An Off-line Physical Traveling Salesman Problem Solver

Lucas Guerrette, Bryan McKenney
Wheeler Ruml (mentor)

August 3, 2020

1 Introduction

This report describes a program that solves the *Physical Traveling Salesman Problem* (PTSP). The PTSP is simple to state, but hard to solve: given a set of waypoints on a 2D plane, generate a sequence of controls for a robot vehicle such that the robot visits all the waypoints as quickly as possible [11]. The PTSP is a generalization of the Traveling Salesman Problem (TSP), which is one of the most famous optimization problems of all time (at least three books have been entirely devoted to the problem [1, 4, 8]). In the TSP, one is given a matrix in which entry i, j is the cost to go from city i to city j, and the goal is to find the cheapest 'tour' that visits each city exactly once [3]. In contrast to the PTSP, actual robot actions do not have to be selected — the cost of a tour is just the sum of the appropriate matrix entries. In general, the TSP is intractable to solve: the best known methods would take longer than the expected remaining lifetime of the universe to solve even medium-sized problems. (After decades of study, fast algorithms are now known for certain special cases, however [2].) The PTSP is even more difficult. Unless the robot vehicle is able to stop instantly and turn on a dime, the fastest tour might require taking a longer route than the corresponding TSP solution — it might be faster, for example, to go straight and fast, taking a single sweeping turn, than to zigzag back and forth among scattered waypoints. Figure 1 shows an example of this: in the normal TSP example, the quickest path between the nodes is traveling in a straight line between each one, but if the robot in the PTSP example attempted that, it would have to come to a complete stop at each node, turn, and then continue towards the next node. That would be far slower than going straight through nodes 1, 3, and 5 and then turning to speed through nodes 4 and 2. (Unlike in a TSP, in a PTSP the vehicle does not need to return to the start location; the goal is only to reach each node in the fastest possible time.)

One can conceptualize the PTSP as two interdependent parts: 1) a TSP-like subproblem in which, given estimates of how long it takes to go from every waypoint to every other, one decides in what order to visit

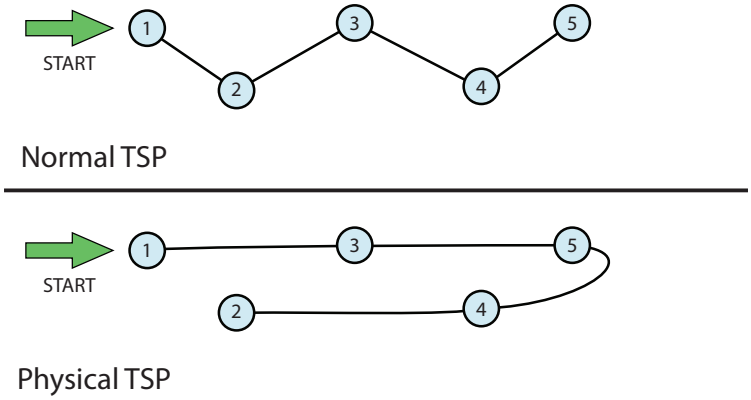

Figure 1: A solution for the TSP (top) versus a solution for the physical TSP (bottom).

the waypoints, and 2) a motion planning-like subproblem in which, given a sequence of waypoints, one plans the fastest robot trajectory along that sequence. Of course, one cannot really compute the ordering in step 1 without the travel times that are computed in step 2, and as explained earlier, the travel times depend on the ordering. In effect, the speed and direction of the robot as it travels through a waypoint is an additional implicit decision that must be optimized. This interdependency is part of what makes the PTSP such an interesting and relevant problem. There is an entire family of similar problems, which go under the name 'combined task and motion planning problems,' and now that the individual problems of task planning and motion planning are becoming better understood, these combined-type problems have been the subject of increasing research interest and numerous scientific workshops in recent years [6].

2 Research Objective

The main research objective was to write an algorithm that could solve PTSP problems, as described in Section 1. The original idea for the PTSP solver was to divide it into three separate sections. The first section would take a set of waypoints and return the most optimal visitation order based on the time it would take the vehicle to reach all of them. The second, "middle" section would take that waypoint ordering and find the best heading for the craft to hit each waypoint at so that it could continue on its way as fast as possible. The third and final section would take the waypoint ordering and headings given to it by the other two parts and return a set of controls that would get the craft to each waypoint at the desired heading and in the desired order.

3 Accomplishments

Over the course of the summer, we achieved our main research objective of developing a working PTSP solver and then improved upon it to make it more robust. In the beginning, we read papers pertaining to the two main sections that would be necessary for our algorithm — the regular TSP solver [5, 9, 12] and the motion planner [7] — and then constructed basic versions of those. These programs enabled us to learn about the parts of the overall solver we were making and get a working MVP, or Minimum Viable Product (which we called "Version 0"), in an easier manner than trying to attack the most complex algorithms to begin with.

3.1 Version 0

Version 0 was composed of an algorithm that found the most efficient order to travel between the waypoints in (the TSP solver) and an algorithm that determined what the vehicle would have to do to get between all of the waypoints using the desired vehicle physics (the motion planner).

3.1.1 TSP Solver

The TSP solver took a matrix filled with the travel-times between waypoints and then checked each set of possible waypoint orderings using a method called a *tree search* to find the one that would be the shortest. As this could become very slow with a large number of waypoints, a *branch and bound* strategy was implemented to reduce computation time; this technique involves keeping a record of the best ordering found so far and using it to "prune," or skip, orderings that become longer than it part-way through their calculation.

3.1.2 Motion Planner

The motion planner was based off of an algorithm called Rapidly Exploring Random Trees (RRT) [7]. This algorithm explores the whole map by growing a "tree" of motions towards a randomly selected point during every iteration of the loop. Each node in the tree represents a state of the vehicle, with each node being connected by a set of controls. To grow the tree, it performs many random controls starting at the node in the tree closest to the randomly selected point. The best control (the one that gets closest to the random point) is then added to the tree. Occasionally, this random point is set to the goal, so as to grow the tree in the desired direction.

3.1.3 Heading Generation

We attempted to implement the "middle" piece of the PTSP solver, as described in Section 2, but as the headings we generated were simplistic (the requested angle for each waypoint was just pointing straight

towards the next waypoint, ignoring the existence of obstacles, and the requested speeds were "hallucinated"), they did not lead to very good solutions and we found that removing this component made the algorithm more effective.

3.1.4 Mastermind

In addition to these pieces of the PTSP solver, it was necessary to create a program to make them all work together. This we called the Mastermind. It read in a PTSP problem file from command line input, ran the TSP solver to find the optimal ordering for that problem, and then fed the ordering to RRT, which would find a set of vehicle controls needed to hit those waypoints and update the time matrix with how long it took to do so. Then the process would repeat until the time matrix had been completely filled with new time values based on RRT runs rather than what they were to begin with — straight Euclidean distances between waypoints divided by a constant "hallucinated" speed value. Once this had happened, the TSP solver and RRT would be run one more time to find (it was theorized) the fastest set of controls yet, which would be written to a file. As this did not work as well in practice as in theory, a safeguard was added in which the best set of controls was tracked throughout all iterations of the Mastermind's main loop and returned at the end (so even if the final run was not the best, the best found would still be written to the file).

3.1.5 Visualization

The controls output from the Mastermind were not very easy to verify by just looking at them, so there was a need for some form of visualization. At first, we made the motion planner also create a file containing the state (direction, velocity, and position vectors) of the vehicle every 0.05 seconds and used that to draw triangles representing the vehicle's position and direction to a PDF, thus allowing the user to see where the vehicle went during the run. This was only a temporary tool, as it did not even display the location of waypoints, and was soon replaced by the animator — a program that took the controls file and played an animation of what the vehicle (still shown as a triangle) was doing. It also showed waypoint locations and radii and drew a trail after the vehicle to show what path the vehicle took once the animation was complete.

3.2 Version 1

After Version 0 of our PTSP solver was complete, we improved almost every aspect of it to make it more robust and effective. Enter Version 1.

3.2.1 TSP Solver

The TSP solver is the only piece of the PTSP solver that was left unchanged since Version 0. This is because, although it is a simple implementation of a TSP solver and not very scalable to problems with large numbers

of waypoints, it was not within the scope of this project to handle problems of that size — and for fewer than ten waypoints, the TSP solver is quite fast enough. There was no need to improve it, and we focused our efforts where they were more impactful.

3.2.2 Motion Planner

RRT was the biggest issue with Version 0 — while it could usually find a solution for single-waypoint problems, it did not often succeed in solving multi-waypoint problems. And even on single-waypoint problems, due to the mostly random generation of its trajectories, the vehicle could take unnecessarily long paths to get to the goal.

For these reasons, we replaced RRT with a newer motion planning algorithm in order to improve the speed and quality of our solutions. This algorithm is called DIRT, which stands for Dominance-Informed Region Trees [10]. While DIRT is similar to RRT in the way that it also uses random controls to grow to randomly selected points on the map, it does so in a much more informed way than RRT. A node in a DIRT tree keeps track of the same information as a node in an RRT tree but with the addition of information about its perceived quality. This new information includes the cost to reach the node (in this case a time value) and the estimated cost from the node to the goal (calculated by what is called a *heuristic function*). Lastly, it includes a radius value that is calculated based on the two previous values and the node's position relative to the other nodes in the tree. This radius, otherwise known as a Dominance-Informed Region, or DIR, is what allows DIRT to make informed decisions on which nodes are promising in the search for a good solution. The larger a node's DIR is, the more likely it is to be selected for expansion towards one of the randomly selected states.

Another big difference between RRT and DIRT is that RRT finds a solution and then quits, while DIRT finds a solution and then continues to improve it within its user-allotted number of iterations. This, as well as the more complex nodes, results in DIRT finding solutions of better quality than RRT on average (although it also takes it longer to do so).

Figure 3 shows two plots depicting the runtimes and solution costs of DIRT and RRT instances compared side-by-side. The left-hand figure shows that on average, RRT has a faster runtime than DIRT, meaning that it can find solutions to the problems faster. This can be attributed to the fact that RRT requires a lot less information and calculations than DIRT does to find a baseline solution. The right-hand image provides evidence towards the fact that DIRT produces better solutions than RRT on average, as shown by the lower average solution cost distribution of DIRT. It also shows that it produces better "high cost" solutions than RRT, as shown by the rather large difference in their highest solution costs.

Figure 4 shows the same information as Figure 3, but with all of the data points from every run of every map, instead of the averages of each map. As with Figure 3, the left-hand image shows data that

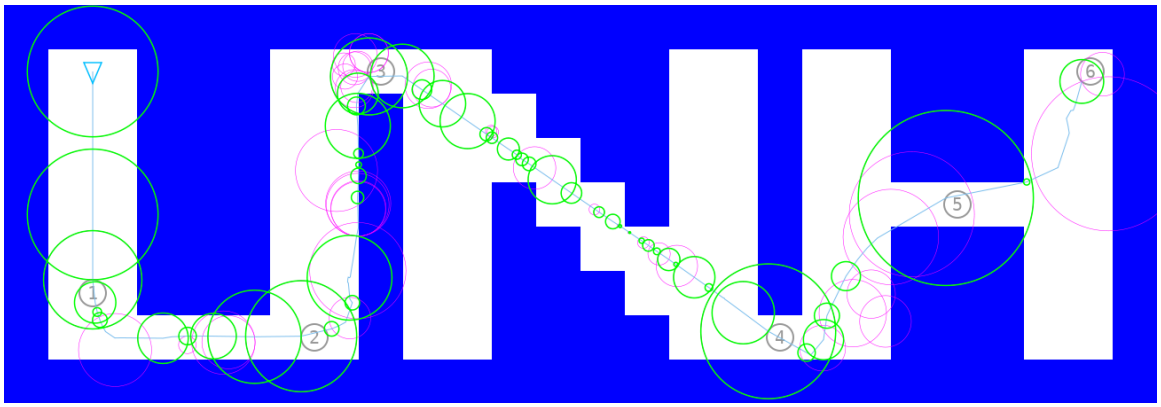

Figure 2: A visualization of the Dominance-Informed Regions (DIRs) after a run of the PTSP solver on the "UNH" problem. The initial state of the vehicle is shown as the blue triangle in the upper left, and the line coming from it is the trajectory of the vehicle during the run. The blue squares are obstacles, the gray numbered circles are waypoints, the pink circles are DIRs around nodes that were expanded but not used in the final solution, and the green circles are DIRs around nodes that are part of the final solution trajectory.

proves DIRT often has significantly longer runtimes than RRT. The right-hand image shows the relationship between DIRT and RRT runtimes, and similar to Figure 3, shows that on average, DIRT solution costs and are lower than RRT.

Figure 5 uses a different method of comparing DIRT and RRT runtimes and solution costs. Instead of stating their values outright, they display each algorithm's solution costs and runtimes as ratios of the other. This was done in order to show how many times more or less each value was when compared to the same value for the opposite algorithm. The data again lines up with previous figures, as the figure depicting DIRT to RRT runtimes (left) has a majority of values higher than one, which means DIRT had a greater runtime than RRT in that case. Similarly, most of the values comparing DIRT to RRT solution costs have values less than one, meaning that DIRT's solution costs are more often lower than RRT's.

3.2.3 Heading Generation

Heading generation was ignored in Version 0 because of the difficulty in determining what angle and speed to hit a waypoint at knowing only the ordering of the waypoints, but in Version 1 we integrated it into the Mastermind, as described below.

3.2.4 Mastermind

The Mastermind was overhauled for Version 1 with a built-in heading generation system. Essentially, for every one run of RRT in the first Mastermind, the new version would run DIRT several times, trying out

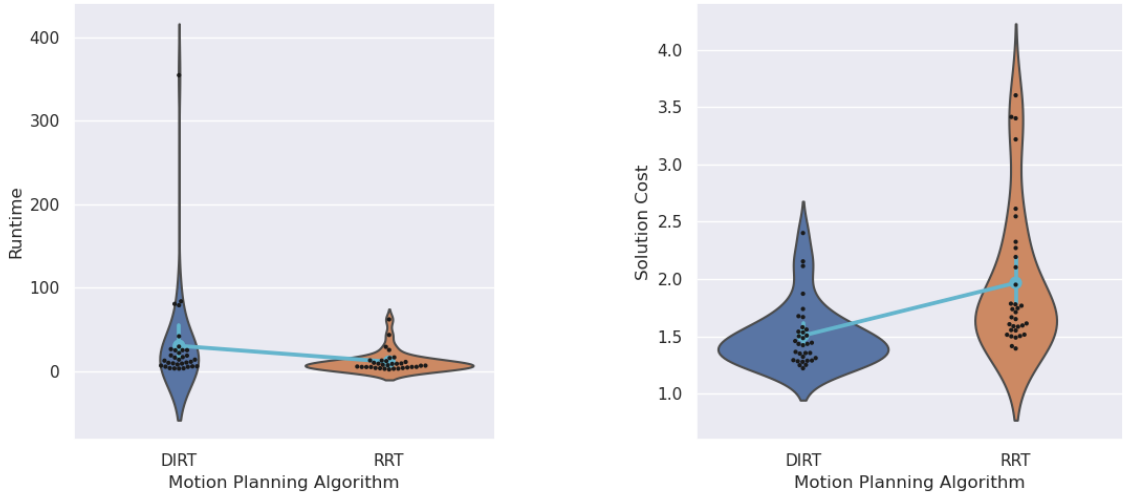

Figure 3: The left-hand image shows the average runtimes (in seconds) for RRT and DIRT on each of the 32 maps. The right-hand image shows each algorithm's average solution cost (also in seconds) for the same maps.

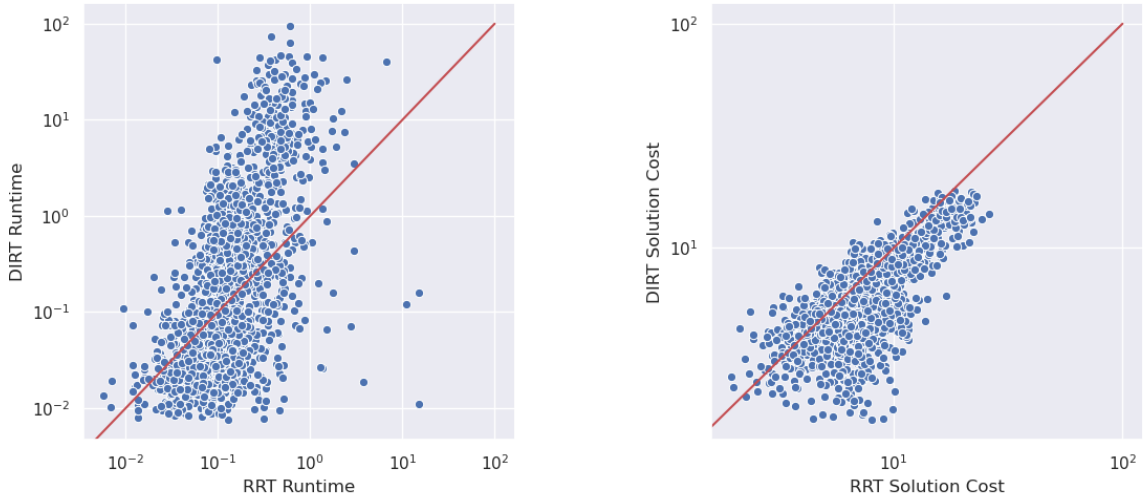

Figure 4: The left-hand image shows the comparison in runtimes (in seconds) for RRT and DIRT for 50 trials of each of the 32 maps. The right-hand image shows each algorithm's solution cost (also in seconds) for the same trials. The line x = y is plotted on both for reference.

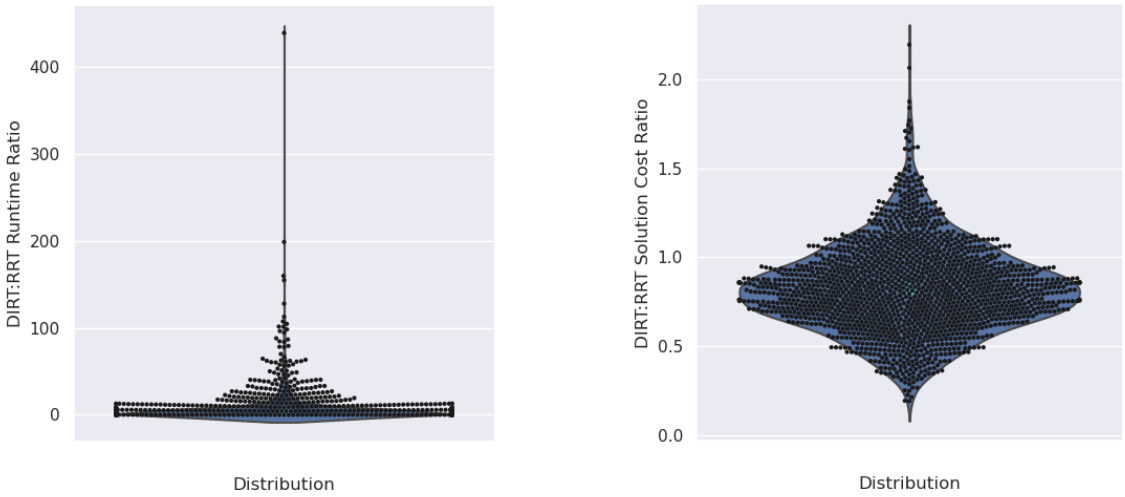

Figure 5: The left-hand image shows the distribution of the runtime ratios between DIRT and RRT. The right-hand image shows the distribution of all solution cost ratios between DIRT and RRT from all 50 instances of the 32 maps we created. In both cases, numbers above 1 indicate that DIRT's value for that statistic was larger, and numbers less than 1 indicate that DIRT's value was smaller.

different angles and velocities to hit the next waypoint at and choosing the set of controls that hit that waypoint *and the one after it* (at any heading and speed) in the shortest time. This solved a bug we had encountered previously where DIRT's heuristic function would convince it that running the vehicle straight into walls was the fastest way to get to the goal.

As running DIRT forty-eight times (two legs were run for every combination of eight different angles and three different velocities) every iteration of the main loop caused the problem to take a long time to solve, we sped it up by introducing a system by which all DIRT runs were cached and, when a similar motion-planning situation arose later in execution, either the cached run would be reused (if it was successful) or DIRT simply would not be run again (if the cached run was a failure, indicating that this situation was "impossible" to motion-plan).

3.2.5 Visualization

In Version 1, the animator was improved in various ways, including the ability to create GIF files of the animations for easier viewing of results.

We also created a DIRT tree visualizer that displayed the Dominance-Informed Regions (radii) of all the DIRT nodes that were in the tree at the end of a run (whether successful or not). We used this to debug DIRT during implementation; we could see what the program was trying to do and use that information to

discover why it might be failing. The tree visualization for a successful run of the "UNH" problem is shown in Figure 2.

3.3 Results

To prove the effectiveness of our PTSP solver, we created a program that would randomly generate PTSP problems and then ran the solver once on each of 1000 maps generated in this manner. In addition, we hand-crafted several maps to test how the solver performed in specific situations and ran the solver 50 times on each of these. Animations of a run on each hand-crafted map are displayed on the following webpage: `https://imgur.com/a/SnuDWzQ`. Out of the 1000 random runs, only one failed to find a trajectory, giving the solver an overall success rate of 99.9% on these problems. The solver failed on Problem 378, which we ran again afterwards and found a solution for (this solution is shown at the bottom of the aforementioned page). It should not have been a particularly hard problem to solve, and as the animation proves, it *is* solvable by our algorithm, so we attribute its failure during the testing runs to a bad random seed — some seeds are guaranteed to cause failures for any algorithm that relies on pseudo-random number generation.

When evaluating the data from the 1000 runs, one can very quickly begin to see a pattern involving the number of orderings and three other variables: runtime, solution cost, and number of calls to the motion planner. The number of orderings on a map directly effects the runtime and number of calls to the motion planner for obvious reasons. The more orderings it has to solve paths for, the more calls are going to be made to the motion planner and the longer the program will take overall, which can be seen in the left-hand image of Figure 9. Solution cost is also often effected by the number of orderings, as in most cases, the more points the vehicle has to go between, the longer the path is, and the higher the cost. In some cases, however, the waypoints can be grouped close together, which could lead to a shorter solution cost than a map with less orderings but with waypoints that are spread out more. This can be indirectly seen in the somewhat visible bunches of data in Figure 10 which correspond to the same bunches that can be seen in Figure 9.

We also decided to investigate the median run of the 1000 based on CPU time. We found this run to have a CPU time of around 190 seconds. We took this problem and ran it 1000 times, in an attempt to get more fine-grained data about the "average" PTSP problem (from our data set). When we did this, we found that on average, this map was actually nowhere near the median CPU time of all of the maps, as we were originally led to believe. As can be seen in Figure 6, it actually had an average CPU time of around 50 seconds, and 190 seconds is one of the highest CPU times out of the 1000 runs on that problem. This shows that chance — that is, random seeds — can still play a large role in the quality and runtime of solutions, even with the upgrade from RRT to DIRT. The single run on this problem in the original test set was simply "unlucky."

As for the hand-crafted problems, they each test a different aspect of the PTSP solver. "Bugtrap"

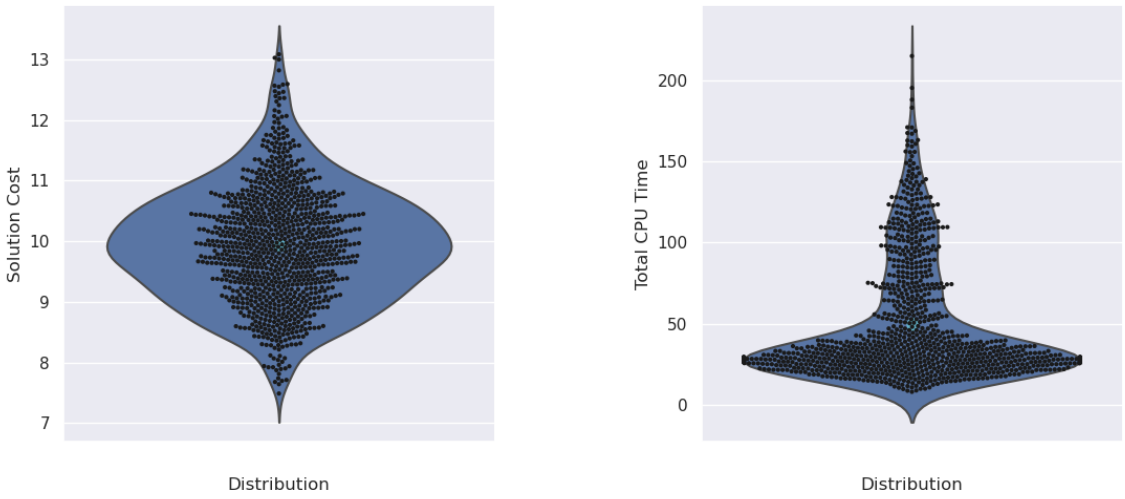

Figure 6: Violin plots comparing the distributions of solution cost and total CPU time of all 1000 runs of the "median" random map.

and "Scatter" showcase the solver's ability to navigate around obstacles, and "Scatter" further shows the precision of DIRT in allowing the vehicle to slip through the crack where two walls touch at the corner. While this is not necessarily a realistic interaction (unless the square obstacles are considered to be bounding boxes around smaller, perhaps not rectangular, objects), it is expected, as the vehicle only takes up a single point in space (the triangle in the animations is just to show the viewer the vehicle's position and direction more easily). "Diagon Alley" demonstrates the effectiveness of the headings chosen by the solver — eight different arrival headings are tried for every waypoint, and these are angles in increments of 45 degrees, but they are offset by the angle from the previous waypoint to the next waypoint so that in situations like in "Diagon Alley" the vehicle can keep heading straight rather than slowing down to hit each waypoint at 0 or 90 degrees. "U-Turn" showcases the effect of updating the time matrix after every iteration of the main loop within Mastermind — the initial ordering was chosen as 1-2-3-4-5 (taking the shortest path between each waypoint), but soon the solver "learned" that it was faster overall to go straight through 1-3-5 and then loop around to 4-2. This problem exemplifies the difference between the TSP and the PTSP, as predicted in Section 1 (when referring to Figure 1). "Corners" and "Selector" target a specific problem that we were having with the solver, which is described in Section 3.2.4. They show that the vehicle no longer runs through waypoints at full speed when that would be detrimental (like when it would hit an obstacle or the edge of the map), but instead slows down and turns to better get to the next waypoint. Lastly, "UNH" proves that the solver can handle large maps with many obstacles and waypoints (the animation is shown at half the scale of all the others due to the map's size). While all of the other hand-crafted maps had a 100% success

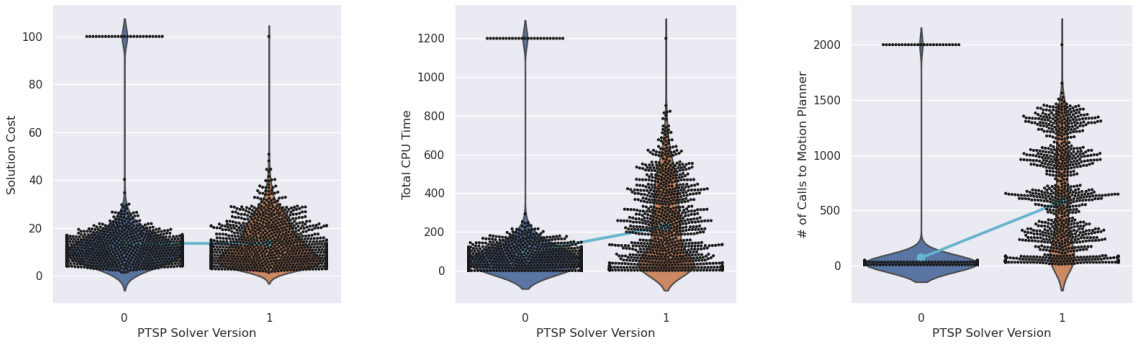

Figure 7: Violin plots comparing the solution cost, total CPU time, and number of calls to the motion planner between Version 0 and Version 1 of the PTSP solver. The numbers at the very tops of each graph are placeholder values that represent failed runs.

rate over the 50 trials, "UNH" only had a 32% success rate. This poor performance is almost certainly the result of the size of the map and the fact that we limited DIRT during these runs to performing 10,000 iterations, which it seems is not usually enough for the solver to find a successful path between some of the waypoints. Increasing the number of iterations that DIRT runs for would undoubtedly increase the success rate for "UNH" and other large problems.

As a final test, we decided to compare the original version of our solver (called Version 0) to our most recent and (theoretically) most effective solver (called Version 1). We used the same 1000 random problems that we tested Version 1 with for Version 0, and to our surprise, Version 0 was still quite versatile. Version 0 had a 97.5% success rate, compared to 99.9% for Version 1, which shows that both were effective, with a performance increase in Version 1. However, the results show that the average solution cost between the two (when accounting for failures as well) is almost exactly the same. Along with this, the overall CPU time for Version 1 is much greater than Version 0. These results can be seen in Figure 7.

4 Challenges

Throughout the course of our REAP project we have encountered numerous challenges, ranging from simple typos or errors in code to having to build up a relatively advanced understanding of topics that we have never worked on or seen before. The smallest errors, while being the easiest to fix (once located), were also the most common. Examples of these include mistyping a variable name or not deleting parts of code that were no longer necessary, both of which would give us an error message when we ran our program. As time went on however, these errors occurred less and less.

The only other trouble we had specifically with code failure (as opposed to the program just not working

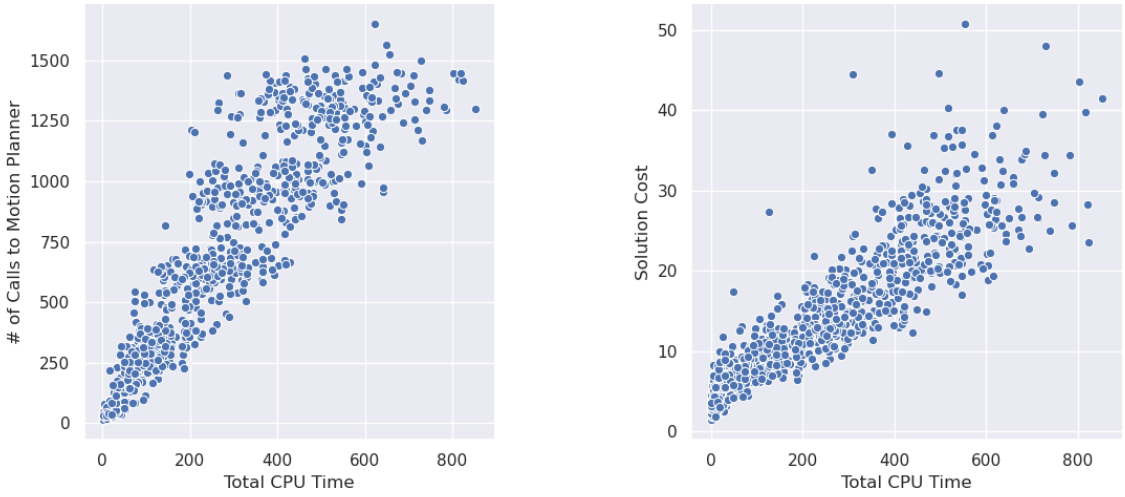

Figure 8: Scatter plots showing the positive correlation between the number of calls to the motion planner and solution cost versus the total CPU time.

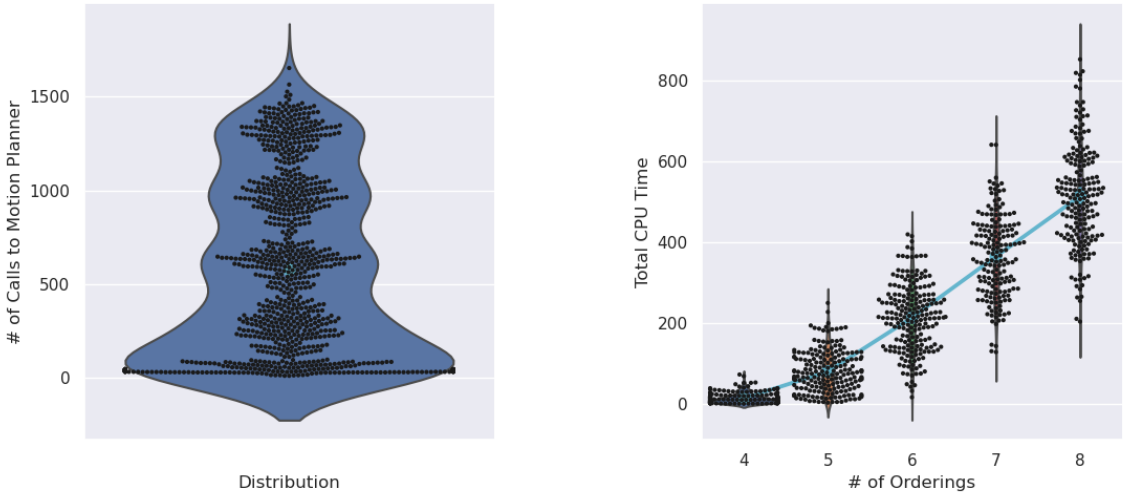

Figure 9: The plot on the left shows the distribution of how many calls were made to the motion planner in each of the 1000 runs, and the plot on the right shows the distribution of the runtime for each possible number of orderings tried for those same maps.

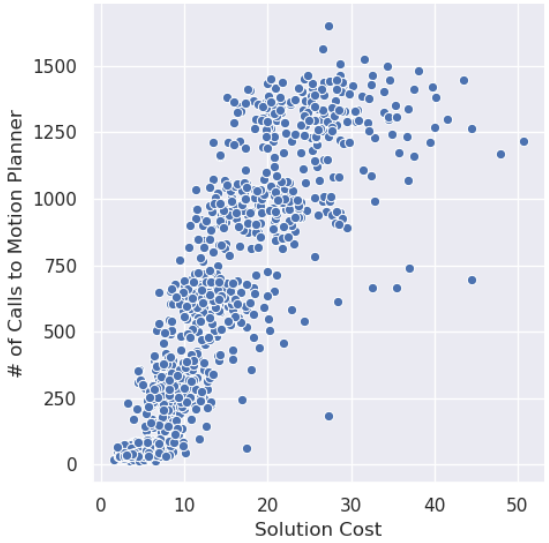

Figure 10: A scatter plot showing the distribution of number of calls to the motion planner compared to solution cost.

as intended) was with null pointer exceptions. These errors happen when you tell the program that it is going to get data from somewhere, and then it never receives that data. These would occur quite frequently, and were much harder to diagnose than simple syntax errors like the ones mentioned above. Fixing these errors usually required going through much of the surrounding code where the error occurred to see why this data was missing. Though they were harder to fix than other bugs, we eventually started seeing patterns in their appearances and were able to find the source of the problems more quickly.

Another challenge that we quickly ran into when starting this project was the lack of a crystal clear direction to go in at the start. For example, with all of the projects that we did in CS 415/416 over freshman year, we were given a relatively detailed instruction sheet on what was expected to be in the program, and how each piece would work. For this project, we were forced to plan out almost everything about the solver ourselves, except for the actual algorithms that we would use (and even within those we had some discretion). While this was at first a somewhat daunting task, with the help of Professor Ruml and good planning and communication between the two of us before writing anything big, we were able to overcome it.

At the beginning of the summer, neither of us knew very much about our problem and research goal. Because of this, we had to start off with a lot of background reading on the algorithms we would be using, and on the problem itself. This was so we could bridge the gap between our knowledge and the knowledge needed to understand the problem and effectively create a program with the ability to solve it.

Similar to the last issue, understanding the research papers that we had to read in order to learn about

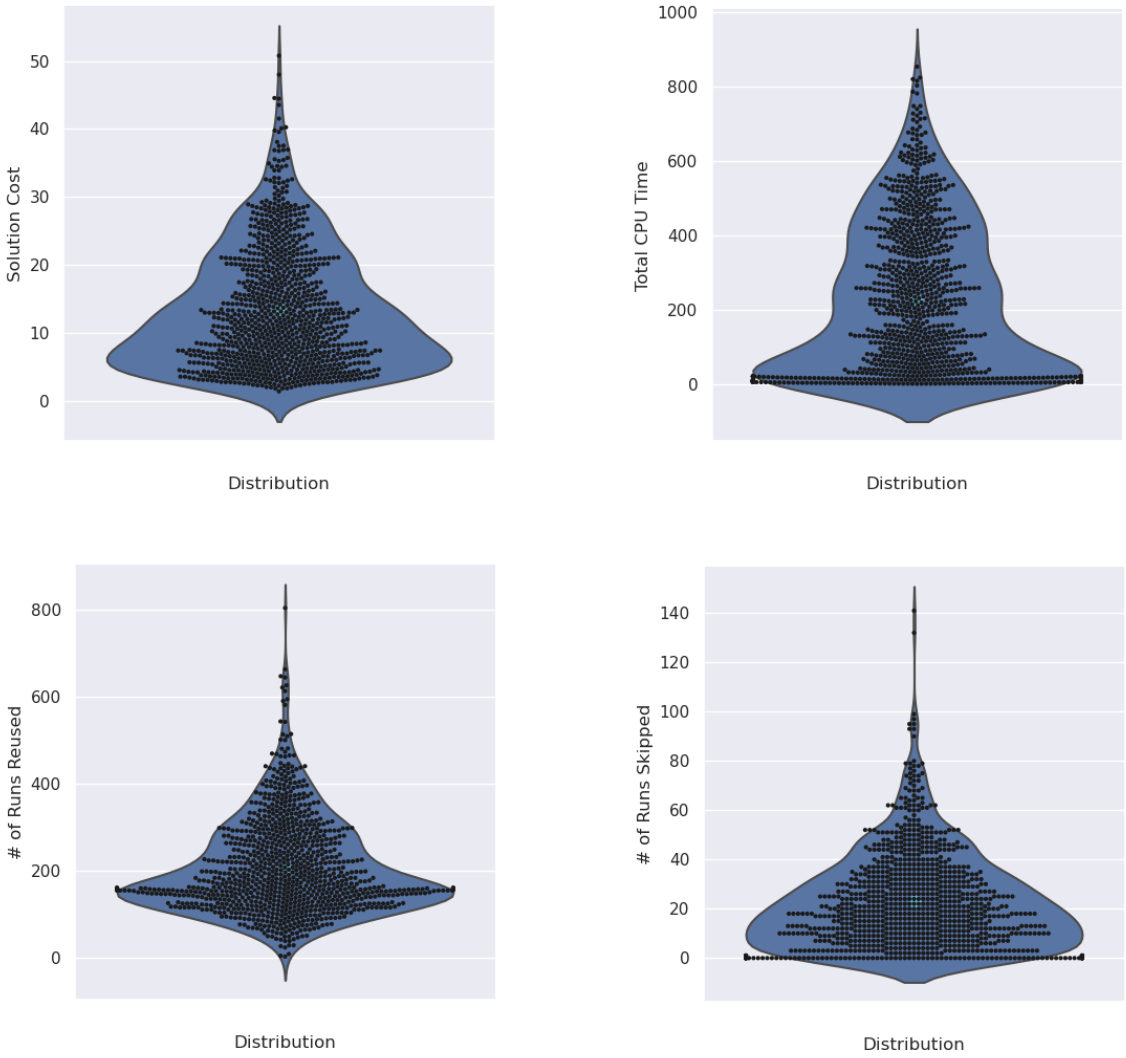

Figure 11: Four violin plots displaying the distributions of individual sets of data gathered from the 1000 random map test.

our problem was a challenge itself. Typically these papers are meant for people with a decently high-level understanding of the topic. As you can imagine, trying to read and understand them as rising sophomores who just completed their Intro to Computer Science classes was a somewhat difficult task. Again, with the help of Professor Ruml, we were able to build up a good understanding of the papers we read.

One unique challenge that we faced was handling automation of testing our code. To do this effectively, we had to learn about a language that neither of us had used before called Bash. Bash is useful for running a program (or programs) many times automatically instead of doing it manually. Similar to our other challenges, while this was rather difficult to do at first, over the course of the project we have continued to improve, and we are both decently skilled in it now.

A problem that frequently plagued us throughout the project that we never managed to completely get rid of was our vehicle spinning for no apparent reason. This can almost certainly be attributed to the fact that controls are randomly chosen, and it has little effect on performance, but it was something that we wished to fix and never could.

An issue that we ran into when trying to create our implementation of DIRT was actually with the paper that introduced and described it. Due to the constraints of paper size that can be submitted to conferences (which more or less act as faster-moving journals), the authors omitted details about implementation of the algorithm that would have been very helpful for us. Examples of this include a lack of an explanation of pruning (keeping the number of tree nodes small so the algorithm runs faster) and mismatching names of important variables in one part. Without this information, there was some guesswork involved in our implementation, making it take a bit longer than it would have otherwise.

Finally, DIRT was a bit more difficult to fine-tune than RRT, because DIRT needed a heuristic function to work effectively. Only the nature of the function was given to us (it had to predict how long it would take to get from the current state to the goal), so we had to figure out with how to implement it on our own. This was done incorrectly at first, which caused DIRT to run into walls and not find solutions for certain problems. Professor Ruml was able to help us come up with a better one, and we never experienced this problem after that.

5 Research Significance

While we did not make a significant contribution to the field of computer science with our research, we did build up our own knowledge and skills in a very important area. The Traveling Salesman Problem, as mentioned in Section 1, is a significant "unsolved" classic computer science problem itself, and we took that a step further by working on the Physical Traveling Salesman Problem, which brought us into the realm of what is called "task and motion planning." This line of research is very relevant for areas like robotics and artificial intelligence. These two fields are becoming hugely relevant in today's society, and many of

the algorithms we learned about can be generalized to other problems or fields, giving them a much larger range of possible applications. As both of us are majoring in computer science, it is clear to see how this experience will help us attain our educational goals, and the concepts and skills that we have learned apply to nearly any computer-science-related position as well.

6 Discussion

The budget for this project was more than sufficient, as we did not even need to use most of it (the money for gas) and we did not find ourselves needing anything that we did not request.

Over the course of our REAP project, both of us have learned a lot of new information and skills that will undoubtedly be useful throughout college and into our professional lives. The most valuable aspect of our learning was probably when it came to the organization of complex algorithms and what pieces go into them — things like *heuristic functions* and *branch and bound* that we had never heard of before but that are essential to know about in the field of computer science. We also greatly improved our programming skills in three languages — Java, Python, and Bash. We had not even known of Bash before beginning this research, but we learned how powerful it can be for running a series of commands automatically. We also used more efficient data structures that were completely new to us, such as hash maps. Another major thing we learned over the duration of this project was how to read and understand high-level research papers. At the beginning, a lot of the terms and math in the papers may as well have been a foreign language, but over time, by reading lots of papers and getting help from Professor Ruml and other students with the questions we had, we began to pick up more and more information from each paper. While we are still certainly by no means experts, we are usually able to read a paper and understand rather well what is going on. Finally, working on this project has improved both of our collaboration soft-skills, as well as our ability to use collaborative software. In typical CS classes, you are not often (if ever) allowed to do group work on big projects, as it is hard to assess what you as an individual know. However, most work done in the real world is done collaboratively, so bigger and better programs can be created. By being able to work as a team throughout the summer on one project, both of us have developed sought-after skills that one typically wouldn't get from a classroom environment. Use of collaborative software was also a key aspect of what we learned, as it allowed us to get much more work done in a shorter amount of time than we would have otherwise. Our first important piece of software that we learned how to use was Git. Git (which we used with the website GitHub) is a version-control system that lets you share a codebase with other people over the Internet, which was key for being able to work on the same project in a remote work environment. The second piece of software was Visual Studio Code (VS Code). VS Code facilitates something called "pair programming," in which multiple people work on the same part of a program together at the same time. VS Code does this in a similar manner to Google Docs. By using both of these pieces of software, implementing

DIRT was considerably faster than when we implemented RRT without them. Other useful software that we explored includes LaTeX, which we used (on a site called Overleaf) to write and format this paper, and Seaborn, a Python library that we used to create all of the plots for this paper. In addition, we had to learn how to set things up to work on a remote file server (UNH's Agate and Mica servers).

The best part of this experience was learning so many new things and being able to collaborate to achieve an end product far more advanced than either of us would have been able to accomplish alone. The worst part was when we encountered problems and bugs that took us much longer than we thought they should to solve, but this is a staple of computer science and is to be expected.

In the future, if we were to continue this project, a few immediate ideas for improvement come to mind. First, the stopping condition for our solver is not ideal, as currently the main loop is just run a set number of times based on the number of waypoints in the problem (many potential orderings are left unexplored), which could lead to a sub-optimal solution being chosen. We could change it so the solver caches orderings it has tried and only stops once it is about to repeat an ordering — for that ordering should theoretically be the closest-to-optimal one. Machine learning could potentially be utilized as well for updating the time matrix intelligently and saving time by not having to run DIRT as much. Second, we were unable to get pruning to work for DIRT, but figuring that out would likely decrease the runtime of the solver considerably. Another way to decrease runtime would be to migrate the codebase to a more efficient programming language, like C++, rather than Java. Lastly, we could prevent the vehicle from spinning so much by introducing a penalty for it doing so in the heuristic function.

7 Acknowledgments

Thanks to the Hamel Center for Undergraduate Research and the donors who provided us with the funding for this opportunity, to Professor Ruml for agreeing to take on two "REAPers" and giving us guidance all throughout this project, and to Matt Westbrook, Zakary Littlefield, and Kostas E. Bekris for providing us with their DIRT implementations, which we attempted to use for debugging our own.

References

[1] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press, 2007. Princeton Series in Applied Mathematics.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, third edition, 2010.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[4] Gregory Gutin and Abraham P. Punnen, editors. *The Traveling Salesman Problem and Its Variations*. Springer, 2002. (Combinatorial Optimization Book 12).

[5] Paris-C. Kanellakis and Christos H. Papadimitriou. Local search for the asymmetric traveling salesman problem. *Operations Research*, 1980.

[6] Scott Kiesel, Ethan Burns, Christopher Wilt, and Wheeler Ruml. Integrating vehicle routing and motion planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS-12)*, 2012.

[7] Steven M. LaValle and James J. Kuffner Jr. Randomized kinodynamic planning. *The International Journal of Robotics Research*, 2001.

[8] Eugene L. Lawler, Jan Karel Lenstra, Alexander H. G. Rinnooy Kan, and David B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, 1985.

[9] S. Lin and B. W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 1973.

[10] Zakary Littlefield and Kostas E. Bekris. Efficient and asymptotically optimal kinodynamic motion planning via dominance-informed regions. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018.

[11] Diego Perez, Philipp Rohlfshagen, and Simon M. Lucas. The physical travelling salesman problem: WCCI 2012 competition. In *2012 IEEE Congress on Evolutionary Computation*, pages 1–8, 2012.

[12] César Rego, Dorabela Gamboa, Fred Glover, and Colin Osterman. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research*, 2011.