TSP Solver Using Multiple iOS Devices

Sixian Hong 95370161

1 Introduction

Many people nowadays have more than one electronic devices in hand. However, most of them are only using one at a time. There are some tasks that require a lot of computation, and if all of those devices can be linked together to distribute computations evenly and then do computations together, a lot of time may be saved.

For my project, I explored the possibility of implementing Distributed-Memory parallel computing on iOS Devices. Several frameworks provided in the iOS are used to achieve my implementation. I implemented a Traveling Salesman Problem Solver to demonstrate the performance and effectiveness.

Due to the limit of iOS devices I have, I only tested my implementation with two devices connecting to each other. After evaluation, the implementation can finish the same amount of computation in about 53% of the time compared with performing all the computations on only one device when the amount of computation is large.

2 Implementation Description

There are mainly 3 parts in my program. They are the User Interface part, the connection manager part, and the calculation managing part.

2.1 Program UI

I used two tabs for my program's User Interface (as shown in figure 1). The first tab is mainly to select the devices that will be connected, and messages can be sent between devices and displayed on the textfield of this tab. The "Select Devices to Connect" button will lead to a view controller which lists all the devices that can be connected, and user can tap on a device to select devices that will be connected. Once a device is successfully connected, the "connection status" row will display the names of devices that are connected. The second tab is the place to select the cities to perform the traveling salesman problem calculation. The upper section is cities' airport names with their switches to indicate whether this city will be visited or not. Two buttons are there are the screen, one is the "Confirm Selection" button and the other one is the "Compute Route" button. Once the "Confirm Selection" button is pressed, the lower region will display how many cities had been selected. After pressing the "Confirm Route" button, the program will start to perform calculations, and once all the calculations are finished, it will display the traveling distance and the order of cities visiting. The lower region is where the result will be displayed.

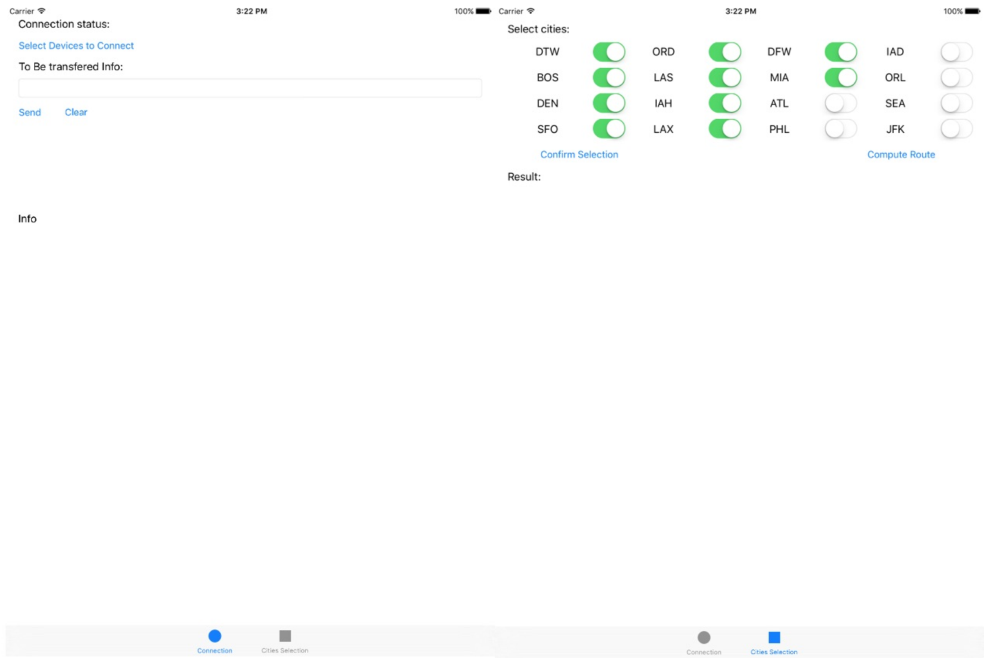


Figure 1. UI of the program

2.2 Connection Manager Part

The connection manager used the ”MultipeerConnectivity” framework provided by Apple to establish connections and transfer data between devices. I referred to both the ”iOS & Swift Tutorial: Multipeer Connectivity” by Ralf Ebert and the ”Multipeer Connectivity Framework Reference” by Apple to implement this part.

To recognize devices that are currently running the same program (or programs that are under the same communication protocol), a string is used to identify devices. This part will also save the ID of the controller for the whole system (for my implementation, the controller is the device on which user selected cities and pressed the ”Confirm Selection” button), which will be used for later communications. The management of the service advertisement and connection request handling is achieved using functions in the ”MultipeerConnectivity” framework.

I implemented several functions for sending different types of data, and to different devices. In terms of the target of those sending functions, the first one is to send the data to all the devices, and the second one is to send the data to only the controller that is mentioned previously. The controller is the device which will send control signal, like ”rebalance all the jobs” (it will be explained in section 2.3.5), ”halt computation”, and ”continue computation”, to all the connected devices. It will monitor how many devices have finished all of their computations, and how many devices have finished the redistribution phase so as to determine the next action for all the devices. As for the data types that can be sent, there are mainly four different data types. The first type is the general ”NSData” type that is supported in the ”MultipeerConnectivity” framework. The second type is a string, which is used for sending the message in the first tab. The third type is a double which, in my program, is the execution time (the use of this data will be explained in section 2.3.2). The fourth type is an array of the class ”route” that I implemented. The use of this function is mainly to redistribute the workload between different devices.

When the program received data from other devices, it will first decode the data into the correct data type. Then according to the type of the data, it will call functions from other part to handle the received data. The functions that can be called are defined in a protocol and implemented in other classes.

2.3 Calculation Management

In this section, I will explain the management of calculation jobs. Since there is no MPI or OpenMP on iOS devices, I need to maintain the data transfer/management and calculation interruption using some of the existing libraries that I can found.

For this part, I used the NSOperation and NSOperationQueue provided by Apple. I referred to the ”NSOperation and NSOperationQueue Tutorial in Swift” by Richard Turton, ”NSOperation Class Reference” by Apple, and ”NSOperationQueue Class Reference” by Apple for my implementation.

The basic idea is as follows: calculations are divided into small jobs, and then these jobs are added into one queue which is different from the main queue. Normally all the calculation will be performed in the order that they are added to the queue. However, when load-balance is needed, the queue will be suspended, and the remaining jobs will be divided and sent to other devices. After redistribution is finished, the queue will continue to execute, and repeat the previous process until all the calculations had finished execution.

2.3.1 Use of Operation Queue

The NSOperationQueue is the method that I used to manage data transfer and calculation interruption. There are two queues in general, one is the main queue, and the other one is the calculation queue. The main queue is used to handle operations that are related to the User Interface, and the handling of signals or data from other devices. The calculation queue is the queue where all the calculations are stored and performed.

The main queue is used to monitor data transfer and perform UI-related operation. All the button press and textfield update operation are added to the main queue as specified by the Apple’s iOS programming requirement. As for data transfer, the consideration for performing them in the main queue is as follows: when one device finished calculation and need to get more jobs from other devices to balance run time, it is needed to suspend the current calculation and stop all the remaining calculations. However, without the use of operation queue, all the operations will be executed on only one queue and in the order that they are added (if all the operations are of the same priority). This means that operations cannot be stopped or suspended when load-balance is needed. By putting the data-transfer monitoring operations on the main queue, I can suspend only the other queue to add operations to it or to pack unfinished operations in an array and send them to another device for calculation. When the calculation queue is suspended, I can continue to execute needed operations on the main queue. With only one queue, I cannot stop specific operations (as all the operations in the queue need to be cancelled), thus it would be nearly impossible to redistribute load and to act according to the signal from the controller.

2.3.2 Initial Load Balancing

Compared with parallel computers that had been used for previous homework, one problem with iOS devices is that the CPU performance difference between devices may be significant. For example, between the iPhone 6 Plus and iPad Air 2, the CPU performance difference is about 10%. It would be more significant between different generations of devices. Moreover, iOS devices will execute other operations like monitoring incoming SNS or phone calls while the program is running, unlike the commercial parallel computers, thus load balancing is an important task for my project.

For initial load balancing, I first timed the execution for three levels of breadth-first search (BFS). Since all the devices will perform the same amount of calculation, the time that is recorded can represent the computational power of the device. Next, the time will be sent to all the connected devices, and they will store the information. After receiving all the timing information, the program will calculate the ratio between different devices' computing power and all the connected devices' computing power. This ratio will be used to guide both the initial load balancing and the later rebalancing. At the initial load balancing phase, it will calculate how many jobs in the 3-level-BFS results should be assigned to this device, and then add those jobs in to the calculation queue.

2.3.3 Calculation operation

For the calculations, I designed a calcOperation class which inherits the NSOperation class. The class can be initialized using children generated from the 3-level-BFS result. In terms of the algorithm that is used in the class, it is depth-first-search with branch-and-bound. Initially, each class have a route that had visited 3 cities and the distance till now. At the beginning of the calculation, it will check to see if it had been cancelled, if not, it will look at and store the global min distance from this device. With this information, it will find the minimum TSP distance starting with the three cities, and if the result distance is smaller than the global min, it will store it as the local min distance, and update at the end of the execution. During this process, it will keep track of how many cities had been explored in this class, and update the global "cities expanded" counter at the end.

With the use of NSOperation, I can add calculations to the calculation queue and be able to keep track of them. During the process of creating calcOperation, I put them in a dictionary (hash-map in swift) to keep track of the correspondence relationship between each operation and the route it corresponds to. In this way, I can look at the operation queue to determine which route should be distributed to other devices (this will be explained in 2.3.5).

2.3.4 Signal Handling

There are five signals that I used to control the operations of all the devices.

The first signal is "START", it is used by the controller to tell all the devices that they should start to do calculations according to the initial load balancing's result.

The second signal is a "CONTINUE" signal, it will lift the suspension on the calculation queue, which means that calculations for the TSP route will continue.

The third signal is a "HALT" signal, it will suspend the calculation queue. The idea for this signal is to provide a method similar to the BARRIER in MPI to check if all the devices are on track. The calculation queue needs to be suspended because at the rebalancing phase, it is possible that the calculations in the queue had just been finished, and then it sends its result to all the other devices. This will disrupt the rebalance phase. By using a "HALT", I can make sure that all the calculations are halted, even though this will hurt performance.

The fourth signal is a "REBALANCE" signal, it will be explained in detail in the next section. Overall, it asks all the devices to rebalance the routes that hadn't been calculated.

The fifth signal is a "ALLFINISHED" signal, it will lift the suspension on the calculation queue, but it doesn't influence the overall execution for one round of the TSP solving.

There are two signals that are sent from devices to the controller.

The first signal is a "FINISHED" signal. If there are no more operations in the calculation queue, it will send a finished signal to the controller. If controller received this signal from all the devices, it means that the overall calculation had finished, it will send out the "ALLFINISHED" signal and display the final result on the textfield.

The second signal is a "ALLRECEIVED" signal. When the device had received the the connected devices' amount of route array, it will tell the controller that it had received them all. If all the devices had sent the controller a "ALLRECEIVED" signal, the controller will issue a "CONTINUE" signal to all the devices.

2.3.5 Rebalancing Load

In my program, once one device had finished calculation, it will send the results to all the other devices. For non-controller devices, they will just use the result to update its own global min distance variable at first. For the controller, once the update is finished, it will send a "HALT" signal to all the devices, and then it will ask all the devices to rebalance.

At this time, all the devices will go through all the operations in the calculation queue and check to see if the operation had been cancelled or is being executed, if not, the operation will be stored into an array. After going through all the operations in the queue, it will mark all the operations in the calculation queue as cancelled, and start to generate arrays of route that will be sent to other devices using the correspondence dictionary generated from the calculation operation (the operations that belongs to itself will be added back to the queue using a new calculation operation, as old one had been cancelled). The amount of routes that will be sent depends on the ratio calculated from the initial load balancing part. The higher the ratio, the more routes will be assigned to that device.

Once the device received the route array from other devices, it will start to add them into the calculation queue. It will generate a new calculation operation, add it to the dictionary, and at last add it into the calculation queue.

After receiving the amount of arrays equal to the amount of connected devices, it will send a signal to the controller that all the routes had been received. If the controller found out that all the devices had finished the rebalancing phase, it will send a ”CONTINUE” signal to all the devices.

The data that is sent between devices is an array of the class route. There is a protocol provided in iOS which is NSCoding. It provides a protocol to encode and decode the variables in the class. When one class follows the requirement of NSCoding, it can be converted to NSData using the NSKeyedArchiver class. By using these two frameworks, I was able to convert the data into the datatype that can be sent between devices and later decode the data into variables in the class.

3 Performance Analysis

For the following analysis, the devices tested are one iPhone 6 Plus, and one iPad Air 2. I looked at the single core CPU benchmark score for them on Geekbench Browser, the score for iPad Air 2 is 1810, and for iPhone 6 Plus is 1603. This means that there should be an about 10% computing power difference. For my program, since there is only one operation queue, and all the calculations are performed on this queue as the main queue only handles UI-related tasks and task reassignment, I assumed that only one core should mainly be used and thus the benchmark score taken into consideration is those for single core.

For the below results, the time information is only for the part which starts from performing the branch-and-bound Depth-First-Search, and ends after getting the final result. For the number of cities explored, it represents how many nodes had been expanded in the branch-and-bound depth-first-search step. It represents how many calculations are performed without considering the communication part.

3.1 Single Device Runtime Result

# of Cities	First Trial iPhone	Second Trial iPhone	First Trial iPad	Second Trial iPad	# cities being explored
8	0.3811s	0.3753s	0.3440s	0.3453s	23484
9	1.6414s	1.6659s	1.5178s	1.3978s	120171
10	7.1927s	7.3296s	6.4653s	6.5468s	545721
11	38.5977s	39.8846s	35.3183s	34.7615s	2992741
12	136.2869s	138.0866s	123.3584s	123.3418s	10630155
13	716.7711s	720.4814s	626.9299s	620.1934s	54765499

Table 1. Table for Raw Single Device Runtime Result

As can be seen from the above table, we can see that iPhone takes about 10% more time to perform the same amount of computation. This is consistent with the benchmark result. Another thing is that there is some small difference between different trials, as the device may have some other operations going on at the same time.

When trying to compare the performance between only using one device and using multiple devices, since there is a performance difference between devices that are used, there needs a way to uniform the above result. Here I used the method of averaging the four runtimes that I got for the same amount of cities. The consideration is that, when distributing computations, the most ideal case is that all the devices spend the same amount of time performing computations. By averaging all the runtime that I got for the same amount of cities, it is a way to get an estimate of how much time is needed to perform those amount of computation using a single device which is a ”combination” of the two devices.

# of Cities	Average Time	Divide by above Time	# cities being explored	Divide by above Cities #
8	$0.3614s$	-	23484	-
9	$1.5557s$	4.3046	120171	5.1171
10	$6.8836s$	4.4248	545721	4.5412
11	$37.1405s$	5.3955	2992741	5.4840
12	$130.2684s$	3.5074	10630155	3.5520
13	$671.0940s$	5.1516	54765499	5.1519

Table 2. Table for Average Single Device Runtime Result

For the above table, generally the amount of time increased is consistent with the amount of cities explored increased. However, there are some operations that need to be executed despite the amount of cities, thus the increase in time is a little bit smaller than the increase in # nodes being explored.

3.2 Multiple Device Runtime Result

# of Cities	Device	First Trial Time	First Trial # nodes	Second Trial Time	Second Trial # nodes
8	iPhone	$0.2647s$	4972	$0.3258s$	5218
8	iPad	$0.2747s$	19055	$0.3163s$	18717
9	iPhone	$0.9647s$	43601	$1.1246s$	49839
9	iPad	$1.0678s$	81129	$1.1251s$	73285
10	iPhone	$3.9522s$	240504	$3.9327s$	269802
10	iPad	$4.2627s$	342786	$3.9570s$	317550
11	iPhone	$20.5025s$	1593323	$20.5551s$	1593323
11	iPad	$20.7346s$	1640569	$20.9083s$	1661729
12	iPhone	$69.3651s$	5578441	$71.8152s$	5692242
12	iPad	$70.7304s$	5966922	$70.6966s$	5878940
13	iPhone	$350.7039s$	27795835	$358.1671s$	28602862
13	iPad	$357.0377s$	30256140	$352.9257s$	29519735

Table 3. Table for Raw Multiple Devices Runtime Result

As we can see from the above table, the runtime for each trial is pretty close between the two devices. This means that the load balance worked well. However, when the number of cities is small, we can see that the number of nodes being explored is not that well balanced. This may be caused by halting the operation queue for a relatively long time. But as the number of cities increased, the number of nodes being expanded by each device is also well balanced considering the performance difference between the two devices.

To convert table 3 to data that can be compared with table 2, I performed the following operation: for each trial, I took the longer time that is spent on both the two devices. This is the runtime using two devices performing the specific TSP. I averaged the time from previous step between the two trials. The number of nodes is the average of the sum of two devices' expanded nodes in each trial.

# of Cities	Average Time	Divide by above Time	# nodes being expanded	Divide by above Nodes #
8	$0.3003s$	-	23981	-
9	$1.0965s$	3.6513	123927	5.1664
10	$4.1099s$	3.7482	585321	4.7231
11	$20.8215s$	5.0662	3244472	5.5431
12	$71.2728s$	3.4230	11558273	3.5625
13	$357.6024s$	5.0174	58087286	5.0256

Table 4. Table for Average Multiple Device Runtime Result

As we can see, the general trend are similar between table 2 and table 4. There are one thing that worth note, which is the difference between time ratio and nodes ratio. There is a relatively large difference between this two number when the number of cities is small. The reason would be that since I need to halt the calculation queue when I try to rebalance load, it would hurt runtime when amount of calculation is small. The number of nodes being expanded only considers about the computations related to solving TSP. However, as the number of cities increases, load balance is a lot more beneficial to the runtime as calculations are spread on the two devices.

3.3 Multiple Device and Single Device Comparison

# of Cities	Single Time	Multiple Time	Speedup	Single #nodes	Multiple #Nodes	Computation Efficiency
8	$0.3614s$	$0.3003s$	1.203	23484	23981	0.979
9	$1.5557s$	$1.0965s$	1.419	120171	123927	0.970
10	$6.8836s$	$4.1099s$	1.675	545721	585321	0.932
11	$37.1405s$	$20.8215s$	1.784	2882741	3244472	0.889
12	$130.2684s$	$71.2728s$	1.828	10630155	11558273	0.920
13	$671.0940s$	$357.6024s$	1.877	54765499	58087286	0.943

Table 5. Single and Multiple Device Runtime and Calculation Comparison Table

As we can see from the above table, the speedup is not good when the amount of calculation is small, however, as the number of calculation increases, the speedup achieved around 1.88 when using two devices. The efficiency is around 90%, however, it should note that here, for efficiency, I only considered the amount of nodes expanded, which is different from the efficiency that have been discussed in class, as communication are ignored when calculating efficiency here.

3.3.1 Speedup Discussion

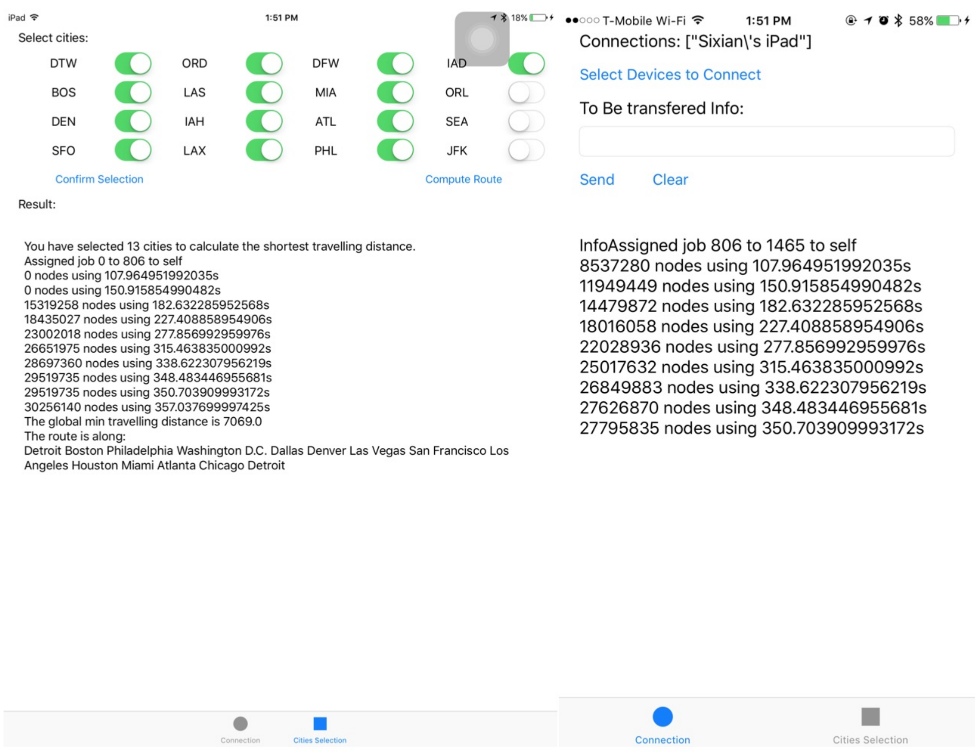
Ideally, for two devices, the speedup should be 2. For my implementation, the speedup is smaller than 2, for which I can think of two reasons.

First, when using multiple devices, data needs to be transferred between those devices so as to perform load balancing. When solving TSP on only one device, all the operations are directly related to finding the minimum traveling distance. However, when multiple devices are used, so as to minimize the time needed, devices need to communicate and send some of the jobs around so as to balance the load. This is a process that takes time, which would decrease the speedup.

Second, the performance of the two devices that I used are different, which means that it is not accurate to use the average runtime to calculate the runtime for single device. Since the difference between my two devices is only about 10%, I still used this method. However, when the difference is significant, the single-device runtime should be approximated using more accurate method. For example, one method may be to approximate the computational power for each device using the timing information, and then use these approximated results to calculate the expected runtime when treating all of them as a single device. Besides, for iOS devices, there may be some other operations being executed at the same time when I was testing my program. I closed all the background applications before I start my testing so as to minimize the influence of other apps and repeated each timing for two times, however, due to other tasks being executed at the same time, it is possible that the runtime is a little bit off from the situation that the program is the only operation that is executing.

I also want to discuss about the speedup difference when the number of cities is different.

When the amount of cities is small, we can see that the speedup is bad. The reason is that when the amount of computation is small, suspend the calculation queue would hurt the runtime a lot. Initially, my implementation only balance the load at the beginning of the calculation. However, I found from my testing that in this situation, one device would stop its calculation very early, and the other device would spend three or four more times time still doing calculations. I printed out the amount of nodes being expanded, and the first device expanded nodes much less than the second device. The reason for this behavior may be that most of the routes for the first device is bad, so it does not need to expand many nodes to finish the calculation. However, for the second device, it is possible that many routes have only a small difference, so it needs to expand most of them so as to find the minimum distance. This can also be seen from a sample output which is provided in figure 2. For my program, since the device's nodes counter is updated only when one calcOperation is finished (the calculation for one of the three-level-BFS result is finished), as iPhone (on the right) had finished all the initial jobs assigned to it (both devices would print out the number of nodes expanded and the time when one of it had finished all the jobs in its calculation queue), the iPad (on the left) still hadn't finished the first calcOperation. Because of this, I implemented the rebalancing step so as to achieve a good speedup when the amount of computation is large. But this leads to a bad speedup when the number of cities is small. When the number is small, every suspension and rebalancing would take time, and the improvement in time by rebalancing tasks may not be significant compared with the time spent on rebalancing steps.



Select cities:

DTW	●	ORD	●	DFW	●	IAD	●
BOS	●	LAS	●	MIA	●	ORL	○
DEN	●	IAH	●	ATL	●	SEA	○
SFO	●	LAX	●	PHL	●	JFK	○

Result:

```
You have selected 13 cities to calculate the shortest travelling distance.
Assigned job 0 to 806 to self
0 nodes using 107.964951992035s
0 nodes using 150.915854990482s
15319258 nodes using 182.632285952568s
18435027 nodes using 227.408858954906s
23002018 nodes using 277.856992959976s
26651975 nodes using 315.463835000992s
28697360 nodes using 338.622307956219s
29519735 nodes using 348.483446955681s
29519735 nodes using 350.703909993172s
30256140 nodes using 357.037699997425s
The global min travelling distance is 7069.0
The route is along:
Detroit Boston Philadelphia Washington D.C. Dallas Denver Las Vegas San Francisco Los
Angeles Houston Miami Atlanta Chicago Detroit
```

Connections: ["Sixian\'s iPad"]

To Be transfered Info:

```
InfoAssigned job 806 to 1465 to self
8537280 nodes using 107.964951992035s
11949449 nodes using 150.915854990482s
14479872 nodes using 182.632285952568s
18016058 nodes using 227.408858954906s
22028936 nodes using 277.856992959976s
25017632 nodes using 315.463835000992s
26849883 nodes using 338.622307956219s
27626870 nodes using 348.483446955681s
27795835 nodes using 350.703909993172s
```

Figure 2. Results for Calculating 13 Cities TSP

When the number of cities is large, my current implementation had a problem that during rebalancing, it needs to send all the rebalanced routes from one device to another. A significant amount of time would be spent on communication if the data is large in size. This may be a reason that the speedup is not close to 2 even when the program executed for around 6 minutes. There are simple solutions to this, like storing all the 3-level-BFS results rather than deleting unneeded ones at initial load balancing, but due to time limit, I didn't implement it.

3.3.2 Efficiency Discussion

For the efficiency in table 5, they just represent the amount of nodes being expanded, so their values are close to one. For actual efficiency analysis, both computation and communication operations should be considered.

For my current implementation, the minimum traveling distance is broadcasted only when one device had finished all the operations in the calculation queue. Also, for each operation, they only acquire the minimum distance at the beginning of its execution. Because of this, for multiple devices TSP solving, it is reasonable that more nodes are expanded compared with single device TSP solving. Under this implementation, less route would be pruned compared with single device TSP solving. Besides, I found that during the rebalancing phase, in some rare cases, one device would first receive the route array from the other device and start to add those routes to the calculation queue, and then it starts to assign jobs that it hadn't finished to other devices. In this case, some duplicated routes may be explored by different devices, leading to more nodes being expanded. This can be solved by adding two signals that can be received and sent by the controller to make sure that adding operations to the queue happens after all the un-started job had been finished reassigning.

4 Conclusion

In this project, I implemented a Traveling Salesman Problem Solver on multiple iOS devices. Several frameworks provided by Apple are used to communicate, exchange data, and manipulate on the calculations and operations to achieve the goal. I tested my implementation using one iPhone 6 Plus and one iPad Air 2. In terms of the result, the program can achieve a speedup of around 1.88 for 2 devices when the amount of calculations is large. However, some of my implementation are still not efficient enough, which leads to a speedup that is not perfect.

I think there are several possible directions to continue this project. The first one is to test the program using more devices, and these devices may preferably be of different generations' iOS devices. By testing the program using more devices, a better understanding of the program and its performance can be generated, and more improvements can be made to this program to further improve its efficiency. Second, MultipeerConnectivity framework supports both iOS and OS X right now, so one way may be to build a distributed-memory computing system using both iOS devices and OS X computers. In this way, some of computational-extensive job like MATLAB computation can be carried out on both computers and iOS devices, which will bring this method to a more realistic situation and bring it to everyday usage.

5 Reference

Apple. Multipeer Connectivity Framework Reference (Sep. 18th, 2013).
`https://developer.apple.com/library/ios/documentation/MultipeerConnectivity/Reference/MultipeerConnectivityFramework/`

Apple. NSCoding Protocol Reference (Aug. 8th, 2013).
`https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Protocols/NSCoding_Protocol/`

Apple. NSKeyedArchiver Class Reference (Sep. 18th, 2013).
`https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSKeyedArchiver_Class/`

Apple. NSOperation Class Reference (Sep. 17th, 2014).
`https://developer.apple.com/library/mac/documentation/Cocoa/Reference/NSOperation_class/`

Apple. NSOperationQueue Class Reference (Sep. 17th, 2014).
`https://developer.apple.com/library/mac/documentation/Cocoa/Reference/NSOperationQueue_class/`

Ebert, Ralf. iOS & Swift Tutorial: Multipeer Connectivity (Apr. 28th, 2015).
`https://www.ralfebert.de/tutorials/ios-swift-multipeer-connectivity/`

Primate Labs Inc. iPhone, iPad, and iPod Benchmarks (retrieved on Dec. 13th ,2015).
`https://browser.primatelabs.com/ios-benchmarks`

Turton, Richard. NSOperation and NSOperationQueue Tutorial in Swift (Oct. 7th, 2014).
`http://www.raywenderlich.com/76341/use-nsoperation-nsoperationqueue-swift`