Ramseys Bane: A Distributed Approach to Ramsey Numbers

Nikolas Chaconas, Sujaya Maiyya, Bjarte Sjursen

October 10, 2017

Abstract

This project seeks to improve the known bounds on Ramsey(10,10), which is currently known as 798 to 23556, using a distributed architecture for increased compute power. To achieve this goal, we used over 7 different cloud technologies, 15281 total cores, and an estimated $1.25492869588 * 10^18$ CPU cycles. Our implementation went through many different iterations, with each iteration progressively improving in performance over the previous.

1 Introduction

Solving Ramsey numbers using computers becomes increasingly difficult as the graph size increases. Upon discovering the difficulty that a single computer has in solving a Ramsey graph, we developed a fault tolerant and scalable client-server architecture, which served as the foundation for our success in this project.

2 Technologies Employed

We tried to use as many cloud technologies as possible. Because we employed so many different technologies on different platforms, we found it necessary to create a script for deploying our code to every cloud service, and automatically building and running it. Furthermore, due to processes our being stopped, or errors within our clients, we also created a script which would sleep and periodically wake up and check if our program was running. If it wasn't, it would automatically restart it. If it was running, it would sleep again. This was helpful both for restarting our program when there were errors (memory leaks, etc) as well as when our program would be shut off for using high resources on shared machines. Another technology we used to monitor our system was the slack messaging service. We created a command, which when run in slack, responds with an update of how clients are doing. Furthermore, every time a counter example was solved, it would send the example to a slack channel. This served a dual purpose - we had more replication of our counter examples, and we could be directly notified when another Ramsey counter example had been found.

2.1 Client Side

On the client side, our protocol was multi-threaded. We always ensured that our thread count was based on the number of cores on the system, in the hope of achieving parallelism. We also chose to implement our clients in C, as they were responsible be handling the most computation-heavy work, and we wanted to maintain speed with scale. Multi-threaded clients performed much faster, as they could signal other threads when better graphs were found - without the use of any networking. We tried to employ as many different cloud technologies as possible, with the hope that our program's speed would increase linearly with the number of cores we threw at it.

2.1.1 ECI Eucalyptus

We had 15 single core nodes on the ECI Eucalyptus cloud, as well as a 12 core node, for a total of 27 cores on Eucalyptus.

2.1.2 Condor

We had five different configurations for Condor:

· **500 single core nodes**

· **250 four core nodes**

· **250 eight core nodes**

· **250 16 core nodes**

· **250 24 core nodes**

This gave us a total potential compute power of 13,500 cores on Condor.

2.1.3 CloudLab

On Cloudlab, we would spawn 3-4 sets of 40 servers, 8 cores each for a total of 1280 cores.

2.1.4 Chameleon

On chameleon, we hosted a single 8 core client.

2.1.5 Cornell

We hosted a 2 core client, as well as a 12 core client on Cornell's eucalyptus platform, for a total of 14 cores on Cornell.

2.1.6 Comet

We attempted to use Comet on multiple occasions, however we always had network difficulties, and comet would automatically kill our process after experiencing any sort of networking failure. However, during the limited time we used Comet, we deployed batches of 72 nodes, with 24 cores on each, for a total of 1728 cores.

2.1.7 CSTL Computers

Previous to speaking with the ECI staff, we had used all of CSIL/Honea/CSTL computers. Later in the quarter, we switched to just 36 CSTL computers, each with 4 cores.

2.1.8 Home Desktop

My home desktop was our final client, with four cores on it.

Although not all technologies could be used simultaneously (Condor frequently halts processes), we had the total potential of 15425 cores/threads working on our solving Ramsey Numbers.

2.2 Server Side

For the server side, our architecture has 3 parts: a loadbalancer, 5-7 coordinators or servers and 3-way replicated database. The loadbalancer is the built-in LB provided by the Aristotle Eucalyptus. For database, we used CockroachDB. CockroachDB is a distributed SQL database built on a transactional, consistent key-value store. It scales horizontally and survives failures on different layers with minimal latency. CockroachDB is an auto-replicated database where number of replication is a configurable parameter. It is based on RAFT consensus protocol and the DB us available as long as majority of the DB nodes are available. In our project, we have used 3-way replication i.e., there are 3 DB nodes.

Every client is aware of the loadbalancer's URL and it contacts the LB. LB then forwards the request to one of the servers. LB maintains the list of active servers and forwards client requests only to active nodes. Based on the input received, if the server decides to make changes in the database, it connects to one of the DBs and performs the operation. The performed operation is replicated in the remaining 2 nodes. Each VM on the server side was deployed on Aristotle and each instance was configured with 4 CPUs and 16GB memory. We also created 5 volumes and attached them to the nodes running the database in order to survive VM failures.

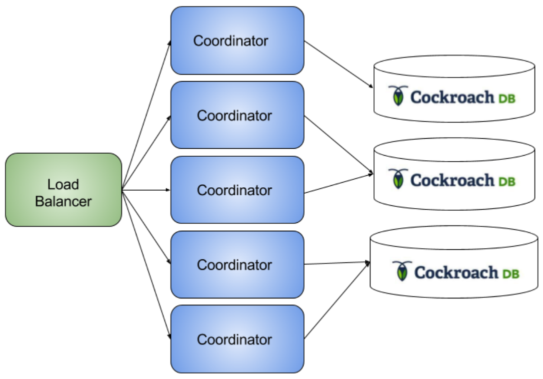

Figure 1: Screenshot of output.

3 Random Approach

3.1 Server Implementation

In the random approach implementation, we did not utilize the clique count property of the graphs. Clients would randomly flip bits till at least one edge-set of the graph yielded a zero clique count. Server would accept the client's message if the number of nodes the client worked on was higher than the highest value known by the coordinators; else the client messages will be discarded and the server sends the highest known node value on which the client should work on. Every time a server receives a counter example for higher node count, the graph would be stored in the database (one of the DBs and then CockroachDB replicates this on all nodes). Since servers do not communicate with each other, one may not be aware of the progress made by the other server. But the DBs always have the latest state. So, before replying to the clients, each server reads the highest node value and the corresponding counter example from the DB and send it to client. This way, every server always responds to the client with the highest solved graph and node value.

In the random approach, the role of server was minimal: accept the message if the counter example was found for a node value higher than the value known by servers; reject the message otherwise. Always reply to the client with the highest solved node value and counter example graph.

3.2 Client Implementation

3.3 Algorithm

We employed a couple of simple randomized algorithms in the beginning of the project to get the whole process going.

3.3.1 Randomized search

Initially we started out with a purely random algorithm that simply generated an upper triangular adjacency matrix and checked if it were a counter example. This basic methodology nudged the project forward in its infancy but did not provide any interesting results. The pseudocode for our first algorithm can be seen in algorithm 1.

Algorithm 1 RANDOM-PERMUTE

```
Require: $|V|$
  while true do
    $K_{|V|} \leftarrow$ RAND-GRAPH($|V|$)
    $c \leftarrow$ CLIQUECOUNT($K_{|V|}$, $|V|$)
    if $c == 0$ then
      $|V| \leftarrow |V| + 1$
    end if
  end while
```

3.3.2 Randomized graph with balanced edge distribution

To get more familiar with the problem and understand the search space of it we performed a couple of experiments. We wrote python scripts that would generate every possible edge permutation of complete graphs with $|V|$ vertices, and then we would see how many of those graphs that had N-cliques for different N. A result of one of these experiments can be seen in figure 2.

We furthermore calculated the total number of possible permutations for every number of 1-edge (we denote the number of 1-edges by T1) in a graph by using the binomial coefficient $\binom{|E|}{T1}$.

Hence we calculated the number of R(N,N) counter examples for different values of T1 by using $\binom{|E|}{T1} - |N - \text{cliques}|$.

The next step was to get the probability of finding an R(N,N) counter examples for certain values of T1. This could be done by

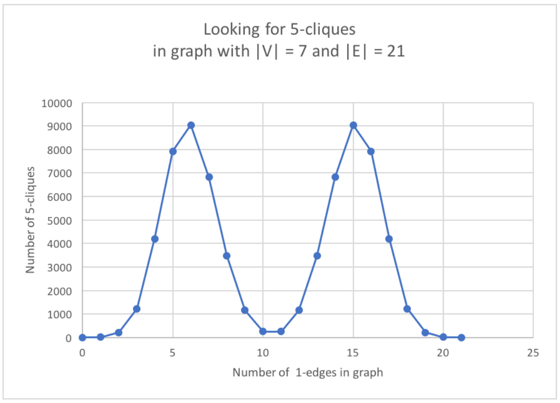

Figure 2: Number of 5-cliques for different number of 1-edges present in graph.

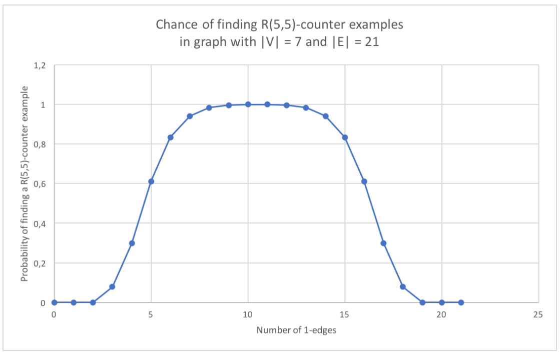

Figure 3: Probability of finding R(5,5) counter examples in a graph G with different number of 1-edges present.

dividing the number of R(N,N) counter examples by the total number of permutations for a certain value of T1 . The result of this when looking for R(5,5) in a graph with $|V|$ = 7 and $|E|$ = 21 can be seen in figure 3.

The insight we gained from this was that the probability of finding R(N,N) counter examples is largest when we look at graphs that have an equal distribution of 1-edges and 0-edges.

This new information paved way for our second algorithm, which essentially also generated random graphs. The change we made compared to the first algorithm was that we make sure the generated graph has an even distribution of 1 - and 0-edges. This got us a lot further, and we got to counter examples around 140 using this methodology. The updated pseudocode can be seen in algorithm 2.

Algorithm 2 BALANCED-PERMUTE

```
Require: $|V|$
  while true do
    $K_{|V|} \leftarrow$ BALANCED-GRAPH($|V|$)
    $c \leftarrow$ CLIQUECOUNT($K_{|V|}$, $|V|$)
    if $c == 0$ then
      $|V| \leftarrow |V| + 1$
    end if
  end while
```

3.3.3 Randomized search with balanced node edge distribution

The last iteration of our random search approach built on top of the previous one, but in addition to making sure that the distribution of the edges in the graph are equally distributed, we also make sure that the edges emanating from every node are evenly distributed. This minor improvement got us a couple steps further in the search for Ramsey-number counter examples. The method can be seen in algorithm 3.

Algorithm 3 BALANCED-NODE-PERMUTE

```
Require: $|V|$
  while true do
    $K_{|V|} \leftarrow$ NODE-BALANCED-GRAPH($|V|$)
    $c \leftarrow$ CLIQUECOUNT($K_{|V|}$, $|V|$)
    if $c == 0$ then
      $|V| \leftarrow |V| + 1$
    end if
  end while
```

4 Distributed Systematic Flip

4.1 Server Implementation

In the distributed systematic flip approach, we used the total clique count property in computing new counter examples. Instead of maintaining just the node count and the respective counter example, the servers now maintained the lowest known clique count for the highest known node count and the corresponding graph in the database. In the distributed systematic approach, the server distributes the indexes on which the client should work on (the client's role is explained in the next section). Each server, on receiving a better solution with a lower clique count, updates the entries in the database and maintains a queue of all the indexes of the graph for which the edge weight was 1 in the graph. In order to avoid giving out redundant indexes by different servers, each server only maintains every 7th index (as there are 7 servers) which has an edge value of 1. Each time a client contacts the server asking for the graph and the index to work on, server replies with the latest known graph which has least clique count and pops an index randomly from the queue and replies to the client, which then works on it. Clients, in their messages, send the node count, best known clique count and graph. If the sent message has a clique count lower than the known clique count by server, it updates its state and the database. Servers read from DB to see if other servers have made any progress before replying to the client; if a better clique count was read from DB, then the server updates the index queue and starts giving out the indexes for the new graph. This way server distributes the indexes amongst all clients, who deterministically work on finding counter example with clique count 0.

4.2 Client Implementation

The client protocol begins with the client contacting the coordinator for an index to begin working on, if an index is received, the client will be begin switching that index with every other zero in that row and checking the clique count on the new graph. If the clique count lowers, the client will signal the other threads on the same machine, and contact the coordinator with the new clique count and graph. If the clique count raises or stays the same, the client will undo the switch, and continue to the next zero in the row.

Throughout this process, the client polls the coordinator every 10 iterations of the above protocol, to ensure that if a better clique count has been found by another client, the polling client will stop searching on the current graph and query the coordinator for a new graph and index to begin searching on. When polling, the client would set a flag in the message indicating that it did not need a new index from the coordinator, and that it was simply contacting to be updated on the latest counter example and clique count.

In the event that all indices are exhausted on a graph (all indices have been distributed by coordinators, and none have yielded a zero clique count), the clients go into a random flipping stage where they randomly flip edges and explore parts of the graph. We found this to not be very effective, and a different heuristic could have been used. We did find, however, that indices for a graph were not often exhausted, thus we rarely entered this stage.

4.3 Algorithm

We quickly realized that search techniques purely based on generating random adjacency matrices would not be feasible for continued progression towards higher number of counter examples. So we started using a new search heuristic, the number of cliques in a graph. The idea was to have a greedy algorithm that would perform a systematic permutation of a graph, and the moment it flipped an edge that resulted in a graph with a lower clique count, then it would become the foundation for a new systematic flip. However we would utilize our previous guideline of only generating graphs with a balanced distribution of 1 and 0 edges on each nodes. So for every 1 edge we flipped, we flipped a 0 edge attached to the same node. The conceptual pseudocode can be seen in algorihtm 4. Every time we find a graph that has a lower clique count than what we already have, then it becomes the basis for a new search. So the method GREEDY-PERMUTE essentially gets called every time we have a new graph.

Algorithm 4 GREEDY-PERMUTE

```
Require: $K_{|V|}$
  $c \leftarrow$ CLIQUECOUNT($K_{|V|}$, $|V|$)
  for all vertices $v \in K_{|V|}.V$ do
    for all edges $e1 \in v.E == 1$ do
      for all edges $e0 \in v.E == 0$ do
        $e1 \leftarrow FLIP(e1)$
        $e0 \leftarrow FLIP(e0)$
        $d \leftarrow$ CLIQUECOUNT($K_{|V|}$, $|V|$)
        if $d < c$ then
          return $K_{|V|}$
        else
          $e1 \leftarrow FLIP(e1)$
          $e0 \leftarrow FLIP(e0)$
        end if
      end for
    end for
  end for
  return $K_{|V|}$
```

After we had gotten this working on a single computer we realized that this method easily could be distributed to a big number of servers. The idea was to give each machine participating in the search a 1-edge in a certain row from the adjacency matrix and make it flip that 1-edge with every 0-edge in the same row. If it found a cliqe count lower than the current best clique count, then we would quit the flipping on the worker machine, and send the better graph along with the clique count to the server. The flipping process is illustrated in figures 4 - 6.

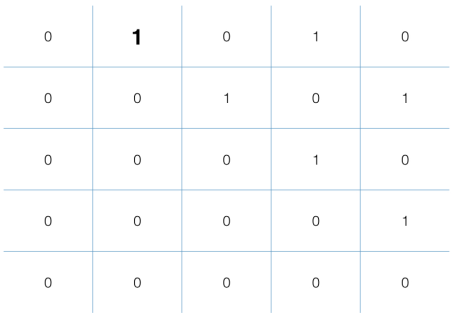

Figure 4: An adjacency matrix before the flipping process starts. The 1-edge to flip is highlighted in bold.

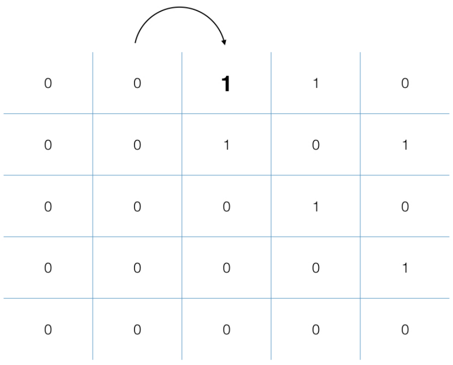

Figure 5: The same adjacency matrix as in figure 4 after the first flip of the highlighted 1-edge.

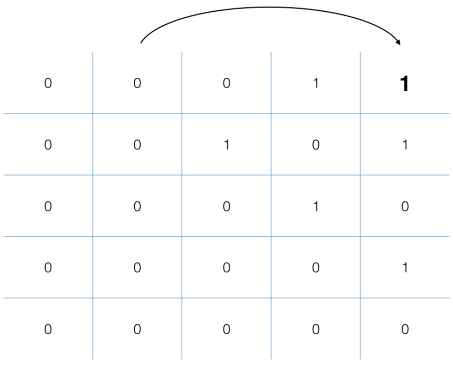

Figure 6: The adjacency matrix after the last flip of the highlighted 1-edge.

The scalability of distributing this would depend upon the size of the graph we were looking at. Assuming that the graph we have has an even distribution of 1- and 0-edges on each node, the relationship between the graph size and how many computer that could participate in the search is given in equation 1.

$$\text{scability} = \sum_{v=0}^{v=|V|-1} \left\lfloor \frac{|V|-v}{2} \right\rfloor \tag{1}$$

In addition an upper bound on the total number of flips that can be done in a graph is given by equation 2.

$$\text{flip-count} = \sum_{v=0}^{v=|V|-1} \left\lceil \frac{|V|-v}{2} \right\rceil^2 \tag{2}$$

5 Final Iteration: Most Common Clique Sets

5.1 Algorithm and Client Implementation

Our final iteration ended with a mixture of our algorithm from (4) and a new search heuristic. This new method involved keeping track of all type one and type zero clique sets in the graph, utilizing a clique counter made by a Finnish mathematician [1]. Using these sets, we created two structures for both type zero edges and type one edges:

1) List of sets of nodes occurring in the most cliques, sorted from largest occurrence to smallest occurrence.

2) The set of most common edges to each node for each set in the list created in (1)

After extracting the two nodes which contributed most to cliques in the graph, we would flip the link between them. If the total clique count went down, we would consider it a good flip and contact the coordinator with the new graph and clique count. If the clique count went up, we would consider it a bad flip, and continue to the next most common node in the list.

We found that this heuristic, when combined with other clients working on the algorithm in (4), was very effective at solving clique counts down to 1 or 2 within minutes. However, our algorithm almost always could not achieve progress past this point, leading us to believe that a new heuristic should have been implemented (such as Tabu search), after bringing a graph down to a low clique count.

6 Results

We believe that the one downfall of our approach was our final search heuristic on a graph with a low clique count. For future work, we would look into optimizing this, or using the knowledge of clique count nodes set more to our advantage.

7 Conclusion

In summary, we realized the difficulties involved in building a robust client-server architecture in a distributed setting. Through the motivation of finding higher counter examples for R(10,10), we have become more familiar with various cloud platforms, their behavior, their benefits and their pitfalls. And through the course of this project, we are more equipped at solving real, large scale problems.

References

[1] P. R. O. Sampo Niskanen, "Cliquer users guide," 2003.