2001

Searching for a Quantum Algorithm to Solve the Traveling Salesman Problem

Buckley Hopper

2001-2.0

Searching for a Quantum Algorithm to Solve the Traveling Salesman Problem

Buckley Hopper, The University of Alabama in Huntsville

Dr. Shawn Pethel and Dr. Charles Bowden, U.S. Army AMCOM, Advisors

Abstract

The traveling salesman problem - given n cities, what is the shortest path that visits each city only once - is a problem in the computational complexity class known as NP-Complete. This class of problems is assumed to be intractable on a classical computer. Quantum computers show promise of being able to perform operations that are intractable on a classical computer in a finite amount of time. I seek to apply current known quantum computer algorithms to solve the traveling salesman problem. In addition, there exist approximate solutions to the traveling salesman problem, offering solutions no greater than 1.5 times the length of the shortest possible route through the cities. I also seek to find out if a quantum computer can solve the approximation problem and provide better bounds (less than 1.5 times the shortest path length) on the approximate solution.

Introduction

The traveling salesman problem exists in various forms such as the directed path traveling salesman problem or the n-dimensional traveling salesman problem. For my purposes, I chose to solve the traveling salesman problem in its traditional form. A traveling salesman must make a tour of n cities. Each city has roads connecting it to all other cities. Find the shortest tour of the cities that visits each city only once.

Although it sounds simple enough, the traveling salesman problem is a member of a group of problems known as NP-Complete. These problems are assumed to be intractable on a classical computer, meaning that as the number of elements in the problem (in this case cities) increases, the time required to find a solution to the problem increases exponentially. Given a traveling salesman problem with enough cities, a classical computer cannot solve it.

Approximations to these problems can be made, and depending on the approximation, the new problem may be tractable on a classical computer. However, since the new problem is just an approximation of the original one, the solution is never quite as good as the exact solution to the original problem.

Enter the quantum computer. Quantum computers, although still in the theoretical phase of development, have already been shown to be able to drastically improve on the time required to solve a problem by a classical computer. Grover's algorithm can find an element in an unordered list of n elements using only order square-root of n operations. A classical computer requires order n operations to perform the same task. Shor's algorithm takes the problem of finding the prime factors of a number and makes it tractable, whereas on a classical computer it is intractable.

The field of quantum computing draws upon fields that have before been far-removed from one another. An understanding of computer science is essential because the aim of quantum computing is to come up with useful algorithms for the computer. At the same time, an understanding of physics is necessary because only a quantum physicist knows what the quantum computer might be capable of doing. For this reason, papers published in physics journals regarding the search for quantum computer algorithms often look like they might be more appropriate in a computer science journal.

I set two goals for the project. First, the traveling salesman problem is intractable on a classical computer, but quantum computers offer the possibility to solve classically intractable problems, so I decided to look for a quantum algorithm to solve the traveling salesman problem. Second, classical approximation schemes allow for finding an approximate solution to the traveling salesman problem, but I wanted to see if a quantum computer could improve on the

approximation, so I decided to try to come up with an approximation algorithm of the traveling salesman problem that a quantum computer could solve more efficiently than a classical computer.

Background

Grover's and Shor's Quantum Computer Algorithms

A quantum computer works by exploiting superposition on the quantum level. A classical particle exists in only one state at a time: its physical properties are known with certainty. A quantum particle exists in a superposition of states. An example is the spin state of an electron. In quantum mechanics, the electron's spin state is not just one of spin up or spin down, but it is a superposition of the two states - the electron's spin state is spin up and spin down at the same time. Whereas a classical computer has bits (binary digits) that can represent either 0 or 1, a quantum computer has q-bits (quantum bits), and these bits can represent a superposition of 0 and 1. A classical computer can store a number in a register and increment this number to go through all the numbers one at a time, but a quantum computer can store a superposition of states in its register, allowing it to simultaneously represent all the numbers in a given range at once. By operating on this register, a quantum computer can try all possible combinations at once, drastically improving on the classical computer's computation time of trying each number in turn. Grover's and Shor's algorithms are two ways in which a quantum computer can solve problems faster than a classical computer.

Grover's algorithm is useful for finding an element in an unordered list of n elements. This can be thought of as looking up a name in a phone book or finding the combination to a lock. It relies on the use of an "oracle" as well as Grover's diffusion matrix. This oracle is a

quantum operator with 1's everywhere except for the position of the desired element, and -1 in that position. The algorithm works by starting with a uniform superposition of the elements in the list. Then the oracle and diffusion matrix are applied to this list. After applying these order square-root of n times, the desired element of the list can be measured with near certainty. In this way, Grover's algorithm allows a search that would take order n operations on a classical computer to be performed in only order square-root of n operations.

Shor's algorithm is useful for finding the prime factors of a number. This may not seem that valuable on the surface, but it is important for security reasons. Current encryption schemes rely on the intractability of the prime factorization problem to make it impossible to find the decryption key for encrypted information using a classical computer. If quantum computers become a reality, Shor's algorithm will make it necessary for us to find a different way to encrypt sensitive information. Shor's algorithm works by generating a periodic function from a superposition of all possible factors of the number in question. The system is then measured and the wave function is partially collapsed. By using the quantum Fourier transform on the resulting wave function, the algorithm can find the period of the wave function. Then Euler's Totient Function and the Chinese Remainder Theorem are used to find the prime factors.

Classes of Computational Complexity

Computational algorithms can be divided into two major groups. One of these groups is polynomial-time problems, or class P. A problem in class P with n elements requires order n^k operations to find the solution, where k is a finite number. These problems are tractable on a classical computer because the time required to solve them increases in a polynomial manner with the complexity of the problem. The other major grouping of computational complexity is exponential-time problems. A problem in this class with n elements requires order k^n operations

to find the solution, where k is a finite number. These problems are intractable on a classical computer because as the number of elements increases, the amount of time required to solve them increases exponentially.

A subclass of the exponential-time problems is nondeterministic polynomial-time problems, or class NP. These problems can be solved in polynomial time on a nondeterministic computer. A nondeterministic computer can be thought of as a combination of a guessing unit and a checking unit. The guessing unit generates random guesses at the solution to the problem and the checking unit checks this solution to see if it is correct. If a problem is in NP, a solution can be checked in polynomial time.

NP-Complete problems are a subclass of NP. These are problems that have been proven to be at least as hard as the hardest NP problem. If an algorithm to solve one of these problems in polynomial time can be found, then all problems in NP can be solved in polynomial time. The traveling salesman problem is NP-Complete, so finding an algorithm that makes the traveling salesman problem tractable will make all problems in NP tractable.

NP-Complete Optimization Problems

The NP-Complete class consists of hundreds of different problems, but there are six that form a representative set. These are 3-satisfiability, 3-dimensional matching, vertex cover, clique, Hamiltonian circuit, and partition.

First in the set of canonical NP-Complete problems is the 3-satisfiability problem, which consists of a collection of internal variables and a set of clauses. Each variable can take on a value of either true or false. A clause looks at 3 of the variables and if the variables have the correct values, the clause is satisfied. Given these conditions, is there a set of variables for which all the clauses are satisfied?

Next is the 3-dimensional matching problem. This is the 3-dimensional version of a problem known as the marriage problem, which has 2 dimensions. In the marriage problem, n women and n men are going to be married to one another. Is there a matching of women to men that results in all couples being happy? The 3-dimensional matching problem is the same except instead of men and women forming couples, there would be 3 groups forming triples.

The vertex cover problem is a set of vertices (points) joined by edges (lines) and a positive number k. Is there a set of k or less vertices such that each edge is touched by at least one of them?

The clique problem consists of a set of vertices joined by edges and a positive number j. Is there a set of j or more vertices such that each member of the set is connected to all other members by an edge? One way to think of this is to think of a group of people (a clique) who all know each other.

The Hamiltonian circuit problem consists of a set of vertices joined by edges. Is there a Hamiltonian circuit through the points that only visits each point once? This is similar to the traveling salesman problem, except the traveling salesman problem has each city connected to all other cities and the solution is the shortest path. Here the solution is whether there exists a path, and not all the "cities" are connected to each other.

The final member of the representative set of NP-Complete problems is the partition problem. One way to think of this problem is as a set of different-length straws. Can the straws be divided into 2 groups such that the sum of the lengths of the straws in one group is equal to the sum of the lengths of the straws in the other group?

Approximation Techniques for the Traveling Salesman Problem

In order to make NP-Complete problems tractable, researchers have come up with various approximations that yield better computation speed at the expense of accuracy. For instance, there is an approximation to the traveling salesman problem that is known to produce results to within one and a half times the length of the shortest path in polynomial time. The algorithm to do this begins by finding what is called a minimum spanning tree. For a set of n cities, a spanning tree is a set of n-1 roads that connect all the cities. The minimum spanning tree is simply the shortest such tree, and algorithms to find the minimum spanning tree work in polynomial time. The length of a minimum spanning tree must be less than the length of a traveling salesman tour because deleting a single road from the minimum tour produces a spanning tree (not necessarily the minimum spanning tree). The next step takes the minimum spanning tree and adds a minimum weight matching to make it an Eulerian graph. A minimum weight matching is, by definition, the minimum-length set of roads that, when added to the minimum spanning tree, produces an Eulerian graph. The minimum weight matching is less than or equal to half the minimum tour length. An Eulerian graph is a set of cities and roads such that each city has an even number of roads connecting it to other cities. Once all these steps have been taken, measuring the length of the Eulerian graph produces a value that is no greater than 1 and a half times the length of the minimum traveling salesman tour.

Methods

I began my search for quantum algorithms for the traveling salesman problem by gathering information from the fields involved. I gathered information from the computer science field on classical algorithms for the traveling salesman problem. In order to understand

the problem better, I wrote a program to compute minimum traveling salesman tours on a classical computer and display them to the screen. Source code for that program and sample plots are available in the appendix.

At the same time, I began trying to figure out how to use a quantum computer to solve the traveling salesman problem. I started by understanding Shor's algorithm and how it could offer a polynomial-time solution for a classically intractable problem. The drawback with Shor's algorithm was that it was a solution to the prime factorization problem, and that problem was not NP-Complete. This meant it could not be mapped into an NP-Complete problem like the traveling salesman problem.

Next, I looked at Grover's algorithm. It offered only a square-root increase in performance, so this meant that the solution would still be intractable. Nevertheless, it would be a performance increase over classical computers.

Having looked at Shor's and Grover's algorithms, I began thinking of them as a "quantum toolbox" of sorts. I wanted to use parts of these algorithms as part of my eventual solution.

Bringing these things together, I began looking at just how a quantum computer could solve the traveling salesman problem. The power of Shor's algorithm was that it used a quantum Fourier transform to find the period of a periodic function. I tried to think of ways to represent the traveling salesman problem as a periodic function with the shortest path as the period, but I did not come up with a solution along those lines. The only solution I found for the exact traveling salesman problem was using Grover's algorithm, and even this did not produce a problem that was tractable.

Finally, I turned my attention to approximation problems. I saw how the approximate

solution for the traveling salesman problem worked, and in order to improve the bounds on the approximation, I needed to find something about a group of cities and roads that a quantum computer could compute. The classical approach used the ideas of a minimum spanning tree and an Eulerian graph, so I tried to think of similar geometric interconnections between the cities that a quantum computer could find in polynomial time.

Results and Discussion

Using Grover's Algorithm to Solve the Traveling Salesman Problem

Although not a satisfying result because the problem is still intractable, this solution allows the minimum path for 2n cities to be computed in the time once required for n cities. This involves representing the traveling salesman tours in a quantum register so that only valid tours are present. If the cities are just placed in a register and placed in a superposition state, there will be invalid tours among the ones that get tried. The solution is representing the possible traveling salesman tours on a number line and loading a register with a superposition of all allowed numbers. The oracle flips the phase off all paths that have a length less than, say, L. By Grover's method we can then find such a path in square root of the time it would take classically. We measure the system and test the resulting path to see if it is indeed less than length L. If it is, then we change the oracle to mark all paths with length less than L/2 and repeat the process. If not, then we double L and try again. In this way we can quickly narrow down the precise length of the shortest path. Using this procedure we can systematically remove edges in the original problem while monitoring the effect this has on the length of the shortest path. If the length changes (it can only get longer), then we know that particular edge must have been used in the shortest route. Since there are only n squared edges, we can check them all in polynomial time.

Using a Quantum Computer to Find an Approximate Solution

I did not find a satisfactory result for an algorithm for computing approximate solutions to the traveling salesman problem. The classical algorithm already produces one and a half times the minimum path, and it does so using techniques that do not lend themselves to conversion to a quantum algorithm. Shor's algorithm is proficient at periods, but this does not seem to have any use in the traveling salesman problem.

Conclusions

This problem did not lend itself to a systematic problem-solving approach. I hoped that by looking at what others in the field of quantum computing had done, I would be able to link some important principles together and have a new algorithm. Grover's algorithm is generically applicable to almost any problem, but it only provides a square root improvement. Therefore it cannot make a classically intractable problem tractable on a quantum computer. Shor's algorithm, on the other hand, does convert a presumably intractable prime factor problem into a tractable one on the quantum machine. Unfortunately, this method seems suitable for only a very specific problem. Similarly, the approximation schemes for making the traveling salesman problem tractable were more like leaps of intuition than systematic progressions. As such, the approximation techniques did not lend themselves to use on a quantum computer. In spite of the fact that my results were not as satisfying as I had hoped, I learned a great deal about the subject areas of quantum computing and computational complexity. The main conclusion is that allowing a computer to exploit quantum effects does not seem to change the approximation bounds currently know for classical machines.

Acknowledgements

I would like to thank Dr. Shawn Pethel for his help and guidance during this project. He helped me understand the concepts of Shor's and Grover's algorithms through the use of lectures. Also, he was able to tell if we were going down a dead-end path and redirect the work in a different direction. I would also like to thank Dr. Charles Bowden. He gave me a brief overview of Shor's algorithm, and he supplied some of the ideas we tried.

References

Bentley, Jon. "Faster and Faster and Faster Yet." *Unix Review.* June 1997.

Boyer, Michel, et. al. "Tight bounds on quantum searching." *PhysComp96.* 23 May 1996.

Ekert, Artur and Jozsa, Richard. "Quantum computation and Shor's factoring algorithm." *Reviews of Modern Physics.* Vol. 68, No. 3, July 1996.

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

Grover, Lov K. "Quantum Mechanics Helps in Searching for a Needle in a Haystack." *Physics Review Letters.* Vol. 79, No. 2, 14 July 1997.

Hogg, Tad and Portnov, Dmitriy. "Quantum Optimization." *Quantum Physics.* 20 June 2000.

Lloyd, Seth. "Quantum search without entanglement." *Physical Review A.* Vol. 61, 1999.

Appendix

Following are sample times for the execution of the traveling salesman program as a function of the number of cities and the c source code for the program. Also included are plots of solved traveling salesman paths generated by the program.

Number of Cities:	Time (sec):
3	0.0021
4	0.0036
5	0.0093
6	0.0341
7	0.0568
8	0.0555
9	0.0851
10	0.2284
11	1.1882
12	4.6476
13	37.5036

```
/*      Buckley Hopper
 *      Traveling Salesman Solver*/

#include <X11/Xlib.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

#define N 13
#define X 0
#define Y 1
#define MINX 10
#define XSIZE 620.0
#define MINY 10
#define YSIZE 460.0
#define WINX 640
#define WINY 480

#define swap(a,b) temp = city[a]; city[a] = city[b]; city[b] = temp;

/*      TSP Global Variables        */
int city[N];
int besttour[N];
int thistour[N];
int coord[N][2];
int temp;
int percent_done = 0;
int goflag = 1;
double tour = 0;
double d_tour;
double bestdist = 0;
double dist[N][N];
```

```
double fact[N];

/*      X11 Global Variables        */
Display *display;
Window root, window;
long fg, bg;
int screen;
int eventmask = KeyPressMask|ExposureMask|ButtonPressMask;
XEvent event;
XGCValues gcval;
GC white_gc, black_gc, red_gc, yellow_gc, blue_gc;
Colormap colormap;
XColor red, yellow, blue;
char title[10];

double factorial (double n)
{
        double f;
        for (f = 1; n > 1; n--)
                f *= n;
        return f;
}

void gen_fact (void)
{
        int i;
        for (i = 0; i < N; i++)
                fact[i] = factorial(i);
}

void gen_city (void)
{
        int n;
        for (n = 0; n < N; n++)
                city[n] = n;
}

void gen_coord (void)
{
        int i;
        srand((unsigned int) time(NULL));
        for (i = 0; i < N; i++)
        {
                coord[i][X] = MINX + (int) (XSIZE * rand() / (RAND_MAX + 1.0));
                coord[i][Y] = MINY + (int) (YSIZE * rand() / (RAND_MAX + 1.0));
        }
}
```

```
void gen_dist (void)
{
        int i, j;
        double x, y, mindist;
        for (i = 1; i < N; i++)
                for (j = 0; j < i; j++)
                {
                        x = (double) (coord[i][X] - coord[j][X]);
                        y = (double) (coord[i][Y] - coord[j][Y]);
                        dist[i][j] = sqrt(x*x+y*y);
                        dist[j][i] = dist[i][j];
                }

/*
        for (i = 1; i < N; i++)
        {
                mindist = dist[city[i-1]][city[i]];
                for (j = i+1; j < N; j++)
                {
                        if (dist[city[i-1]][city[j]] < mindist)
                        {
                                swap(i,j)
                                mindist = dist[city[i-1]][city[i]];
                        }
                }
                bestdist += mindist;
        }
        bestdist += dist[city[0]][city[N-1]];
*/

        for (i = N-2; i >= 0; i--)
        {
                mindist = dist[city[i+1]][city[i]];
                for (j = i-1; j >= 0; j--)
                {
                        if (dist[city[i+1]][city[j]] < mindist)
                        {
                                swap(i,j)
                                mindist = dist[city[i+1]][city[i]];
                        }
                }
                bestdist += mindist;
        }
        bestdist += dist[city[0]][city[N-1]];

        for (i = 0; i < N; i++)
```

```
                besttour[i] = city[i];

}

/*
void print_coord (void)
{
        int i;
        for (i = 0; i < N; i++)
        {
                printf("%d\t%d\n", coord[i][X], coord[i][Y]);
        }
}

void print_dist (void)
{
        int i,j;
        for (i = 0; i < N; i++)
        {
                for (j = 0; j < N; j++)
                {
                        printf("%lf\t", dist[i][j]);
                }
                printf("\n");
        }
}

void print_cities (void)
{
        int n;
        for (n = 0; n < N; n++)
                printf("%d ",city[n]);
        printf("\n");
}

void print_besttour (void)
{
        int n;
        for (n = 0; n < N; n++)
                printf("%d ",besttour[n]);
        printf("\n%lf\n",bestdist);
}

double compute_tour (void)
{
        int i;
        double sum;
```

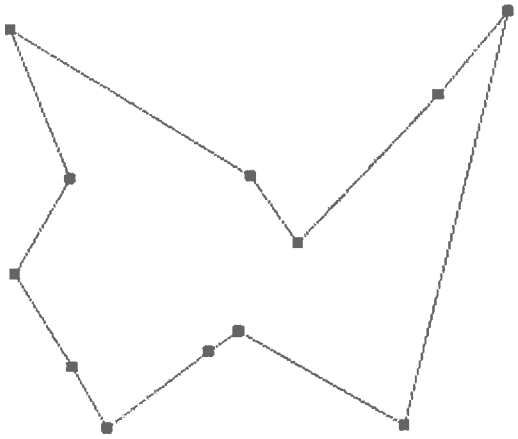
```
        sum = dist[city[0]][city[N-1]];
        for (i = 1; i < N; i++)
        {
                sum += dist[city[i]][city[i-1]];
        }
        return sum;
}
*/

void draw_tour (int *tour, GC gc)
{
        int n, i, j;

        i = tour[N-1];
        j = tour[0];
        XDrawLine(display, window, gc, coord[i][X], coord[i][Y],
                coord[j][X], coord[j][Y]);

        for (n = 1; n < N; n++)
        {
                i = j;
                j = tour[n];
                XDrawLine(display, window, gc, coord[i][X], coord[i][Y],
                        coord[j][X], coord[j][Y]);
        }
}

void draw_cities (GC gc)
{
        int n;

        for (n = 0; n < N; n++)
        {
                XFillRectangle(display, window, gc,
                        coord[n][X] - 5, coord[n][Y] - 5, 10, 10);
        }
}

void check_event (int type)
{
        switch (type)
        {
                case Expose:
                        XClearWindow(display, window);
                        draw_tour(besttour, yellow_gc);
                        draw_cities(blue_gc);
                        break;
```

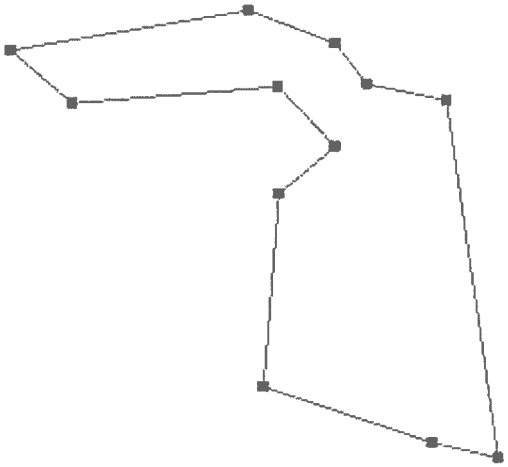
```
                case ButtonPress:
                        goflag = 1;
                        break;
                case KeyPress:
                        XDestroyWindow(display, window);
                        exit(0);
                default:
                        fprintf(stderr, "Unexpected event: %d\n",
                                event.type);
        }
}

void update_title (void)
{
        int t;
        t = (int) floor(tour);
        if (t > 0)
        {
                tour -= t;
                percent_done += t;
                sprintf(title, "%d%%", percent_done);
                XStoreName(display, window, title);
        }
}

void check_tour (double sum)
{
        int i;
        sum += dist[city[0]][city[N-1]];

        draw_tour(thistour, black_gc);
        for (i = 0; i < N; i++)
                thistour[i] = city[i];
        if (sum < bestdist)
        {
                bestdist = sum;
                draw_tour(besttour, black_gc);
                draw_tour(city, yellow_gc);
                for (i = 0; i < N; i++)
                        besttour[i] = city[i];
        }
        else
        {
                draw_tour(city, red_gc);
                draw_tour(besttour, yellow_gc);
        }
        draw_cities(blue_gc);
```

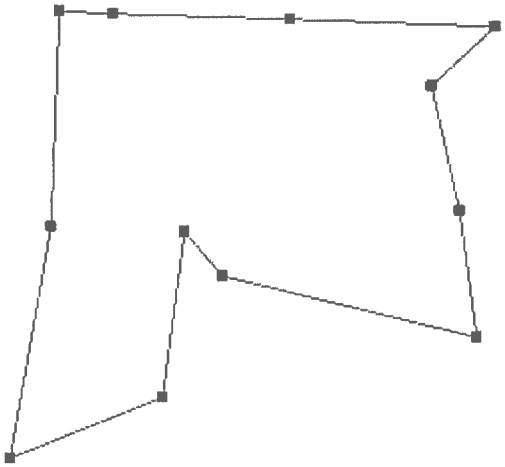
```
}

void search (int m, double sum)
{
        int i;

        if (sum > bestdist)
        {
                for (i = 0; i < m; i++)
                        if (city[i] > city[N-2])
                        {
                                tour += fact[m-1];
                        }
                update_title();

                if (XCheckWindowEvent(display, window, eventmask, &event))
                        check_event(event.type);
                return;
        }

/*
        XDrawLine(display, window, draw,
                coord[city[m-1]][X], coord[city[m-1]][Y],
                coord[city[m]][X], coord[city[m]][Y]);
*/

        if (m == 1 && (city[0] > city[N-2]))
        {
                check_tour(sum + dist[city[0]][city[1]]);
                tour += d_tour;
                update_title();

                if (XCheckWindowEvent(display, window, eventmask, &event))
                        check_event(event.type);
        }
        else
        {
                for (i = 0; i < m; i++)
                {
                        swap(i, m-1)
                        search (m-1, sum + dist[city[m-1]][city[m]]);
                        swap(i, m-1)
                }
        }
}

int main (int argc, char **argv)
{
```

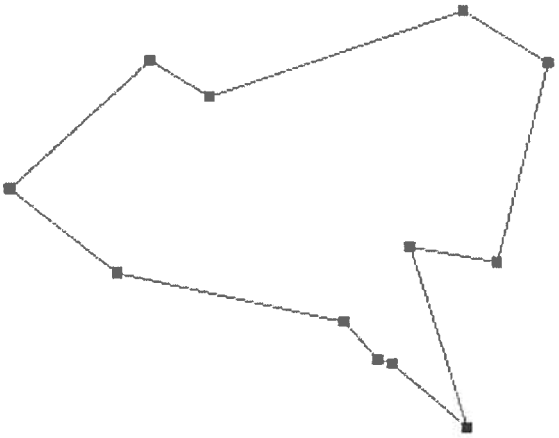
```
        int i = 0;

        /*      X initialization        */
        if (!(display = XOpenDisplay(argv[1])))
        {
                perror("XOpenDisplay");
                exit(1);
        }

        screen = DefaultScreen(display);
        root = RootWindow(display, screen);
//      fg = BlackPixel(display, screen);
        fg = WhitePixel(display, screen);
//      bg = WhitePixel(display, screen);
        bg = BlackPixel(display, screen);

        window = XCreateSimpleWindow (display, root, 0, 0,
                WINX, WINY, 2, fg, bg);

        gcval.foreground = fg;
        gcval.background = bg;
        white_gc = XCreateGC(display, window, GCForeground|GCBackground,
                &gcval);
        gcval.foreground = bg;
        black_gc = XCreateGC(display, window, GCForeground|GCBackground,
                &gcval);
        colormap = DefaultColormap(display, 0);
        XParseColor(display, colormap, "red", &red);
        XAllocColor(display, colormap, &red);
        gcval.foreground = red.pixel;
        gcval.line_style = LineOnOffDash;
        red_gc = XCreateGC(display, window,
                GCForeground|GCBackground|GCLineStyle, &gcval);
        XParseColor(display, colormap, "yellow", &yellow);
        XAllocColor(display, colormap, &yellow);
        gcval.foreground = yellow.pixel;
        yellow_gc = XCreateGC(display, window, GCForeground|GCBackground,
                &gcval);
        XParseColor(display, colormap, "blue", &blue);
        XAllocColor(display, colormap, &blue);
        gcval.foreground = blue.pixel;
        blue_gc = XCreateGC(display, window, GCForeground|GCBackground,
                &gcval);

        XSelectInput(display, window, eventmask);
        XMapWindow(display, window);
        /* end X initialization         */
```

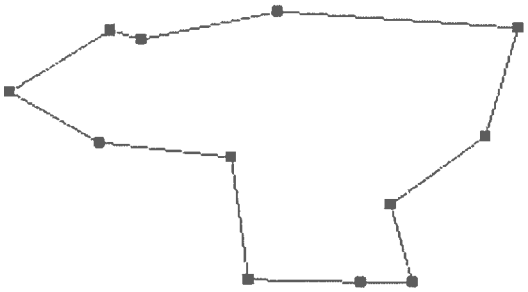
```
        gen_fact();
        d_tour = 200.0/fact[N-1];
        for (i = 0; i < N; i++)
                fact[i] *= d_tour;

        for (;goflag;)
        {
                tour = 0;
                bestdist = 0;
                percent_done = 0;
                XStoreName(display, window, "0%");

                gen_city();
                gen_coord();
                gen_dist();
                XClearWindow(display, window);
                draw_tour(besttour, yellow_gc);
                draw_cities(blue_gc);

                search (N - 1, 0);

                XClearWindow(display, window);
                draw_tour(besttour, yellow_gc);
                draw_cities(blue_gc);
                XStoreName(display, window, "Done");

                /*      X event handling     */
                goflag = 0;
                for (;!goflag;)
                {
                        XWindowEvent(display, window, eventmask, &event);
                        check_event(event.type);
                }
                /*      end X event handling        */
        }
}
```

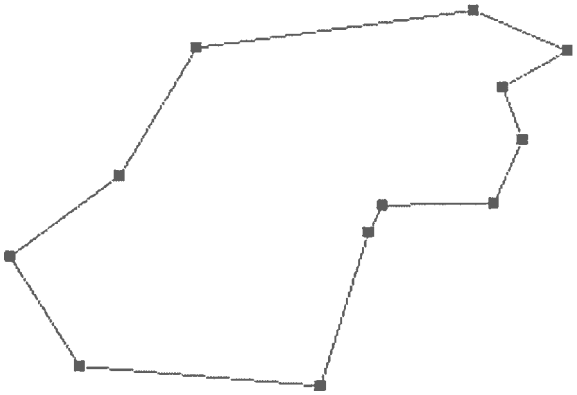












Honors Senior Project
Approval

Form 3 – Submit with completed thesis. All signatures must be obtained.

Name of candidate: Buckley Hopper

Department: Physics

Degree: Bachelors

Full title of project: Searching for a Quantum Algorithm to Solve the Traveling Salesman Problem

---

Approved by:

[signature]	Nov. 29, 01
Project Advisor	Date
[signature]	13 Dec 01
Department Chair	Date
[signature]	12/13/01
Honors Program Director for Honors Council	Date