Corbin Montague, Christopher Word

EIDs: cm35972, cword

Maximum Weight Quartet Compatibility

Introduction

The “tree of life” project is an attempt at reconstructing the evolutionary history of all known species. This is a central problem in biology for the study of evolution and is computationally difficult. The “tree of life” is a Supertree problem: Given a collection of trees, find a tree that agrees with all the input trees. Supertree is provably NP-Hard, therefore heuristics and approximation algorithms are necessary to solve this “real world” problem.

We have developed a heuristic to solve the Maximum Weight Quartet Compatibility (MWQC) problem. The input consists of weighted, unrooted quartet trees and the output is a tree that agrees with a maximum weight subset of the quartets. Simply put, this is an extension of the quartet compatibility problem, where we now care about the relative weights of the quartets more than the quantity of quartets we satisfy.

While we could find no heuristics developed for MWQC, we researched heuristics for similar problems such as the Quartet Tree Amalgamation (QTA) problem. QTA aims to find a tree which agrees with a maximum number of the given quartet trees. QMC, one such solution, runs in polynomial time but provides no guarantees on the optimality of its answer [1]. We also looked at solutions to the more general Supertree problem, such as Matrix Representation with Parsimony (MRP), which encodes source trees as a matrix of partial binary characters and analyzes it using a heuristic for maximum parsimony.

The High-Level Algorithm

Finding the optimum tree for MWQC is NP-hard, so we use a hill-climbing/greedy algorithm on a randomly generated starting tree to find a local optimum. We have 2 ways to run our program: Double Greedy and Precision.

Double Greedy mode uses 2 greedy choices. We take the best neighbor tree, but we only consider neighbors that are good for an individual edge as well.

Double Greedy Pseudocode:

```
Generate some tree T with the appropriate number of leaves
{
  Create T' = {set of all neighbors of T who have greater values for a particular edge}
  Max = maxscore { T, T' }
  if score(Max) == score(T)
    return T
  else
    recurse on Max
}
```

Precision mode uses only 1 greedy choice. We take the best neighbor tree, but we consider all neighbors of every edge.

Precision Pseudocode:

```
Generate some tree T with the appropriate number of leaves
{
  Create T' = {set of all neighbors of T}
  Max = maxscore { T, T' }
  if score(Max) == score(T)
    return T
  else
    recurse on Max
}
```

Since the output is a local optimum with no guarantee of being a global optimum, the algorithm can be run multiple times given different random starting trees. This can drastically improve the quality of the answer returned, as the starting tree has the most significant impact on the final tree's score.

Input: Weighted Quartets

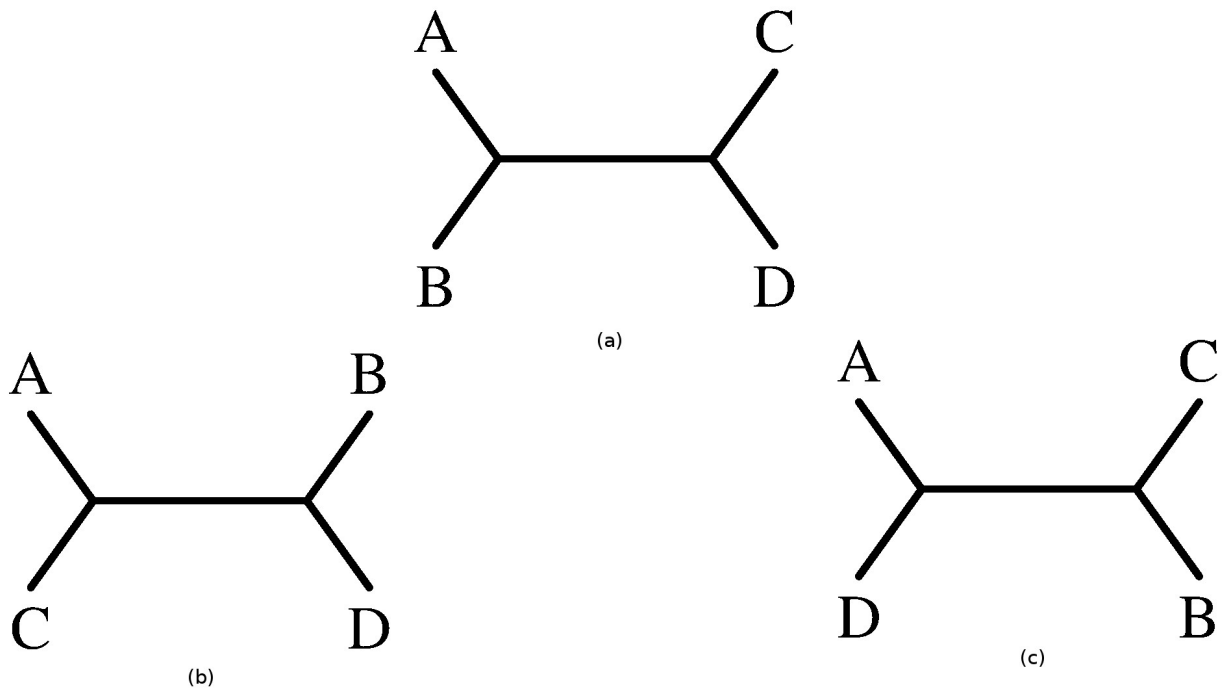
Input is given as a list of weighted quartets of the form: "((A,B),(C,D)); weight".
Each quartet is named ":A:B:C:D" in sorted order. Quartets are stored in a HashMap with their name as a key. Due to our naming convention, there can be up to three quartets mapped to the same name. The purpose of this naming will be discussed further in the "Work to be done" section later.

Generating Neighbors

Our approach uses NNI (Nearest Neighbor Interchange) [2] to generate all neighboring trees which works as follows:

Pseudocode:

```
T' = {}
For all edges E in T:
{
  Swap subtrees B and C to generate T1
  Swap subtrees B and D to generate T2
  add T1 and T2 to T'
}
```



Given an edge (a), we generate its two neighbors (b) and (c). (b) and (c) both represent how this edge would look in the neighboring trees induced by the exchange. In our Double Greedy mode we only return the largest neighbor whose value is also larger than our own. In precision mode we return both.

Tree Scoring

At construction a tree scores itself by summing up the weight of its edges. Each edge is scored during its construction by iterating over all given quartets. An edge satisfies a quartet if the edge contains the leafs of the quartet in each of its sets (exactly one leaf in each set).

Evaluation

Below are evaluations of our Double Greedy algorithm vs our precision algorithm on the given quartet input files: r3, r5, and r20.

	Double Greedy	Precision
Avg Time per Iteration-r3	79.2243365785384	474.962968594625
Total Iterations-r3	320	130
Best Score-r3	17233.3750767086	17963.2310270004
Avg Time per Iteration-r5	95.0860096765	472.316870218571
Total Iterations-r5	160	80
Best Score-r5	16084.3488290928	16336.7972247998
Avg Time per Iteration-r20	89.9075446998	455.647393248667
Total Iterations-r20	160	70
Best Score-r20	17177.3095738327	16471.9410484665

	Avg Speedup-Greedy (Times Faster)	Percent (Greedy's best/Precise's best)
r3	5.995	96%
r5	4.967	98.5%
r20	5.068	104%

Our data only compares these two approaches since we do not have a way of comparing our final output to the optimal score. While Double Greedy was our original idea, we considered an algorithm like Precise which looks at more cases may yield significantly better trees. Double Greedy's goal is to reduce the running time as much as possible by

looking at far less cases. Our data supports that we are successful as the Double Greedy algorithm finishes 5 to 6 times faster than Precision and maintains nearly identical correctness in its final tree compared to Precision (96% - 104%). The data also suggests that a good starting tree is more important than any other factor. Since Double Greedy can look at trees 5 to 6 times faster than Precision, it can look at more starting trees increasing the score of the final choice.

Work to be done

- Edges only iterate over quartets concerning it when scoring. If we only look at edges we have a chance to satisfy we could call (Edge.satisfies(Quartet)) much less.
- Spawn multiple threads: This kind of algorithm would benefit from parallelism.
-Research into which language would optimize our running time. This is a proof of concept and as such is implemented in Java. C could be faster with pointers and a different data representation.

Classes

MWQT.java

Our driver class. It is a static class that consists of main, and many helper methods. It is the glue that puts together our approach.

1)Reads input to determine the number of taxa

2)Loop for iteration(args[2]) times

3)Calls QMC to generate a random starting graph

4)Generates a map of Quartets

5)Generates the Tree object(T)

6)Calls the tree findMaxNeighbor method that returns the best neighbor to this tree(T'). This is our heuristic. It is purely a greedy approach.

7)If T == T' we return T

8)Else we go to 4 and use T' to generate the T.

9)end Loop

Tree.java

A class of Edges that represent a tree. It is constructed from a Newick string and stored as an adjacency list.

Edge.java

A set representation of an edge. It is able to generate its own neighbors using NNI and can compute the delta-score for each neighbor. Sets {A,B,C,D} are lists of all leaves beneath neighboring edges. Each edge has a value equal to the sum of the weights of all quartets it satisfies.

Quartet.java

A data structure for storing quartets and their information. A quartet has a method to if it is satisfied or not and is of the following form: ((A,B),(C,D)); weight

Works Cited

[1] S. Snir, S. Rao, “Quartets MaxCut: A Divide and Conquer Quartets Algorithm ,” *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 7, no. 4, pp. 704-718, 2010.

[2] B. DasGupta, X. He, T. Jiang, M. Li, J. Tromp, L. Zhang, “On Computing the Nearest Neighbor Interchange Distance,” *Proc. 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 427-436, 1997.