examine a possible case where we can solve this problem. I name this imaginary city Rubensberg and it looks like the figure marked “Rubensberg” below.



Rubensburg - Euler Example

In this case, a causal look at the graph we can see that we have two odd numbered degreed nodes, Box A with five connectors and Box D with one. That means we can walk from one node to another, crossing every bridge once, starting with Node A or Node D. The complete graph can be represented in C++ like this (Nodes are capitalized and edges use small letters):

```
tree::Node<int> A {"Island A", 0};
tree::Node<int> B {"Island B", 0};
tree::Node<int> C {"Island C", 0};
tree::Node<int> D {"Island D", 0};
tree::Node<int> E {"Island E", 0};
tree::Node<int> F {"Island F", 0};
tree::Node<int> G {"Island G", 0};

tree::Node<int> * city[] = {&A, &B, &C, &D, &E, &F, &G};
```

4 Bard, Jonathan, “Principles of Evolution”, 2017, Garland Science

5 Allmon, Yaccobucci, editors, “The Fossil Record”, University of Chicago Press 2016

6 Brusatte, Stephen, Carr, Thomas, “The phylogeny and evolutionary history of tyrannosauroid dinosaurs”, Nature, Feb 2nd, 2016, online http://www.nature.com/articles/srep20252

```
/* For Edges, we use an open/close integer.  zero is open.  Once it is crossed
 * we can close it setting it to one */

tree::Edge<int> a {"Tunnel A", 0};
tree::Edge<int> b {"Tunnel B", 0};
tree::Edge<int> c {"Tunnel C", 0};
tree::Edge<int> d {"Tunnel D", 0};
tree::Edge<int> e {"Tunnel E", 0};
tree::Edge<int> f {"Tunnel F", 0};
tree::Edge<int> g {"Tunnel G", 0};
tree::Edge<int> h {"Tunnel H", 0};
tree::Edge<int> i {"Tunnel I", 0};
tree::Edge<int> j {"Tunnel J", 0};

/* Creating all the predicates */
tree::Predicate<int, int> crossing_1 {A,B,a};
tree::Predicate<int, int> crossing_5 {B,E,b};
tree::Predicate<int, int> crossing_10 {A,E,c};
tree::Predicate<int, int> crossing_4 {A,F,d};
tree::Predicate<int, int> crossing_3 {B,F,e};
tree::Predicate<int, int> crossing_2 {A,B,f};
tree::Predicate<int, int> crossing_6 {A,G,g};
tree::Predicate<int, int> crossing_7 {E,G,h};
tree::Predicate<int, int> crossing_8 {E,C,i};
tree::Predicate<int, int> crossing_9 {C,D,j};
tree::Predicate<int, int> * crossings[] = {&crossing_1 ,&crossing_2 ,&crossing_3
,&crossing_4 ,&crossing_5 ,&crossing_6 ,&crossing_7 ,&crossing_8 ,&crossing_9
,&crossing_10};
```

Our testing for the existence of a valid Euler path can use the same code as we had before. That leaves us with process of actually walking through all possible paths, until we find one that is valid. It is important to know that we can have more than a single valid path. If we start with one of the odd nodes, lest say A in this case, we can systematically test each attached path, one at a time, and then repeating the process until we either succeed or run out of paths. This lends itself to a stack and it can look like the following code. The problem solved either when we reach the last odd node, or a node that has no more valid paths.   How do we know that we finished and how do we know we have a correct result? First we check to see our final node is an odd node. How do we know if it is a valid result (which I term as a true result)? Well, if we count all the connections that we crossed, if the number of uniquely crossed connections are equal to the total number of connections, which in this case is 10 tunnels (or bridges, etc), then we have a winning route. If not, its a failure and we need to back track, and mark all the crossed connectors uncrossed on the way out. Here is an example code:

```
/*
 * =====================================================================================
 *
 *       Filename:  euler.cpp
 *
 *    Description:  Demonstration Program for Eulers Konisgburg problem
 *    using a path decision tree
 *
 *        Version:  1.0
 *        Created:  02/12/2017 10:21:23 AM
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Ruben Safir (mn), ruben@mrbrklyn.com
 *        Company:  NYLXS Inc
 *
 * =====================================================================================
 */

#include "nodes.h"
#include<vector>
#include<stack>
/*Create all the nodes */
/* data can be anything useful to describe a node.  In this case
 * it will be an int to represent degree (or number of connections */

tree::Node<int> A {"Island A", 0};
tree::Node<int> B {"Island B", 0};
tree::Node<int> C {"Island C", 0};
tree::Node<int> D {"Island D", 0};
tree::Node<int> E {"Island E", 0};
tree::Node<int> F {"Island F", 0};
tree::Node<int> G {"Island G", 0};

tree::Node<int> * city[] = {&A, &B, &C, &D, &E, &F, &G};
/* For Edges, we use an open/close integer.  zero is open.  Once it is crossed
 * we can close it setting it to one */

tree::Edge<int> a {"Tunnel A", 0};
tree::Edge<int> b {"Tunnel B", 0};
tree::Edge<int> c {"Tunnel C", 0};
tree::Edge<int> d {"Tunnel D", 0};
tree::Edge<int> e {"Tunnel E", 0};
tree::Edge<int> f {"Tunnel F", 0};
tree::Edge<int> g {"Tunnel G", 0};
tree::Edge<int> h {"Tunnel H", 0};
tree::Edge<int> i {"Tunnel I", 0};
tree::Edge<int> j {"Tunnel J", 0};

/* Creating all the predicates */
tree::Predicate<int, int> crossing_1 {A,B,a};
tree::Predicate<int, int> crossing_5 {B,E,b};
tree::Predicate<int, int> crossing_10 {A,E,c};
tree::Predicate<int, int> crossing_4 {A,F,d};
tree::Predicate<int, int> crossing_3 {B,F,e};
tree::Predicate<int, int> crossing_2 {A,B,f};
tree::Predicate<int, int> crossing_6 {A,G,g};
tree::Predicate<int, int> crossing_7 {E,G,h};
tree::Predicate<int, int> crossing_8 {E,C,i};
tree::Predicate<int, int> crossing_9 {C,D,j};
tree::Predicate<int, int> * crossings[] = {&crossing_1 ,&crossing_2 ,&crossing_3 ,&crossing_4
,&crossing_5 ,&crossing_6 ,&crossing_7 ,&crossing_8 ,&crossing_9 ,&crossing_10};

int tunnel_count = 0;

int is_euler(tree::Node<int> ** city )
{
   int odd = 0;
   int size = 7;
   for(int i=0;i<size;i++){
//    std::cout << "__LINE__ " << __LINE__ << std::endl;
      if( (city[i]->data % 2) == 1 )
      {
         odd++;
         std::cout << "Found odd number " << odd << std::endl;
      }
   }
   if(odd == 0 || odd == 2){
      return true;
   }
   return false;
}

bool testnode(tree::Node<int>* in, int isodd)
{
   if(tunnel_count == 10)
   {
      std::cout << "FINISHED Node==>" << in->get_name() << std::endl;
      return true;
   }
   //std::vector<tree::Predicate<int,int> *> attached;
   std::stack<tree::Predicate<int,int> *> attached;
      if(isodd == 1)
      {
         //start on the first crossing and find matching nodes from our selected odd node_path
         std::cout << "Starting Node==>" << in->get_name() << std::endl;

         for(int index = 0; index< 10; index++)
         {
            //find all the attached nodes and crossings
        if( (crossings[index]->search_for_matching_node(in) ) &&  (crossings[index]->c.data == 0) )
            {
               std::cout << "\tPushing =>" << crossings[index]->c.get_name() << std::endl;
               attached.push(crossings[index]);
            }
         }
         //have we reached an end?
         if(attached.size() == 0)
         {
            std::cout << "\tSize of Attached is =>" << attached.size() << "  ";
            std::cout << "\tTunnel Count is  =>" << tunnel_count << std::endl;
            if(tunnel_count == 10)
            {
               return true;
            }

         }
         //Search the attached crossings for a working path
         tree::Predicate<int,int> * tmp;
         while((! attached.empty() ) && ( tunnel_count != 10 ) )
         {
               //check all the paths
            std::cout << "\t\tCurrent Node==>" << in->get_name() << std::endl;
            tmp = attached.top();
            std::cout << "\t\tTraveling Path==>" << tmp->c.get_name() << std::endl;
            attached.pop();
            tunnel_count++;
            std::cout << "\t\tCurrent Tunnel Count is==>" << tunnel_count << std::endl;
            tmp->c.data = 1;
            //recursion
            if(testnode(tmp->next(in), isodd) && (tunnel_count != 10)  )
            {
               std::cout << __LINE__ << "\t\tCurrent Node==>" << in->get_name() << std::endl;
               std::cout << "\t\t\t\t***Returned TRUE***" << std::endl;
               std::cout << "\t\t\t\t\tSize of Attached is =>" << attached.size() << std::endl;
            }else{
               if(tunnel_count == 10)
               {
                  return true;
               }
               std::cout << __LINE__ << "\t\tCurrent Node==>" << in->get_name() << std::endl;
               std::cout << "\t\t\t\t** returned FALSE **" << std::endl;
               tmp->c.data = 0;
               std::cout << "\t\t\t\t\t\t** tunnel marked unvisited **" << std::endl;
               tunnel_count--;
               std::cout << "\t\t\t\t\tCurrent Tunnel Count is ==>"  << tunnel_count <<  std::endl;

            std::cout << "\t\t\t\t\tNumber of attached paths is=>" << attached.size() << std::endl;
            }
            std::cout << "check while condition again do we have attached tunnels**" << std::endl;
         }
         std::cout << "\tOut of while loop : Size of Attached is =>" << attached.size() << " ";
         std::cout << "For Node " << in->get_name() << std::endl;
//       std::cout << __LINE__  << std::endl;

      }

   return false;
}

int main(int argc, char * argv[])
```

```
{
   std::cout << A.get_name() << " " << A.data << std::endl;
   std::cout << B.get_name() << " " << B.data << std::endl;
   std::cout << C.get_name() << " " << C.data << std::endl;
   std::cout << D.get_name() << " " << D.data << std::endl;
   std::cout << E.get_name() << " " << E.data << std::endl;
   std::cout << F.get_name() << " " << F.data << std::endl;
   std::cout << G.get_name() << " " << G.data << std::endl;
   std::cout << "Is it walkable ==> "  << is_euler(city) << std::endl;

//Find an odd degree nodes
   int cntr = 0;
   int isodd =0;
// tree::Node<int> * cur;
   tree::Node<int> * start;
   std::stack < tree::Node<int> * > node_path;
   //store the attached crossings for future reference
   while( ( (city[cntr]->data % 2) == 0 ) && cntr < 7 )
   {
      cntr++;
   }
   if(cntr == 7 )
   {
      start = city[0];
   }else{
      start = city[cntr];
      isodd = 1;
   }

   node_path.push(start);
   std::cout << "TOP Node =>" << node_path.top()->get_name() << std::endl;
   testnode(start, isodd);

   return EXIT_SUCCESS;
}
```

```
/*
 * =====================================================================================
 *
 *       Filename:  nodes.h
 *
 *    Description:  Basic Node for Euler Graph
 *
 *        Version:  1.0
 *        Created:  02/11/2017 06:14:42 PM
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Ruben Safir (mn), ruben@mrbrklyn.com
 *        Company:  NYLXS Inc - LIU Thesis
 *
 * =====================================================================================
 */

#ifndef NODES_H
#define NODES_H
#include<iostream>
#include<string>
#include<vector>
namespace tree {

   /*
 * =====================================================================================
 *        Class:  Node
 *  Description:  This class describes a single node and its data
 * =====================================================================================
 */
template < class T >
   class Node
   {
      public:

         /* ====================  LIFECYCLE     ======================================= */
         Node (std::string n, T d): data{d}, name{n} {

         };                             /* constructor */

         /* ====================  ACCESSORS     ======================================= */
         std::string get_name()const{
            return name;
         }

         bool operator==( const Node first) const{
            if(first.get_name() == name)
            {
               return true;
            }
            return false;
         };
         bool operator==(Node *first) const {
            if(first->get_name() == name)
            {
               return true;
            }
            return false;
         };

         /* ====================  MUTATORS      ======================================= */
         /* ====================  OPERATORS     ======================================= */
         /* ====================  DATA MEMBERS  ======================================= */
         T data;
      protected:

      private:
         std::string name;

   }; /* ----------  end of template class Node  ---------- */

/*
 * =====================================================================================
 *        Class:  Edges
 *  Description:  Bridges - usually I wouldn't make edges but we are being consistant
 *  with the Luger description of the problem.
 * =====================================================================================
 */
template < class T >
   class Edge
   {
      public:

         /* ====================  LIFECYCLE     ======================================= */
         Edge (std::string n, T d): data{d}, name{n}{};                             /*
constructor */

         /* ====================  ACCESSORS     ======================================= */
         std::string get_name(){
            return name;
         }

         /* ====================  MUTATORS      ======================================= */

         /* ====================  OPERATORS     ======================================= */

         /* ====================  DATA MEMBERS  ======================================= */
         T data;
      protected:

      private:
            std::string name;

      }; /* ----------  end of template class Edges  ---------- */
   template< class node_type, class bridge_type>
   class Predicate{
      public:
         Predicate(Node<node_type>& in_a, Node<node_type>& in_b, Edge<bridge_type>& in_c)
            : a{in_a}, b{in_b}, c{in_c}{
               a.data++;
               b.data++;
            };
         Node<node_type>& a;
         Node<node_type>& b;
         Edge<bridge_type>& c;

         //== is true if we have a single matching node
         bool operator==(const Predicate *first) const
         {
            if( (first->a == a) || (first->a == b)|| (first->b == a)
                  ||(first->b == b) )
            {
               return true;
            }
            return false;
         };
         bool operator==(const Node<node_type> * first) const
         {
            if( ( *first == a) || (*first == b)|| (*first == a)
                  ||(*first == b) )
            {
               return true;
            }
            return false;
         };

         bool search_for_matching_node(const Node<node_type> * first) const
         {
            if( (*first == a)|| (*first==b) || (*first == a) || (*first == b ) )
            {
               return true;
            }
            return false;
         };

         Node<node_type> * next(const Node<node_type> * first)
         {
            if(*first == a){
               return &b;
            }
            return &a;
         }
   };

}//close of namespace tree

#endif
```

The output traces the entire decision making tree:

```
Island A 5
Island B 4
Island C 2
Island D 1
Island E 4
Island F 2
Island G 2
Found odd number 1
Found odd number 2
```

```
Is it walkable ==> 1
TOP Node =>Island A
Starting Node==>Island A
       Pushing =>Tunnel A
       Pushing =>Tunnel F
       Pushing =>Tunnel D
       Pushing =>Tunnel G
       Pushing =>Tunnel C
              Current Node==>Island A
              Traveling Path==>Tunnel C
              Current Tunnel Count is==>1
Starting Node==>Island E
       Pushing =>Tunnel B
       Pushing =>Tunnel H
       Pushing =>Tunnel I
              Current Node==>Island E
              Traveling Path==>Tunnel I
              Current Tunnel Count is==>2
Starting Node==>Island C
       Pushing =>Tunnel J
              Current Node==>Island C
              Traveling Path==>Tunnel J
              Current Tunnel Count is==>3
Starting Node==>Island D
       Size of Attached is =>0      Tunnel Count is  =>3
       Out of while loop : Size of Attached is =>0 For Node Island D
140           Current Node==>Island C
                            ** returned FALSE **
                                   ** tunnel marked unvisited **
                                   Current Tunnel Count is ==>2
                                   Number of attached paths is=>0
check while condition again do we have attached tunnels**
       Out of while loop : Size of Attached is =>0 For Node Island C
140           Current Node==>Island E
                            ** returned FALSE **
                                   ** tunnel marked unvisited **
                                   Current Tunnel Count is ==>1
                                   Number of attached paths is=>2
check while condition again do we have attached tunnels**
              Current Node==>Island E
              Traveling Path==>Tunnel H
              Current Tunnel Count is==>2
Starting Node==>Island G
       Pushing =>Tunnel G
              Current Node==>Island G
              Traveling Path==>Tunnel G
              Current Tunnel Count is==>3
Starting Node==>Island A
       Pushing =>Tunnel A
       Pushing =>Tunnel F
       Pushing =>Tunnel D
              Current Node==>Island A
              Traveling Path==>Tunnel D
              Current Tunnel Count is==>4
Starting Node==>Island F
       Pushing =>Tunnel E
              Current Node==>Island F
              Traveling Path==>Tunnel E
              Current Tunnel Count is==>5
Starting Node==>Island B
       Pushing =>Tunnel A
       Pushing =>Tunnel F
       Pushing =>Tunnel B
              Current Node==>Island B
              Traveling Path==>Tunnel B
              Current Tunnel Count is==>6
Starting Node==>Island E
       Pushing =>Tunnel I
              Current Node==>Island E
              Traveling Path==>Tunnel I
              Current Tunnel Count is==>7
Starting Node==>Island C
       Pushing =>Tunnel J
              Current Node==>Island C
              Traveling Path==>Tunnel J
              Current Tunnel Count is==>8
```

```
Starting Node==>Island D
       Size of Attached is =>0      Tunnel Count is  =>8
       Out of while loop : Size of Attached is =>0 For Node Island D
140           Current Node==>Island C
                             ** returned FALSE **
                                    ** tunnel marked unvisited **
                                    Current Tunnel Count is ==>7
                                    Number of attached paths is=>0
check while condition again do we have attached tunnels**
       Out of while loop : Size of Attached is =>0 For Node Island C
140           Current Node==>Island E
                             ** returned FALSE **
                                    ** tunnel marked unvisited **
                                    Current Tunnel Count is ==>6
                                    Number of attached paths is=>0
check while condition again do we have attached tunnels**
       Out of while loop : Size of Attached is =>0 For Node Island E
140           Current Node==>Island B
                             ** returned FALSE **
                                    ** tunnel marked unvisited **
                                    Current Tunnel Count is ==>5
                                    Number of attached paths is=>2
check while condition again do we have attached tunnels**
              Current Node==>Island B
              Traveling Path==>Tunnel F
              Current Tunnel Count is==>6
Starting Node==>Island A
       Pushing =>Tunnel A
              Current Node==>Island A
              Traveling Path==>Tunnel A
              Current Tunnel Count is==>7
Starting Node==>Island B
       Pushing =>Tunnel B
              Current Node==>Island B
              Traveling Path==>Tunnel B
              Current Tunnel Count is==>8
Starting Node==>Island E
       Pushing =>Tunnel I
              Current Node==>Island E
              Traveling Path==>Tunnel I
              Current Tunnel Count is==>9
Starting Node==>Island C
       Pushing =>Tunnel J
              Current Node==>Island C
              Traveling Path==>Tunnel J
              Current Tunnel Count is==>10
FINISHED Node==>Island D
```

Understanding this basic algorithm is essential for understanding the bulk of the computational theory on phylogeny and evolutionary computations. We are using multiple levels of comparisons, whether it is genetic sequences or cladistic phenotypes. We need to examine multiple possible trees and apply order to them to determine the ones that have the highest likelihood of representing the actual historical progression of species.

Dikstra's Shortest Path Algorithm and Introduction to Costs

In basic graph theory, unlike in our example, crossings, or what is generally term as edges, can have costs. An edge between two nodes can have costs, and the costs between different nodes have various numeric representations, which can determine any abstraction, for example, distance, resistance, likelihood for an event to happen, the number of hops in a network, or anything that can describe the difference between two states. Each state is a node and between nodes are edges that have costs. Trying to determine the shortest route between states or a group of states is an area of study in computational math. The classic example which paleontologists need to be familiar with is the Dijkstra algorithm. Dijkstra's Algorithm is classically described in the seminal text, Cormen, et al, "Introduction of Algorithms" which in the second edition is included in Chapter 24.3. Quoting Cormen[7][8]

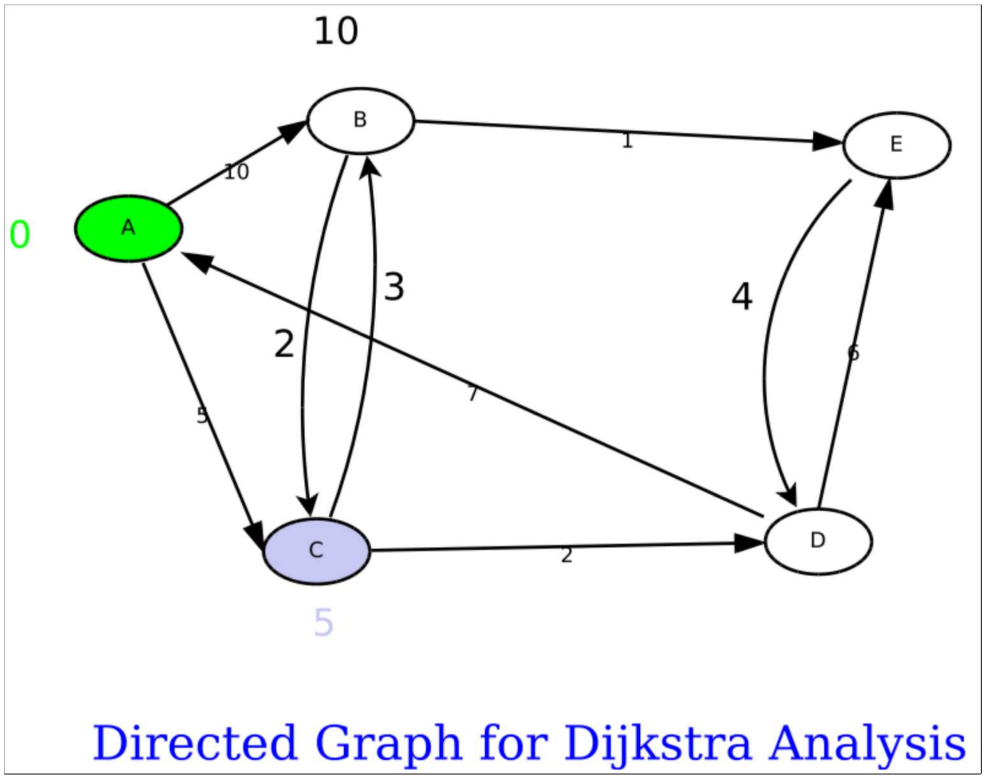
> Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G = (V, E) for the case in which all edge weights are nonnegative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.
>
> Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined. That is, for all vertices v in S, we have d[v] = (s, v).

7 Cormen, Leiserson, Rivest, Stein "Introduction to Algorithms" Second Edition, The MIT Press, 2001

8 Dijkstra, E.W., "A Note on Two Problems in Connexion with Graph", Numerische Mathematatlk 1, pg269 - 27, 1959

> The algorithm repeatedly selects the vertex u $\in$ V - S with the minimum shortest-path estimate, inserts u into S, and relaxes all edges leaving u. [9]

What this means is that we have a directed graph, such as this one based on Cormen:[10]



A couple of things should be noted about this chart. Unlike the previous one, the arrows mean that the relationship between those two vertexes are directed, and one can only travel one way. This kind of constraint is useful for paleontology and species don't presume to devolve, although features can at times be lost. Even whales that lose legs still have remnants.

In the Dijkstra algorithm we chose a starting point, which in this case is vertex A. We divide the graphs total number of vertexes between vertexes that are visited and vertexes that are unvisited. **We call the set of visited vertexes as** S and t**he unvisited are V - S where V is the complete set of vertexes** for graph G.

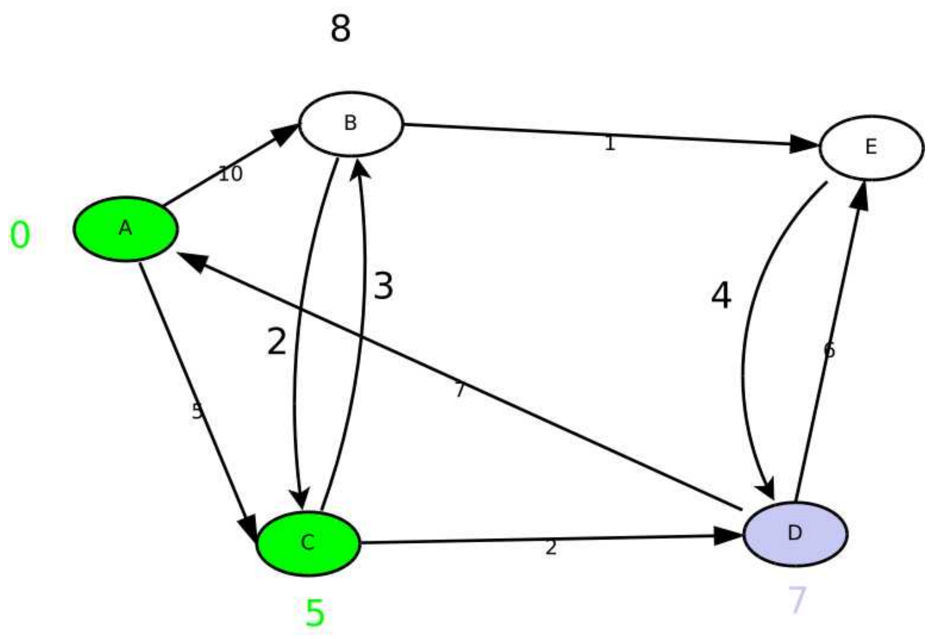
S is the Set of Visited Vertexes
V is all the Vertexes in a graph
V-S is unvisited Vertexes

We examine all the vertexes that are attached to S (Just vertex A at the start) and compare all the costs. Each examined vertex is marked with the cost from the current (or starting) position to the attached

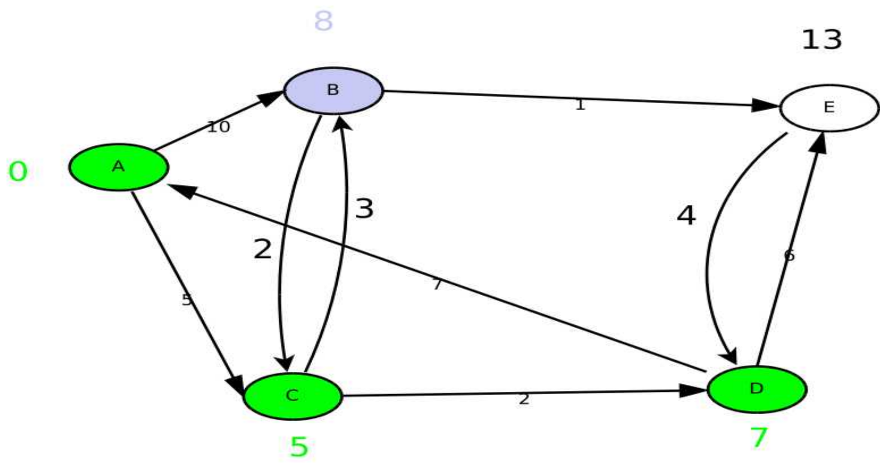
9 Cormen, Leiserson, Rivest, Stein "Introduction to Algorithms" Second Edition, The MIT Press, 2001
10 Ibid.

vertex. We then look for the attached vertex with the smallest cost, and chose that as our next “current’ vertex. So in this case, we look at vertex B and C. B is marked 10 and C is marked 5, and C is now our *current vertex*.

Now we repeat with C as out start. Attached to C is B (which was marked previously with a value of 10), and D. B is remarked to 8. Why? Because C is valued currently at 5, and the cost from C to B is 3. 5 and 3 is 8. 8 is less than the current value of 10, so we mark it down to 8. This implies that the cheapest way to B is through C. We now mark D with 7 (5 plus the cost of 2 between C and D). Our lowest attached point is now D at 7. D is now marked as current.

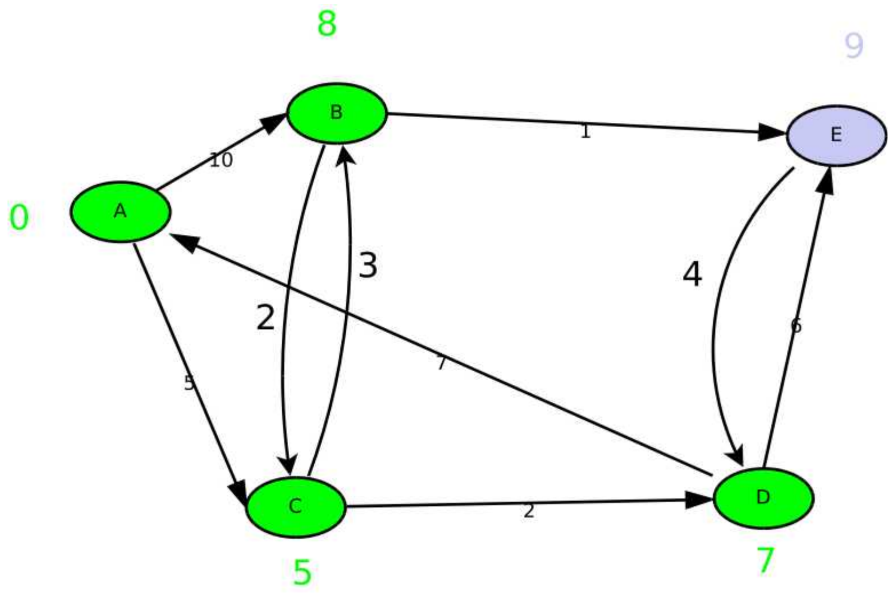


Repeat from D. D is attached to E and A. A however is part of S (it has been visited and marked *green* in our graph) and we don’t revisit. S is now a set of {A, C} and our current vertex, D being evaluated. E is evaluated to 13. B is still evaluated at 8 and is now the minimum valued node and *is marked current*. **D is added to the set of V.**



Repeat for B which has a value of 8. It is attached to the last free vertex, E which is currently valued at 13. The cost from B to E is 1, and since B is 8, we get 8 (which is B) + 1 (which is the cost from B→E) we get a new minimum cost for E of 9. B is included in the Set of V and we now repeat with E

E is the last node and remains at its value of 9.



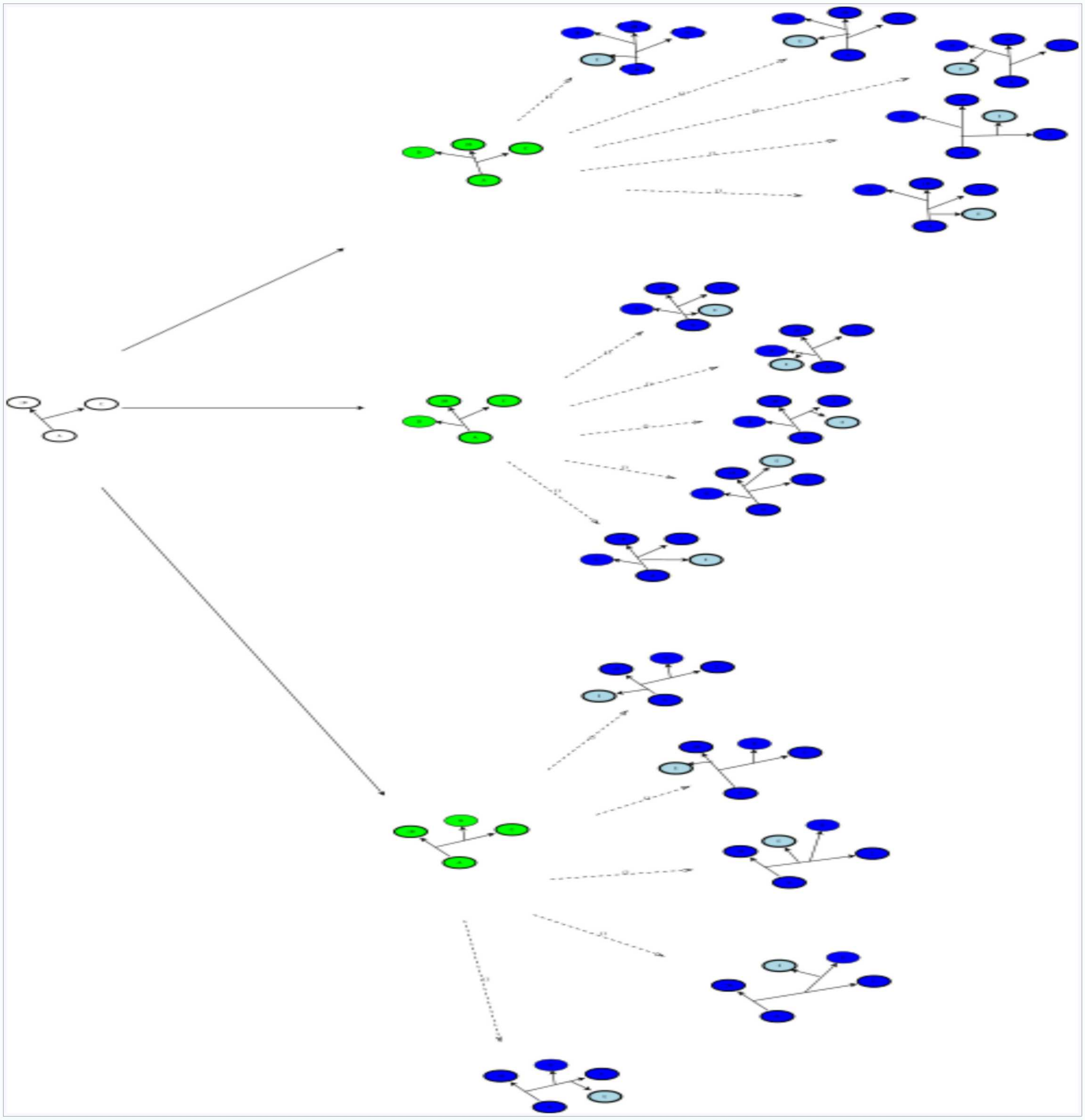
In theory, if we have proper cost analysis, and direction. We should be able to create a decision tree through this algorithm that would produce the most correct cladogram between any number of species. Unfortunately, things are never that simple.

Determining a proper cost is the key to using Dijkstra. Lets assume, for one useful example that we have 10 species that we know are closely related, and we want to try to determine a most likely evolutionary tree of said species. In this case, we can build a **finite number of trees which represent the complete set of possible relationships**, assuming that we can have at most 2 species branching from any other single species. We can find the most likely tree if we build the tree, starting by evaluating just two species, and then adding a third. When we add a fourth species, we can now create 3 possible trees of 4 vertexes each. Each tree is related to the previous 3 node tree. We can determine the cost from the 3 vertex tree to any of the three 4-vertex trees and apply Dijkstra's algorithm to the graph. We can then add a 5th vertex, we can then create 6 more trees for each of the 3 previous cases, and each with a cost in relationship to their parent. We can then apply Dijkstra again and determine the shortest path according to latest diagram and data. And we can do so again for as many nodes that we want, determining the shortest path for any x where N(x) is the designation for any vertex where the the number of vertexes in the graph is the set of all $x \rightarrow \{x\}$ where x is an integer from $0 .. x_{max}$.

Defining the cost determines the nature of the graph, and the evolutionary information that we can extract from that data. One recent lectures from the Royal Tyrrell Museum in Drumheller, Alberta in the 2017 lecture series which are available on their youtube channel evaluated just tooth morphology. [11] Those can be quantified and the degree of difference from species to another can be assigned points based on morphology criteria. The differences between teeth from species to species is a subtraction of these costs. Assigning them stepwise to trees can give a grand total of parsimony for the teeth for an entire tree at each level. We can then compare those summations from proposed tree to proposed tree and find the shortest path between and two trees, from 2 vertex trees to 10 vertex trees (since this is a 10 species problem). A shortest path is now the most likely correct tree and evolutionary path based on our information.

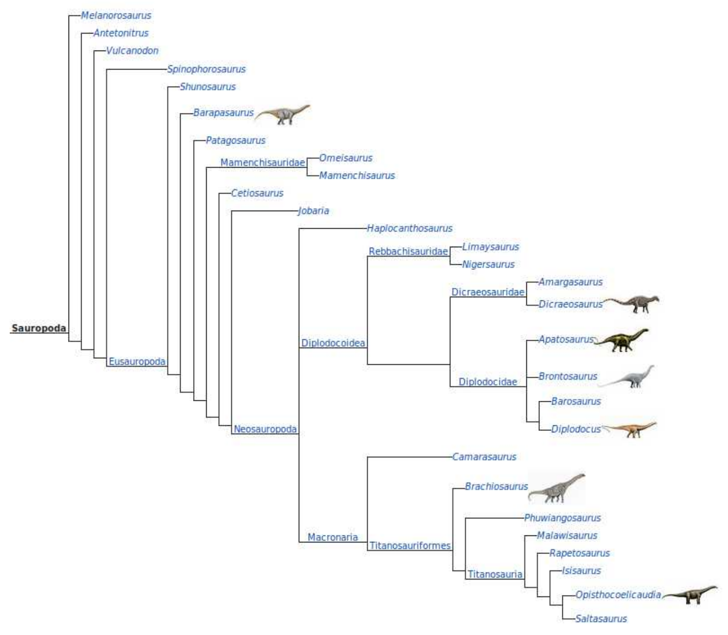
As we can see, the number of trees that can serve as vertexes rises rapidly as each individual tree in our graph increases in the number of species. This is worth looking at for a moment because of its consequences to computational complexity and because it allows us to clarify some of the assumptions we are making about evolutionary tree development which differs from classical tree graphs. Lets assume for this example that we only have binary trees. When we add a node to our graph we are adding it to an edge that we select, forming a new leaf, and internal unlabeled node. The common evolutionary point of view says that species X splits and evolved into two new species, Y and Z. But having such transitional forms in hand is like catching lightning in a bottle. That is why instead, when we are adding new species to a tree, we add them to the edges, forming an unknown transitional node which is unnamed, by splitting an existing edge, and extending a new edge from the transitional node to our added species. This creates a large diagram which is on the following page

11 Derek Larson, “Morphology, Disparity, and Evolution of Theropod Teeth in the Late Cretaceous in Alberta”, The Royal Tyrrell Museum Speaker Series, March 1, 2017, https://www.youtube.com/watch?v=SJh0Izvw2DM


Example of Stepwise Addition for preparation of Dijkstra's Algorithm

This is the model that has been developed and is largely explored in the literature. Like any other scientific model, it has limitations. It is possible, after all, that a species we are looking at is a transitional form. Not all species are evolutionary dead ends. To make matters even more difficult, there is a lack of consensus in the literature as to what standards are to being used to describe trees and graphs. Sometimes the shapes of trees are to be considered, and sometimes they are not. As computer scientists and mathematicians, our goal is often to simplify descriptions to the least restrictive generalization in order to apply logical abstractions. But for evolutionary computations, one has to read carefully the currently proposed tree structure, and not take for granted terminology without seeing the formal definition. There is no consensus on terminology within these papers or even textbooks.

For example, one might study rooted bifulcrum trees, unrooted bifulcrum trees, and trees that are unrooted but can be rooted by any given species on the tree, and others with an unlabeled species at the root, and even trees that represent time in the length of edges. There are papers and theories that are proposed that handle all these kinds of cases. **This adds to the confusion when trying to gather the known work on phylogenies into a comprehensive whole body of knowledge.** It gets harder to connect the dots. Most commonly, when one sees a graph within articles on paleontology, unlike the ones on computations, we see a chart where the nodes nearly disappear, such as this sample from Wikipedia: [12]



Wikipedia sample evolutionary diagram

That format is particularly useful because it is easy to render in HTML tables. Unfortunately, it takes even more deep thought to translate from the graph models in theory to the visuals used in descriptions of results. We will discuss this further in the upcoming section. Also, keep in mind that we have not discussed how costs are determined. This is the bread and butter of phylogenies work, but is not addressed at all with Dijkstra's Algorithm.

We can code Dijksta's Algorithm in C++. Examples exist in the literature and around the internet. [13] A fair example is this code which was borrowed from Stack Overflow:[14]

12 https://en.wikipedia.org/wiki/Sauropodomorpha March 2017
13 Robert Sedgewick https://www.cs.princeton.edu/~rs/AlgsDS07/15ShortestPaths.pdf
14 https://stackoverflow.com/questions/3447566/dijkstras-algorithm-in-c/3448361

```
working.addNode(Start, 0); // No cost to get to start.


for( (node, cost) = working.popHead(); node != End; (node,cost) = working.popHead())
{
    // If we have already processed this node ignore it.
    if (finished.find(node))
    {    continue;
    }

    // We have just removed a node from working.
    // Because it is the top of the list it is guaranteed to be the shortest route to
    // the node. If there is another route to the node it must go through one of the
    // other nodes in the working list which means the cost to reach it will be higher
    // (because working is sorted). Thus we have found the shortest route to the node.

    // As we have found the shortest route to the node save it in finished.
    finished.addNode(node,cost);

    // For each arc leading from this node we need to find where we can get to.
    foreach(arc in node.arcs())
    {
        dest = arc.dest();
        if (NOT (finished.find(dest)))
        {
            // If the node is already in finished then we don't need to worry about it
            // as that will be the shortest route other wise calculate the cost and add
            // this new node to the working list.
            destCost = arc.cost() + cost;
            working.addNode(dest,destCost); // Note. Working is sorted list
        }
    }
}
```

or this working example offered under the Creative Commons on www.geeksforgeeks.org by an seemingly anonymous author: [15]

```
// A C / C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph

#include <stdio.h>
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
// Initialize min value
int min = INT_MAX, min_index;

for (int v = 0; v < V; v++)
       if (sptSet[v] == false && dist[v] <= min)
               min = dist[v], min_index = v;

return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
```

15 http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

```
printf("Vertex Distance from Source\n");
for (int i = 0; i < V; i++)
       printf("%d \t\t %d\n", i, dist[i]);
}

// Funtion that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
       int dist[V];    // The output array. dist[i] will hold the shortest
                                    // distance from src to i

       bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
                                    // path tree or shortest distance from src to i is finalized

       // Initialize all distances as INFINITE and stpSet[] as false
       for (int i = 0; i < V; i++)
              dist[i] = INT_MAX, sptSet[i] = false;

       // Distance of source vertex from itself is always 0
       dist[src] = 0;

       // Find shortest path for all vertices
       for (int count = 0; count < V-1; count++)
       {
       // Pick the minimum distance vertex from the set of vertices not
       // yet processed. u is always equal to src in first iteration.
       int u = minDistance(dist, sptSet);

       // Mark the picked vertex as processed
       sptSet[u] = true;

       // Update dist value of the adjacent vertices of the picked vertex.
       for (int v = 0; v < V; v++)

              // Update dist[v] only if is not in sptSet, there is an edge from
              // u to v, and total weight of path from src to v through u is
              // smaller than current value of dist[v]
              if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                                                                     && dist[u]+graph[u][v] < dist[v])
                     dist[v] = dist[u] + graph[u][v];
       }

       // print the constructed distance array
       printSolution(dist, V);
}

// driver program to test above function
int main()
{
/* Let us create the example graph discussed above */
int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                                    {4, 0, 8, 0, 0, 0, 0, 11, 0},
                                    {0, 8, 0, 7, 0, 4, 0, 0, 2},
                                    {0, 0, 7, 0, 9, 14, 0, 0, 0},
                                    {0, 0, 0, 9, 0, 10, 0, 0, 0},
                                    {0, 0, 4, 14, 10, 0, 2, 0, 0},
                                    {0, 0, 0, 0, 0, 2, 0, 1, 6},
                                    {8, 11, 0, 0, 0, 0, 1, 0, 7},
                                    {0, 0, 2, 0, 0, 0, 6, 7, 0}
                                    };
       dijkstra(graph, 0);

       return 0;
}
```

Calculating Costs

While we use some form of the shortest path algorithm, such as Dijkstra’s algorithm for use to find the best trees or paths for phylogenies, the increase in the number of permutations that could be vertexes would challenge the computational power of even today's powerful multi-core workstations. It becomes undoable to solve. Threaded coding technique in the application of algorithms can relieve some of this computational burden but ultimately this is a practical constraint forcing paleontologists to focus on statistical analysis. When you consider how much uncertainty is in the fossil record to begin with, statistical models are essential. In the end though, such analysis are still a cost factors that can be plugged in the Dijkstra’s algorithm. And that is why I believe that the lack of broader education on basic graph theory is detrimental to the Paleontology field.

There is a general concept that the simplest means of explaining something is the most correct. This is termed ***parsimony***. In paleontology parsimony is defined as the evolutionary tree that requires the least about of evolution. As early as 1963 Edwards and Cavalli-Sforza wrote: [16]

> In our isotropic space we may think of evolution as a branching random walk, with a constant probability of branching and a constant rate of walking: that is, after the elapse of a certain time interval, the probability distribution in space of the position of a population will be normal, with mean at its original position and variance proportional to the time elapsed. The constant of proportionality will depend on many parameters, such as the population size and the type and intensity of selection, and will usually be unknown. Since an evolutionary tree uniting n points (without loops) is bound to contain n - 1 branching points, the probability of branching is not a parameter which has to be estimated, although it is worth noting that the theory of birth and death processes may have some interesting things to say about the expected form of an evolutionary tree.
>
> Now the probability density at a distance d spatially and t temporally from a point in the space-time of p spatial dimensions is given by

16 Edwards, A. W. F., and Cavallli-Sforza, L. L., 1963a. The reconstruction of evolution. Unpublished paper read at 142nd meeting of the Genetical Socie(Y of Great Britain, London, July 1963. (Abstract in Anlt. hum. Genet., 27: 104-105, and in Heredity, Land., 18: 553.)

$$\sigma^{-p}\ (2\pi t)^{-\frac{1}{2}p} \exp\{-d^2/(2t\sigma^2)\}$$

where σ^2 is the constant of proportionality mentioned above. The log-likelihood is therefore given by

$$-\{d^2/(2t\sigma^2) + \tfrac{1}{2}p \log (2t\sigma^2) + \tfrac{1}{2}p \log\pi\}$$

Writing T for $2t\sigma^2$, omitting the constant, and changing the sign, the expression becomes

$$d^2/T + \tfrac{1}{2}p \log T$$

Each arm of a postulated evolutionary tree will have an expression of this type associated with it, d being its spatial length and T proportional to its temporal length. **Thus the likelihood of the tree will be maximized if the sum of these expressions, over all the arms, is minimized (since we have changed the sign, and the branching points, being constant in number, are irrelevant).**[17]

If we want to determine a proposed cost between any pair of trees, then we need to assign some value to the nodes. Since we are looking at trees, one proposed and most basic means of evaluations is to review minimum tree parsimony for a specifically proposed tree. Since we will evaluate all the possible trees for any specific number of species, then we can assign all of them parsimony values and run Dijkstra's algorithm.

A decent example of the method which for determining a parsimony value is included in standard texts, including Felsenstein [18]who works with a small set of characteristics in a binary comparison chart

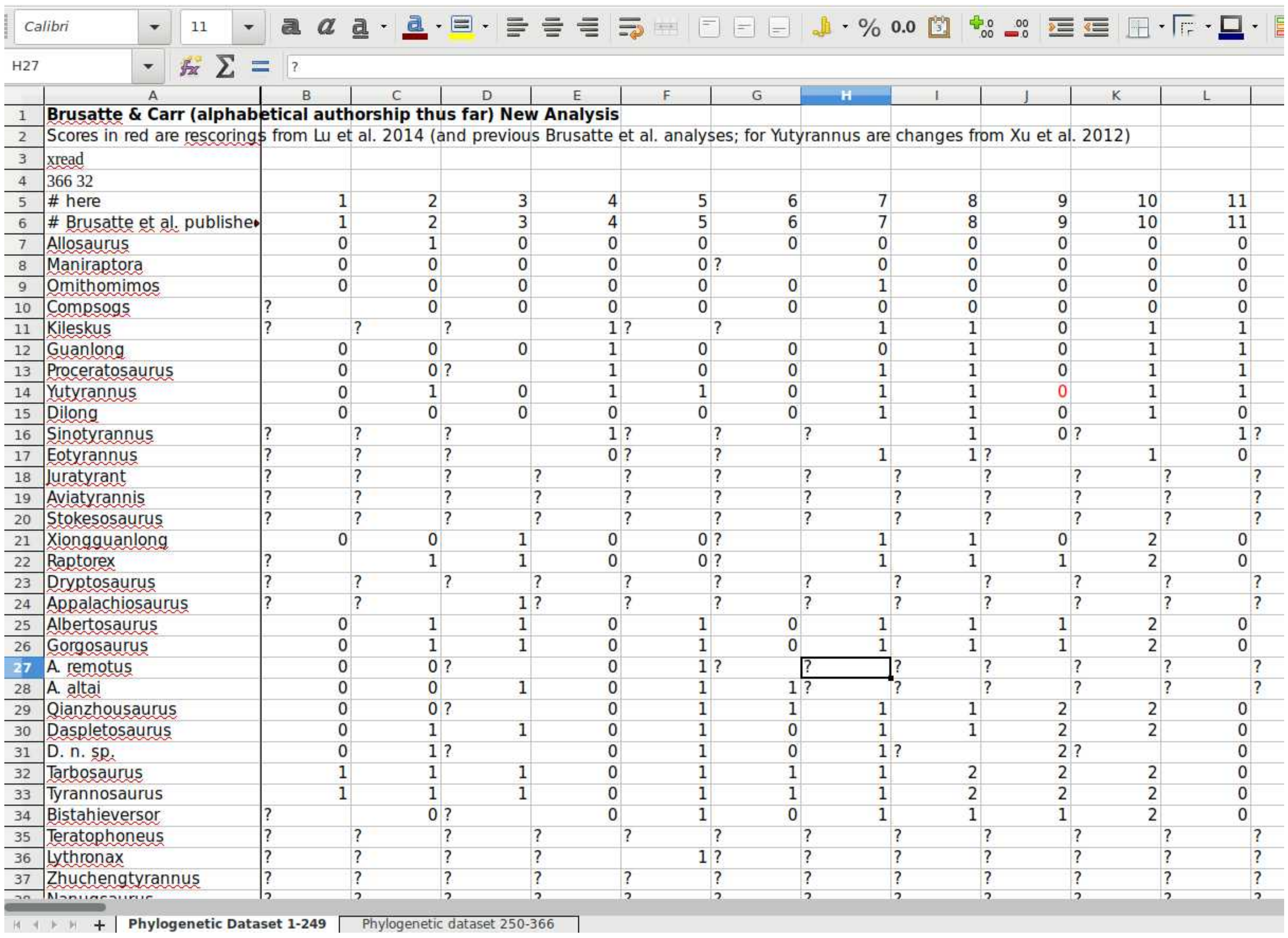
Characteristics

Species	**1**	**2**	**3**	**4**	**5**	**6**
Alef	1	0	0	1	1	0
Bet	0	0	1	0	0	0
Gimel	1	1	0	0	0	0
Delad	1	1	0	1	1	1
Hey	0	0	1	1	1	0

This is a simple chart of 5 species with a mapping of 6 characteristics which are selected by the wisdom of the modeler and investigator. Now, it is worth pointing out that this is where errors are produced in understanding techniques. Mathematical models and algorithms are foolproof in that they

17 A. W. F. Edwards & L. L. Cavalli-Sforza, December 1964, " Reconstruction of Evolutionary Trees" SYSTEMATICS ASSOCIATION PUBLICATION NUMBER 6 PHENETIC AND PHYLOGENETIC CLASSIFICATION, pp. 67-76
18 Felsenstein, 2004, "Inferring Phylogenies" , Sinauer Associates Inc: pg2

are based on solid formal logic, which can be reduced to computational notation and computer programming. The selection of data, on the other hand, and the appropriate use of models is easy to error with. This case uses a simple binary characteristics. In the real world, we can easily have a much more complex set of data. This, for example, is a ***view*** of the set of data used by Stephen L. Brusatte and Thomas Carr, on their analysis of Tyrannosaurs with 366 characteristics across 32 species:



H27 ?

	A	B	C	D	E	F	G	H	I	J	K	L	
1	**Brusatte & Carr (alphabetical authorship thus far) New Analysis**												
2	Scores in red are rescorings from Lu et al. 2014 (and previous Brusatte et al. analyses; for Yutyrannus are changes from Xu et al. 2012)												
3	xread												
4	366 32												
5	# here	1	2	3	4	5	6	7	8	9	10	11	
6	# Brusatte et al. publishe	1	2	3	4	5	6	7	8	9	10	11	
7	Allosaurus	0	1	0	0	0	0	0	0	0	0	0	
8	Maniraptora	0	0	0	0	0	?	0	0	0	0	0	
9	Omithomimos	0	0	0	0	0	0	1	0	0	0	0	
10	Compsogs	?	0	0	0	0	0	0	0	0	0	0	
11	Kileskus	?	?	?	1	?	?	1	1	0	1	1	
12	Guanlong	0	0	0	1	0	0	0	1	0	1	1	
13	Proceratosaurus	0	0	?	1	0	0	1	1	0	1	1	
14	Yutyrannus	0	1	0	1	1	0	1	1	0	1	1	
15	Dilong	0	0	0	0	0	0	1	1	0	1	0	
16	Sinotyrannus	?	?	?	1	?	?	?	1	0	?	1	?
17	Eotyrannus	?	?	?	0	?	?	1	1	?	1	0	
18	Juratyrant	?	?	?	?	?	?	?	?	?	?	?	?
19	Aviatyrannis	?	?	?	?	?	?	?	?	?	?	?	?
20	Stokesosaurus	?	?	?	?	?	?	?	?	?	?	?	?
21	Xiongguanlong	0	0	1	0	0	?	1	1	0	2	0	
22	Raptorex	?	1	1	0	0	?	1	1	1	2	0	
23	Dryptosaurus	?	?	?	?	?	?	?	?	?	?	?	?
24	Appalachiosaurus	?	?	1	?	?	?	?	?	?	?	?	?
25	Albertosaurus	0	1	1	0	1	0	1	1	1	2	0	
26	Gorgosaurus	0	1	1	0	1	0	1	1	1	2	0	
27	A. remotus	0	0	?	0	1	?	?	?	?	?	?	?
28	A. altai	0	0	1	0	1	1	?	?	?	?	?	?
29	Qianzhousaurus	0	0	?	0	1	1	1	1	2	2	0	
30	Daspletosaurus	0	1	1	0	1	0	1	1	2	2	0	
31	D. n. sp.	0	1	?	0	1	0	1	?	2	?	0	
32	Tarbosaurus	1	1	1	0	1	1	1	2	2	2	0	
33	Tyrannosaurus	1	1	1	0	1	1	1	2	2	2	0	
34	Bistahieversor	?	0	?	0	1	0	1	1	1	2	0	
35	Teratophoneus	?	?	?	?	?	?	?	?	?	?	?	?
36	Lythronax	?	?	?	?	1	?	?	?	?	?	?	?
37	Zhuchengtyrannus	?	?	?	?	?	?	?	?	?	?	?	?

Phylogenetic Dataset 1-249 | Phylogenetic dataset 250-366

The full dataset can only be viewed as a database file or imported (as in this case) into a spreadsheet. The R programming language has excellent facilities for viewing such data in matrices. There is a key to the numeric codes in this example which describes the meaning of each column. All of this can be found attached to his research paper for peer review. [19]

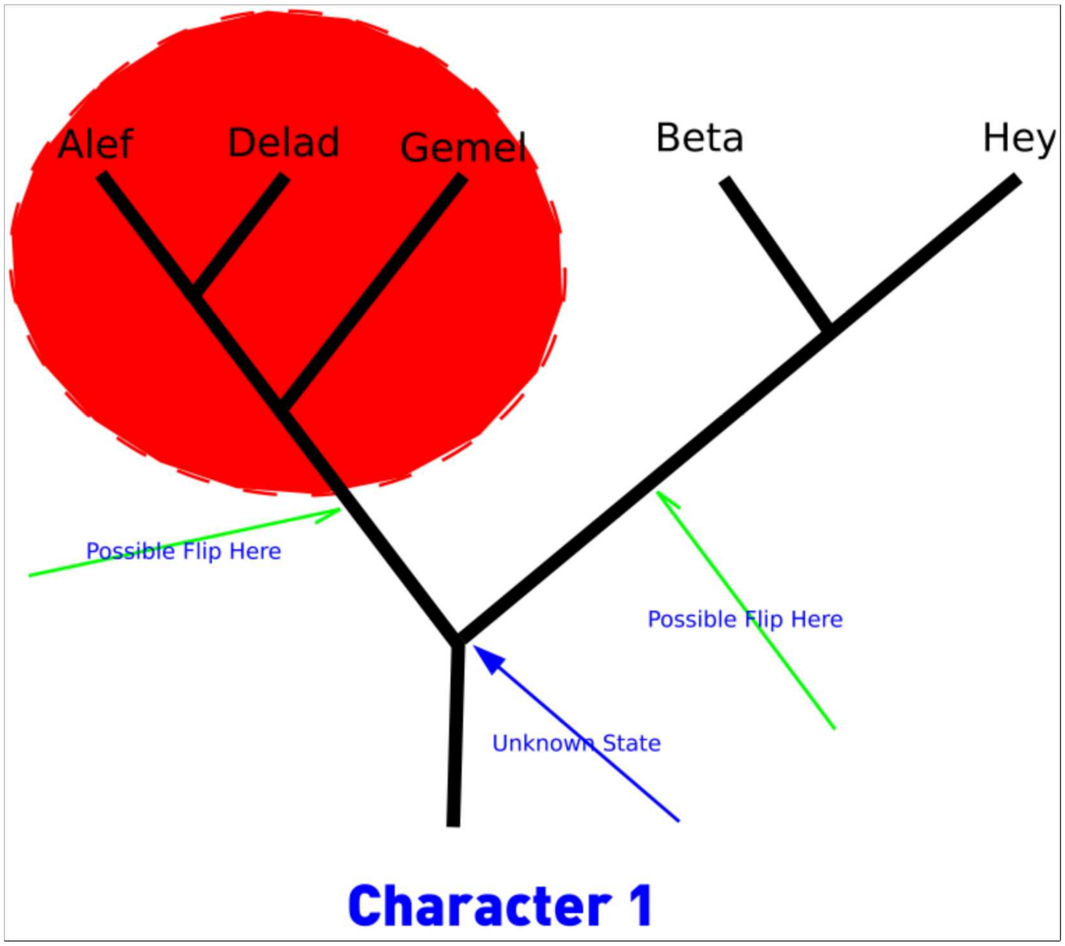
One of the measured characteristics, characteristic 13, is the angle between the bottom of the premaxilla (where the teeth are) and the snout. How important is this angle in distinguishing species is

19 Brusatte, Stephen L. and ,Carr, Thomas D "The phylogeny and evolutionary history of tyrannosauroid dinosaurs", 2016/02/02/, Scientific Reports, http://www.nature.com/articles/srep20252#supplementary-information

a good question. Is it more likely to vary with individuals of a single species? This is a problem with morphological data. We just don't have a big enough fossil record to know and Paleontologist John Harsshman wrote:

> I tried comparing a few images of Tyrannosaurus and Dilong skulls, and there seems to be quite a lot of variation. Some of the T. rex skulls do seem to show an inflection point, but others seem pretty smoothly curved. [20]

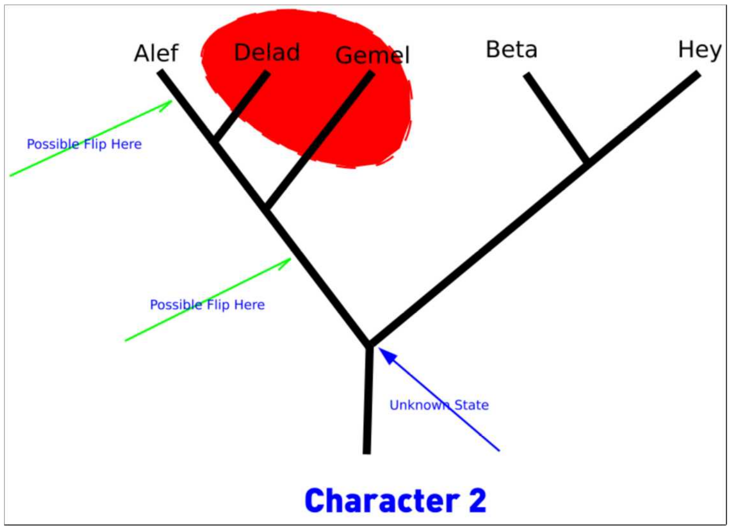
In Felsenstein's simpler example, he starts with an arbitrary tree which is rooted by a unlabeled node and evaluates each characteristic one by one, trying to determine on which edges need change to happen for each of the species on the tree to have the proper state for a specific characteristic. This is worth reviewing because while the process is simple, it is the foundation of other analytic theories. It is also confusing and easy to forget. Lets look at a randomly proposed tree and the breakdown of the distribution of Characteristic 1. Note, that in this example, we don't know the state of the root node. Alef, Gimel and Delad are all in state 1 for characteristic 1. We have a diagram that looks like this.



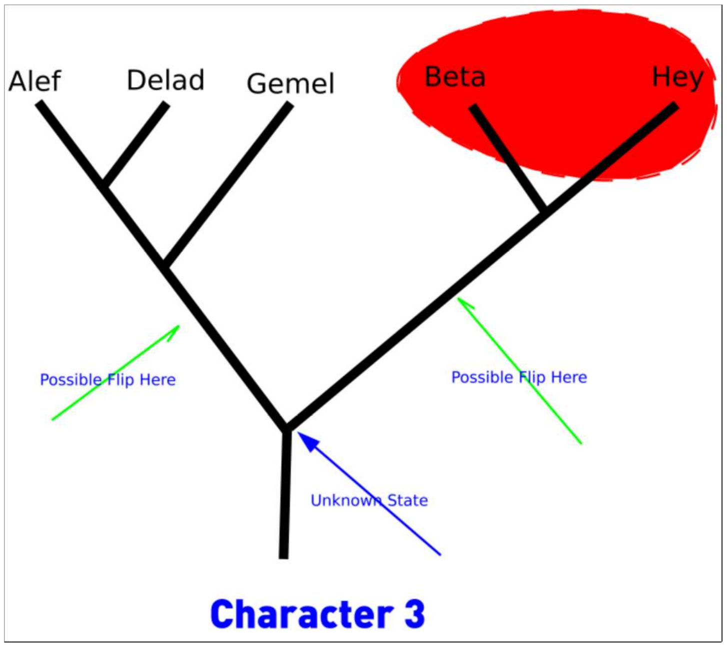
The red area denotes the sections that are known to be in state 1 for character 1. The purple arrow shows the unknown state of the root node. That implies that the character 1 could have flipped in one of those two positions, with an equal chance for either. Either way, the Parsimony value for character 1 in this random tree is 1.

20 John Harshman, usenet, sci.bio.paleontology, 08 Oct 2016, phylogeny tyrannosauroid dinosaurs

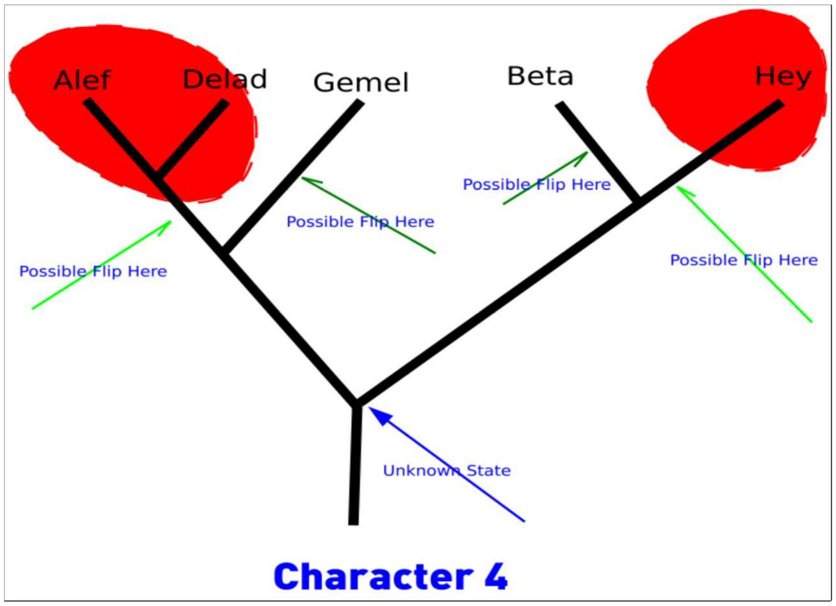
Let's look at Character 2. Gimel and Daled are in red and have a state of 1 for character 2. There are several ways that we can produce this result, for which I showed you one. All of them require two flips and have a parsimony value of 2, bringing our total parsimony value for this tree to 3.



The third characteristic is in state 1 for species Bet and Hey, This would require a single flip for an additional parsimony value of 1. We can apply that flip in one of tow places as you can see with this chart. So all together we have a total parsimony value of 4 (1 + 2 + 1).



Characteristic 4 is for Alef, Delad and Hey. These are disconnected on either side of the tree. If this tree is valid, it might be an example of convergent evolution. It creates a more complex parsimony hypothesis as we can see in the diagram:

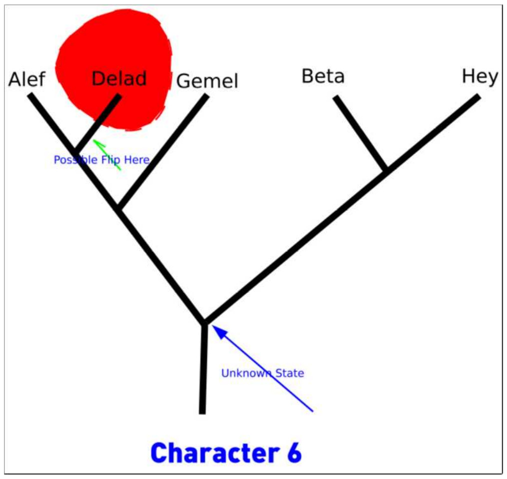


There is no doubt that we have a parsimony of two here, but it is interesting to note that we can use either the dark green marks or the light green marks. The dark green assumes that the unknown state of the root is red (or 1). Our total is now 6

Characteristic 5 is the same as Characteristic 4, raising our total parsimony value to 8.

Character 6 is 1 only for Delad. We can only have one minimal means of doing that as we can see from the last chart.

This gives us a total Parsimony value for this tree of 9.



The question arises, is this the lowest parsimony total we can acquire for any arraignment this set of species. It so happens the answer is no. There is at least one other tree with this same set of species and characteristics with a lower parsimony. However, we do we have a process that will find not only the tree with the minimum parsimony, but also if we examine all the trees and evaluate them as vertexes on Dijkstra’s shortest path algorithm, we can find the lowest parsimony by the shortest path. If we build them up from *simple organisms to complex ones, and directed over time,* the shortest path will give us the most likely evolutionary pathway, because the minimal total amount of evolutionary change through the path is discovered, not just a specific tree without consideration for how that tree was arrived at.

With straight parsimony, all changes in a character trait are equal. But in reality, this is not the case. Many traits change more readily than others, and some traits might not even indicate a difference in species. Size, for example, in human populations vary greatly, within the confines of normal. Even size among populations can change greatly as, for example in recent generations, the Dutch population has dramatically grown taller, gaining over eight inches in height over the past century and a half.[21] The Dutch are not a new species, to the best of our knowledge. We noted that the variation of the angle of the snout of the Tyrannosaurs seems to vary, and with this group, the shape of the nostrils and snout largely separates them from other therapod lineages. [22] How does one account for such differences in value?

Light can be shed on these these problems with multiple approaches, which can involve DNA sequences and morphology. Computationally, the key matter is giving a variety of values to characteristics and standardized costs for changes from one state to the next. The key algorithm for this kind of evaluation grew from the increasing use of genetics and is known as the Sankoff Algorithm [23]

In the case of Sankoff, we want to extrapolate unknown historical states as we derive them from a known table of costs. An easy way to demonstrate the problem is to use the nucleotides of DNA and values related to the chances of random exchanges. DNA nucleotides are either cytosine (C), guanine (G), adenine (A), or thymine (T). The nucleotides bind DNA strands together by forming base pairs of A with T, and C with G. We can create a chart of probable transformation that looks like this (after Felsenstein)

from-to	**A**	**C**	**G**	**T**
A	0	2.5	1	2.5
C	2.5	0	2.5	1
G	1	2.5	0	2.5
T	2.5	1	2.5	0

The key to understanding the Sankoff algorithm is understanding that it recursively hunts for the state with the minimal amount of costs all along the tree, memorizion of the data as we search, walking through a tree of related entities, node by node. In this regard, it is similar to other algorithms which use memoization that keep lists of known states to prevent duplication of efforts, but in this case, these intermediate states themselves are of interest. [24]

21 Gert Stulp, Louise Barrett, Felix C. Tropf, Melinda Mills, "Does natural selection favour taller stature among the tallest people on earth?", 8 April 2015, Proceedings of the Royal Society B 282

22 Stephen L Brusatte; Mark A Norell; Thomas D Carr; Gregory M Erickson; John R Hutchinson; Amy M Balanoff; Gabe S Bever; Jonah N Choiniere; Peter J Makovicky; Xing Xu, "Tyrannosaur Paleobiology: New Research on Ancient Exemplar Organisms", September 17th, 2010, Science. 329, no. 5998: 1481

23 Felsentein pg 13.

24 Cormen et al pg 347

The function that does the recursion searches for all child nodes for the current node. If they exist then it examines all the states for each child node, and finds the state combination with the most minimal cost for each of the nodes possible states. Then it adds all the results for all the children together and memoize this for each possible state in the node. When all the states are computed, start again with the parent node. Let’s look at this in action. We are using the data as it is presented in Felsenstein but this explanation is geared for the programmer who understands graph theory and algorithms.

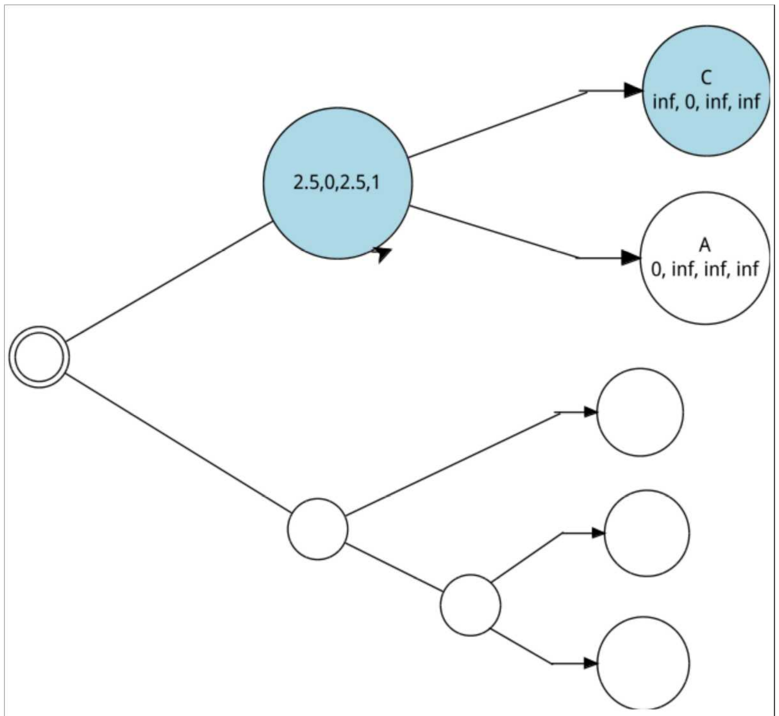


Figure 1 - starting in the upper right leaf.

We are examining a tree and trying to determine the minimal costs for this tree. Once this cost is determined, the tree can be used as a node for the Dijkstra evaluation. We start on the first node, which is a leaf in the upper right of the above chart. It has, like all the possible nodes in our tree, each state for a possible species, an array of 4 states {A,C,G,T}. The values for this leaf node is A=>infinity, C=>0, G=>infinity, and T=>infinity. The values of infinity can NEVER be minimal values (I guess in theory they are positive infinity). That means this node is actually in a C state overall. So this species has a phenotype of a C.

Our routine looks for child nodes, and if they don’t exist then in runs itself recursively on its parent node, marked in light purple in the graphs below (fig 2).

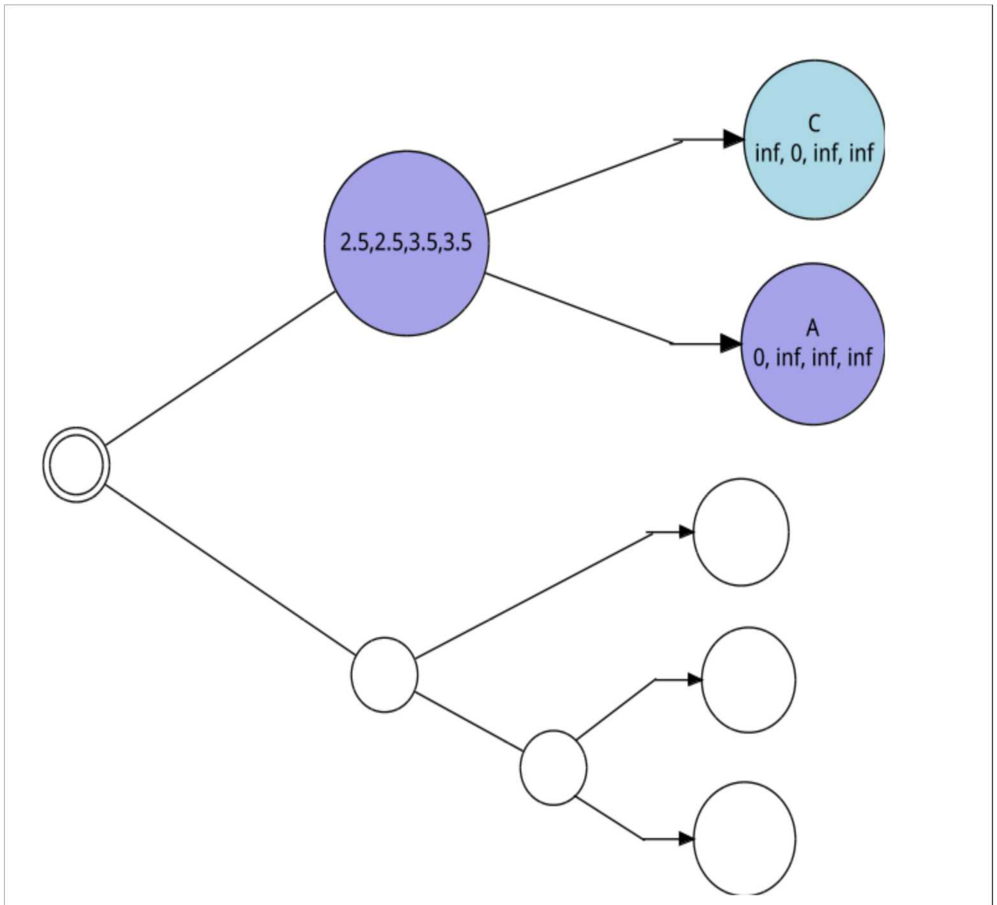


Figure 2

Once we move up the tree and set the parent as the current node, it now proceeds to search for the minimum costs for each possible state.

- It starts with the first property "A" and compare it to the child on the left.
- A comparison of A => A would be zero, but then we add the cost inherited from the child, which is positive infinity, and it can not be a minimum.
- A=>C is 2.5 with the added inherited cost of 0 from the child gives us 2.5.
- A=>G is 1 but we again add the inherited positive infinity.
- A=>T is 2.5 but we again add the inherited positive infinity

The resulting minimum for A is therefore 2.5. Record this in the parent node.

- C=>A is 2.5 plus the inherited infinity is infinity
- C=>C is 0 plus 0 is 0
- C=>G is 2.5 plus infinity which is infinity
- C=>T is 1 plus infinity which is infinity

The resulting minimum for C is therefore 0. Record this in the parent node.

- G=>A is 1 plus the inherited infinity is infinity
- G=>C is 2.5 plus 0 is 2.5
- G=>G is 0 plus infinity is infinity
- G=>T is 2.5 plus infinity is infinity

The resulting minimum for G is therefore 2.5. Record this in the parent node.

- T=>A is plus infinity is infinity
- T=>C is 1 plus zero is 1
- T=>G is 2.5 plus infinity is infinity
- T=>T is 0 plus infinity is infinity

The resulting minimum is T is 1. Record this in the parent node.

Now we examine the next child node which has the cost array of {0, infinity, infinity, infinity} and repeat the procedure:

- A=>A is 0 plus inherited cost of 0 is Zero.
- A=>C is 2.5 plus infinity is infinity
- A=>G is 1 plus infinity is infinity
- A=>T is 2.5 plus infinity is infinity

The resulting minimum for A is 0. We add the current A value of 2.5to this result for a total minimum of 2.5.

- C=>A is 2.5 plus 0 is 2.5
- C=>C is 0 plus infinity is infinity
- C=>G is 2.5 plus infinity is infinity
- C=>T is is 1 plus infinity is infinity

The resulting minimum for C is 2.5. We add the current C value of 0 to this result for a total minimum of 2.5.

- G=>A is 1 plus zero is 1
- G=>C is 2.5 plus infinity is infinity
- G=>G is 0 plus infinity is infinity
- G=>T is plus infinity is infinity

The resulting minimum for G is 1. We add the current G value of 2.5 for a total minimum of 3.5.

T=>A is 2.5 plus infinity is 2.5
T=>C is 1 plus the inherited infinity is infinity.
T=>G 2.5 is plus infinity is infinity
T=>T is 0 plus infinity is infinity
The resulting minimum for T is 2.5. We add the current T value of 1 for a total minimum of 3.5.

This gives the parent node a complete cost array of {2.5, 2.5, 3.5, 3.5}

This node is finished and we repeat the process

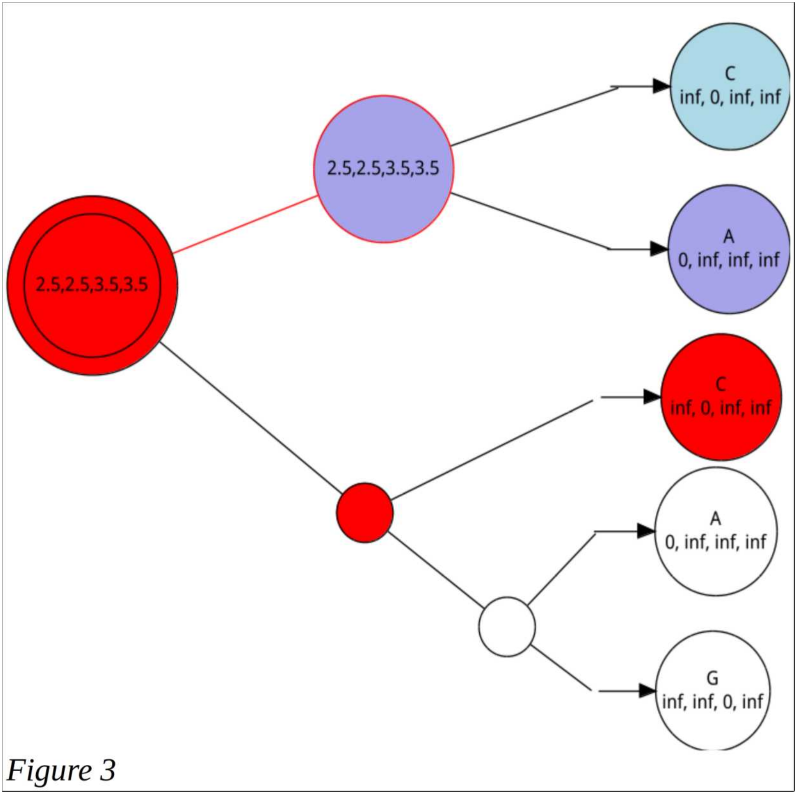


Figure 3

We drop to the parent node which is the root node in this example and run the same process again, first on the left, which results in the array of {2.5, 2.5, 3.5, 3.5 }

- A=>A is 0 plus inherited cost of 2.5 is 2.5.
- A=>C is 2.5 plus 2.5 is 5
- A=>G is 1 plus 3.5 is 4.5
- A=>T is 2.5 plus 3.5 is 6

The resulting minimum for A is 2.5. Record 2.5.

- C=>A is 2.5 plus 2.5 is 5.0
- C=>C is 0 plus 2.5 is 2.5
- C=>G is 2.5 plus 3.5 is 6
- C=>T is is 1 plus 3.5 is 4.5

The resulting minimum for C is 2.5. We record it.

- G=>A is 1 plus 3.5 is 4.5

- G=>C is 2.5 plus 3.5 is 6
- G=>G is 0 plus 3.5 is 3.5
- G=>T is 2.5 plus 3.5 is 6

The resulting minimum for G is 3.5. We record it.

- T=>A is 2.5 plus 2.5 is 5
- T=>C is 1 plus the inherited 2.5 is 3.5.
- T=>G 2.5 is plus 3.5 is 6
- T=>T is 0 plus 3.5 is 3.5

The resulting minimum for T is 3.5. We record it to the node.

Now, when we check the right, however, we have an unknown condition on the right child. So now we run the process on the unknown right child, making it current, and it finds values on the left

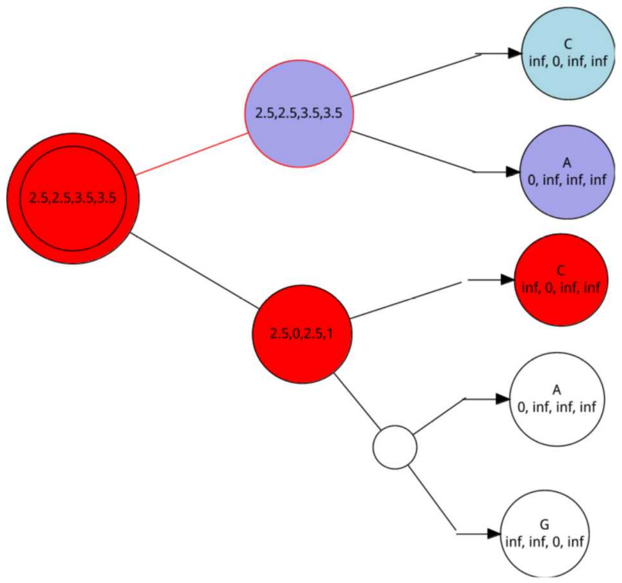


Fig 4

A comparison of A => A would be zero, plus infinity is infinity

A=>C is 2.5 with the added inherited cost of 0 from the child gives us 2.5.

A=>G is 1 but we again add the inherited positive infinity.

A=>T is 2.5 but we again add the inherited positive infinity

The resulting minimum for A is therefore 2.5. Record this in the parent node.

C=>A is 2.5 plus the inherited infinity is infinity

C=>C is 0 plus 0 is 0

C=>G is 2.5 plus infinity which is infinity

C=>T is 1 plus infinity which is infinity

The resulting minimum for C is therefore 0. Record this in the parent node.

G=>A is 1 plus the inherited infinity is infinity

G=>C is 2.5 plus 0 is 2.5

G=>G is 0 plus infinity is infinity

G=>T is 2.5 plus infinity is infinity

The resulting minimum for G is therefore 2.5. Record this in the parent node.

T=>A is plus infinity is infinity

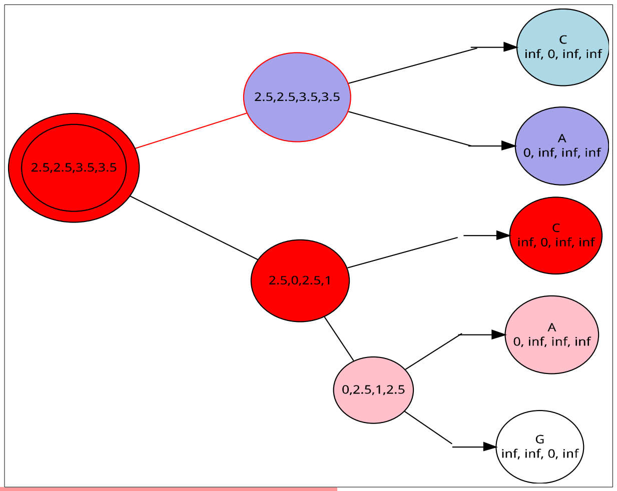
T=>C is 1 plus zero is 1

T=>G is 2.5 plus infinity is infinity

T=>T is 0 plus infinity is infinity

The resulting minimum is T is 1. Record this in the parent node.

The right child is undefined so we trigger the process again, with no panic, on the child. Mark the child current and run the procedure on it. It has a valid left which we mark in pink here.



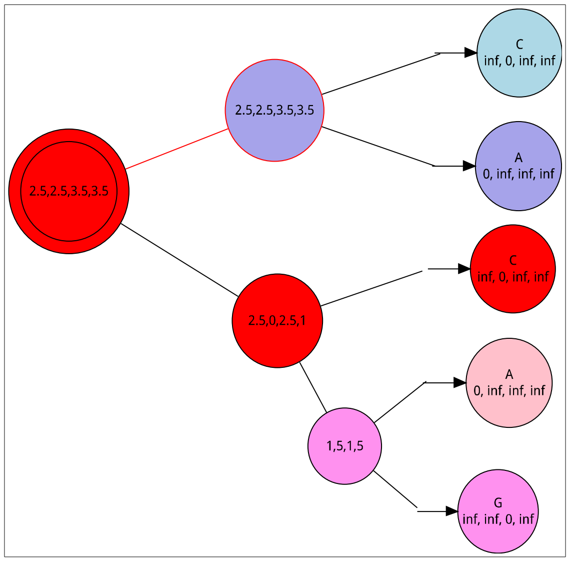
A=>A is 0 plus inherited cost of 0 is Zero.
A=>C is 2.5 plus infinity is infinity
A=>G is 1 plus infinity is infinity
A=>T is 2.5 plus infinity is infinity
The resulting minimum for A is 0. Record 0

C=>A is 2.5 plus 0 is 2.5
C=>C is 0 plus infinity is infinity
C=>G is 2.5 plus infinity is infinity
C=>T is is 1 plus infinity is infinity
The **resulting minimum for C is 2.5. Record 2.5.**

G=>A is 1 plus zero is 1
G=>C is 2.5 plus infinity is infinity
G=>G is 0 plus infinity is infinity
G=>T is plus infinity is infinity
The resulting minimum for G is 1. Record 1

T=>A is 2.5 plus infinity is 2.5
T=>C is 1 plus the inherited infinity is infinity.
T=>G 2.5 is plus infinity is infinity
T=>T is 0 plus infinity is infinity
The resulting minimum for T is 2.5. Record 2.5

Now evaluate the right child:



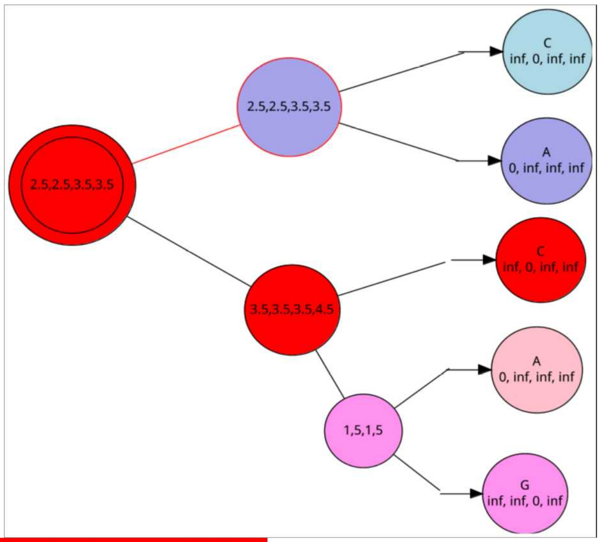
A=>A is 0 plus inherited cost of infinity is infinity
A=>C is 2.5 plus infinity is infinity
A=>G is 1 plus 0 is 1
A=>T is 2.5 plus infinity is infinity
The resulting minimum for A is 1. Add 1 to the parent memoized value of 0 and record 1

C=>A is 2.5 plus infinity is infinty
C=>C is 0 plus infinity is infinity
C=>G is 2.5 plus 0 is 2.5
C=>T is is 1 plus infinity is infinity
The resulting minimum for C is 2.5. Add the parents memoized value of 2.5 and record 5.0

G=>A is 1 plus infinity is infintiy
G=>C is 2.5 plus infinity is infinity
G=>G is 0 plus 0 is 0
G=>T is 2.5 plus infinity is infinity
The resulting minimum for G is 0. Add the parent memoized value of 1 and record 1

T=>A is 2.5 plus infinity is infinity
T=>C is 1 plus the inherited infinity is infinity.
T=>G 2.5 is plus 0 is 2.5
T=>T is 0 plus infinity is infinity
The resulting minimum for T is 2.5. Add the parent memoized 2.5 and record 5.
This node is now {1, 5, 1, 5 }

With the end of those routine, recursion happens back to the red nodes left side. It runs routinely even though the result looks a little differently. We are looking for the minimal values for all possible costs and then adding them to the recorded values from the left side.



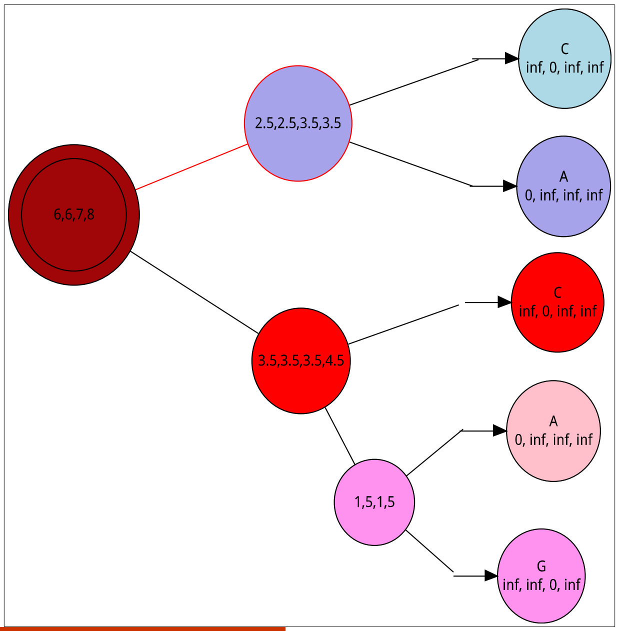
A=>A is 0 plus inherited cost of 1 is 1
A=>C is 2.5 plus 5 is 7.5
A=>G is 1 plus 1 is 2
A=>T is 2.5 plus 5 is 7.5
The resulting minimum for A is 1. Add 1 to the parent memoized value of 2.5 and record 3.5 which is recorded.

C=>A is 2.5 plus 1 is 3.5
C=>C is 0 plus 5 is 5.0
C=>G is 2.5 plus 1 is 3.5
C=>T is is 1 plus 5 is 6
The resulting minimum for C is 3.5. Add the parent memoized value of 0 and record 3.5.

G=>A is 1 plus 1 is 2
G=>C is 2.5 plus 5 is 7.5
G=>G is 0 plus 1 is 1
G=>T is 2.5 plus 5 is 7.5
The resulting minimum for G is 1. Add the parent memoized value of 2.5 and the result for G 3.5.

T=>A is 2.5 plus 1 is 3.5
T=>C is 1 plus the inherited 5 is 6.
T=>G 2.5 is plus 1 is 3.5
T=>T is 0 plus 5 is 5
The resulting minimum for T is 3.5. Add the parent memoized 1 and record 4.5.
This red parent node is now {3.5, 3.5, 3.5, 4.5 }

When this evaluation is finished, the node is finished and the process ends. We now return to the Root Node's right child evaluation. The end result looks like this



- A=>A is 0 plus inherited cost of 3.5 is 3.5
- A=>C is 2.5 plus 3.5 is 6.0
- A=>G is 1 plus 3.5 is 4.5
- A=>T is 2.5 plus 4.5 is 7.0

The resulting minimum for A is 3.5. Add 3.5 to the parent memoized value of 2.5 and record 6 which is recorded.

- C=>A is 2.5 plus 3.5 is 6
- C=>C is 0 plus 3.5 is 3.5
- C=>G is 2.5 plus 3.5 is 6
- C=>T is is 1 plus 4.5 is 5.5

The resulting minimum for C is 3.5. Add the parent memoized value of 2.5 and record 6.

- G=>A is 1 plus 2.5 is 3.5
- G=>C is 2.5 plus 2.5 is 5
- G=>G is 0 plus 3.5 is 3.5
- G=>T is 2.5 plus 3.5 is 6

The resulting minimum for G is 3.5. Add the parent memoized value of 3.5 and the result for G 7.

- T=>A is 2.5 plus 2.5 is 5
- T=>C is 1 plus the inherited 2.5 is 3.5.
- T=>G 2.5 is plus 3.5 is 6
- T=>T is 0 plus 3.5 is 3.5

The resulting minimum for T is 3.5. Add the parent memoized 4.5 and record 8.
This dark red parent node is now {6, 6, 7, 8 }

Sankoff’s Algorithm is very important in understanding how costs can be constructed for the purposes of phylogenies. The most important thing to understand is that it is a recursive algorithm and we can code it in C++ (and other languages) fairly quickly. It also lends itself particularly well for functional programming idioms. A work up of Sankoff’s Algorithm that is expressed like the above diagrams can be used to produce values representing trees. These values can then be used in Dijkstra’s shortest path in order to determine cost between vertexes.

If you start by coding objects that represent traits with costs values, then you can embed those into node objects. Just as we did in the decision tree, then we can stitch them together as a path and walk the path recursively. So in the end, the algorithm uses elements of the decision tree in the Königsberg problem, and elements like the Dijkstra problem, which results in values that can be used to create stepwise addition of trees by the species for a further Dijkstra analysis in trying to determine the most likely evolutionary paths. Furthermore, C++’s templating mechanism allows us to use any analysis we choose for determining costs, from any statistical method or phenotype character trait.

Let’s start out by looking at an individual state object:

```
/*
 * =====================================================================================
 *        Class:  State
 *  Description:  This describes a possible individual state or charactoristic
 * =====================================================================================
 */
template<class cost>
class STATE
{
	public:
		/* ====================  LIFECYCLE     ======================================= */
		                      /* constructor */
		//r is the memoized value of the minimum cost of a single charactor when compared to the child nodes
		explicit STATE<cost>(std::string const xa, cost xr)
		:_nam{xa}, _r{xr}
		{
			std::cout << std::endl << "Building a state pair" << std::endl;
			std::cout << "nam => " << nam()  << "\tcost=> " << r() << std::endl;
		};


		/* ====================  ACCESSORS     ======================================= */
		std::string nam()const {return _nam;};
		cost r() const {return _r;};
		void r(cost b ){_r = b;};
		/* ====================  MUTATORS      ======================================= */

		void change(cost a, cost b){
			_nam = a;
			r(b);
		}
		/* ====================  OPERATORS     ======================================= */
		friend std::ostream & operator << ( std::ostream & os, const STATE & obj )
			{
				os << std::endl << "name => " << obj.nam() << ":cost=> " << obj.r() ;
				return os;
			}		/* -----  end of function operator <<  ----- */
		STATE & operator = (cost b)
		{
			r(b);
			return *this;
		}

		/* ====================  DATA MEMBERS  ======================================= */
	protected:

	private:
		std::string _nam;
		cost _r;

}; /* -----  end of class State  ----- */
```

A state has 2 data members, a name and a value. Using templating they can be of any kind of data type, although in our example we use ints. We overloaded the output stream to make printing of the values easy. And we overloaded the assignment operator to make the value assignment easy, something we will be doing frequently in evaluating these values as we move through the tree.

These states are grouped together in C++ vectors. In our example we have 4 per node. This is not etched in stone for the algorithm so I provide flexibility in its type and assignment.

```
/*
 * =====================================================================================
 *        Class:  Pstates
 *  Description:  vector of all possible states
 * =====================================================================================
 */
template<class cost_type>
class Pstates
{
	public:

		/* ====================  LIFECYCLE     ======================================= */
		/* constructor      */
		Pstates (std::vector< STATE<cost_type>  > x)
			: _vstate{x}
		{
				for( auto& y : _vstate)
					std::cout << std::endl <<"adding state " << y << std::endl;
		};

		~Pstates (){};                            /* destructor       */

		/* ====================  ACCESSORS     ======================================= */
		const std::vector< STATE<cost_type> >& vstate() const{
			return _vstate;
		}
		void vstate(std::vector< STATE<cost_type> > in ){
			_vstate = in;
		}

		/* ====================  MUTATORS      ======================================= */

		/* ====================  OPERATORS     ======================================= */

		const Pstates& operator = ( const Pstates &other ); /* assignment operator */
		const Pstates& operator = ( const std::vector< STATE<cost_type > >  &other ){
			vstate(other);
			return *this;
		} ; /* assignment operator */
		STATE<cost_type>& operator [] ( unsigned int const index ){
			return (_vstate[index]);
		} ; /* [] operator */

		friend std::ostream & operator << ( std::ostream & os, const Pstates & obj )
			{
				for(auto state_pair : obj.vstate() )
					std::cout << std::endl << "state pair" << state_pair;
				return os;
			}                  /* -----  end of function operator <<  ----- */

		/* ====================  DATA MEMBERS  ======================================= */
	protected:

	private:
		std::vector< STATE <cost_type> > _vstate;

}; /* -----  end of class Pstates  ----- */
```

The Pstates constructor takes a vector of type **`STATE<class>`** . Those are paired data in objects, assigned to the vector and in fact there are a number of C++ libraries I could have employed including enums and pairs. This method gives me a little more flexibility if I choose to alter the program, but using standard libraries is certainly valid. One design choice here was that one needs to construct the **`vector< STATE<class> >`** externally and feed it whole. Once the vector is created, of course, all the tools for vectors can be leveraged both within the object construction as one builds this out, and externally which bends the containment rules of object oriented programming, but I can live with it.

This syntax:

```
for( auto& y : _vstate)

        std::cout << std::endl <<"adding state " << y << std::endl;
```

is relatively new to C++. These are the new auto variables that are available in C++ and allow one to deduce the data type from the context of the expression, specifically the type of the data on the initialization. This can be particularly useful for templates or allocators where the data type might well change with the implementation, or just changed in debugging or normal development. If you change your mind about the data type, auto remains correct.

Also, the for(:) syntax is known as a Range-for Statement.[25] It gives the programmer access of each member of a range, such as vectors. This syntax will be seen more and more in C++ code going forward.

If you look at the STATE object, you will recall that the standard output object is conveniently overloaded, and I can also use the assignment overload to conveniently use this syntax to edit the values of STATE<class>.

Now we can construct our NODE object:

```
template<class unk>
class NODE{
          public:
      //========================LIFECYCLE=======================================
      /* Constructor */
                   NODE( std::string  xt , const unk &x , NODE<unk> *xcl= 0, NODE<unk> *xcr = 0, NODE<unk> *xp =
0 ):_trait{xt},_states{x}, _cl{xcl}, _cr{xcr}, _p{xp} {//child left, child right etc
                   std::cout << std::endl <<"Constructing node  "<< trait() << std::endl << trait() << " has the following states => " <<
states() << std::endl;
                   };

                   NODE(const NODE &other); //copy constructor
             ~NODE();

      /* ===================  ACCESSORS     ====================================== */
//no need for inline when you are defineing functions within the class definition
                   std::string trait()const{
                             return _trait;
                   }
                   unk states()const{
                             std::cout << std::endl << __LINE__ << ": sending _states" << std::endl;
         return _states;
      };
                   NODE<unk> * cl()const{
         return _cl;
      };
                   NODE<unk> * cr()const{
         return _cr;
      };
                   NODE<unk> * p()const{
         return _p;
      };
                   void states(const unk &x){
         _states = x;
      }
                   void trait(std::string xt){
                              _trait = xt; //With genetics this is a A C G T although it is just a label
                    }
                   void cl(NODE<unk> *xcl){ //This is the left child node in the tree
         _cl = xcl;
      }
                   void cr(NODE<unk> *xcr){//This is the right node child node in the tree
         _cr = xcr;
      }
                   void p(NODE<unk> *xp){//This is the parent node in the tree
         _p = xp;
      }
      /* ===================  MUTATORS      ====================================== */

      /* ===================  OPERATORS     ====================================== */
                   template <class T>
                   friend std::ostream& operator<<(std::ostream& os, const T&);
```

25 Bjarne Stroustrup, “The C++ Programming Language”, 4th Edition 2013, Addison-Wesley pg 233

```
                template<class T>
                friend std::ostream& operator<<(std::ostream& os, const NODE<T> &);

   protected:
      /* ====================  DATA MEMBERS  ======================================= */


   private:
      /* ====================  DATA MEMBERS  ======================================= */
                 std::string _trait;
                 unk _states;
                 NODE<unk> * _cl, * _cr, * _p;


}; /* -----  end of template class NODE  ----- */
```

NODEs have three attached NODEs linked together, a left child, a right child and a parent. This is a little bit like a linked list. A complete list of NODEs allows for a graph, as it does in the decision tree program, although we don't even need a container. The attached NODEs default to 0, which allows for testing. Ostream is overloaded

```
template <class T>
friend std::ostream& operator<<(std::ostream& os, const T&);

template<class T>
friend std::ostream& operator<<(std::ostream& os, const NODE<T> &);
```

With all our objects nicely constructed (and with room to grow them if need be later), the next thing to build is the crawler. The crawler will be a recursive function. What needs to be recognized is that this formulation is a standard binary tree, but upside down. Therefore,we can crawl the tree using a post-order traversal. We can define the recursive function as follows:

Preliminary activity: Load the cost table for state. Load any node. Follow it's parents to the root. Once we reach the root then do the following

1. Examine the upper left child. If it doesn't exist, examine the right hand child. If it doesn't exist => exit the function..
2. If the left exists, examine its Pstates. If the Pstates are undefined, do a recursion passing the child node as current. Otherwise, Examine your Pstates. Starting with the first character in the current pState, compare it against each character in the child's Pstates vector. Walk through all the characters.
3. Look at the cost for each pairing. Add the cost to the child's Pstate character value. Peek at the nodes temporary buffer. If the sum is less than the buffer, write it to the buffer. Continue to the next child Pstate character. Repeat through all the child's Pstate characters.
4. This will give you the minimum cost + inherited value in the buffer. Add whatever value is in the current Pstate's character value to the buffer and then write the sum to the Pstate character of the current node. Write infinity to the temporary buffer. Continue to the next character for the current nodes Pstate and repeat from 3.

5. Examine the right child. If it doesn't exist, exit the function. Otherwise examine the child's Pstate. If the childs Pstate is undefined call recursion with the child marked as current (sent as a param). Otherwise, repeat from step 3.
6. Return from function.

Fossil Recognition and Pattern Recognition in Graphics

Much of the debate relating to phylogeny involves the application of static data which is processed by trained observers. This is where the field has been at for over twenty years. Field workers find bones in rock, and highly trained preparers try to separate the rock from mineralized bone. Then paleontologists measure, scan and draw the specimens, dividing observations into what is considered important differentiation which are then processed through computational algorithms, not much different from what is described above. But what has happened over the last 20 years has been nothing short of astonishing. Soft tissue and soft tissue remnants have been found in Cretaceous era bone beds, and even Jurassic age fossils like the famous German specimen of Archaeopteryx have soft tissue traces in the surrounding rock. Hundreds of feathered dinosaurs have been found in China, and now that people are looking, such specimens are now being found the Canadian Rockies, the Gobi Desert and even in the collections standing in museums worldwide. Proteins have been found in Tyrannosaur bone tissue, and several Dinosaur mummies have been found which have crops and soft tissue. We have extrapolated the color of feathers from melanosome in feather impressions. One thing has become very obvious, tearing the fossils from the rock is not always a good idea. Within the rock surrounding bones is a huge amount of soft tissue information. Scanning technology is expanding, and the use of CAT Scans, and Florescence techniques are literally shedding a whole new life to paleontological research. Brain cases are a now a wide area of study and the dream is to go from encased rock to 3d graphics printer, without being destructive to the specimen.

What we need to develop, as time moves on, is the ability to not only scan rock, but also identify likely fossils and attempt to classify them by artificial intelligence, hopefully we remove much of the human factor and its associated biases. Can this be done? Yes, and within a few short years we should have such technology ubiquitously available. And with worldwide databases of fossils, we should be much better and developing phylogeny maps of species and remove much of the human bias that now plagues the field.

If we CAT scan a rock with a fossil within it, how can we identify the fossil? This is an enormously important question which has an ethical wrinkle. Most CAT scans are using proprietary graphics formats. Getting specimens into an international database is going to require ending the dependency of these private formats and to get them into formats based on international copyright and patent free formats, that is copyleft formats. The dependency on proprietary file formats for scanning equipment is the biggest impediment of future software development for the community. The effort to go from CAT scan to the 3d modeling program, *Blender*, is just too difficult and even the legality of tools is questionable. Development of artificial intelligence is going to depend on patent free, and royalty free international standards which allow for reading and manipulation.

From a technological perspective, consider the depth of the problem. Even the simplest graphic recognition problem requires considerable thought and application of theory. How do we discover and identify, for example, a black circle on a white sheet of 4 x 4 inch paper? Engineers have been grappling with this problem for decades and its solution is the cornerstone for a broad swath of computational problems from biometrics, driverless cars, facial recognition, and fossil evaluation.

So how does one identify a circle on a sheet of paper? Or a circle in a PNG file? The answer involves the engineering principle of the Circle Hough Transform.[26]

The Hough Transform was a patented method for recognition of shapes patented by IBM in the 1960's. It was later updated and has become quite famous in engineering circles. It involves introduction of the concept of parameterized space within euclidean equations. This can best be shown by an example:

With a straight line we have the familiar equation of

$y = mx + c$ where m is the slope and c is a translational constant. For any pair of x,y we can sock them into this equation and plot them on the line with simple subsitution.

$y_i = mx_i + c$

As a specific example we can have a graph of

$$\mathbf{y = 3x + 2}$$

where x is 2, y is 8.

x is 3, y is 11.

x is 4, y is 14 … and so on.

This has been coined as an equations in **geometric space.**

If we flip the equation and look at it from a different perspective we can develop another set of related equations that are coined as “parameter space”. For any (x_i,y_i) we can treat m and c as variable for a specific point. This is accomplished by using the parameterized form of the equation which might look like this for a line

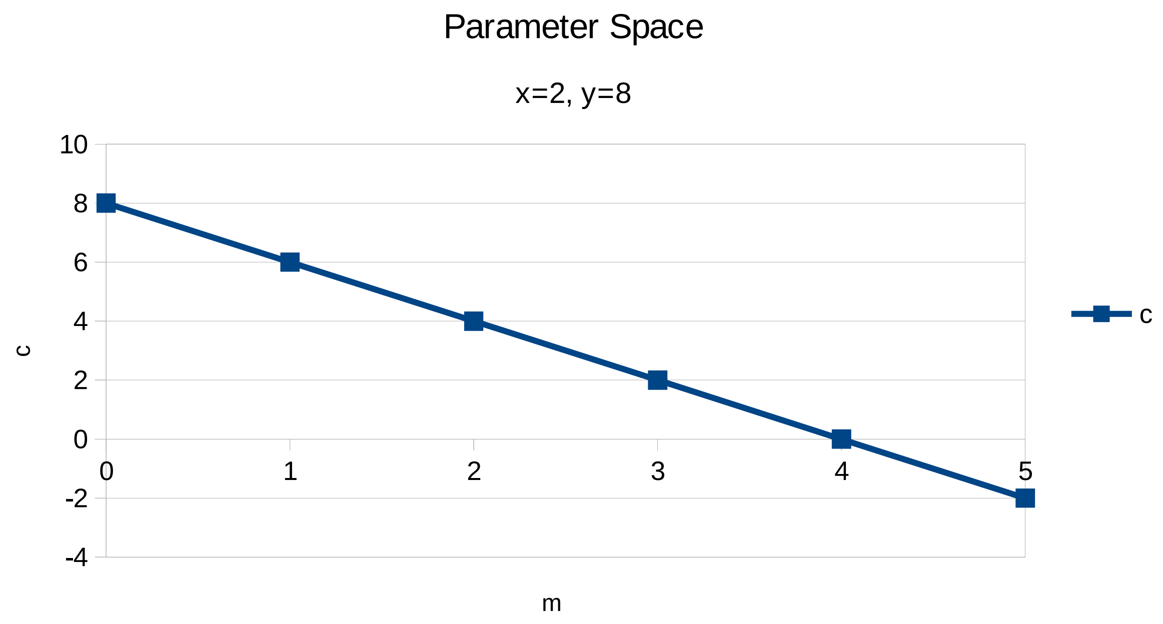
$c = -mx_i + y_i$

When we plot this equation, it is yet another line fixed on (xi,yi). The relationship between the parametrized space and the geometric space is of interest within the theorem of Hough Transformance. When we evaluate our previous example where x = 2 and y = 8, x is now the slope and y the translational constant.

26 R. O. Duda and P. E. Hart, “Use of the Hough transformation to detect lines and curves in pictures,” Communications of the ACM, vol. 15, no. 1, pp. 11–15, 1972.

c = -m2 + 8

m	c	x	y
0	8	2	8
1	6	2	8
2	4	2	8
3	2	2	8
4	0	2	8
5	-2	2	8



Here, m is the independent variable and c is the dependent variable within the parametrized form.

For each point on the geometric graph, we produce another line.

Let's add other points from the original equation of 3x+2 = y :

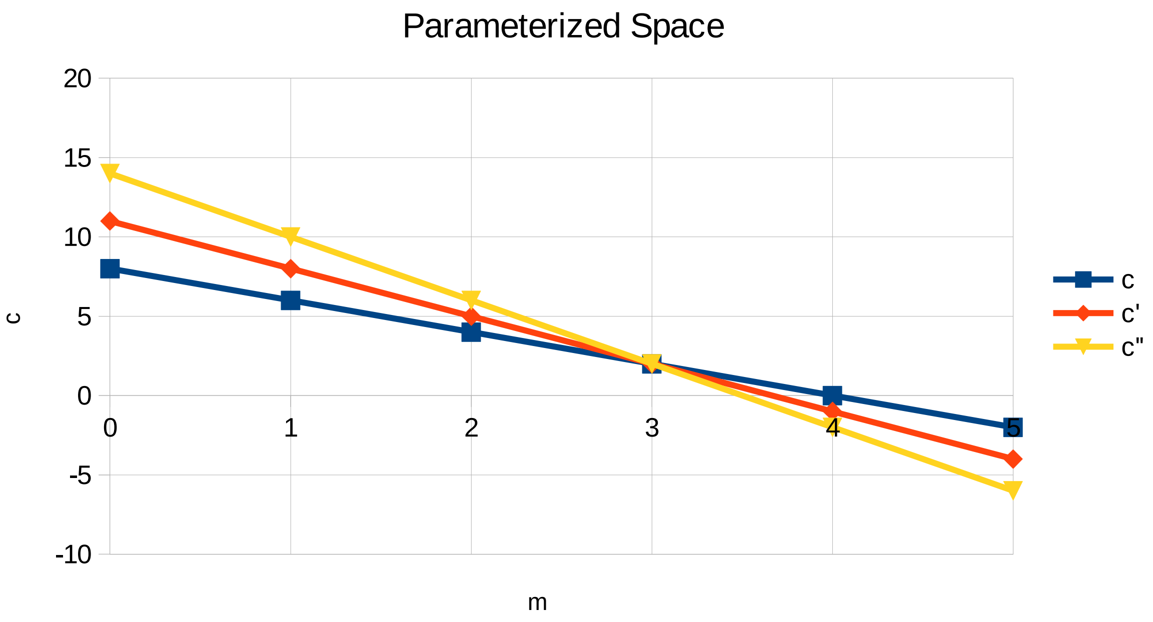
point of (3, 11). This produces the parameterized line of: c' = -3m' + 11;

point of (4, 14). This produces the parameterized line of c''=-4m'' + 14;

We can describe the data for these three lines in this color coded table and accompanying graph.

m	c	c'	c''	x	y	x'	y'	x''	y''
0	8	11	14	2	8	3	11	4	14
1	6	8	10	2	8	3	11	4	14

2	4	5	6	2	8	3	11	4	14
3	2	2	2	2	8	3	11	4	14
4	0	-1	-2	2	8	3	11	4	14
5	-2	-4	-6	2	8	3	11	4	14



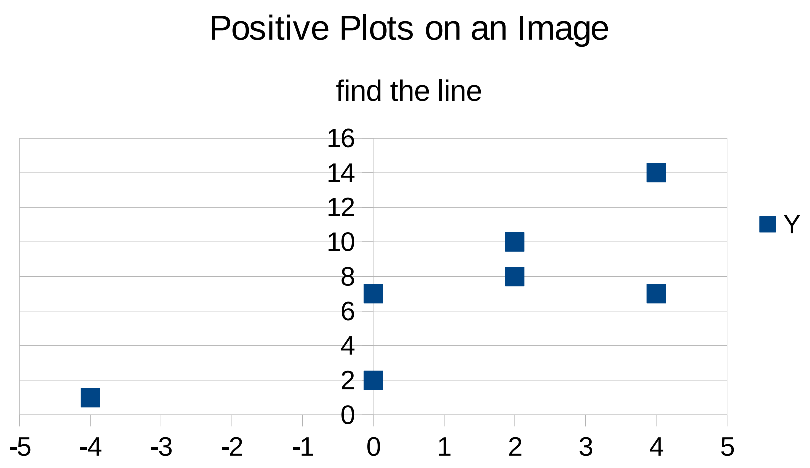
In Parameter Space, the crossing points represent lines. If you have more than a single cross point, that would represent more than one line. The relationship is 1:1 so that a single crossing point in Parameter Space can only represent a single line, while all the lines that go through a crossing point infer a point on the line which is determined by the crossing point.

Lets say we have a graph with a known set of positive points:

{ (2, 8), (4,7), (4,14) (0,2), (2,10), (-4,1), (0,7) }

Do we have lines from these points assuming they are pixels from a graphic?

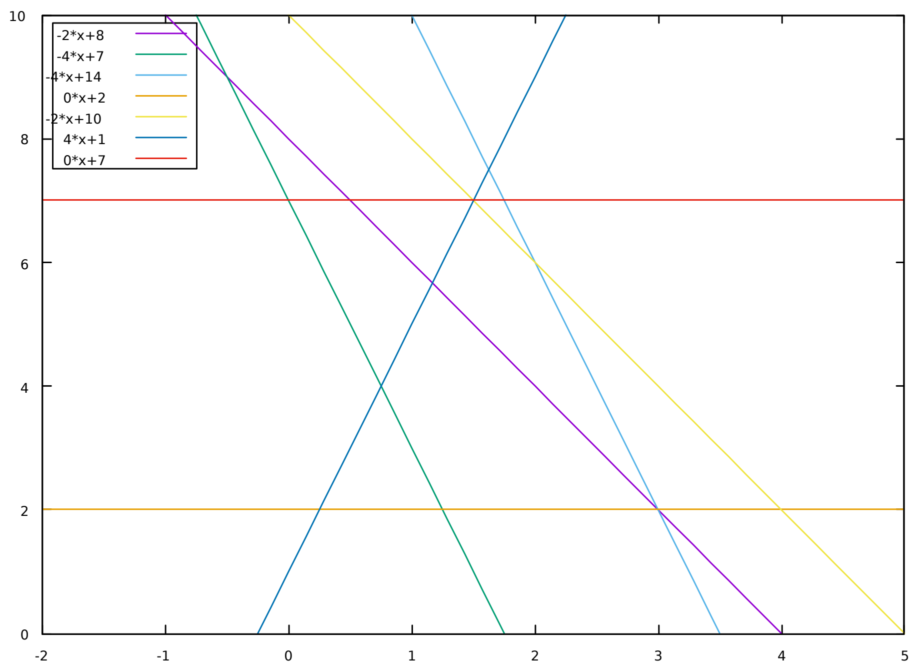
When we plot them we get the following graph which is not clear.



We can convert them to parameter functions and see how many crossings we have

X	Y	Parameter Functions
2	8	c = -2m + 8
4	7	c = -4m + 7
4	14	c = -4m + 14
0	2	c = 2
2	10	c = -2m + 10
-4	1	c = 4m + 1
0	7	c = 7

Lets analyze this “images” now though its parameters:



Every crossed point within the parameter graph represents a potential line. The only exception to this is that vertical lines where in the geographic graph they would have a slope (m) of infinity and are therefor parallel. In our example, I created these points as a demonstration from an image that has 2 lines and a single random point. The lines pixels can be described as **`y=3x+2`** and **`y=3/2x+7. The individual pixel is 4,7.`**

Since 2 points determine a line, our parameter space displays a bit of noise. Any crossing is a line, yet not all these lines are intentional. The solution to determining intended lines is having a significantly large sample of pixels. If we get, say, 40 crossings of a point, that is likely a representative of a real line in the image.

How can we compute this?

Let's divide our parametric space in 100 x 100 pixel divisions and mark them (just as a random choice).

m_i, c_j

If we have N number of sampled points in geometric space, each point can be described as (Xn,Yn) where

$0 \le n \le (N - 1)$

where n starts at zero and indexes up to N-1.

If we stack them in a C++ vector, we can walk though each point and compute the parameter:

The line is c=mx +y

$c_i = -m_i x_n + y_n$ is a specific point. Find that division that this point is within

{ $m_{j,}c_j$ } represents an area that is incremented

At the end of the vector we find the final values for the { $m_{j,}c_j$ } pairs. Those with the highest values represent lines.

Commonly, this transformation is performed using polar coordinates for lines using angles (Θ).

$\rho = x \cos \theta + y \sin \theta$

where ρ = shortest distance from the origin to the line

θ = angle between the x-axis and the right angle from the origin to the line.

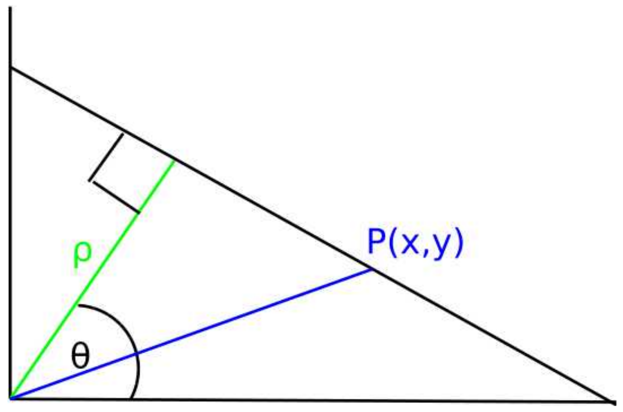
The domain of ρ and θ are bounded: $0 \le \rho \le R$,

where R = maximum distance from a pixel to the origin of the image and $0 \le \theta \le 2\pi$.

Lines that pass through point (x_n , y_n) have parameters (ρ, θ) that satisfy

$\boldsymbol{\rho = x_n \cos \theta + y_n \sin \theta}$ [27]

All the point on the line can be described by that line and that line is defined as a function of θ, the angle, and radius).



f(θ, ρ) = L where L is a line.

27 Rhody, Harvey "Lecture 9: Object Description and Location – Hough Transform", October 6, 2005, Chester F. Carlson Center for Imaging Science Rochester Institute of Technology

Any point P(x,y) can be described as two vectors, one along ρ, and one along L (that is the line).

Let the vector along ρ be called Q

Let the vector along L be called R

a matrix of [Q, R] is associated one to one and onto with point P(x,y), which is vector [x,y] for a parameters of θ, ρ.

The Parameter Space table now looks like this:

We can calculate the parameters for any point on the line with this formula.

$\rho = x \cos \theta + y \sin \theta$

Using the line we defined previously

$\mathbf{y = 3x + 2}$

$\boldsymbol{\rho = x \cos \theta + y \sin \theta}$

where x = 2, 3, 4

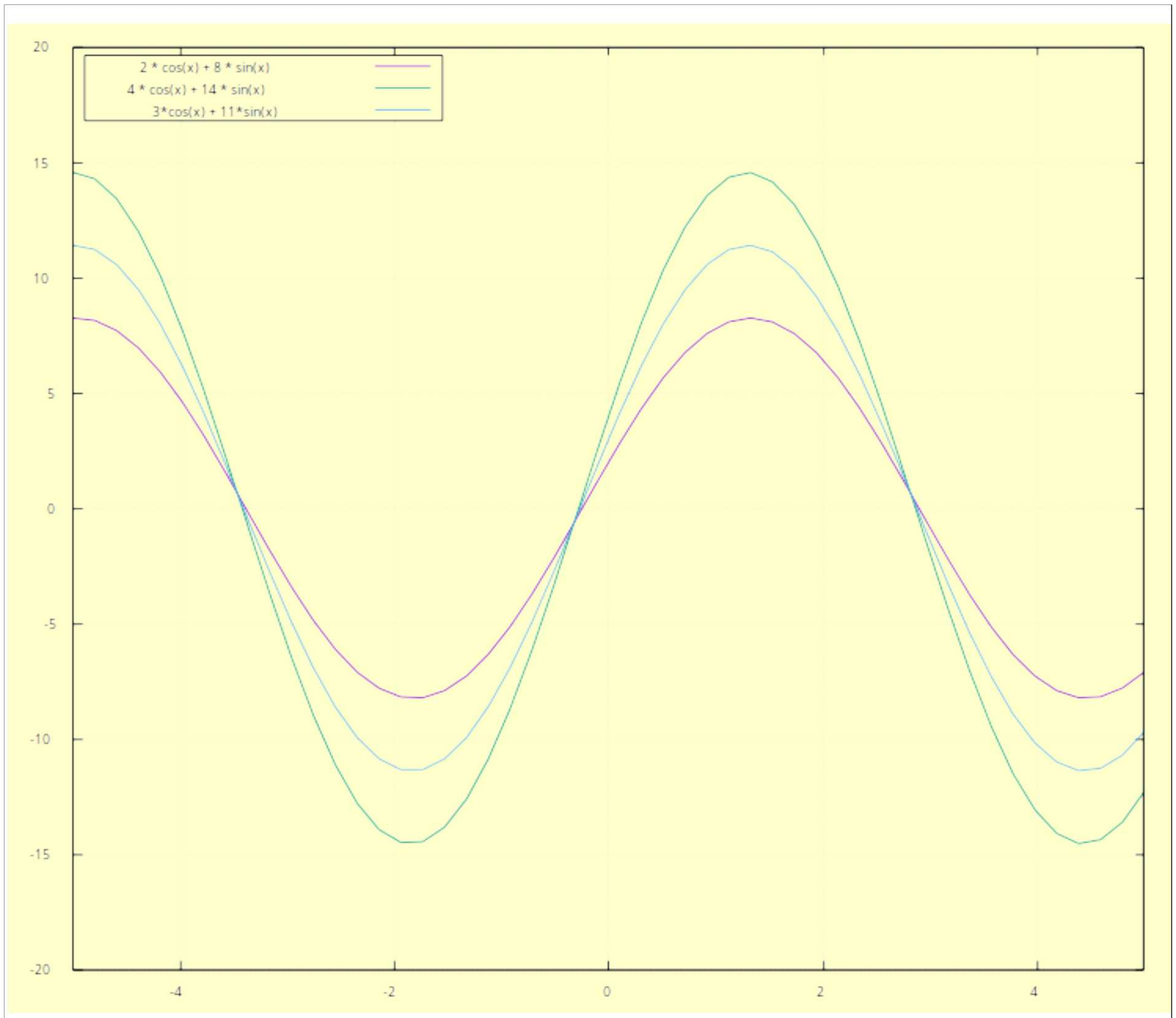


figure 1: Parameter Space: Theta vr Rho for a line

Note: x is θ on this plot

Implementation of this algorithm, which is termed the normal, involves the same technique as before, but our walk and polling of the parameter space involves a 180 degree sweep of the the parameter space.

This can simplify implementation of the algorithm by creating counters for each parameter pair (θi, ρj) which divides up the space.

Now, for each (x,y) compute the parameter with:

ρ= x cosθ + y sinθ

Find the container closest to $\{\theta, \rho\}$ and increment it.

Hough Circle Transform

Of bigger interest is the extrapolation of the Hough Transform to the Hough Circle Transform. The implementation of this algorithm has allowed for software to readily do facial recognition, and many other applications. For fossils, it can be particularly useful for examining skulls which have many circular features, particularity the orbit, and the shape of the premaxilla, the temporal fenestra and more. For quite a while it has been a popular area of study and development because of the increasing use of surveillance software, and facial recognition for cameras. This algorithm is also of interest because it is included in the opencv library.[28]

The equation for a circle is as follows:

$$r^2 = (x - a)^2 + (y - b)^2$$

We can see that unlike a line, a circle has three parameters, {a, b, r} where a and b are the (x, y) of the center, and r is the radius of the circle. Without proving the math, we know that we can find the parameter space by the following formula.[29]

$$x = a + r*\cos(\theta)$$
$$y = b + r*\sin(\theta)$$

Lets examine an example of parameter space evaluation. We can perform a method as follows:

```
For each pixel(x,y) which is positive
  For each radius r = 10 to r = 60 // the possible radius
    For each $\theta$ // the possible  theta in radians
      a = x – r * cos($\theta$ ); //polar coordinate for center
      b = y – r * sin($\theta$);  //polar coordinate for center
      A[a,b,r] +=1; //voting
    end
  end
end
```

θ is the interdependent variable and a and b is the dependent variables.

28 http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_houghcircles/py_houghcircles.html aquired March, 2017

29 Carolyn Kimme, Dana Ballard, and Jack Sklansky. Finding circles by an array of accumulators. Communication of the ACM, Volume 18, Number 2, February 1975.

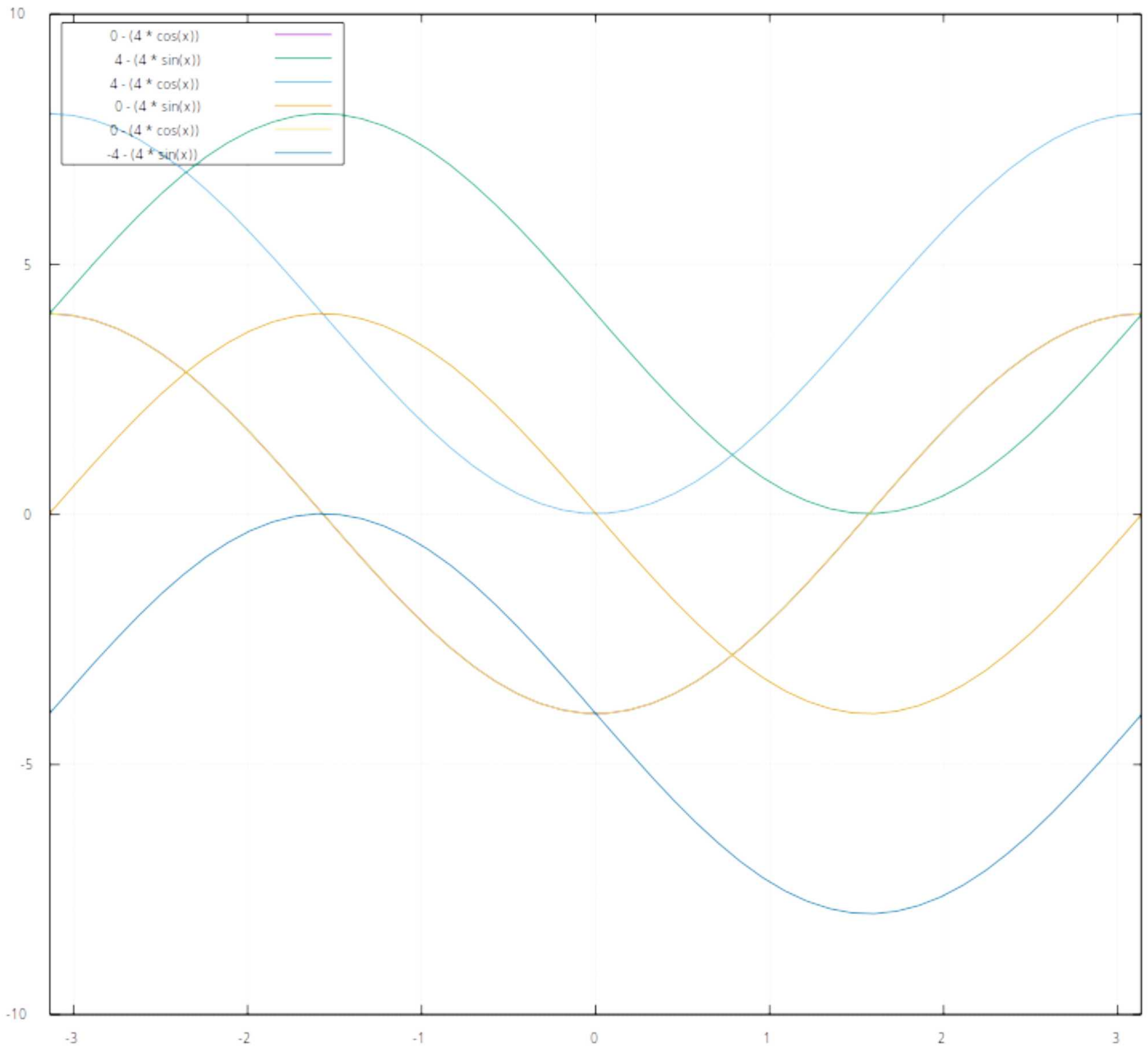


figure 2: Three{0,0} intercepts for the parameter space of a circle with a radius of 4 centered on the origin

Lets apply an example. Lets say I have a circle with a radius of 4 about the Origin (0,0). We can have on this circle 3 points on the edges of (0,4), (4,0), (0,-4), among an infinite number of other points. When I substitute these x and y values as pairs into the parameter formula, I get 3 pairs of curves (one is repeated) that intersect at 0,0 like this: Note that since we are looking for {0,0} I can combine the x and y graphs.

```
a = 0 - 4 *cos(0)
b= 4 -  4*sin(0)

a = 4 - 4 *cos(0)
b= 0 -  4*sin(0)
```

```
a = 0 - 4 *cos(0)
b= -4 -  4*sin(0)
```

The usage of image recognition software for evolutionary biology is one of the largest areas of advancement and opportunity in the coming generation of computational research and expertise. What drives this is the technological breakthroughs in the scanning technologies, which are getting reduced in size, cheaper and increasingly available to museums and research labs. A few years ago a dinosaur mummy with soft tissue intact was discovered in the Dakotas. It was termed “Leonardo” and was scanned with several tools as described by its website:[30]

What kind of Xrays were used to look inside the fossil?

The science team used several different types of radiation to create the Xrays of the dinosaur mummy. The first they used was a water-cooled Xray tube. This is similar to what is used to take Xrays of people, only it is about 5,000 times more powerful. This kind of power was needed because it had to look through more than two feet of rock. Even this wasn’t powerful enough, so the team took Leonardo down to Houston, Texas were NASA made room for him in a special lead-lined jet hangar at the Johnson Space Center. While there, the science team used Iridium and Cobalt radiation to look inside the fossil. ... The yellow container holds a small but highly radioactive isotope. The Cobalt in particular was extremely powerful. The science team had to leave the building and hide behind another building to be certain they were safe from the radiation. However, the most powerful radiation used in the study of the fossil was used on Leonardo’s arm. A Linear Accelerator fired radiation at the fossil at Stanford University’s SLAC (Stanford Linear Accelerator Center) also known as the National Accelerator Laboratory. Yes, Leonardo is important enough of a fossil to have these kind of resources used in the research!

How was the digital image of the fossil created?

The Dinosaur Mummy was the first dinosaur to be replicated in an entirely new way. Because the skin on the surface of Leonardo was so fragile, if the mold necessary to make casts, or replicas, of the fossil had been created in the same way it has been done for many years, the skin would have been damaged or destroyed. This made the traditional method of pouring silicon rubber on the fossil and peeling it off out of the question. The science team had to come up with another way. The answer was a new technology called White Light Scanning. A grid of light and shadow was projected on the fossil and special cameras recorded this grid and the surface of the fossil in 3D. The resolution of the cameras is thinner than a human hair, so that the digital information they provided shows every fine detail of the original fossil. ... This information was then used by a computer to control a machine called a rapid prototyper that was able to make a perfect replica of the dinosaur mummy without any possibility of damaging the original fossil.

30 http://www.dinosaurmummy.org/guide-to-dinosaur-mummy-csi.html accesses March 2017

They needed to ship the entire fossil, which weighs tons, from the Phillips County Museum, in Montana, all the way to Texas to do scans. In March of 2017, Don Brickman of the Royal Tyrrell Museum, in Drumheller Calgary, Alberta told me that they had set up florescence equipment in their labs to do 3d imaging. This is a complete breakthrough in technology and we can expect that in the future that CAT Scanners will likely become standard equipment in most labs. With the discovery of so many soft tissue remains, the days of tearing up fossils is largely over without first doing analysis of the possible soft tissue remains inside the surrounding rock.

In 2011, SLAC did a wonderful lecture on the analysis of Archaeopteryx.[31] The presentation done by Uwe Bergmann walks through several types of images, showing trace materials. But this leaves us with several problems, the first being, what kind of image formats are being used to image fossils in CAT scanners and other scanning technologies. How can we prepare these images for analysis and for the analysis by artificial intelligence. There are no standards. In correspondence with Ramon Nagesan, the PhD resident at the Royal Tyrrell, he explains the difficulty of currently working with CAT Scanners data. He wrote,

> The file type that is output from a CT scanner is a "diacom"(sic), these need to be converted to either a jpeg or tif in a diacom(sic) converter, and then processed into an image stack in a software called "ImageJ". These image stacks are then uploaded to "AMIRA" (FEI) for processing into 3D models. I'm curious to see if you'll be able to streamline this process.

Dicom Cat Scan Format

Dicom is a huge copyright problem promoted as “the internet of medical information”. According to its standards page, “It defines the formats for medical images that can be exchanged with the data and quality necessary for clinical use. “[32] It is controlled by The Medical Imaging & Technology Alliance (MITA), a division of the National Electrical Manufacturers Association (NEMA). They claim to be the leading organization and collective voice of medical imaging equipment conglomerate.[33] They claim to represent companies whose sales make up more than 90 percent of the global market for advanced imaging technologies. They are not an open forum for digital standards but are created as a profit center for a few select and well placed medical device companies.

There are, nevertheless, some free tools to access some of their formats. On the sourceforge network there is Open Dicom Viewer[34] which is licensed under GNU Library or Lesser General Public License version 3.0 (LGPLv3). It is a Java application (written in Java 1.7).

Another tool is DCMLinux [35], and it is released under the GPL2. It promotes itself as, “a complete PACS system, free of charge. Its core is an Ubuntu 10.04 system fully updated and it contains the

31 https://www6.slac.stanford.edu/community/past-lectures/archaeopteryx-bringing-dino-bird-life 2017 accessed
32 http://dicom.nema.org/Dicom/about-DICOM.html
33 http://www.medicalimaging.org/about-mita/ accessed in April 2017
34 https://sourceforge.net/projects/opendicomviewer/ accessed in April 2017
35 https://sourceforge.net/projects/dcmlinux/ accessed in April 2017

DCM4CHEE as its PACS server. In the near future it will contain many other addons such as Weasis, Oviyam, Care2x, etc."

There is an "OpenSourced" C++ library for dicom with java and python hooks which looks promising for new developers. Called Imebra[36], and licensed under the GPL2, and it's development is current. "The Imebra SDK is a multiplatform, open source, C++ library for handling DICOM files, both raw and compressed." It actually has an impressive set of features and looks like it is worth development. It's main class is the `imebra::DataSet` class and it is fully documented. It is compiled with cmake. You need to include imebra/imebra.h, to your source files to access the library. It uses uniqueptr and can open files like this according to the official documentation:

```
std::unique_ptr<imebra::DataSet> loadedDataSet(imebra::CodecFactory::load("DicomFile.dcm"));
// Retrieve the first image (index = 0)
std::unique_ptr<imebra::Image> image(loadedDataSet->getImageApplyModalityTransform(0));

// Get the color space
std::string colorSpace = image->getColorSpace();

// Get the size in pixels
std::uint32_t width = image->getWidth();
std::uint32_t height = image->getHeight();

// let's assume that we already have the image's size in the variables width and height
// (see previous code snippet)

// Retrieve the data handler
std::unique_ptr<imebra::ReadingDataHandlerNumeric> dataHandler(image->getReadingDataHandler());

for(std::uint32 scanY(0); scanY != height; ++scanY)
{
    for(std::uint32 scanX(0); scanX != width; ++scanX)
    {
        // For monochrome images
        std::int32_t luminance = dataHandler->getSignedLong(scanY * width + scanX);

        // For RGB images
        std::int32_t r = dataHandler->getSignedLong((scanY * width + scanX) * 3);
        std::int32_t g = dataHandler->getSignedLong((scanY * width + scanX) * 3 + 1);
        std::int32_t b = dataHandler->getSignedLong((scanY * width + scanX) * 3 + 2);
    }
}
```

Dicom has a networking layer in the specification, something that PNG also made room for in its specification. This is a template for views an image:

```
 // The transforms chain will contain all the transform that we want to
// apply to the image before displaying it
imebra::TransformsChain chain;

if(imebra::ColorTransformsFactory::isMonochrome(image->getColorSpace())
{
    // Allocate a VOILUT transform. If the DataSet does not contain any pre-defined
    //  settings then we will find the optimal ones.
    VOILUT voilutTransform;

    // Retrieve the VOIs (center/width pairs)
    imebra::vois_t vois = loadedDataSet->getVOIs();

    // Retrieve the LUTs
    std::list<std::shared_ptr<imebra::LUT> > luts;
    for(size_t scanLUTs(0); ; ++scanLUTs)
    {
        try
        {
```

36 https://imebra.com/ accessed in April 2017

```
            luts.push_back(loadedDataSet-
>getLUT(imebra::TagId(imebra::tagId_t::VOILUTSequence_0028_3010), scanLUTs));
        }
        catch(const imebra::MissingDataElementError&)
        {
            break;
        }
    }

    if(!vois.empty())
    {
        voilutTransform.setCenterWidth(vois[0].center, vois[0].width);
    }
    else if(!luts.empty())
    {
        voilutTransform.setLUT(*(luts.front().get()));
    }
    else
    {
        voilutTransform.applyOptimalVOI(image, 0, 0, width, height);
    }

    chain.add(voilutTransform);
}

// If the image is monochromatic then now chain contains the VOILUT transform

// We create a DrawBitmap that always apply the chain transform before getting the RGB image
imebra::DrawBitmap draw(chain);

// Ask for the size of the buffer (in bytes)
size_t requestedBufferSize = draw.getBitmap(image, imebra::drawBitmapType_t::drawBitmapRGBA, 4, 0, 0);

// Now we allocate the buffer and then ask DrawBitmap to fill it
std::string buffer(requestedBufferSize, char(0));
draw.getBitmap(image, imebra::drawBitmapType_t::drawBitmapRGBA, 4, &(buffer.at(0)),
requestedBufferSize);
```

3d Printing and File Formats

Likewise there are copyright and patent issues with 3d printers. A resent visit to Makerbot, the makers of 3d printers who’s offices are located at 1 Metrotech Plaza, in Downtown Brooklyn, proved that they have a rather large Tyrannosaur 3d model that they printed hung right next to the receptionist desk. They have intense interest in getting scans for printing and they are using, according to the presenter, Drew Lentz, STL files (StereoLithography) and the Wavefront .obj file. [37] The good news about these formats is that they are both are open and adopted by numerous free and proprietary software solutions. So everyone is interested, but these systems are generally not talking to each other. CAT Scans are the biggest problem as their graphics formats are different from machine to machine and all are proprietary, even secret formats at times.

Despite this, there is a growing number of CAT Scan and images in archives which have been created and which are publicly available. The University of Texas High-Resolution X-ray Computed Tomography Facility, has DigiMorph.Org, which is, “part of the National Science Foundation Digital Libraries Initiative, develops and serves unique 2D and 3D visualizations of the internal and external structure of living and extinct vertebrates, and a growing number of 'invertebrates.' The Digital Morphology library contains more than a terabyte of imagery of natural history specimens that are

37 Object Files Specification http://www.cs.utah.edu/~boulos/cs3505/obj_spec.pdf, accessed March 2017

important to education and central to ongoing cutting-edge research efforts. The Digital Morphology library site now serves imagery, optimized for Web delivery, for more than 1000 specimens contributed by almost 300 collaborating researchers from the world's premiere natural history museums and universities." Of course, copyright problems hinder this library, and it can benefit from a copyleft license. [38]

Applying Artificial Intelligence and Machine Learning to Fossil Data and Images

With the growing numbers of 3d and 2d images of fossils, the use of artificial intelligence to analyze these resources will revolutionize the field. Understanding some basic principles of AI will be essential for future paleontologists.

There are a number of Artificial Intelligence paradigms that have evolved over the last 60 years. The classic reference is **George Luger – Artificial Intelligence.** We can examine two very different kinds of artificial intelligence systems, "Expert Systems" which can be demonstrated with CLIPS. Such systems gain expertise, accumulate facts and data, and apply them to a series of rules which analyzes the facts. CLIPS uses the Rete Algorithm to perform its search results and syntax. The system might repeat itself, allowing for adaptation, that is learning. A typical example is the nodal learning task that can be adapted for our needs. An excellent text for mastering this basic skill set is "Expert Systems: Principles and Programming" by Joseph C. Giarratano and, Gary D. Riley.[39]

The second methodology which we will explore is neural net and perceptrons. The object here is to use learning algorithms in order to recognize images of fossils for analysis and identification. We can demonstrate the process by showing how systems can be taught to recognize shapes without instruction. We can then discuss means to scale up the process to entire cat scans of fossils.

CLIPS and Inference Engine classification and Decision Tree

Lets look at the inference engine and the expert systems first. One can quickly mock up a standard decision tree that allows for the input of data which is slowly organized into a hierarchical format for discovery. This is a classic algorithm covered in multiple sources.

38 http://digimorph.org/aboutdigimorph.phtml accessed 2017
39 Joseph C. Giarratano, Gary D. Riley, Expert Systems: Principles and Programming, Third Edition, Course Technology, 1998

CLIPS is an inference engine that has its own syntax. It has templates and rules which use pattern matching. It builds internal data structures to enforce its rules. It can interact with the external environment in order to obtain facts that are evaluated by the rules. After applying all its rules on the data, it reaches a state that it can report, and which might represent a result.

With a basic understanding, we can now look at the full code. There are a few things that the novice CLIPS programmer needs to understand.

- At its heart and soul, CLIPS is a huge regular expression engine. In many regards, using it reminds me Unix power tools like GAWK and even Perl. The top of every rule, CLIPS defrules has a pattern that has to be matched according to a specific set of, not nearly obvious, rules.
- On the other hand, CLIPS leans heavily on the functional language paradigm. I think that the authors of CLIPS view its shell like many LISP coders view EMACS.
- Getting started with CLIPS is easy. Learning it well, that can take a long time. It is a fickle programming language and having a good programming text editor is vital to spotting the inevitable lost parenthesis.
- Good programming languages shape the way that you think. CLIPS does this. After knocking your head around with it for a while, you begin to see its roots in the Artificial Intelligence area. Standard textbooks become easier to read and understand. And new possible solutions to problems arise as you develop a sense of what it does.
- **Every time you assert a fact into a scope, the entire scope is reevaluated in full.** You can't say this enough.
- *Learning CLIPS means learning how its inference engine works.*
- CLIPS has the ability to create modules that allow for results to be passed from one scope to another.

In this case, we want a program that tries to determine a species by an accumulations of facts. It will acquire those facts by interacting with individuals. In order to do that, it needs a template for a node that can formulate questions. The syntax rules are available from the CLIPS online documentation.[40]

Each node of our decision tree needs to have a structure that links it with two other nodes, one for an affirmative result, and another for a negative result. It requires space of a question that it asks the user. We can create an adequate node design as follows. It can always be altered later if you decide you need to store more information.

```
(deftemplate node
        (slot name )
        (slot type )
        (slot question )
        (slot yes-node)
        (slot no-node )
```

40 http://clipsrules.sourceforge.net/documentation/v630/ug.pdf accessed March of 2017

```
        (slot answer )
)
```

so in this case, we are creating facts which contain “nodes”. “nodes” are of types answers and questions. They are linked to a yes or a no node, depending on which rules are triggered. We can define out facts as follows:

```
(deffacts tree
      (node (name root)(type decision)
            (question "Does it have sharp teeth?" )
            (yes-node node2) (no-node node1))
      (node (name node1)(type decision)
            (question "Does it have horns?")
            (yes-node node3) (no-node node4))
      (node (name node2) (type answer) (answer tyranosaur))
      (node (name node3) (type answer) (answer triceratops))
      (node (name node4) (type answer) (answer diplodocus.))
      (current-node root)
)
```

So we are creating initial facts which is represent a tree of 5 nodes, 2 of type decision, and 3 of type answers.

The decision nodes are:
root
node1

Answer nodes are
node2
node3
node4

We are also creating a current fact that the current-node is root.

Next, we want to create rules for evaluating facts. Let’s make a rule that asks a question from the user and collects answers. On the top part of the rule is a pattern match. This methodology for developing a decision tree in CLIPS is borrowed from the Rete Algorithm.[41] It has a syntax as follows:

```
(defrule <rule-name> [<comment>]
   [<declaration>]              ; Rule Properties
   <conditional-element>*       ; Left-Hand Side (LHS)
   =>
   <action>*)                   ; Right-Hand Side (RHS)
```

41 Forgy, Charles L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem." Artificial Intelligence. Carnegie-Mellon University, 1979. Web. 23 Mar. 2016. <http://www.csl.sri.com/users/mwfong/Technical/RETE%20Match%20Algorithm%20-%20Forgy%20OCR.pdf>.

Rules (and the rest of CLIPS syntax) have LHS and RHS statements. LHS are matched in some way which cause RHS statements to activate. The list of selected rules form what is called an "agenda" for a module. And a module is a scoped area of a clips programing. Rules are fired, resetting the "agenda". It is notated in the CLIPS documentation that:

> Redefining a currently existing deffacts causes the previous deffacts with the same name to be removed even if the new definition has errors in it. There may be multiple deffacts constructs and any number of facts (either ordered or deftemplate) may be asserted into the initial fact-list by each deffacts construct.
>
> Dynamic expressions may be included in a fact by embedding the expression directly within the fact. All such expressions are evaluated when CLIPS is reset.
>
> Redefining a currently existing defrule causes the previous defrule with the same name to be removed even if the new definition has errors in it. The LHS is made up of a series of conditional elements (CEs) that typically consist of pattern conditional elements (or just simply patterns) to be matched against pattern entities. An implicit and conditional element always surrounds all the patterns on the LHS. The RHS contains a list of actions to be performed when the LHS of the rule is satisfied. In addition, the LHS of a rule may also contain declarations about the rule's properties immediately following the rule's name and comment (see section 5.4.10 for more details). The arrow (=>) separates the LHS from the RHS. There is no limit to the number of conditional elements or actions a rule may have (other than the limitation placed by actual available memory). Actions are performed sequentially if, and only if, all conditional elements on the LHS are satisfied.
>
> If no conditional elements are on the LHS, the rule will automatically be activated. If no actions are on the RHS, the rule can be activated and fired but nothing will happen.

Our defrules look as follows: Notice the implementation of both a left and right side of the rule and the use of variables (demarked with a ?syntax).

```
(defrule ask-decision-node-question
        ?node <- (current-node ?name)
        (node (name ?name)
                        (type decision)
                        (question ?question))
                        (not (answer ?))
=>
        (printout t ?question "(yes or no) ")
        (assert (answer (read)))
)

(defrule bad-answer
        ?answer <- (answer ~yes&~no)
=>
        (retract ?answer)
)
```

The first question reads a node, which must match the type “decision” and loads the ?question field. When it has such a matching node, it prints the question and awaits for yes or no, which is then asserted as the answer. The next rule, bad-answer, retracts the answer from the agenda if it doesn’t match yes or no .

The next rule we establish is designed to be triggered only when there is a match fact which is a node of a type decision.

```
(defrule proceed-to-yes-branch
      ?node <- (current-node ?name)
      (node (name ?name)
      (type decision)
      (yes-node ?yes-branch))
      ?answer <- (answer yes)
=>
      (retract ?node ?answer)
      (assert (current-node ?yes-branch))
)
```

If we match a node of type decision, then we assert that the new current-node to be the value in “?yes-branch” of the matching node.

The next rule we define, defrule, walks the opposite path, down the no node:

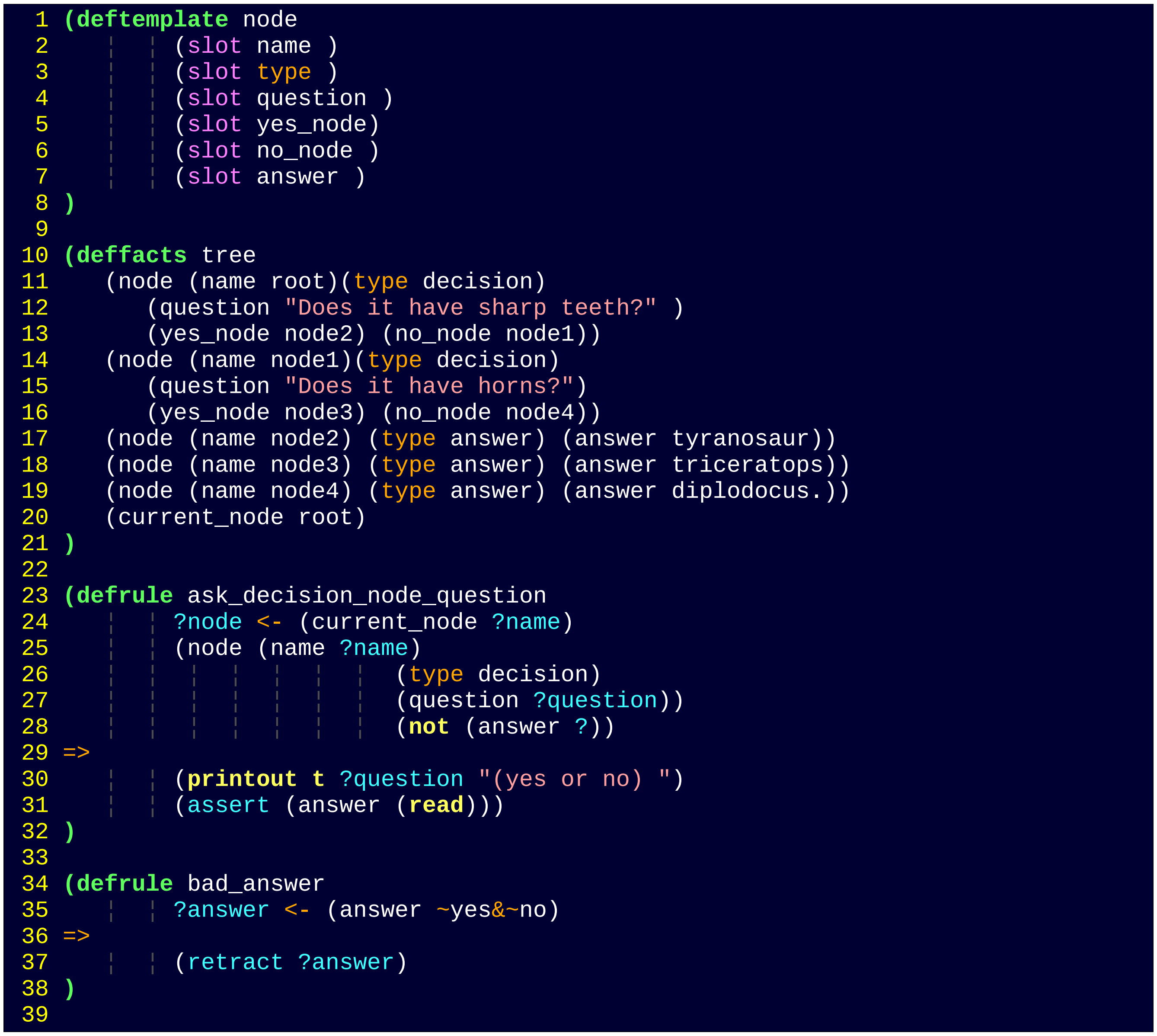
```
(defrule proceed-to-no-branch
      ?node <- (current-node ?name)
      (node (name ?name)
      (type decision)
      (no-node ?no-branch))
      ?answer <- (answer no)
      =>
      (retract ?node ?answer)
      (assert (current-node ?no-branch))
)
```

When we reach a leaf node on the decision making tree we need a rule to see if our result is correct:

```
(defrule correct
      ?node <- (current-node ?name)
      (node (name ?name) (type answer)(answer ?value))
      (not (answer ?))
      =>
      (prinout t "I guess it is a " ?value crlf)
)
```

```
(defrule answer-node-guess-is-correct
      ?node <- (current-node ?name | (node (name ?name) (type answer))
      ?answer <- (answer yes)
=>
      (assert (ask-try-again))
      (retract ?node ?answer))



(defrule answer-node-guess-is-incorrect
      ?node <- (current-node ?name)
      (node (name ?name) (type answer))
      ?answer <- (answer no)
=>
      (assert (replace-answer-node ?name))
      (retract ?node ?answer))
```

A complete working formulation, which is adapted from the example presented by Gary Riley and Joseph Giarratano: in “Expert Systems Principles and Programming,Third Edition” looks like this:[42]



```
(deftemplate node
        (slot name )
        (slot type )
        (slot question )
        (slot yes_node)
        (slot no_node )
        (slot answer )
)

(deffacts tree
   (node (name root)(type decision)
      (question "Does it have sharp teeth?" )
      (yes_node node2) (no_node node1))
   (node (name node1)(type decision)
      (question "Does it have horns?")
      (yes_node node3) (no_node node4))
   (node (name node2) (type answer) (answer tyranosaur))
   (node (name node3) (type answer) (answer triceratops))
   (node (name node4) (type answer) (answer diplodocus.))
   (current_node root)
)

(defrule ask_decision_node_question
        ?node <- (current_node ?name)
        (node (name ?name)
                        (type decision)
                        (question ?question))
                        (not (answer ?))
=>
        (printout t ?question "(yes or no) ")
        (assert (answer (read)))
)

(defrule bad_answer
        ?answer <- (answer ~yes&~no)
=>
        (retract ?answer)
)
```

42 Joseph C. Giarratano, Gary D. Riley, Expert Systems: Principles and Programming, Third Edition, Course Technology, 1998

```
(defrule proceed_to_yes_branch
   ?node <- (current_node ?name)
   (node (name ?name)
   (type decision)
   (yes_node ?yes_branch))
   ?answer <- (answer yes)
=>
   (retract ?node ?answer)
   (assert (current_node ?yes_branch))
)

(defrule proceed_to_no_branch
   ?node <- (current_node ?name)
   (node (name ?name)
   (type decision)
   (no_node ?no_branch))
   ?answer <- (answer no)
   =>
   (retract ?node ?answer)
   (assert (current_node ?no_branch))
)
(defrule correct
   ?node <- (current_node ?name)
   (node (name ?name) (type answer)(answer ?value))
   (not (answer ?))
   =>
   (printout t "I guess it is a " ?value crlf)
)



(defrule answer_node_guess_is_correct
   ?node <- ( current_node ?name )
   ( node (name ?name) (type answer) )
   ?answer <- ( answer yes )
=>
   (assert ( ask_try_again ))
   (retract ?node ?answer)
)


(defrule answer_node_guess_is_incorrect
   ?node <- (current_node ?name)
   (node (name ?name) (type answer))
   ?answer <- (answer no)
=>
   (assert (replace_answer_node ?name))
   (retract ?node ?answer)
)

(defrule ask_if_answer_node_is_correct
   ?node <- (current_node ?name)
   (node (name ?name) (type answer)
   (answer ?value))
   (not (answer ?))
=>
   (printout t "I guess it is a " ?value crlf)
   (printout t "Am I correct? (yes or no) ")
   (assert (answer (read)))
)
```

```
(defrule answer_node_guess_is_correct
   ?node <- (current_node ?name)
   (node (name ?name) (type answer))
   ?answer <- (answer yes)
=>
   (assert (ask_try_again))
   (retract ?node ?answer))

(defrule answer_node_guess_is_incorrect
   ?node <- (current_node ?name)
   (node (name ?name) (type answer))
   ?answer <- (answer no)
=>
   (assert (replace_answer_node ?name))
   (retract ?node ?answer)
)

(defrule ask_try_again
   (ask_try_again)
   (not (answer ?))
=> (printout t "Try again? (yes or no) ")
   (assert (answer (read)))
)

(defrule one_more_time
   ?phase <- (ask_try_again)
   ?answer <- (answer yes)
=>
   (retract ?phase ?answer)
   (assert (current_node root) )
)

(defrule no-more
   ?phase <- (ask_try_again)
   ?answer <- (answer no)
=>
   (retract ?phase ?answer)
   (save-facts "dinos.dat" local node)
)

(defrule replace_answer_node
   ?phase <- (replace_answer_node ?name)
   ?data <- (node (name ?name) (type answer) (answer ?value))
=>
   (retract ?phase)
; Determine what the guess should have been
   (printout t "What is the animal? ")
   (bind ?new_animal (read))
   ; Get the question for the guess
   (printout t "What question when answered yes ")
   (printout t "will distinguish " crlf " a ")
   (printout t ?new_animal " from a " ?value "? ")
   (bind ?question (readline))
   (printout t "Now I can guess " ?new_animal crlf)
   ; Create the new learned nodes
   (bind ?newnode1 (gensym*) )
   (bind ?newnode2 (gensym*))
   (modify ?data (type decision)
      (question ?question)
      (yes_node ?newnode1)
```

```
        (no_node ?newnode2)
     )
     (assert (node (name ?newnodel)
        (type answer)
        (answer ?new_animal)
        )
     )
     (assert (node (name ?newnode2)
        (type answer)
        (answer ?value)
        )
     )
; Determine if the player wants to try again
     (assert (ask_try_again))
)
```

The running output looks like this. As we can see, the system does learn and build a database, but this method is slow. It can be adapted to pull into comma separated tables of information, if it is thought out carefully. Systems like EXPECT[43] used to do this with PTP connections and chats for dial up and analog modems.

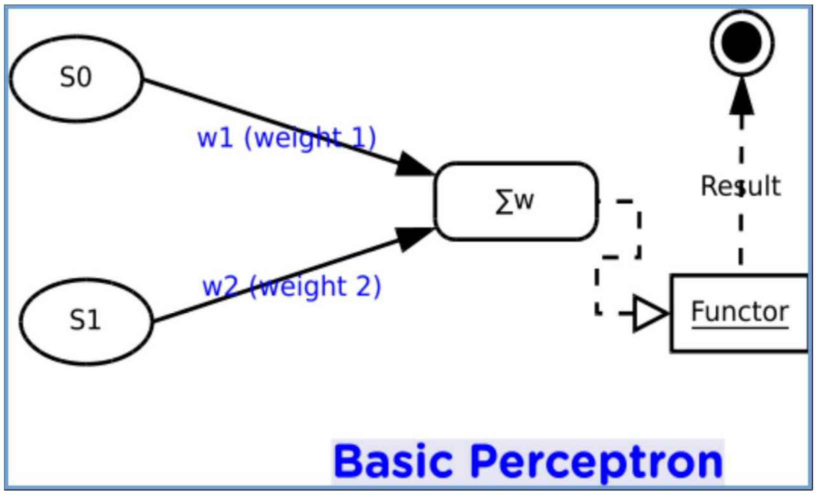
```
 [ruben@flatbush Documents]$ clips -f ./species.clps
        CLIPS (6.30 3/17/15)
  …

; Determine if the player wants to try again
        (assert (ask_try_again))
)
CLIPS> (run)
CLIPS> (reset)
CLIPS> (run)
Does it have sharp teeth?(yes or no) no
Does it have horns?(yes or no) yes
I guess it is a triceratops
I guess it is a triceratops
Am I correct? (yes or no) no
What is the animal? Stegasaurus
What question when answered yes will distinguish
 a Stegasaurus from a triceratops? does it have spikes on its tail
Now I can guess Stegasaurus
Try again? (yes or no) yes
Does it have sharp teeth?(yes or no) no
Does it have horns?(yes or no) yes
does it have spikes on its tail(yes or no) no
I guess it is a triceratops
I guess it is a triceratops
Am I correct? (yes or no) no
```

43 EXPECT http://www.tcl.tk/man/expect5.31/expect.1.html accessed April 2017

```
What is the animal? Achelousaurus
What question when answered yes will distinguish
 a Achelousaurus from a triceratops? spikes on the frill
Now I can guess Achelousaurus
Try again? (yes or no) yes
Does it have sharp teeth?(yes or no) no
Does it have horns?(yes or no) yes
does it have spikes on its tail(yes or no) yes
I guess it is a Stegasaurus
I guess it is a Stegasaurus
Am I correct? (yes or no) yes
Try again? (yes or no) yes
Does it have sharp teeth?(yes or no) no
Does it have horns?(yes or no) yes
does it have spikes on its tail(yes or no) no
spikes on the frill(yes or no) yes
I guess it is a Achelousaurus
I guess it is a Achelousaurus
Am I correct? (yes or no) yes
Try again? (yes or no) no
```

Perceptrons and Learned Pattern Recognition

Artificial Intelligence, and all intelligence, actually, requires feedback from stimulation. Attempts have been made to create artificial software systems that can reproduce the properties of our biological neural system. In truth, I think that much of what we have learned about intelligence in living things has outstripped these models, but nevertheless, many decent proposals have evolved with usable results which are being unleashed in the world. Most of this can be classified as machine learning. For example, with the Hough Transformation, we program our systems with basic logic needed to recognize and identify shapes. But do we have to? Can we create systems where the machine, through feedback and regulated change, can find shapes and learn to do that on their own, without the math? This question has been explored for decades, and the answer is yes. There are a wide range of questions where machines can learn from repeated patterns by themselves. These systems are being explored in self-driving cars, walking robots, and robotic vacuums. They can learn, essentially, by trial and error.



Work on this was started as early as 1943 with McCulloch an Pitts[44]. Perceptrons are a process that has inputs, side effects and outputs. It’s not very different, despite all the advanced mathematical symbols about it in discussion, from the models we have seen elsewhere and everywhere.

You have two sensors tagged S0 and S1. They receive input signals of some sort. They are connected to a *summation node* which is connected to the sensors through a weighted multiplier. The *summation* of weights is then sent to a *functor* which determines if the signal is “on” or “off”, which is passed as a result.

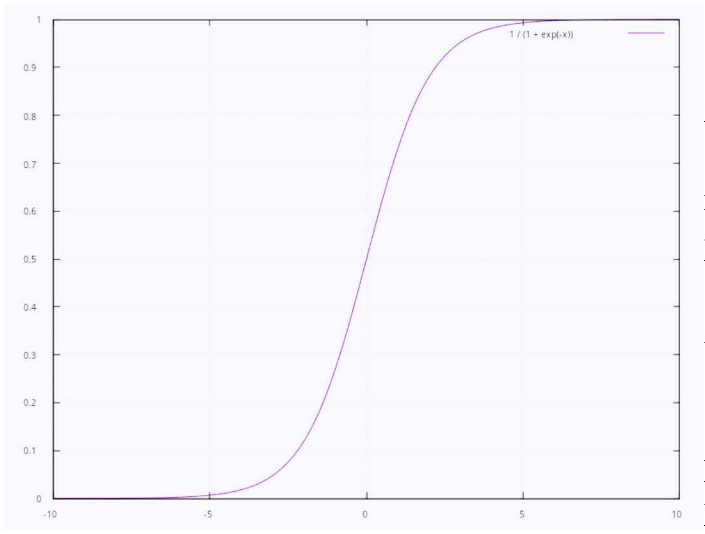
The summation node (∑w) adds together the weights of all the attached sensors. The result of this (which is termed “dot multiplication” in vector math), is then put through a step function which we can define as follows:

```
f(x) = { 0 if x < t, 1 if x ≥ t }
     where t is an arbitrary threshold
```

Step equations can be difficult to deal with mathematically but we can readily substitute them with a near approximation with a sigmoid function that looks like this:

$$f(x)=\frac{1}{1+e^{-x}}$$

44 McCulloch and Pitts, “A Logical Calculus of the Idea Immanent in Nervous Activity” , 1943, Bulletin of Mathematical Biophysics Vol. 5, pg 115-133



This gives us a nice binary result in most cases. Perceptrons also get a dummy input with an input of 1 (Sψ = 0) with its own weight. A single perceptron might not seem so exciting, but when multiple perceptrons are put together in a variety of arrays and organizations, and when linked into feedbacks, they are capable of extraordinary feats of machine learning. One of the things they do most well is pattern recognition of graphics, which is perfect for fossil identification from CAT scans for which we have libraries being built up.

A basic perceptron can be worked up in C++ rather quickly and we can build such a library and link them together. I would build a perceptron object to look like the following sample code:

```
/*
 * =====================================================================================
 *
 *       Filename:  perceptron.h
 *
 *    Description:  Basic Unit for AI machine learning
 *
 *        Version:  1.0
 *        Created   Mon Mar 27 22:33:19 EDT 2017
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Ruben Safir (mn), ruben@mrbrklyn.com
 *        Company:  NYLXS Inc - LIU Thesis
 *
 * =====================================================================================
 */

#ifndef PERCEPTRON_H
#define PERCEPTRON_H
#include<iostream>
#include<string>
#include<vector>
namespace tree {

    /*
     * =====================================================================================
     *        Class:  Perceptron
```

```
      ¦*  Description:  This class describes a single node and its data
      ¦*
=============================================================================
==
      ¦*/
      template < class T, class W >
         class Perceptron
         {
            public:

               /* ====================  LIFECYCLE
======================================= */
               Perceptron (T s1, T s2, W w1, W w2, T thresh): sensor_1{s1},
sensor_2{s2}, weight_1{w1}, weight_2{w2}, threshold{thresh} {
                  weighted_sum();
               };                              /* constructor */

               /* ====================  ACCESSORS
======================================= */
               T get_sum()
               {
                  return sum;
               };
               T get_threshold()
               {
                  return threshold;
               };
               /* ====================  MUTATORS
======================================= */
               int functor()
               {
                  if(sum < get_threshold())
                  {
                     return 0;
                  }else{
                     return 1;
                  }
               }
               /* ====================  OPERATORS
======================================= */
               /* ====================  DATA MEMBERS
======================================= */
               T data;
            protected:

            private:
               T sensor_1;
               T sensor_2;
               W weight_1;
               W weights_2;
               T threshold;
               T sum;
               T weighted_sum()
               {
                  sum = (sensor_1 * weight_1) + (sensor_2 * weight_1);
                  return sum;

               };

         }; /* ----------  end of template class Perceptron  ---------- */
```

```


}//close of namespace tree

#endif
```

This is a basic set up for a single Perceptron. A single Perceptron can imitate a logic circuit fairly easily. A logic circuit takes two binary inputs and results in a single output. For example, if we want a logical AND then we can create a truth table that looks like this:

0=F, 1=T	AND	Truth Table
	0	1
0	0	0
1	0	1

```
/*
 *
=====================================================================================
==
 *
 *       Filename:  perceptron.h
 *
 *    Description:  Basic Unit for AI machine learning
 *
 *        Version:  1.0
 *        Created   Mon Mar 27 22:33:19 EDT 2017
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Ruben Safir (mn), ruben@mrbrklyn.com
 *        Company:  NYLXS Inc - LIU Thesis
 *
 *
=====================================================================================
==
 */

#ifndef PERCEPTRON_H
#define PERCEPTRON_H
#include<iostream>
#include<string>
#include<vector>
namespace network {

    /*
   *
=====================================================================================
==
   *        Class:  Perceptron
   *  Description:  This class describes a single node and its data
   *
=====================================================================================
==
   */
    template < class T, class W >
      class Perceptron
      {
        public:
            /* ====================  LIFECYCLE
======================================= */
            Perceptron ( std::vector<T>  s1, std::vector <W> w1, float thresh ):
sensors{s1}, weights{w1}, threshold{thresh} {
              sensors.push_back(1);
              weighted_sum();
            };                              /* constructor */

            /* ====================  ACCESSORS
======================================= */
            W get_sum()
            {
//              std::cout << "Called get sum"  << std::endl;
              return sum;
            };

            float get_threshold()
            {
              return threshold;
            };

            /* ====================  MUTATORS
======================================= */
            int functor()
            {
              if(sum < get_threshold())
              {
                return 0;
              }else{
                return 1;
              }
            }
         /* ====================  OPERATORS =======================================
*/
        /* ==================== DATA MEMBERS
======================================= */
            T data;
        protected:

        private:
            std::vector<T> sensors;
            std::vector<W> weights;
            float threshold;
            float sum = 0;
            float weighted_sum()
            {
              if( sensors.size()  != weights.size() )
              {
                std::cerr << __LINE__ << " You broke my perceptron by failing to
give a weight for the bias";
                exit(1);
              }
              for (int i = 0; i < static_cast<int> ( sensors.size()) ; i++)
              {
                std::cout << "sensor =>" << sensors[i] << std::endl;
                std::cout << "weight =>" << weights[i] << std::endl;
                sum += weights[i] * sensors[i];
//                std::cout << "sum =>" << sum << std::endl;
//                std::cout << "get sum =>" << get_sum() << std::endl;

              };
              return sum;
            };



      }; /* ----------  end of template class Perceptron  ---------- */


}//close of namespace network

#endif
```

We switched the parameters to vectors so that they can be more flexible and we attached to our Perceptron object with methods to do the dot multiplication for sensors and added the bias. We can run out Perceptrons through testing by emulating the logic circuits.

```
/*
 *
=====================================================================================
==
 *
 *       Filename:  learn.cpp
 *
 *    Description:  Learn with a Neuronet and Perceptrons
 *
 *        Version:  1.0
 *        Created:  03/29/2017 04:14:13 PM
 *       Revision:  none
 *       Compiler:  gcc
 *
 *         Author:  Ruben Safir (mn), ruben@mrbrklyn.com
 *        Company:  NYLXS Inc
 *
 *
=====================================================================================
==
 */
#include "perceptron.h"
const float LEARN_RATE = 0.2;
const int PRECISION = 3;
const int NUMWIDTH = PRECISION + 3;

void tt(std::vector<int> truth_table ){
        std::cout << std::endl << "__Truth Table__" << std::endl;
        for(auto x : truth_table)
        {
                std::cout << x << "\t";
        }
        std::cout << std::endl;
}
int main (int argc, char ** argv )
{
        std::vector<int> * sensor_inputs[4] = {
                new std::vector<int> {0,0},
                new std::vector<int> {0,1},
                new std::vector<int> {1,0},
                new std::vector<int>{1,1}
        };

        std::vector<float> weight_inputs = { 0.4, 0.4, 0.0 };
        std::vector<int> truth_table(4);
        for(int i = 0; i<4; i++){
                network::Perceptron <int, float> p_and
(*sensor_inputs[i],weight_inputs, 0.5);
                std::cout << "AND sum =>" << p_and.get_sum() << std::endl <<
std::endl;
                std::cout << "AND threshold =>" << p_and.get_threshold() <<
std::endl;
                std::cout << "AND Functor =>" << p_and.functor() << std::endl <<
std::endl;
                truth_table[i] = p_and.functor();
        }

        tt(truth_table);

        weight_inputs[0] = 0.6;
        weight_inputs[1] = 0.6;
        weight_inputs[2] = 0.0;
        for(int i = 0; i<4; i++){
                network::Perceptron <int, float> p_or
(*sensor_inputs[i],weight_inputs, 0.5);
                std::cout << "OR sum =>" << p_or.get_sum() << std::endl;
                std::cout << "OR threshold =>" << p_or.get_threshold() <<
std::endl;
                std::cout << "OR Functor =>" << p_or.functor() << std::endl;
                truth_table[i] = p_or.functor();
        }
        tt(truth_table);

        weight_inputs[0] = -0.4;
        weight_inputs[1] = -0.4;
        weight_inputs[2] = 0.0;
        for(int i = 0; i<4; i++){
                network::Perceptron <int, float> p_or
(*sensor_inputs[i],weight_inputs, -0.5);
                std::cout << "NAND sum =>" << p_or.get_sum() << std::endl;
                std::cout << "NAND threshold =>" << p_or.get_threshold() <<
std::endl;
                std::cout << "NAND Functor =>" << p_or.functor() << std::endl;
                truth_table[i] = p_or.functor();
        }
        tt(truth_table);

        return 1;
}
```

We created a truth table called TT which we feeding into four different Perceptron objects, one of AND, NAND and OR. Adjusting the weights, and thresholds for each Percetron gives us a desired outcome. Of course, this is all fixed and we did all the intelligence. The system becomes more exciting when the Perceptrons are are put in arrays and the weights are fed back to the system depending on the result.

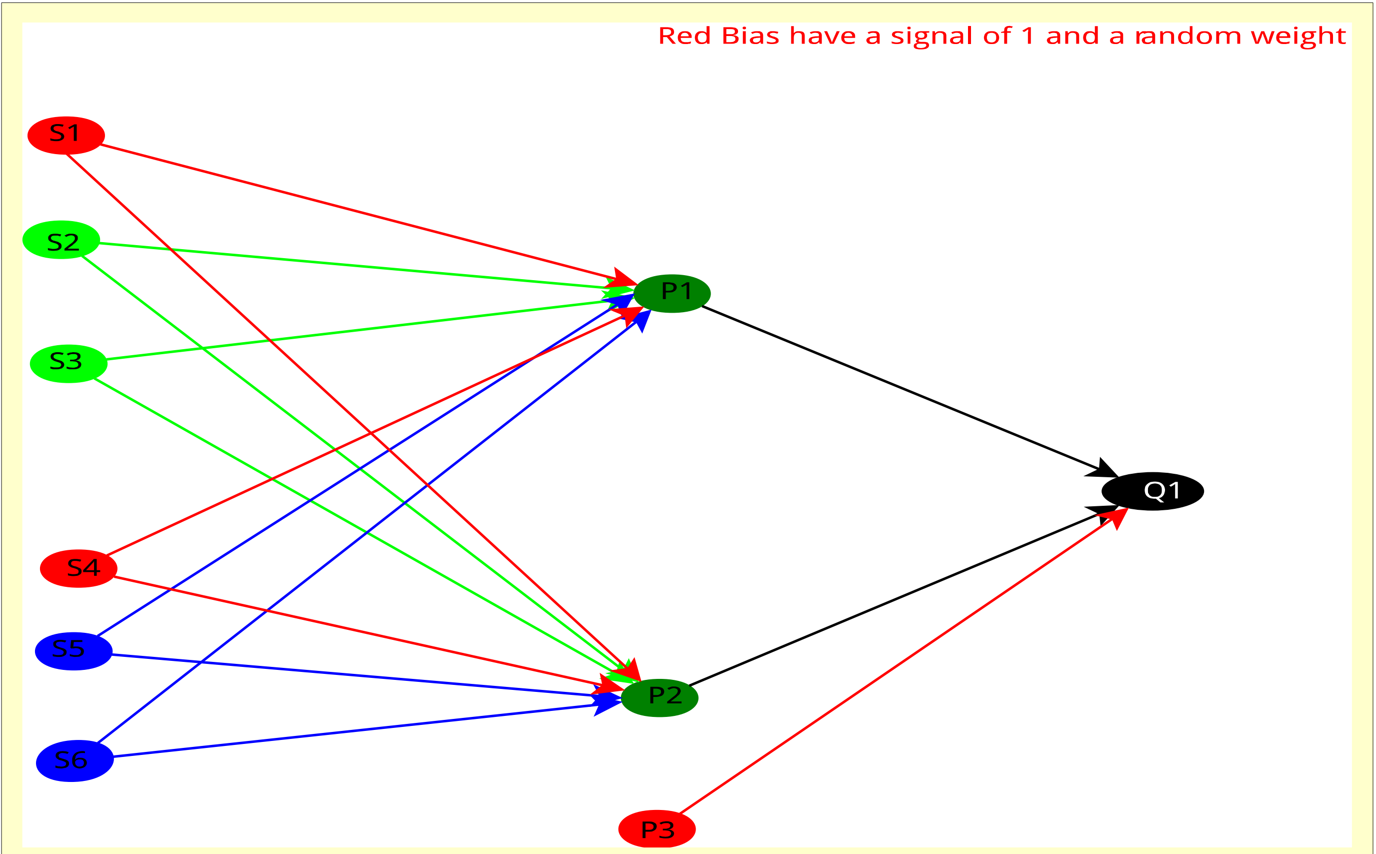
The outcome of our testing programming looks as follows:

```
sensor =>0
weight =>0.4
sensor =>0
weight =>0.4
sensor =>1
weight =>0
AND sum =>0

AND threshold =>0.5
AND Functor =>0

sensor =>0
weight =>0.4
sensor =>1
weight =>0.4
sensor =>1
weight =>0
AND sum =>0.4

AND threshold =>0.5
AND Functor =>0

sensor =>1
weight =>0.4
sensor =>0
weight =>0.4
sensor =>1
weight =>0
AND sum =>0.4

AND threshold =>0.5
AND Functor =>0

sensor =>1
weight =>0.4
sensor =>1
weight =>0.4
sensor =>1
weight =>0
AND sum =>0.8

AND threshold =>0.5
AND Functor =>1


__Truth Table__
0     0     0     1
sensor =>0
weight =>0.6
sensor =>0
weight =>0.6
sensor =>1
weight =>0
OR sum =>0
OR threshold =>0.5
OR Functor =>0
sensor =>0
weight =>0.6
sensor =>1
weight =>0.6
sensor =>1
weight =>0
OR sum =>0.6
OR threshold =>0.5
OR Functor =>1
sensor =>1
weight =>0.6
sensor =>0
weight =>0.6
sensor =>1
weight =>0
OR sum =>0.6
OR threshold =>0.5
OR Functor =>1
sensor =>1
weight =>0.6
sensor =>1
weight =>0.6
sensor =>1
weight =>0
OR sum =>1.2
OR threshold =>0.5
OR Functor =>1

__Truth Table__
0     1     1     1
sensor =>0
weight =>-0.4
sensor =>0
weight =>-0.4
sensor =>1
weight =>0
NAND sum =>0
NAND threshold =>-0.5
NAND Functor =>1
sensor =>0
weight =>-0.4
sensor =>1
weight =>-0.4
sensor =>1
weight =>0
NAND sum =>-0.4
NAND threshold =>-0.5
NAND Functor =>1
sensor =>1
weight =>-0.4
sensor =>0
weight =>-0.4
sensor =>1
weight =>0
NAND sum =>-0.4
NAND threshold =>-0.5
NAND Functor =>1
sensor =>1
weight =>-0.4
sensor =>1
weight =>-0.4
sensor =>1
weight =>0
NAND sum =>-0.8
NAND threshold =>-0.5
NAND Functor =>0

__Truth Table__
1     1     1     0
```

When perceptrons are stacked in a multi-leveled they become more dynamic in their behavior. A single perceptron is cable of 2 outcomes, and any system it analyzes is limited to some version of a step function adjusted by the weighted average of its inputs. Regardless of the domain of inputs from sensors or the number, it still can only chart a step response to adjustments of its weights. However, when linked together with full connectivity, it can have a much broader graph of responses. A schematic of our network might look like the figure below. What is important to note that the biases, marked in red, are usually set to a 1, and the weights can vary. We can greatly expand this model to allow for image recognition. Image recognition is the first step to automated analysis by artificial intelligence. In work by Michael A. Nielsen [45][46], he used handwriting samples are are 28 by 28 pixel images.

45 Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015

46 Raiko, Valpola, LeCun, “Deep Learning Made Easier by Linear Transformations in Perceptrons”, Proceedings of the 15th International Conference on Artificial Intelligence and Statistics (AISTATS) 2012, La Palma, Canary Islands. Volume XX of JMLR: W&CP XX. C, 2012



Fully Connected Neural Network

For each image, for which there is 784 pixels, we apply a sensor to each pixel data. Then add a middle layer of percetrons, with 15 perceptrons, which then outputs to 10 perceptron nodes that represents a digit, each node representing a digit from 0-9. If yes then we have an image of a '3', we can ascribe to the network random weights for both levels and read the bytes. We have a target of {0,0,0,1,0,0,0,0,0,0}. When we run our network, we get results that will likely differ from this target. We calculate the difference between our result and the target, which we can call delta, and apply a function of that difference to to our weights, attempting to cause a correction. Then we can run the analysis again until we approach a correct answer for the image, and then train the next image. It is important to run analysis on a batch of images until they are correctly identified. The result is that our perceptron network can recognize hand written numbers. And by expanding this concept, we identify scanned images of fossils for identification.

How to calculate the delta and adjust for error is a matter of much discussion and mathematics. Neilson uses a summation method that is described as follows:

What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x. To quantify how well we're

> achieving this goal we define a *cost function (*Sometimes referred to as a *loss* or *objective* function.) ...
>
> $C(w,b) \equiv 1/(2n) \sum_x \| y(x) - a \|^2$.
>
> number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x. Of course, the output a depends on x, w and b, but to keep the notation simple I haven't explicitly indicated this dependence. The notation $\|v\|$ just denotes the usual length function for a vector v. We'll call C the *quadratic* cost function; it's also sometimes known as the *mean squared error* or just *MSE*. Inspecting the form of the quadratic cost function, we see that $C(w,b)$ is non-negative, since every term in the sum is non-negative. Furthermore, the cost $C(w,b)$ becomes small, i.e., $C(w,b) \approx 0$, precisely when $y(x)$ is approximately equal to the output, a, for all training inputs, x. So our training algorithm has done a good job if it can find weights and biases so that $C(w,b) \approx 0$. By contrast, it's not doing so well when $C(w,b)$ is large - that would mean that $y(x)$ is not close to the output a for a large number of inputs. So the aim of our training algorithm will be to minimize the cost $C(w,b)$ as a function of the weights and biases. In other words, we want to find a set of weights and biases which make the cost as small as possible. We'll do that using an algorithm known as *gradient descent*.[47]

It is worth noting that Dr Christopher League has published in his gitlab archive a slightly different method for determining delta and applying it which looks like this:[48]

```
float delta_rule(perceptron& p, vector<float>& input, float target)
{
  float output = feed_forward(p, input);
  float error = target - output;
  for(unsigned i = 0; i < p.weights.size(); i++) {
    float delta = LEARN_RATE * error * input.at(i);
    p.weights[i] += delta;
  }
  return error;
}

float train_one_round(perceptron&p, vector<vector<float>>& inputs, const
vector<float>& targets)
{
  assert(inputs.size() == targets.size());
  float error = 0;
  for(unsigned i = 0; i < inputs.size(); i++) {
    error += fabs(delta_rule(p, inputs[i], targets[i]));
  }
  error /= inputs.size();
  cout << " err = " << setw(NUMWIDTH) << error << "  " << p << '\n';
  return error;
}
```

47 Ibid. chapter 1
48 https://gitlab.liu.edu/cs691s17/public/blob/master/perceptrons/perceptrons.cc

We obtain output from feeding the perceptron with its existing threshold and weights a sensor input to obtain an summation result. The error itself is then reported as an average. We keep doing this until our error approaches a reasonable limit, and then stop.

Summary and Conclusions

Within a couple of months prior to the publication of this paper a paper was released that shook up the Paleontology world. Using the existing software available the authors proposed that the phylogeny of the base clades of Dinosauria was perhaps wrong. [49] Traditionally, and one can see this when you visit the New York Museum of Natural History where the halls are segregated by these clades, the class is divided by the shape of the hip, with theropods, and sauropods grouped together, and Ornithischia, including groups like triceratops, duckbills like Edmontosaurus, and in the others.

> For 130 years, dinosaurs have been divided into two distinct clades—Ornithischia and Saurischia. Here we present a hypothesis for the phylogenetic relationships of the major dinosaurian groups that challenges the current consensus concerning early dinosaur evolution and highlights problematic aspects of current cladistic definitions. Our study has found a sister-group relationship between Ornithischia and Theropoda (united in the new clade Ornithoscelida), with Sauropodomorpha and Herrerasauridae (as the redefined Saurischia) forming its monophyletic outgroup. This new tree topology requires redefinition and rediagnosis of Dinosauria and the subsidiary dinosaurian clades. In addition, it forces re-evaluations of early dinosaur cladogenesis and character evolution, suggests that hypercarnivory was acquired independently in herrerasaurids and theropods, and offers an explanation for many of the anatomical features previously regarded as notable convergences between theropods and early ornithischians.

This lead to some difficult conversation among experts on phylogenies on the Dinosaur mailing list out of the University of Southern California (dinosaur-l@usc.edu), and on usenet. Much of it got right down to the brass tacks of how these relationships are computed. Discussions questioned the basics of the terms “cost” and “model”. David Marjanovic wrote on the topic:

```
"Modeling"?

What modeling?

The distinctive feature of the method called "parsimony" or
"maximum parsimony" (a somewhat misleading name) is that it
doesn't use a model of evolution at all, if that's what you
mean. It's nonparametric. That's the method Baron et al.
used; it's all TNT does, actually (unlike PAUP*, which can
also use distance methods and maximum likelihood).
```

49 Matthew G. Baron, David B. Norman & Paul M. Barrett (2017)
A new hypothesis of dinosaur relationships and early dinosaur evolution.
Nature 543: 501–506 doi:10.1038/nature21700

> Would you like to talk about the advantages and disadvantages of "maximum parsimony" and model-based methods? There's an interesting conversation to be had there.

Further conversations involved costs, neutral transformations, genetic drift, and the terminology was repeatedly a problem. For example, note this discussion of costs by Dr R. Sean Norman of the University of South Carolina:

> You talk about the cost of a genetic or phenotypic change. What is more appropriate is to talk about the fitness of the organism that has that change. High costs combined with even higher reproductive advantage still confer greater fitness value than low cost with even lower reproductive advantage. Cost is only one factor in life. People do buy fantastically expensive consumer goods when far cheaper ones will serve the function just as well or even better. [50]

Both of these quotations demonstrate just how much distance there is now between the biologists and paleontologists in the field and the mathematicians and computer scientists who are compiling the theory and producing the technology that is being used in the field. Central to these misunderstandings is the need for deep rooted and profound understanding of graph theory and costs. The shortest path theorems are being overrun in favor for parsimony models and statistical models, all of which have a place in analysis, but compose only part of a larger mathematical picture. This problem of focus in a model causing a constraint of vision for the purposes of problem solving is not uncommon, and certainly not uncommon in computer sciences. The most famous example that I can think of, and one directly related to discussion of the proposed phylogeny classification changes in the Dinosauria root is the story by Trey Harris of the 520 mile email problem, which I will republish here with permission:

From trey@sage.org Fri Nov 29 18:00:49 2002
Date: Sun, 24 Nov 2002 21:03:02 -0500 (EST)
From: Trey Harris <trey@sage.org>
To: sage-members@sage.org
Subject: The case of the 500-mile email (was RE: [SAGE] Favorite impossible task?)

Here's a problem that *sounded* impossible... I almost regret posting the story to a wide audience, because it makes a great tale over drinks at a conference. :-) The story is slightly altered in order to protect the guilty, elide over irrelevant and boring details, and generally make the whole thing more entertaining.

I was working in a job running the campus email system some years ago when I got a call from the chairman of the statistics department.

50 https://groups.google.com/forum/#!original/sci.bio.paleontology/pvwttT99FSw/PNNmpPoqAQAJ viewed March, 2017

"We're having a problem sending email out of the department."

"What's the problem?" I asked.

"We can't send mail more than 500 miles," the chairman explained.

I choked on my latte. "Come again?"

"We can't send mail farther than 500 miles from here," he repeated. "A little bit more, actually. Call it 520 miles. But no farther."

"Um... Email really doesn't work that way, generally," I said, trying to keep panic out of my voice. One doesn't display panic when speaking to a department chairman, even of a relatively impoverished department like statistics. "What makes you think you can't send mail more than 500 miles?"

"It's not what I *think*," the chairman replied testily. "You see, when we first noticed this happening, a few days ago--"

"You waited a few DAYS?" I interrupted, a tremor tinging my voice. "And you couldn't send email this whole time?"

"We could send email. Just not more than--"

"--500 miles, yes," I finished for him, "I got that. But why didn't you call earlier?"

"Well, we hadn't collected enough data to be sure of what was going on until just now." Right. This is the chairman of *statistics*. "Anyway, I asked one of the geostatisticians to look into it--"

"Geostatisticians..."

"--yes, and she's produced a map showing the radius within which we can send email to be slightly more than 500 miles. There are a number of destinations within that radius that we can't reach, either, or reach sporadically, but we can never email farther than this radius."

"I see," I said, and put my head in my hands. "When did this start? A few days ago, you said, but did anything change in your systems at that time?"

"Well, the consultant came in and patched our server and rebooted it. But I called him, and he said he didn't touch the mail system."

"Okay, let me take a look, and I'll call you back," I said, scarcely believing that I was playing along. It wasn't April Fool's Day. I tried to remember if someone owed me a practical joke.

I logged into their department's server, and sent a few test mails. This was in the Research Triangle of North Carolina, and a test mail to my own account was delivered without a hitch. Ditto for one sent to Richmond, and Atlanta, and Washington. Another to Princeton (400 miles) worked.

But then I tried to send an email to Memphis (600 miles). It failed. Boston, failed. Detroit, failed. I got out my address book and started trying to narrow this down. New York (420 miles) worked, but Providence (580 miles) failed.

I was beginning to wonder if I had lost my sanity. I tried emailing a friend who lived in North Carolina, but whose ISP was in Seattle. Thankfully, it failed. If the problem had had to do with the geography of the human recipient and not his mail server, I think I would have broken down in tears.

Having established that--unbelievably--the problem as reported was true, and repeatable, I took a look at the sendmail.cf file. It looked fairly normal. In fact, it looked familiar.

I diffed it against the sendmail.cf in my home directory. It hadn't been altered--it was a sendmail.cf I had written. And I was fairly certain I hadn't enabled the "FAIL_MAIL_OVER_500_MILES" option. At a loss, I telnetted into the SMTP port. The server happily responded with a SunOS sendmail banner.

Wait a minute... a SunOS sendmail banner? At the time, Sun was still shipping Sendmail 5 with its operating system, even though Sendmail 8 was fairly mature. Being a good system administrator, I had standardized on Sendmail 8. And also being a good system administrator, I had written a sendmail.cf that used the nice long self-documenting option and variable names available in Sendmail 8 rather than the cryptic punctuation-mark codes that had been used in Sendmail 5.

The pieces fell into place, all at once, and I again choked on the dregs of my now-cold latte. When the consultant had "patched the server," he had apparently upgraded the version of SunOS, and in so doing *downgraded* Sendmail. The upgrade helpfully left the sendmail.cf alone, even though it was now the wrong version.

It so happens that Sendmail 5--at least, the version that Sun shipped, which had some tweaks--could deal with the Sendmail 8 sendmail.cf, as most of the rules had at that point remained unaltered. But the new long configuration options--those it saw as junk, and skipped. And the sendmail binary had no defaults compiled in for most of these, so, finding no suitable settings in the sendmail.cf file, they were set to zero.

One of the settings that was set to zero was the timeout to connect to the remote SMTP server. Some experimentation established that on this particular machine with its typical load, a zero timeout would abort a connect call in slightly over three milliseconds.

An odd feature of our campus network at the time was that it was 100% switched. An outgoing packet wouldn't incur a router delay until hitting the POP and reaching a router on the far side. So time to connect to a lightly-loaded remote host on a nearby network would actually largely be governed by the speed of light distance to the destination rather than by incidental router delays.

Feeling slightly giddy, I typed into my shell:

```
$ units
1311 units, 63 prefixes

You have: 3 millilightseconds
You want: miles
        * 558.84719
        / 0.0017893979
```

"500 miles, or a little bit more."

The issue here is understanding costs and basic algorithms. Working so intently with Networking paradigms it becomes difficult to imagine a real event which could affect the basic algorithms which it was based on. Specifically here, a bug in the software created a cost burden which produced a result which was seemingly impossible under network theory and practice. Can the same thing be true for the morphology of the hip of dinosaurs. I think so. I know it is possible, but the answer is unknown. Hopefully this paper discloses some of the many issues and questions that face the field in the near future. But the solutions to specific problems are delightfully beyond the scope of this paper.

Final Notes:

This paper was written with Free Software as defined by the Free Software Foundation. Software used includes:

- LibreOffice
- GIMP
- DIA
- GCC
- Look
- GVIM
- Apache
- Okular
- make

- pan

I’d like to give a tip of the hat to my friend Richard Stallman, and John Tennant for their inspirational work in bringing education and light to humanity.

Please see http://openscience.org/

Please see the Royal Tyrrell Museum in Drumheller, Alberta Canada and enjoy their online lecture series.

The Dinosaur Mailing list is at http://dml.cmnh.org/about.html

usenet paleontology and programming resources: sci.bio.paleontology, comp.lang.c++, and sci.math

Suggested Future Projects:

1. Audit of current software such as TNT for debugging and algorithm verification
2. Bayesian analysis and its place within Shortest Path Analysis
3. Image recognition of Scanned Images
4. DICOM software development and creating a batch program to go from DICOM to PNG to STL for 3d printing
5. R programming development for Stepwise analysis from raw data dumps
6. Perceptron Neural Net on Fossil Image Recognition
7. Parallel Processing implementation of Sankoff with the new threads library in C++