PEW RESEARCH PROPOSAL

1. **Title of the project.**

   Using domain-specific heuristics to search for optimal and "interesting" paths in a genealogy tree

2. **Statement of the end product(s).**

   - Make the tool available to the public through github; since it works with the standard gedcom format, it can be used by anyone working with genealogy, no matter which specific app they use.
   - Present the research at the *Annual National Genealogical Society Conference* in 2019. This is one of the premium genealogy conferences in the United States. The National Genealogical Society's goal is to further the academic legitimacy of genealogy research.
   - Submit an article to the *Journal of the Society of Genealogists*, one of the premier publications of the discipline, first published in 1925.

3. **Explanation of the scholarly activity.**

I. Statement of the Scholarly Activity

The goal of this research is to design an algorithm to compute the relationship(s) between any two people in a family tree. The intent is to use domain-specific heuristics to speed up the search for large trees by preventing redoing previous computations. A second objective is to discern which alternative paths are "interesting" and which are redundant, and to write Prolog rules to filter out the latter.

II. Description of the activity and its goals

Although traversing trees is a well-researched problem, family trees are really more like graphs with loops and no single root. Unlike tree traversal, **graph traversal** may require that some vertices be visited more than once. As graphs become denser, this redundancy becomes more prevalent, causing computation time to increase. In fact, an algorithm that find an optimal solution that works for any graph (*Travelling Salesman Problem*) is still an open-ended question, and an example of a famous NP hard problem.

Because of the computational complexity, professional programs like *myheritage.com* don't display relationships between two random people in a family tree if there are more than 5,000 nodes. There is a wonderful *gedcom4j* java library to parse the standard database format for genealogy data, but even its *RelationshipCalculator* class fails to consistently find the shortest paths.

To tackle this problem, I intend to use **heuristics**, which is an approach to problem solving that employs a practical method not guaranteed to be optimal or perfect, but sufficient for the immediate goals (i.e. not all redundant computations will be avoided, but a sufficient number to be able to tackle the problem in a reasonable time). An example of such a technique is the *branch-and-bound* heuristic often used in game-trees, which abandons a partial path if it would lead to a sub-optimal solution. **Domain-specific** means that these optimizing heuristics take advantage of idiosyncrasies of the target field that would not hold for general graphs – for example traversing a child subtree from the father node should not be repeated when re-visiting the child node from the mother side.

The programming language chosen for this research is a specialized language called **Prolog**. Prolog is a fifth-generation language with roots in first-order logic. It is a declarative language, meaning the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a query over these relations. Prolog is often associated with artificial intelligence and computational linguistics and is well-suited for specific tasks that benefit from rule-based logical queries such as searching "databases". One of the problems that is being addressed in this research is that program control in Prolog is through the *static* ordering of the rules (e.g. navigate *spouse* first, then *children*, then *parents*) and different orderings would be optimal for different queries.

III. Theoretical Framework

Traversing a tree has been the focus of much research, especially in game trees. Finding the path between any two nodes is trivial in a programming language like Prolog, whose goal-resolution principle is based on a built-in depth first search. In other words, it will recursively try all paths until it finds the solution. When it reaches a leaf that is not the intended target, it will simply backtrack to the last choice-point and try the next child.

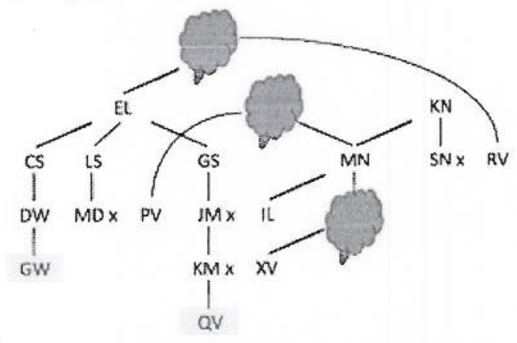
The problem is that a family tree, despite its name, is not a tree at all; each child has **two parents** (although for past generations the records are often incomplete and the information on one of the parents is missing – or in a few instances in my tree the child is a "natural" child who shares the mother's surname, with no father on record).

Moreover, unlike traditional trees, there are **multiple "roots"** in genealogy; for every generation, the number of ancestors doubles (until no more records can be found). The tree I maintain is mostly about my sons' ancestors and *all* the descendants of these roots (female lines included). The parents of spouses of these blood relatives are listed if known (they are typically part of the marriage records in city hall). These too are roots, but very shallow.

A final complication is that there are many "**loops**" in a family tree. The heimat of my great-grandfather is a small farmer's community by the Maas river, and most of its inhabitants are family one way or another. When my great-great grandfather's wife died giving birth, he remarried with her younger sister. So the stepchildren have the same grandparents. Another great-great grandfather whose first wife was a widow almost two decades older, remarried with her niece, 12 years his junior. History repeated itself when one of his grandsons married the niece of his father's second wife. There are several brothers who spouses were sisters, and even couples with the same last name who share distant ancestors. Thus, there are often multiple paths between any two nodes.

Thus, to clarify the goal of this research:

- Find the **most direct path** between any 2 nodes
- Find other **"interesting" paths** (while ignoring trivial alternatives, such as siblings being related through their father, and also through their mother). Examples of noteworthy connections:
    - *Kwinten*, a second-nephew one time removed (shortest path: the mother of his grandfather is a sister of my grandmother) is also connected three different ways via his grandmother through marriages.



    - A distant relative of my grandfather (the first in many generations to leave the heimat) is related to family of his spouse from his adopted country, thus providing a second connection between these 2 families.
    - The spouse of a relative of my dad is related to the spouse of one of my mother's relatives.

Finding "a" path is fairly straightforward in Prolog. The program specifies rules to explore horizontally (through *spouse*), and vertically (*children* and *parents*). However, there are several caveats:

- The built-in backtracking behavior of Prolog is convenient, but gives only limited control to the programmer
    - through **cuts** (!) - which means once satisfied, it doesn't attempt to re-evaluate;

  - and through the **ordering of rules**. Because said rules are tried in the order listed (*spouse*, then *child*, then *parent*), the first solution is rarely the shortest, and can be very convoluted (for example the grandfather of the above-mentioned *Kwinten* is only a few steps away through his mother, but since the grandfather's spouse is tried first, a much longer path is returned.)

Therefore, the following **heuristic** is applied: try a specialized blood-relative rule first:

```
        blood_relative(X,Y):-ancestor(A,X),ancestor(A,Y).
```

paraphrasing: X and Y are blood-relatives if they have a common ancestor A; or stated differently: find an ancestor A for X (referred to as instantiating or binding A), then check if the goal A is (also) an ancestor of Y can be proven/unified. Due to the relational nature of predicates, they can typically be used in several directions.

```
        ancestor(A,D):-parent(A,D). %trivial case: A is an ancestor of D if A is D's
        parent.
        ancestor(A,D):-parent(P,D),ancestor(A,P). %if above fails, try again, and
        see if perhaps the A is the ancestor of D's parent, etc.
```

This way the algorithm only has to look in an upwards direction – the search is quick and almost always returns the shortest path.

A second related heuristic takes advantage of the fact that the family tree is mostly blood-relatives and their spouses (plus in-laws). Only occasionally is the tree expanded to include siblings of the spouses, etc., because they appear in group pictures which are face-tagged, or are mentioned in death notices, or are perhaps other genealogy enthusiasts who maintain trees that partially overlap mine. So if the above `blood_relative()` function fails, before turning to the generic search, the goal is retried for the spouse; if successful, one step is added back to the returned path and offered as solution.

```
        related_via_spouse(X,Y):-married(X,S),blood_relative(S,Y).
        related_via_spouse(X,Y):-married(Y,S),blood_relative(X,S).
```

One interesting datamining rule is to find all couples in the tree who are both blood-relatives, but descending from 2 different ancestors of mine…

- The loops in the tree that were mentioned before are problematic; therefore, the Prolog code checks that the node it is about to explore isn't already in the *Path* traversed so far (this is the *Path* that will be reported eventually upon reaching the Target node). For example, the case mentioned above where 2 brothers (B1 & B2) had wives (S1 & S2) that were sisters: [*start_node - … - **B1** – S1 –Father_of_Sisters – S2 – B2 – Father_of_Brothers – **B1***]. Because we are already exploring ***B1***, there is no need to revisit it.
  BUT this *Path* grows and contracts as Prolog reaches a dead-end and backtracks. With this selective amnesia, nodes that have previously been explored **are** inadvertently revisited. Consider for example the following scenario: [*start_node - … - Husband – Wife*]; suppose the program next exhaustively explores the large subtree of the *Wife* and concludes this is a dead-end. It backtracks to the last choice-point [*start_node - … - Husband*] and tries *Child1* of *Husband* next [*start_node - … - Husband – Child1*]. If *Child1* tries the path through its mother, there is no record that the *Wife* subtree has already been explored, and a lot of time is wasted on this futile revisit. For large trees this causes very long computation times when looking for a path. This is why myheritage.com only reports blood relative relationships between 2 people once a tree grows larger than 5,000 nodes.
  One easy solution to the above problem is to maintain a persistent store of visited nodes that doesn't undo when backtracking: **`assert`**`(visited(X))`. This allows *a* solution to be quickly found, even for large trees. However, it is rarely the shortest path, and often quite convoluted.

- Because the search process in Prolog is driven by the **static order** in which the rules are stated, one trick is to **modify the search terms** instead and keep the optimal result from these back-to-back searches.

```
        find_relationship(Target,Source,Path)
```

may find a shorter path than

```
        find_relationship(Source,Target,Path)
```

because the latter may reach the destination when it explores one of the `Source`'s children, whereas the former may find a shorter alternative path that arrives at `Source` through its parent. Similarly, it pays trying

```
        married(Source,Spouse),find_relationship(Spouse,Target,Path)
```

if the shorter path is indeed through `Source` (because the first rule is to try the *spouse* rule first).

If the goal of the search is to find **all** the possible paths, then this can be accomplished by adding the **`fail`** predicate after a path has been found:

```
        find_relationship(Target,Target,Path):- writeln(Path),fail.
```

This forces Prolog to backtrack to its last choice-point, as if the target was not reached, and try again. In order to find the **shortest** path (and speed up the search), a **branch-and-bound** technique can be used which abandons non-promising paths when they reach a length equal to the shortest solution found so far; continuing along that path would be fruitless even if the destination can be reached because it would guaranteed be longer.

The above solution won't work in practice because of the `visited()` persistent store mentioned earlier. Since the target node has been reached once when the first solution was reported, no other paths will be found because `Target` is off limits. Even if the final node is excluded from being asserted as `visited()`, a shorter path likely also includes some `visited()` nodes. For example

- o [Source – A – B – C – D – E – Target] : 7 nodes
- o backtrack: [Source – A – B]
- o let's say the next choice-point is **`E`**, which would lead to a shorter 5-node path [Source – A – B – E - Target]. Yet it is prevented because **`E`** has been asserted as `visited()`.
- o note that [Source – A – B – F – Target] would still be permitted if such a solution existed.

Dilemma: the persistent store allows finding a first solution very quickly, but prevents finding further solutions (or cuts off most other solutions).

Proposed Solution: use domain-specific heuristics to prevent duplicating traversals **without relying on a persistent store**. Some examples:

- in the first rule (try *Spouse*): `waitForBufferedChildToExplore();` assume the following scenario:
  [*start_node* - ... - Child – Parent1]
  normally this rule would cause the spouse of *Parent1* to be tried next (in other words *Parent2* of *Child*).
  [*start_node* - ... - Child – Parent1 – **Spouse=Parent2**]
  However, upon later backtracking (because of dead-end, or because of looking for additional solutions), *Child* will try its next choice-point, which is *Parent2*
  [*start_node* - ... - Child – **Spouse=Parent2**]
  which would be duplicate effort. So rather than remembering that we have already explored this subtree, the heuristic anticipates the duplication and pre-emptively cancels the spouse traversal when it sees that *Child* is already in the *Path*. An added benefit is that if a solution happens to lie along the *Parent2* path, it is one step shorter than by using the *Parent1* intermediate step (and the solution that included *Parent1* would fall under the category of "trivial" alternatives).
- in the second rule (try *Children*): `father_or_single_mom()`. Don't explore children from mother node (unless there is no father on record). Unlike the previous heuristic, the father doesn't have to be in the *Path* yet.
- in the third rule (try *Parents*): `waitforBufferedSiblingToExplore();` assume the following scenario:
  [*start_node* - ... - Child1 – Parent1 – Child2]
  normally this rule would cause *Child2* to try *Parent2* (not *Parent1* because it is already in the *Path*).
  [*start_node* - ... - Child1 – Parent1 – Child2 – **Parent2**]
  However, upon later backtracking, *Child1* will try its next choice-point, which is *Parent2*
  [*start_node* - ... - Child1 – **Parent2**]
  which would be duplicate effort. So, like heuristic 1, the *Parent2* traversal is pre-emptively cancelled for later exploration by *Child1*.
- A final example, also for the third rule: `spouse_in_path();` assume the following scenario:

[*start_node* - ... - Parent1 – Spouse=Parent2] (*)
let's assume this subtree is explored (successfully or not), and we are backtracking.
[*start_node* - ... - Parent1]
try rule 2
[*start_node* - ... - Parent1 – Child]
try rule 3
[*start_node* - ... - **Parent1** – Child – **Parent1**]: blocked because already in *Path*, and
[*start_node* - ... - **Parent1** – Child – Parent2]: block on account of `spouse_in_path()` rule. We already have explored the *Parent2* subtree earlier (*). This heuristic does look at the past instead of ahead, but uses the dynamic *Path* rather than a persistent store.

These example heuristics prevent duplication by looking for short-distance patterns. The goal of the research is to look for additional re-computation patterns caused by backtracking, and to create domain-specific heuristics to anticipate these or to recognize them as duplicate. One tool to detect the need for a blocking heuristic is to temporarily continue using a persistent store. When it would prevent a node selection (because it has been visited before) and it is not currently blocked by a heuristic, it would yield examples that can be studied for patterns so optimization rules can be created.

IV. Brief examination of scholarly literature or context of the activity within your discipline

See above – here is a listing of references to technical terms, best practices, and current research that were mentioned.

Backtracking:
Labbe, Roger. "Solving Combinatorial Problems with STL and Backtracking". February 1, 2000. Accessed October 17, 2017.
http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/CUJ/2000/0002/labbe/labbe.htm.

Branch and Bound Heuristic:
Clausen, Jens. "Branch and Bound Algorithms - Principles and Examples." March 12, 1999. Accessed October 17, 2017. http://www.diku.dk/OLD/undervisning/2003e/datV-optimer/JensClausenNoter.pdf

Depth-First Search:
Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Introduction to Algorithms", Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 22.3: Depth-first search, pp. 540–549.

Game Trees:
Victor Allis (1994). "Searching for Solutions in Games and Artificial Intelligence" (PDF). Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands. ISBN 90-900748-8-0.

Graph Traversal:
Foerster, Klaus-Tycho; Wattenhofer, Roger (December 2016). "Lower and upper competitive bounds for online directed graph exploration". Theoretical Computer Science. 655: 15–29.
Valiente, Gabriel. (2013) "Algorithms on Trees and Graphs". Springer Science & Business Media. ISBN 978-3-642-07809-5. Chapter 5.

Heuristics:
Pearl, Judea (1983). "Heuristics: Intelligent Search Strategies for Computer Problem Solving." New York, Addison-Wesley. ISBN 978-0-201-05594-8

Prolog programming language:
Clocksin, William F.; Mellish, Christopher S. (2003). "Programming in Prolog." Berlin ; New York: Springer-Verlag. ISBN 978-3-540-00678-7.
Bratko, Ivan (2001). "Prolog programming for artificial intelligence." Harlow, England ; New York: Addison Wesley. ISBN 0-201-40375-7.

Shortest Path Problem:
Ahuja, Ravindra K.; Mehlhorn, Kurt; Orlin, James; Tarjan, Robert E. (April 1990). "Faster algorithms for the shortest path problem". Journal of the ACM. ACM. 37 (2): 213–223.

Travelling Salesman Problem:
Rosenkrantz, Daniel J.; Stearns, Richard E.; Lewis, II, Philip M. "An Analysis of Several Heuristics for the Traveling Salesman Problem". SIAM Journal on Computing. 6 (3): 563–581.

Tree Traversal:
Valiente, Gabriel. (2013) "Algorithms on Trees and Graphs". Springer Science & Business Media. ISBN 978-3-642-07809-5. Chapter 3.

4. **An essay (500 - 1000 words) describing how the Christian faith relates to your understanding of your discipline and how it relates to this scholarly endeavor.**

This brief essay is a reflection of how my faith informs and directs my work as a computer scientist. It can be summarized by the Truth that **all life falls under the Lordship of Jesus Christ**, meaning that we can serve Him equally well as a minister or as a programmer.[1] It is a rejection of a dualistic view, that some careers are "sacred" and guided by the Bible, while others like technical professions are not (the modern-day "Marthas" of the world). The Christian Faith has comprehensive scope and affects all of his Creation. "Each of You should use whatever gift you have received to serve others, as faithful stewards of God's grace in its various forms" (1 Peter 4:10). "So whether you eat or drink or whatever you do, do it for the glory of God" (1 Corinthians 10:31). Derek Schuurman paraphrases a quote credited to Karl Barth that "A Christian is a thinking person who holds the Bible in one hand and a smartphone in the other".[2]

The word *computer* obviously does not appear in the Bible, so what is the **relevance** of our 2,000-year old Christian religion on technology? How does our faith inform our thinking about coding, piracy, privacy and piggybacking?[3] Paraphrasing the early church father Tertullian, "what do chips and code have to do with Jerusalem"?[4]

God reveals Himself not only through the Bible, but through the beauty of His Creation. This isn't limited to stars and continents and animals. John Calvin comments that "God daily discloses himself in the whole workmanship of the universe. […] There are enumerable evidences both in heaven and on earth that declare his wonderful wisdom [including] those more recondite matters for the closer observation of which astronomy, medicine, and all natural science are intended".[5] We can easily pencil in computer science as one of the recondite matters that **demonstrate God's glory**. At the time of creation, God "placed within the world the latent potential for technology and computers" and gave us "the delightful task of opening up [this] potential. […] The study of computer science, like other scientific pursuits, gives us a glimpse of the majesty of a powerful and wise Creator."[6] Computers, like microscopes and telescopes are like the ships in Psalm 107:23-24[7] that gives us new insight in the awesomeness of God's Creation, such as fractals, patterns in data, quantum computing, etc.

We are made in the image of God, and thus have the **capacity to be creative**. Linus Torvalds, creator of Linux, writes how "with computers and programming you can build new worlds and sometimes the patterns are truly beautiful".[8] Frederic Brooks, known for his canonical work *The Mythical Man-Month* similarly writes about the joy of programming: "Why is programming fun? What delights may its practitioner expect as a reward? First is the sheer joy of making things. As the child delights in his mud

[1] As an aside, a point should be made that as teachers at a Christian school, we serve as lay ministers in the classroom.

[2] Derek Schuurman, *Shaping a Digital World- Faith Culture and Computer Technology*. (Intervarsity Press, 2013), p. 76. This book is one of the few scholarly works that provides insight on a Christian approach to Computer Science, and it has greatly shaped my thinking.

[3] unauthorized use of someone's WIFI signal.

[4] "What does Athens have to do with Jerusalem"?

[5] John Calvin, *Institutes of the Christian Religion*, vol 1 (Philadelphia: Westminster, 1960), pp. 52-53.

[6] Derek Schuurman, pp. 31&32.

[7] "Those that went out on the sea in ships […] saw the works of the Lord, his wonderful deeds in the deep".

[8] Linus Torvalds and David Diamond, *Just for Fun: The Story of an Accidental Revolutionary* (New York: HarperCollins, 2001), p. 75.

pie, so the adult enjoys building things, especially things of his own design. I think this delight must be an image of God's delight in making things, a delight shown in the distinctness and newness of each leaf and each snowflake."[9]

Technology being an extension of Creation has implications of how it should be developed and used. It should be done in a way that furthers Gods kingdom and glorifies him.[10] Rather than rejecting technology as the source of all our ills or blindly trusting that it will solve all modern-day problems, we should design software and hardware in ways that honor God. **We must steward computing** as we would do with all other aspects of God's Creation. Some like Robert Buchanan argue that technology is value-neutral, "essentially amoral, a thing apart from values, an instrument which can be used for good or ill".[11] Others, like Derek Schuurman argue that technology is value-laden because designers embed their values or "direction" into their devices. This causes their creations to be biased towards certain uses, which in turn bias the consumer.[12] Neil Postman puts it this way: "Embedded in every tool is an ideological bias, a predisposition to construct the world as one thing rather than another, to value one thing over another, to amplify one thing over another, to amplify one sense or skill or attitude more loudly than another."[13] Or as John Culkin observes, "We shape our tools and thereafter they shape us."[14] I agree with George Grant that computer technology is changing our world, and we are still discerning how extensive these changes are.[15]

In "What Difference Could it Possibly Make?", George Marsden identifies four ways that faith can impact and direct Christian scholarship.[16] Besides the motivation to **do the research well**, faith is helping to determine the **application of my scholarship** (Marsden's first two points). The proposed research deals with **genealogy**. Jesus' ancestry is given in Matthew 1 and Luke 3. Genealogy is an area of very active research by members of the Church of Jesus Christ of Latter-Day Saints. While I am not a Mormon and many of their doctrinal positions are radically different from those of historical, biblical Christianity, our shared belief in eternal life in heaven with God gives some insight into the reason they put so much time and resources into genealogy work. Jessica Grayless is a blogger who is a programmer by profession and a genealogy enthusiast in her spare time. She explains that "The central ideology that drives this effort is that all mankind will have the opportunity to hear and accept or reject the gospel of Jesus Christ, whether in this life or the next. Of course, if someone has passed on to the next life without the opportunity, how can they be baptized and have the other necessary ordinances performed? Mormons believe this can be done by proxy, meaning a living person can stand in the place of, or be proxy for, one who is dead. The ordinances of baptism, endowment and sealing of families together for those who have died are done in LDS temples."[17]

---

[9] Frederick Brooks, *The Mythical Man-Month* (San Francisco: Wiley, 1995), p. 7.

[10] Theodore Plantinga, *Rationale for a Christian College* (Grand Rapids: Paideia Press, 1980), p. 57.

[11] Robert Buchanan, *Technology and Social Progress* (New York: Pergamon Press, 1965), p. 163.

[12] Derek Schuurman, p. 15.

[13] Neil Postman

[14] John Culkin, "A Schoolman's Guide to Marshall McLuhan," *Saturday Review*, March 18, 1967, p. 70.

[15] George Grant, *Technology and Justice* (Toronto: House of Anansi, 1986), p. 19.

[16] George Marsden, *The Outrageous Idea of Christian Scholarship* (New York: Oxford University Press, 1998), pp. 63-64.

[17] Jessica Grayless "Why Genealogy is Important to Mormons". March 20, 2009. Accessed October 17, 2017. http://familyhistoryresource.blogspot.com/2009/03/why-genealogy-is-important-to-mormons.html.

5. **A time frame for the completion and a plan for the dissemination of the project.**

   Feb 28 – Mar 3: attend RootsTech Conference: https://www.rootstech.org/

   Summer 2018: write and test heuristics

   Late Summer/Fall 2018: Preparation of an article for to the *Journal of the Society of Genealogists*: http://www.sog.org.uk/

   May 2019: present at the NGS 2019 Family History Conference: http://conference.ngsgenealogy.org/

6. **A brief budget.**

Conference attendance:	$1,500
Researcher stipend:	$3,000