The Traveling Salesman Project

New Mexico

Supercomputing Challenge

Final Report

March 30th, 2007

Team 50

Los Alamos High School

Team Members

Ben Batha

Daniel Cox

Ryan Marcus

William Phillips

Teachers

Mrs. Diane Medford

Project Mentor

Mr. Neale Pickett

Table of Contents

Executive Summary 3

A. The Problem *3*

B. The Solution *3*

C. The Engine *3*

Problem statement 4

The Solution 5

A. Shortest Route *5*

B. Schedule *5*

C. Salesman *5*

Results 7

Conclusion 8

A. What could be extended *8*

1. Large Map Files 8

2. Dynamic Map Files 8

3. Implementation 8

B. Limits *8*

C. What could have gone better *9*

Appendix A: Code 11

Appendix B: Maps 16

Appendix C: Screenshot 17

Executive Summary

A. The Problem

A primary aspect of our world is location. Humanity has created maps, putting complex networks onto a two- dimensional sheet of paper, in an attempt to make traveling between these locations easier. The map was amazing. A human being was able to look at the map and quickly discover a route to their destination. However, was it always the fastest?

Surely, in a small system, a human can deduce the fastest route, as it is normally a straight line with very few turns. However, a human being may fail to discover the fastest route when the system grows more complex. This presents a clear opportunity for a computer – something consistent, precise, and always correct.

B. The Solution

We've created an original map file system that is simple and to-the-point. It allows for a unlimited size, as well as unlimited distance between points. In short, one vertex can touch a second, through an edge. This edge has a certain weight. If we were talking about locations on a map, the points (vertexes) would be a location, and the weighted values will be the distance between points.

On top of this infinitely expandable map file system, we've created an incredibly flexible and fast engine.

C. The Engine

In creating our engine, we considered elegance and speed to be critical factors. Our program is written in LISP, as LISP makes it easier to concentrate on the algorithm instead of on the small semantics that are embedded in other languages. LISP itself was developed purely for processing lists, as the name suggests. The program will need to take in any set of connected nodes as data, and quickly and reliably find the best path between the two.

The engine has three primary functions:

1. **Shortest distance**: The engine can compute the shortest distance between point A and point B.
2. **Scheduling**: You can find the fastest route passing through any number of specified points.
3. S**alesman**: Find the fastest route to every single point and then back to the starting point.

Problem statement

Location is a major aspect of our world. The mechanisms we have for moving from one point to a another have evolved, as well as the complexity of the world. With these massive changes in the size of cities and the ability to move, man developed the map, a simple sheet of paper designed to assist a traveler in moving from one place to another.

In a small system, a human could deduce the fastest route, as it is normally a straight line with very few turns. However, a human being may begin to fail to discover the fastest route if the system grows more complex. Mathematicians started looking at possible solutions to this problem in ancient times. Theories and algorithms have evolved over the years. The invention of the computer has drastically changed the playing field for these mathematicians, who now have a device that can do these tasks with ease.

Being able to calculate the fastest route through a map, with a computer could save time, money, fuel, power, and all other kinds of resources. Clearly, the traveling salesman problem is a worthy task for our team, our world, and certainly a supercomputer.

The Solution

Our map file system is expandable to an unlimited size, allowing for both an unlimited number of points, as well as unlimited distance between points. On top of this map file, we've created a fast and elegant engine. Depending on what you want to do, one of three actions will occur:

A. Shortest Route

When you tell the engine to calculate the shortest route between point A and point B, the engine starts cranking away, spiraling out of point A in an attempt to reach point B.

As soon as one of these threads reaches point B, the thread adds the route it took to get there to a list, and then sets a variable containing the distance it took to get to point B. Because the first route found may not be the fastest, the engine continues.

Because a variable has been set with the length of the first route, all threads will terminate themselves if they exceed that number. If you are trying to get from Santa Fe to Los Alamos, and you've already found the route that only has to go 50 miles, then there is no sense in continuing a route that has already exceeded 50 miles.

If a shorter route is found, the engine will store the new route, and change the variable.

Once all threads have finished, because they have gone as deep as possible, the engine goes through all the routes that where found and locates the shortest. This may not be necessary, because of the variable used to store the longest possible route, but it may be if a supercomputer can run enough threads at the same time.

B. Schedule

This function of the engine is rather simple. It will just find the shortest route from the first item in your schedule to the second, and from the second to the third, and so on, until it finishes your route. This could be useful if you had to, for example, deliver and pick up some cargo at point A, then proceed to point B, then to point C.

C. Salesman

The third and perhaps the most useful function of the engine is the ability to “salesman.” The engine will take in a point, and then calculate the fastest route to go to all other points, and then return to the start point.

The method used is similar to the method used for getting the shortest route. The engine starts a stream of threads, discovering every possible route that can be taken from the start point. When the thread gets deep enough that it will no longer produce a unique route, it stops and saves its route to a variable.

Once this lengthy task finishes, the engine will remove all the routes that do not pass over every point on the map. We call this “route skimming,” as it is comparable to skimming the top of a glass of liquid to filter the bad, floating particles out.

Once we have every route that goes to every point, we add the original starting point to the end of each route, so that we actually end where we started. After that, we simply go through each route, calculate its distance, and return the shortest one. And thus, the traveling salesman problem is solved.

Results

Our model does not really have any large-scale results, as we lack the technology – namely, a supercomputer – to implement it. We hope to find of some fast machines and do some work for next year, and perhaps even look at applying our project to a network.

While the map file format is infinitely expandable, we do not have the time to enter in anything of an incredibly large scale. Having a computer generated map file is clearly one of the next steps we will have to achieve. The time we used to optimize the code would have been wasted trying to make a large map file.

Another step that would add to the limitless potential of this model would be a map that changes over time. For example, the weight of Interstate 40 would increase during rush hour and decrease during normal hours.

Conclusion

A. What could be extended

1. Large Map Files

Perhaps the entire point of the traveling salesman problem is to preform the problem on a map of an incredibly large size. As much as we would have liked to, we did not have enough time to create, much less test, a map of more then fifteen points. Most practical situations would require many more points then we tested, however, we have faith in our algorithm. We were unable to test it on a map of incredibly large scale, so the scalability of the program is untested.

Making an application scale is perhaps one of the most difficult things for a programmer to do. What is faster on a ten point map might not be faster for a hundred point map. The correct method for all sizes of maps is probably unachievable, so the program should modify its algorithm based on the number of points in a map.

2. Dynamic Map Files

Another essential aspect of a map is change: traffic changes, storms can occur and volcanoes erupt. For example, a mars rover might have to go from Colored Chaos to the Sabis Vallies, but a storm will reach the path between the Mars 2 Lander and the Mars 6 Lander in an hour. Therefore, the model must take into account that the weight of a route might change over time.

3. Implementation

Of course, our entire model has yet to be implemented completely into a single practical situation. Next year, our team will try to tackle the problem of passing period... Getting from one point to another within the ten minutes given. We will have to use a dynamic map file, as well as a large map file. Acquiring the resources to run the application will force us to dive into LISP and discover its methods for multi-processing.

B. Limits

Due to the nature of the map file, the model has virtually no limits. You are only limited by the time you have to enter data, and the power of your computer.

C. What could have gone better

One of the largest issues for our team was collaboration. When we began the year, we had a project idea that we all knew would inevitably fail. It took us about three weeks to figure that out, planning and even writing a little code in the process. After that fiasco, one of our team members presented the idea of modeling the world economy, a slightly more possible, but still way to variable process. However, we went with it, developing a plan and creating a process (thanks to Neale). We began to implement this plan, but it eventually led to the same person deviating from the proposed plan and attempting to write the entire project himself.

This, of course, led to failure. Two other members of the team, throughout the entire project, were busy learning LISP, as well as writing a preliminary version of the Traveling Salesman Problem. After the first two projects failed, we extended the preliminary model until it became a full-fledged project, which turned out to be more substantial then any other piece of code we had.

We definitely learned a lot about planning an application, and how something like this is not a “dive right in” kind of program. As we look back on our successes and our failures, we can definitely “chalk it up as experience.”

References

EL Lawler. *The traveling salesman problem*. Wiley 1985.

“GNU CLISP.” October 13, 2006. <http://clisp.cons.org/>

“GNU Emacs.” <http://www.gnu.org/software/emacs/>.

JB Kruskal Jr. “On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.” Proceedings of the American Mathematical Society, 1956 - JSTOR.

Pablo Mascato. “TSBIB.” <www.ing.unlp.edu.ar/cetad/mos/TSPBIB_home.html>.

“SLIME: The Superior LISP Interaction Mode for Emacs.” <http://common-lisp.net/project/slime/>.

S Lin. “An Effective Heuristic Algorithm for the Traveling-Salesman Problem.” BW Kernighan. Operations Research, 1973 - JSTOR.

Wolfram Mathworld. “Traveling Salesman Problem.” <http://mathworld.wolfram.com/TravelingSalesmanProblem.html>.

Appendix A: Code

```
(setq map (with-open-file (infile "/Users/ryan/Desktop/SCC/Code/mars.txt")
(read infile)))

;; This function will return true if the MAP has location LOC. It will return
false otherwise.
(defun HasLoc(map loc)
 (cond ( (null map) nil)
       ( (equal (first (first map)) loc) T)
       ( T (HasLoc (rest map) loc))))

;; This function will return all of the points in the map
(defun AllPoints(map)
 (cond ((null map) 'nil)
       (t (cons (first (first map)) (AllPoints (rest map))))))

;; This function will return the neighbors of LOC inside of MAP with their
distances.
(defun GetNeighbors_WithDistance(map loc)
 (cond ( (null map) nil)
       ( (equal (first (first map)) loc) (rest (first map)))
       ( T (GetNeighbors_WithDistance (rest map) loc))))

;; This fuction will return E, with every other element removed.
(defun RemoveSecond(e)
 (cond ( (null e) nil)
       (     t    (cons (first (first e)) (RemoveSecond (rest e))))))

;; This is a wrapper to return the neighbors of LOC inside of MAP without the
distances.
(defun GetNeighbors(map loc) (RemoveSecond (GetNeighbors_WithDistance map
loc)))

;; This method was ripped out of GetTouchingDistance. defun in a defun is
messy.
(defun UnderGetTouchingDistance(a b)
 (cond ( (null a) 'nil)
       ( (equal (first(first a)) b) (second (first a)))
       ( t                       (UnderGetTouchingDistance (rest a) b))))

;; This function will get the distance between point A and point B inside of
MAP. It will only work if point A and point B are touching.
(defun GetTouchingDistance(map a b)
 (let* ((theDistances (GetNeighbors_WithDistance map a))
        (findingPoint b))

       (UnderGetTouchingDistance theDistances findingPoint)))

;; This is a wrapper to return how long ROUTE is inside of MAP.
(defun GetDistanceForRoute(map route) (GetDistanceForRouteF map route 0))

;; This is the recursive function that is called by GetDistanceForRoute
(defun GetDistanceForRouteF(map route distance)
 (cond ((null route) nil)
       ((< (length route) 2) distance)
       (t          (GetDistanceForRouteF map (rest route) (+ distance (Get-
TouchingDistance map (first route) (second route)))))))
```

```
;; This function is a wrapper function for FigureOutShortestF.
(defun FigureOutShortest(map routes)
 (setq theShortestRoute nil)
 (setq theShortest 10000000000000000000000000000000000)
 (FigureOutShortestF map routes))

;; This takes a list of routes and returns the shortest one
(defun FigureOutShortestF(map routes)
 (cond ((null routes) theShortestRoute)
       ((< (GetDistanceForRoute map (first routes)) theShortest) (setq theS-
hortestRoute (first routes)) (setq theShortest (GetDistanceForRoute map
(first routes))) (FigureOutShortestF map (rest routes)))
       (t (FigureOutShortestF map (rest routes)))))


;; This is a wrapper function for getting the shrotest route from point A to
point B in MAP
(defun GetShortestRoute(map a b)
 (setq possibleRoutes nil)
 (setq thecurrentLow 1000000000000000000000000000000000)
 (GetDistance map a b () 0)
 (FigureOutShortest map possibleRoutes))


;; This is a function to store all possible routes between two pointes. It
just appends a route to a list.
(defun printRoute (theRoute)
 (setq possibleRoutes (cons (reverse theRoute) possibleRoutes)))


;; This returns the maximum number of recursions we would need to go through.
(defun limit (map)
 (- (length map) 1))


;; This sets the current low to the passed route
(defun setLow (map theRoute)
 (setq thecurrentLow (GetDistanceForRoute map theRoute)))


;; This sees if the passed value is less then the current low. If it is, re-
turn true. else, nil
(defun checkLow (testNumber)
 (cond ( (> testNumber thecurrentLow) t)
       ( t nil)))


;; This gets the distance for a route and then returns the checkLow for that
route.
(defun checkRoute (map theRoute)
 (cond ( (null theRoute) t)
       ( (< (length theRoute) 2) t)
       ( t (checkLow (GetDistanceForRoute map theRoute)))))

;; Ripped this out of GetDistance. defun in a defun was a little messy.
(defun StartOnTheNeighbors (map a b route count theNeighbors)
 (cond ( (null theNeighbors) nil)
       ( t                  (GetDistance map (first theNeighbors) b (cons a
route) (+ count 1)) (StartOnTheNeighbors map a b route count (rest theNeigh-
bors)))))

;; The meet. This is the function that puts all possible routes from point A
to point B in map into printRoute.
(defun GetDistance(map a b route count)

 (setq theNeighbors (GetNeighbors map a))

 (cond ( (= count (limit map)) nil)
       ( (equal a b) (printRoute (cons b route)))
       ( t          (StartOnTheNeighbors map a b route count theNeigh-
bors))))

;; This is the wrapper method that removes the random nil from the schedule-
route
(defun ScheduleRoute(map points) (allButLast (ScheduleRouteF map points)))

;; This method will schedule from stop one to stop two, etc. In essense, go
from point A to point B and append the path... Oh boy.
(defun ScheduleRouteF(map points)
 (cond ((null points) nil)
       (t (cons (GetShortestRoute map (first points) (second points)) (Sched-
uleRouteF map (rest points))))))

;; This method returns all but the last element of the list
(defun allButLast(l)
 (cond ((= (length l) 1) nil)
       (t (cons (first l) (allButLast (rest l))))))

;; This method takes a list that looks like ((1) (2) (3)) and makes it (1 2
3)
(defun listFix(l)
 (cond ((null l) nil)
       (t (cons (first (first l)) (listfix(rest l))))))

;; This method will return the distance needed to preform a schedule
(defun ScheduleTime(map sched)
 (cond ((null sched) '0)
       (t   (+ (GetDistanceForRoute map (first sched)) (ScheduleTime map
(rest sched))))))

;; This method takes a list of neighbors: (("A" 2) ("B" 3)) and returns the
lowest (A)
(defun LowestNeighborF(neighbors)
 (cond ((= (length neighbors) 1) (first(first neighbors)))
       ((< (second (first neighbors)) (second (second neighbors))) (Lowest-
NeighborF (remove (second neighbors) neighbors)))
       (t (LowestNeighborF (rest neighbors)))))

;; This method will take in a list of integers and return the lowest one
(defun GetLowest(l)
 (cond ((= (length L) 1)(first l))
       ((< (first l) (second l)) (GetLowest (remove (second l) l)))
       (t        (GetLowest (rest l)))))

;; This method will search a list for e
```

```
(defun searchList(e L)
 (cond ((null L) nil)
       ((equal (first L) e) T)
       (t (searchList e (rest L)))))

;; This is the wrapper function for GetAllRoutesF
(defun GetAllRoutes(map a)
 (setq theSalesmenRoute '())
 (GetAllRoutesF map a '() 0)
 theSalesmenRoute)

;; This method was ripped out of GetAllRoutesF. Defun in a defun is messy.
(defun UnderGetAllRoutesF (map a route count theNeighbors)
 (cond ( (null theNeighbors) nil)
       ( t                   (GetAllRoutesF map (first theNeighbors) (cons a
route) (+ count 1)) (UnderGetAllRoutesF map a route count (rest theNeigh-
bors)))))

;; This method will give out a list of every possible combination of routes
from A
(defun GetAllRoutesF(map a route count)

 (setq theNeighbors (GetNeighbors map a))

 (cond ( (= count (+ 1 (limit map))) (addRoute route))
       ( t           (UnderGetAllRoutesF map a route count theNeighbors))))

;; This method will add a route to a list
(defun addRoute(route)
 (setq theSalesmenRoute (cons (reverse route) theSalesmenRoute)))

;; This method will check a route to see if it has all points in the map
(defun checkRouteForPoints(map route) (checkRouteForPointsF route (AllPoints
map)))

;; This method will check a route to see if it contains every single point in
thePoints
(defun checkRouteForPointsF(route thePoints)
 (cond ((null thePoints) T)
       ((searchList (first thePoints) route) (checkRouteForPointsF route
(rest thePoints)))
       (t 'nil)))

;; This method is a wrapper for removeBadRoutesF
(defun removeBadRoutes(map routes)
 (setq thenewroutes '())
 (removeBadRoutesF map routes)
 thenewroutes)

;; This method will remove all routes that don't pass checkRoutesForPoints
(defun removeBadRoutesF(map routes)
 (cond ((null routes) thenewroutes)
       ((checkRouteForPoints map (first routes)) (addANewRouteToDone (first
routes)) (removeBadRoutesF map (rest Routes)))
       (t (removeBadRoutesF map (rest routes)))))

;; This method keeps track of all the good routes
(defun addANewRouteToDone(route)
 (setq thenewroutes (cons route thenewroutes)))

;; This method will add point to the end of route, putting the correct jumps
in between.
(defun addRouteToPointToEnd(map route point)
 (cond ((equal (first (last route)) point) route)
       (t (append (allbutlast route) (getshortestroute map (first (last
route)) point)))))

;; This method will take a list of routes and pass them into addRouteToPoint-
ToEnd
(defun addPointToRoutes(map routes point)
 (cond ((null routes) nil)
       (t  (append (list (addRouteToPointToEnd map (first routes) point))
(addPointToRoutes map (rest routes) point)))))

;; This method is IT! It, as in a wrapper.
(defun Salesmen(map point)
 (setq theRoutesToWorkWith (removebadroutes map (getallroutes map point)))
 (setq newRoutes (addPointToRoutes map theRoutesToWorkWith point))
 (FigureOutShortest map newRoutes))
```

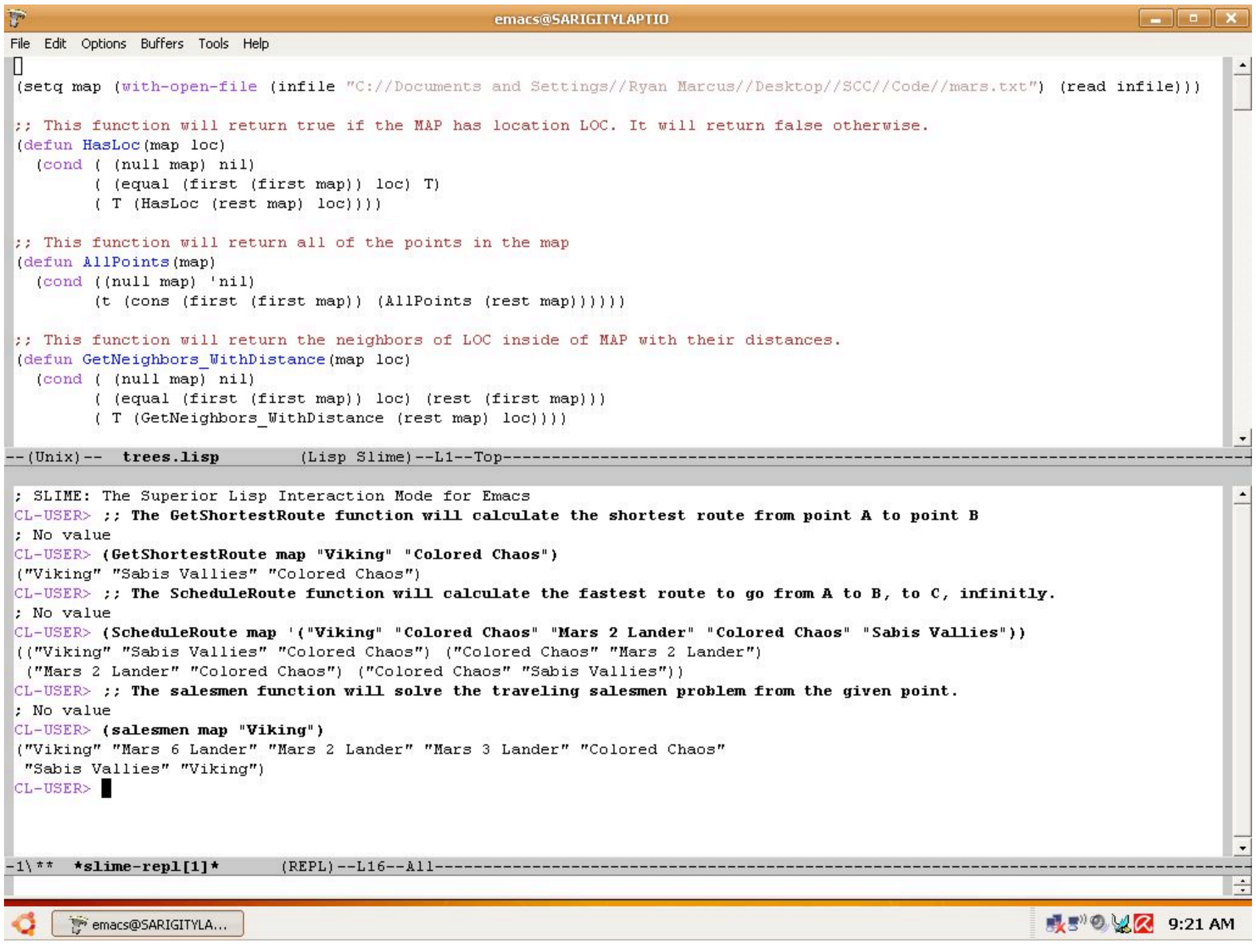
Appendix B: Maps

```
;;Nanotechnology
( ("Cell 1"  ("Cell 2" 20) ("Cell 3" 20) ("Cell 4" 20) ("Cell 5" 20) ("Cell
6" 1))
  ("Cell 2"  ("Cell 1" 20) ("Cell 10" 1) ("Cell 7" 1))
  ("Cell 3"  ("Cell 1" 20) ("Cell 5" 3) ("Cell 9" 2))
  ("Cell 4"  ("Cell 1" 20) ("Cell 10" 8) ("Cell 8" 10))
  ("Cell 5"  ("Cell 1" 20) ("Cell 3" 3) ("Cell 9" 1))
  ("Cell 6"  ("Cell 1" 20) ("Cell 7" 2) ("Cell 10" 2))
  ("Cell 7"  ("Cell 2" 1)  ("Cell 6" 2))
  ("Cell 8"  ("Cell 4" 10) ("Cell 10" 3))
  ("Cell 9"  ("Cell 3" 2)  ("Cell 5" 1) ("Cell 10" 2))
  ("Cell 10" ("Cell 2" 1)  ("Cell 4" 8) ("Cell 6" 2) ("Cell 8" 3) ("Cell 9"
2)))

;;Network
( ("Main Server" ("File Server" 1) ("Web Server" 1))
  ("File Server" ("Main Server" 1) ("Router" 20) ("Users" 10))
  ("Web Server"  ("Main Server" 1) ("Router" 10) ("Users" 5))
  ("Router"      ("File Server" 20) ("Web Server" 10) ("Users" 2))
  ("Users"       ("File Server" 10) ("Web Server" 5) ("Router" 2)) )

;;Mars
( ("Mars 2 Lander" ("Mars 6 Lander" 10) ("Mars 3 Lander" 32) ("Colored Chaos"
33))
  ("Mars 3 Lander" ("Mars 2 Lander" 32) ("Colored Chaos" 2) ("Sabies Vallies"
3))
  ("Mars 6 Lander" ("Viking" 7) ("Mars 2 Lander" 10))
  ("Viking"        ("Mars 6 Lander" 7) ("Sabis Vallies" 8))
  ("Colored Chaos" ("Mars 3 Lander" 2) ("Mars 2 Lander" 33) ("Sabis Vallies"
11))
  ("Sabis Vallies" ("Viking" 8) ("Colored Chaos" 11) ("Mars 3 Lander" 12)))
```

Appendix C: Screenshot


emacs@SARIGITYLAPTIO
File Edit Options Buffers Tools Help
--(Unix)-- trees.lisp (Lisp Slime)--L1--Top
-1** *slime-repl[1]* (REPL)--L16--All
emacs@SARIGITYLA...
9:21 AM

```
(setq map (with-open-file (infile "C://Documents and Settings//Ryan Marcus//Desktop//SCC//Code//mars.txt") (read infile)))

;; This function will return true if the MAP has location LOC. It will return false otherwise.
(defun HasLoc(map loc)
  (cond ( (null map) nil)
        ( (equal (first (first map)) loc) T)
        ( T (HasLoc (rest map) loc))))

;; This function will return all of the points in the map
(defun AllPoints(map)
  (cond ((null map) 'nil)
        (t (cons (first (first map)) (AllPoints (rest map))))))

;; This function will return the neighbors of LOC inside of MAP with their distances.
(defun GetNeighbors_WithDistance(map loc)
  (cond ( (null map) nil)
        ( (equal (first (first map)) loc) (rest (first map)))
        ( T (GetNeighbors_WithDistance (rest map) loc))))
```

```
; SLIME: The Superior Lisp Interaction Mode for Emacs
CL-USER> ;; The GetShortestRoute function will calculate the shortest route from point A to point B
; No value
CL-USER> (GetShortestRoute map "Viking" "Colored Chaos")
("Viking" "Sabis Vallies" "Colored Chaos")
CL-USER> ;; The ScheduleRoute function will calculate the fastest route to go from A to B, to C, infinitly.
; No value
CL-USER> (ScheduleRoute map '("Viking" "Colored Chaos" "Mars 2 Lander" "Colored Chaos" "Sabis Vallies"))
(("Viking" "Sabis Vallies" "Colored Chaos") ("Colored Chaos" "Mars 2 Lander")
 ("Mars 2 Lander" "Colored Chaos") ("Colored Chaos" "Sabis Vallies"))
CL-USER> ;; The salesmen function will solve the traveling salesmen problem from the given point.
; No value
CL-USER> (salesmen map "Viking")
("Viking" "Mars 6 Lander" "Mars 2 Lander" "Mars 3 Lander" "Colored Chaos"
 "Sabis Vallies" "Viking")
CL-USER> 
```

This shows the program in its native runtime environment, SLIME. This uses emacs to run the CLISP interpreter.