6.871, Spring 2005, Prof. Davis

Transit Advisor

Table of Contents

Table of Figures

1. Introduction

The Transit Advisor is a knowledge based system which constructs itineraries for travel around Boston and Cambridge. The program contains knowledge about many modes of transportation, organized with the blackboard and agent problem solving paradigm. Instead of embedding the transportation knowledge into search procedures (i.e. telling the program what to do), this architecture allows knowledge to be expressed in a very direct way (i.e. telling the program what to know).

2. The Task

The Transit Advisor helps its user get from point A to point B within the Cambridge/Boston area. Its input is an origin, destination, and a set of ancillary inputs that tell the program what to know about available modes of transportation. The output of the program is an itinerary, with a low overall monetary and time cost (relative to all options), that will get the user from the origin to the destination.

The program knows about and will consider walking, biking, taking a taxi, taking the T, or using a ZipCar. Each of these options has differentiating factors that make them appropriate in different situations. For example, the user cannot take the T or a ZipCar unless they move to any number of specific locations. Some modes of transportation have constant costs, like $1.25 to ride the T or a 3 minute wait for a taxicab. Each mode has a different speed, and can require you to go in a non-direct path (i.e. the T).

Differences in cost, location, and speed make different modes of transportation appropriate in particular circumstances. The Transit Advisor evaluates all possible itineraries, and presents the best one to the user.

This goal surfaces the question – what makes an itinerary good? First, the itinerary should not include any modes of transportation which are not feasible for the user. For example, if the user is not 21+ years old then they cannot use a ZipCar. If it is raining outside then a bike is not a viable option. After the possible modes of transportation are selected, the system minimizes the total cost of the itinerary, expressed in dollars. Travel time is expressed in dollars using a value-of-time user input expressed as dollars per hour.

2.1. A Concrete Problem Example

Figure 1. Screenshot of a solution. Details are provided in the text

In this section we will go through a single concrete problem, describe the inputs to the program, and discuss the output of the Transit Advisor. A screenshot of the solution to the problem is show in Figure 1. The user wants to get from East Campus to a location near Kenmore Square. The origin and destination of a trip is always shown as open rectangles. The location of the user's bike is always shown as a filled rectangle, unless it is raining outside, in which case the program does not ask where the bike is since the bike's location is not relevant. There are not any ZipCars pictured in this screenshot because the user is not 21+ years old, so ZipCars are not an option.

The final input to the program is the value of the user's time, in dollars per hour. In the case of the problem shown in Figure 1, the user's time is worth $15 per hour.

With that information in hand, the Transit Advisor computes that the best itinerary (as defined in section 2) is to walk to the bike and ride it to Kenmore.

A few observations are in order. First, paths are straight lines. Although the itineraries are on scale with the map, Transit Advisor does not know about roads or rivers that might force someone to travel in an indirect way. The only indirect transportation is via the T. Second, the user can see the cost of each segment of the resulting itinerary. Since walking is much slower than biking, and neither of these have direct monetary costs, walking to the bike in this example costs more than twice the biking segment of the trip.

2.2. Using the Transit Advisor program

The user interface of the Transit Advisor is not very polished. This section will describe the basic user interface.

2.2.1. User Interface Symbols

Most of the symbols are used in the solution shown in Figure 2. Namely:

- Black dots represent the location of a Zip Car
- Red dots and red lines between them represent the Red Line of the T
- Green dots and green lines between them represent the Green Line of the T
- Black, unfilled rectangles represent the origin and destination of the itinerary
- A black, filled rectangle represents the starting location of the user's bike
- Black lines represent segments of an itinerary, which are labeled with the mode and cost of that segment

Figure 2. Screenshot of a solution involving three modes of transportation.

2.2.2. Providing Scenario Input

The Transit Advisor asks the user some questions at the beginning of each session. These questions, outlined in Figure 3, collect information necessary for each mode of transportation to determine its eligibility (e.g. can I ride a bike?) and cost of ownership.

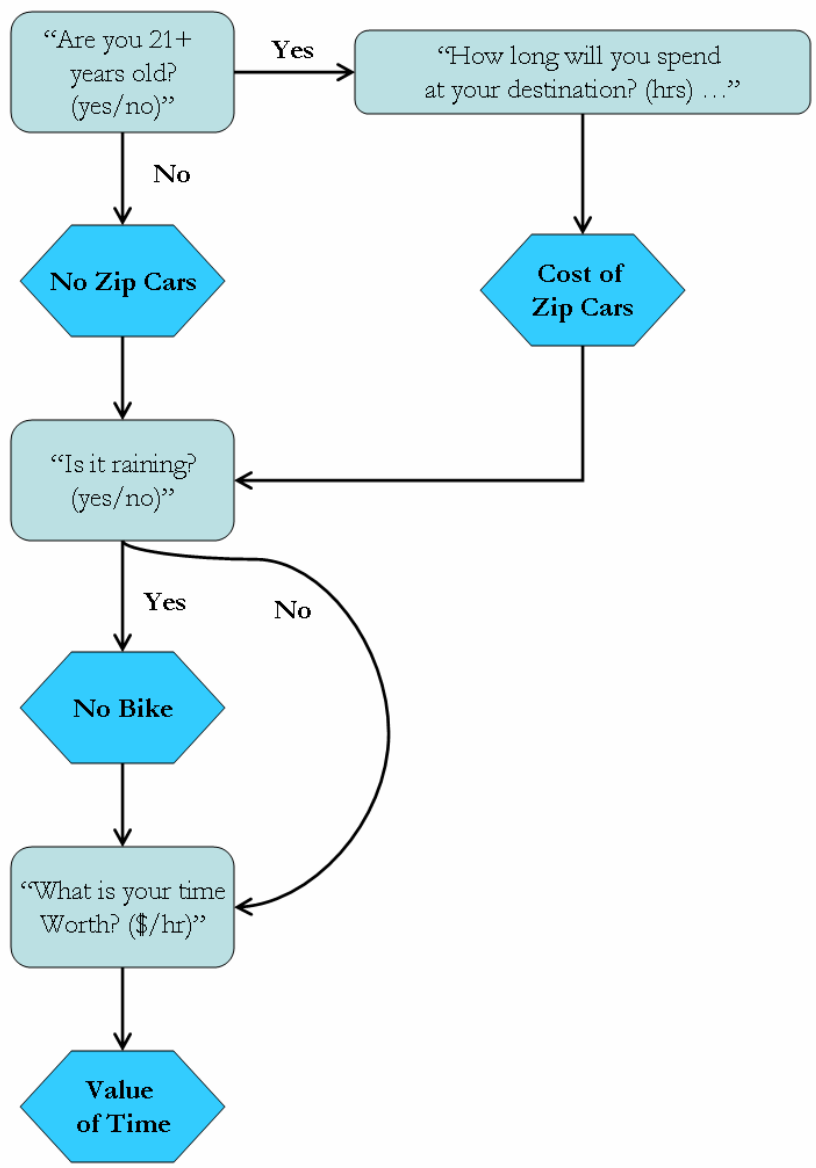

Figure 3. Flow diagram of questions asked to user and corresponding inferences made at beginning of execution.

2.3. User Interface Shortcuts

There are several time-saving mechanisms built into the program. Although obscure, they can help the avid user avoid the hassle of repeated input.

- If you want to recomputed an itinerary without having to restart the program and enter in the scenario input again, then just click the mouse once after an itinerary has been computed and displayed.

- If you type "def" (as in "default") as the answer to the first scenario input question, the program will load with the following parameters:

- Value of time = $15 per hour
- 21+ years old = yes
- Time spent at destination = 3 hours
- Raining = no

- If you type "defXX" instead of "def", where XX is any two digit number, then the same parameters are loaded, except the value of time becomes XX dollars per hour. I.e. "def15" is equivalent to "def"

3. Problem Solving Paradigm

The Transit Advisor uses the GPS (General Problem Solver) paradigm proposed by Newell and Simon (1972). The program has an initial state (the user is at the *origin*), and a goal state (the user is at the *destination*). There are operators (modes of transportation) which can change the state of the system; these operators can be combined in sequences which can, hopefully, achieve the goal state.

I original conceived of the paradigm in a different way using a slightly different vocabulary. Here, I imagine a blackboard which represents the state of the world (as a 2-D map) and a goal (get from A to B). Agents in the system, one for each mode of transportation, devise itineraries which leverage their mode of transportation. The system collects itineraries from all agents and selects the best one.

More specifically, this works in the following way. When the user selects their origin and destination, the system passes these points to each agent, asking for its best itinerary. The system selects the best itinerary of those produced by each agent, and then returns that as the answer. Each agent might need the assistance of another agent, though. For example, the Zip Car agent needs to find a way for the user to travel from the origin to the closest ZipCar. To do this, the Zip Car agent recursively calls the system, asking for the best sub-itinerary in which the origin is the user's original origin, but the new destination is the closest Zip Car. To prevent infinite looping, the Zip Car agent will exclude itself from the sub-itinerary search. Suppose that the only agents in the system are for walking, and taking a Zip Car. Then the sub-itinerary will involve walking from the origin to the Zip Car. The Zip Car agent then concatenates its Zip Car travel to the returned sub-itinerary, creating a complete itinerary. This process is illustrated in
Figure 4.

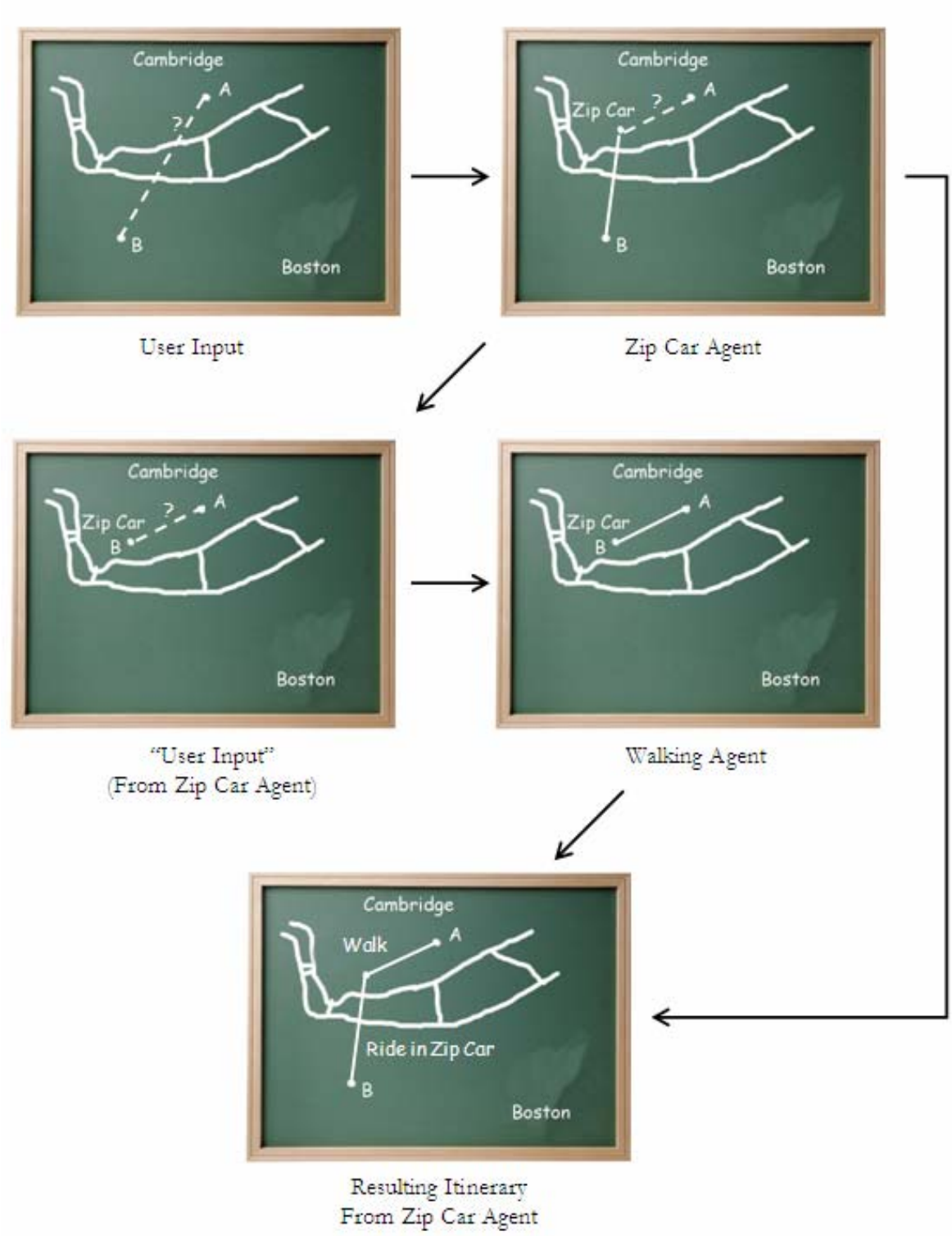

Figure 4. Annotated illustration of greenboard/agent interaction.

4. The Agents

The following sub-sections each describe one agent of the system, each representing a single mode of transportation. Each mode's cost (time and monetary), recursive calls, knowledge, and general operations will be discussed. The "Knowledge Summary" boxes summarize the facts known to the agent, and the main body text describes the common scenarios for that mode's use.

4.1. Walking

Knowledge Summary

Speed: 5 MPH
Recursive calls: None
Monetary costs: None
Requisite conditions for use: None

Walking is the simplest agent in the system. It also serves as a fallback; since the mode does not have any requisites for use, it is always a plausible means. Unfortunately, its slow speed makes it sparsely a good choice for users with a high value of time.

4.2. Biking

Knowledge Summary

Speed: 20 MPH
Recursive calls: (1) To get itinerary to bike
Monetary costs: None
Requisite conditions for use: (1) It's not raining, (2) Must begin at location of bike

Biking is walking's cousin. It is faster, but can only be used when it is not raining outside, and the user has to get to the location of the bike. Thus, biking always requires some other means of transportation. Since both modes are free (aside from time), the walking+biking combo is a popular among users with a low value of time.

4.3. Taxicab

Knowledge Summary
Speed: 45 MPH, after 3 minutes to get a taxicab Recursive calls: None Monetary costs: $1.75 + $0.3*(Miles Traveled /8) Requisite conditions for use: None

The taxicab is the rich man's walking. There is a constant startup cost (from both a 3 minute wait time and Boston's $1.75 starting meter price), which makes it most viable for long-distance transportation. As the cost of time goes to infinity, taxicabs are preferred more and more over walking for distances that take longer than three minutes to walk.

4.4. Zip Car

Knowledge Summary
Speed: 40 MPH Recursive calls: (1) To get itinerary to closest Zip Car Monetary costs: $8.5 * (hours at destination + hours for round trip) Requisite conditions for use: (1) User is 21+ years old, (2) Must begin at location of closest Zip Car

When the value of the user's time is high enough, both Zip Cars and taxicabs become viable. Here, the Zip Car and the taxicab become arch enemies. Since the cost of a Zip Car is directly proportional to the time it is gone, any itinerary using a Zip Car is charged for the time that the user spends at the destination. If that time is short, Zip Cars are usually preferred over taxicabs; if that time is long, Zip Cars are usually not used.

4.5. The T

Knowledge Summary
Speed: 50 MPH Recursive calls: (1) To get itinerary to T, (2) To get itinerary from T Monetary costs: $1.50 Requisite conditions for use: (1) Must begin at location of closest T stop to origin

The T is cheap transportation that moves pretty quickly. Its biggest downfall is that the user cannot travel directly from their origin T stop to their destination T stop. For example, a trip from Kendall on the red line to Kenmore on the green line travels about three times more distance than the straight-line path distance. If the T's route is direct enough, and the price of time is in the right range, using the T becomes a viable mode.

5. Validation of the Problem Solving Principle (AKA "The Point™")

There are really two different representations of roughly the same knowledge in each subsection of section 4. In the "Knowledge Summary" box, the real knowledge stored in the system is described. In the main body of each subsection, though, a different representation exists; the appropriate conditions for each mode are described. For example, the biking description said:

> "Since both modes are free (aside from time), the walking+biking combo is a popular among users with a low value of time."

This is really a colloquial description of the decision points that come out of the Transit System. These decision points are **what to do**, e.g. "if these conditions are present then pick this mode." They are a result of the knowledge of the agents, and the ability of the blackboard to combine solutions.

It is much easier to express knowledge for each mode of transportation than to describe the cases in which one mode is preferred over the other. (We'll call the former "modularized knowledge;" the latter is named "procedure knowledge.") This is true for four reasons.

First, modularized knowledge scales well; if there are 10 different modes of transportation then the system can avoid describing all {10 choose 2} possibilities.

Second, modularized knowledge is a more natural way to express the knowledge. I usually think "The Zip Car will be cheaper than the taxicab" instead of "I prefer the Zip Car if I will be at my destination for less than one and a half hours."

Next, modularized knowledge makes the system easy to debug, maintain, and expand. If, for example, the T is never appearing in itineraries, the system's administrator can trace through application of the T module's knowledge, instead of searching through all of the other modes of transportation. Alternatively, consider the case in which taxicabs change their fares. The administrator only needs to modify the taxicab agent, instead of how all of the other modules interact with the taxicab mode.

Finally, and most importantly, modularized knowledge is an expression of **what the program should know** instead of **what it should do.** The former representation embodies the facts about each mode of transportation; the latter is a database lookup of what to do in

any given scenario. Knowing about each mode of transportation is preferred for the three reasons given above – it is easier to express, more natural to express, and easier to manipulate and work with.

6. Limitations of Transit Advisor

Just like most real-world systems, the Transit Advisor's abilities are limited. There are principally two kinds of situations that the system cannot handle – extra freedom or extra constraints. Each of these kinds of situations will be discussed below.

6.1. Extra Freedom

There are things about the real world that make it possible to construct a better itinerary than the one created by Transit Advisor. These mostly boil down to additional modes of transportation that the system does not know about. For example:

- The user can get a free car ride to a specific point,
- Buses, or
- Skateboarding.

6.2. Extra Constraints

The system should, ideally, be able to cope with the constraints that are encountered in the real world. Some of these constraints are acknowledged by the system, such as the predicate that only allows a bike to be used if it is not raining outside. There are many other constraints, inversely, that are not handled by the system. Some of these are given, with example scenarios, below.

- **Constraint**: Must travel on land, not through buildings, etc.
- **Example**: Since the system plots straight-line segments, itineraries often include drives or walks across the Charles river.

- **Constraint**: No wall clock support
- **Example**: "I need to be at the Zip Car by 6pm, since that is the time of my reservation."

- **Constraint**: No absolute budget constraints (time or money)
- **Example**: "I cannot spend more than $5 cash." or "I must be there in less than 20 minutes."

- **Constraint**: Time limits on modes of transportation

- **Example**: "I cannot bike for more than 2 miles."

- **Constraint**: Cargo space requirements
- **Example**: "I cannot bike because I have 5 bags of groceries."

6.3. Scope of Information

As obvious as it is, the system only knows about a specific geographic region. Furthermore, it only knows about taxicab fares in Boston, the main parts of the red and green lines, and the location of a subset of Zip Cars in the area.

7. Reflections

In this section we will discuss the merits and pitfalls of the system with 20/20 hindsight, and conclude with a list of basic lessons learned.

7.1. What Went Well

First, the problem solving paradigm fits the problem well. Knowledge is abstracted into agents who know a lot about a specific mode of transportation, and the system combines these agents so that they can work together while maintaining modularity. More explanation on this can be found in section 5, but the ultimate validation can be found in the fact that the system was relatively easy to build. The code of each agent is very straightforward, yet the system is powerful enough to produce itineraries involving complex sequences of transportation.

Next, the system seems to give good answers. From a top-down perspective, the Transit Advisor gives itineraries which look like what I would personally do, as a Boston cosmopolitan. From a bottom-up perspective, the costs and availability of various modes of transportation are representative of the real world, as is the combination and search approach.

7.2. What Went Poorly & Lessons Learned

It has been said that one can learn the most from their mistakes, a maxim that certainly holds in this case. This section describes the problems that unexpectedly came up, and what I learned from them.

In short: many things. The Transit Advisor is significantly different from what I originally envisioned. It is not a rule-based system, and has very limited rule inference capabilities (see Figure 3). In essence, the knowledge expressible in these transit situations is

not very deep, so I had to re-focus on the itinerary search. This result brings us to the first lesson, which I have picked up the hard way.

"Spend enough time early on to convince yourself that the problem is well suited for your goals."

I did not do this reflection before picking my topic. I got lucky because my topic had other interesting attributes that I could re-focus on.

There are things that I did right which have paid off in hindsight. Despite the fact that my topic did not have sufficient relevance for a rule-based system, I spent enough time up front on reflecting on the problem solving paradigm to determine that a rule-based approach would not work. This allowed me to re-focus without wasting too much energy on the rule-based approach before trying something else.

"Spend time choosing the correct problem solving paradigm. Choose the best one for the job, not your favorite."

8. Conclusion

In conclusion, the Transit Advisor is a system that builds itineraries for travel around Boston and Cambridge using various modes of transportation. It uses the blackboard and agent problem solving paradigm, also known as GPS, to modularize knowledge about each mode of transportation into single agents. This representation, in which each agent is told what it should know about one mode of transportation, enables complex itineraries to be built using collaboration between agents.

9. Appendix – Knowledge Base

Java code is given for each agent, in the same order as section 4. First, though, the interface for a transportation mode is given.

I will also give an English description of the dynamics of the Walking class. The real action happens inside the findRoute() routine. The routine is only three lines long because the mode of transportation is so simple. First, the travel distance is found, then the cost is computed given the walking speed and the value of time. The procedure then returns with a new itinerary which contains the calculated cost and a pointer back to the Walking class (so that the system knows which mode submitted that calculation).

9.1. XPortMode interface

```
public interface XportMode {

        public Itenerary findRoute(Point origin, Point destination, List availableXportModes,
Gui context);

        public String description();

}
```

9.2. Walking

```
public class Walking implements XportMode {

        private final static double walkingSpeedInMPH = 5;
        private static Walking instance = null;

        private Walking() {     }

        public static Walking getInstance() {
                if(instance == null) {
                        instance = new Walking();
                }
                return instance;
        }

        public Itenerary findRoute(Point origin, Point destination,
                        List availableXportModes,
                        Gui context) {

                double distanceInMiles = Gui.milesPerPixel * origin.distance(destination);
```

```
		double totalCost = (distanceInMiles / walkingSpeedInMPH) *
Gui.dollarsPerHour;
		return new Itenerary(origin, destination, this, totalCost);
	}

	public String description() {
		return "on foot";
	}
}
```

9.3. Biking

```
public class Biking implements XportMode {

	private final static double bikeSpeedInMPH = 20;
	private static Biking instance = null;

	private Biking() {	}

	public static Biking getInstance() {
		if(instance == null) {
			instance = new Biking();
		}
		return instance;
	}

	public Itenerary findRoute(Point origin, Point destination,
			List availableXportModes,
			Gui context) {

		availableXportModes.remove(this);
		Gui yo = new Gui(availableXportModes, origin, destination,
context.bikePosition);

		Itenerary toGetToBike = yo.findRoute(origin, context.bikePosition);

		double distanceFromBikeInMiles = Gui.milesPerPixel *
context.bikePosition.distance(destination);
		double costFromBike = (distanceFromBikeInMiles / bikeSpeedInMPH) *
Gui.dollarsPerHour;

		yo.closeMe();
		toGetToBike.addToEnd(new Segment(context.bikePosition, destination, this,
costFromBike));
		return toGetToBike;
```

```
	}

	public String description() {
		return "riding a bike";
	}
}
```

9.4. Taxicab

```
public class Taxicab implements XportMode {

	private final static double taxiSpeedInMPH = 45;
	private final static double averageHoursWaitingForTaxi = 3/60;
	private final static double startingMeter = 1.75;
	private final static double additionalEighthMile = .3;
	private static Taxicab instance = null;

	private Taxicab() {	}

	public static Taxicab getInstance() {
		if(instance == null) {
			instance = new Taxicab();
		}
		return instance;
	}

	public Itenerary findRoute(Point origin, Point destination,
			List availableXportModes,
			Gui context) {

		double distanceInMiles = Gui.milesPerPixel * origin.distance(destination);
		double timeWaitingForCabCost = averageHoursWaitingForTaxi *
Gui.dollarsPerHour;
		double timeRidingCost = (distanceInMiles / taxiSpeedInMPH) *
Gui.dollarsPerHour;
		double costOfCabRide = startingMeter +
distanceInMiles*8*additionalEighthMile;
		double totalCost = timeWaitingForCabCost + timeRidingCost +
costOfCabRide;
		return new Itenerary(origin, destination, this, totalCost);

	}

	public String description() {
		return "taking a taxi";
	}
```

```
}
```

9.5. ZipCar

```
public class ZipCar implements XportMode {
	private static ZipCar instance = null;

	private final static double carSpeedInMPH = 40;
	private final static double dollarsPerHour = 8.5;
	private static double hoursAtDestination = -1;
   public static List zipcars = new ArrayList();
   private static boolean eligible = true;

	private ZipCar() {
		String in = JOptionPane.showInputDialog(null, "Are you 21+ years old?
(yes/no)");
		if(in.equals("no")) {
			eligible = false;
		// HACKING FOR USER INTERFACE SHORTCUTS
		} else if (in.substring(0,3).equals("def")) {
			hoursAtDestination = 3;
			if(in.equals("def")) {
				Gui.def = 15;
			} else {
				Gui.def = Integer.valueOf(in.substring(3,5)).intValue();
			}
      // END HACKING FOR USER INTERFACE SHORTCUTS
		} else {
			hoursAtDestination =
				Double.valueOf(JOptionPane.showInputDialog(null, "How
long will you spend at your destination? (hrs) (needed to calculate cost of
ZipCar)")).doubleValue();
		}


		if(eligible) {
			zipcars.add(new Point(132, 569));
			zipcars.add(new Point(171, 625));
			zipcars.add(new Point(203, 648));
			zipcars.add(new Point(200, 563));
			zipcars.add(new Point(88, 509));
			zipcars.add(new Point(278, 532));
			zipcars.add(new Point(352, 592));
			zipcars.add(new Point(366, 591));
```

```
                zipcars.add(new Point(71, 471));
                zipcars.add(new Point(21, 413));
                zipcars.add(new Point(17, 235));
                zipcars.add(new Point(73, 145));
                zipcars.add(new Point(41, 92));
                zipcars.add(new Point(45, 76));
                zipcars.add(new Point(27, 70));
                zipcars.add(new Point(304, 55));
                zipcars.add(new Point(253, 141));
                zipcars.add(new Point(186, 180));
                zipcars.add(new Point(181, 267));
                zipcars.add(new Point(476, 216));
                zipcars.add(new Point(354, 351));
                zipcars.add(new Point(355, 355));
                zipcars.add(new Point(510, 304));
                zipcars.add(new Point(702, 321));
                zipcars.add(new Point(716, 352));
                zipcars.add(new Point(625, 438));
                zipcars.add(new Point(539, 474));
                zipcars.add(new Point(450, 462));
                zipcars.add(new Point(604, 584));
                zipcars.add(new Point(728, 631));
                zipcars.add(new Point(596, 671));
                zipcars.add(new Point(530, 188));
                zipcars.add(new Point(337, 245));
                zipcars.add(new Point(289, 317));
                zipcars.add(new Point(180, 368));
                zipcars.add(new Point(188, 145));
        }
}

public static ZipCar getInstance() {
        if(instance == null) {
                instance = new ZipCar();
        }
        return instance;
}

public Itenerary findRoute(Point origin, Point destination,
                List availableXportModes,
                Gui context) {

        if(!eligible) {
                return new Itenerary(origin, destination, this, Double.MAX_VALUE);
        }

        availableXportModes.remove(this);
```

```
                Gui yo = new Gui(availableXportModes, origin, destination,
context.bikePosition);

                // find closest ZipCar
                double closestDist = Double.MAX_VALUE;
                Point closestCar = null;
                for(Iterator i = zipcars.iterator(); i.hasNext(); ) {
                        Point tmpCar = (Point) i.next();
                        if (closestDist > tmpCar.distance(origin)) {
                                closestDist = tmpCar.distance(origin);
                                closestCar = tmpCar;
                        }
                }

                Itenerary toGetToZipCar = yo.findRoute(origin, closestCar);

                double distanceFromCarInMiles = Gui.milesPerPixel *
closestCar.distance(destination);
                double timeCostFromCar = (distanceFromCarInMiles / carSpeedInMPH) *
Gui.dollarsPerHour;
                double moneyCostFromCar = (2*(distanceFromCarInMiles /
carSpeedInMPH) + hoursAtDestination) * ZipCar.dollarsPerHour;
                double totalCostFromCar = timeCostFromCar + moneyCostFromCar;
                double totalItinCost = totalCostFromCar + toGetToZipCar.getCost();

                yo.closeMe();
                toGetToZipCar.addToEnd(new Segment(closestCar, destination, this,
totalCostFromCar));
                return toGetToZipCar;
        }

        public String description() {
                return "using a ZipCar";
        }
}
```

9.6. T

```
public class T implements XportMode {
        public static List stopsInOrder;
        private static T instance = null;
        public static int indexColorChange = 4;

        private final static double tSpeedInMPH = 50;  // they probably don't go this fast,
but
```

```
	// cars don't really go (for example) 40 MPH if you're thinking about a straight line
	// (i.e. no stop lights, turns), so we scale up the T's speed since it doesn't have this
	// assumption
	private final static double dollarsPerRide = 1.5;

	private T() {
		stopsInOrder = new ArrayList();
		stopsInOrder.add(new Point(63, 30));
		stopsInOrder.add(new Point(263, 205));
		stopsInOrder.add(new Point(467, 244));
		stopsInOrder.add(new Point(693, 271));
		stopsInOrder.add(new Point(762, 330));
		stopsInOrder.add(new Point(733, 405));
		stopsInOrder.add(new Point(679, 416));
		stopsInOrder.add(new Point(598, 446));
		stopsInOrder.add(new Point(467, 483));
		stopsInOrder.add(new Point(372, 465));
	}

	public static T getInstance() {
		if(instance == null) {
			instance = new T();
		}
		return instance;
	}



//	[63, 30]
//	[263, 205]
//	[467, 244]
//	[693, 271]
//	[762, 330]
//	[733, 405]
//	[679, 416]
//	[598, 446]
//	[467, 483]
//	[372, 465]
//
//	 harvard
//	 central
//	 kendall
//	 carles/mgh
//	 park st
//	 boylston
//	 arlington
//	 copley
//	 hynes/ica
```

```
//        kenmore


        public Itenerary findRoute(Point origin, Point destination,
                        List availableXportModes,
                        Gui context) {

                availableXportModes.remove(this);
                Gui toTOrigin = new Gui(availableXportModes, origin, destination,
context.bikePosition);
                Gui toTDest = new Gui(availableXportModes, origin, destination,
context.bikePosition);

                // find closest T to origin
                double closestDist = Double.MAX_VALUE;
                Point closestTOrigin = null;
                for(Iterator i = stopsInOrder.iterator(); i.hasNext(); ) {
                        Point tmpCar = (Point) i.next();
                        if (closestDist > tmpCar.distance(origin)) {
                                closestDist = tmpCar.distance(origin);
                                closestTOrigin = tmpCar;
                        }
                }

                // find closest T to destination
                closestDist = Double.MAX_VALUE;
                Point closestTDest = null;
                for(Iterator i = stopsInOrder.iterator(); i.hasNext(); ) {
                        Point tmpCar = (Point) i.next();
                        if (closestDist > tmpCar.distance(destination)) {
                                closestDist = tmpCar.distance(destination);
                                closestTDest = tmpCar;
                        }
                }

                Itenerary toGetToT = toTOrigin.findRoute(origin, closestTOrigin);
                Itenerary toGetFromT = toTDest.findRoute(closestTDest, destination);

                double distanceOnT = 0;
                for(int i = stopsInOrder.indexOf(closestTOrigin); i <
stopsInOrder.indexOf(closestTDest); i++) {
                        distanceOnT += ((Point) stopsInOrder.get(i)).distance((Point)
stopsInOrder.get(i+1));
                }
                distanceOnT *= Gui.milesPerPixel;

                double timeCostFromT = (distanceOnT / tSpeedInMPH) *
Gui.dollarsPerHour;
```

```
                double moneyCostFromT = dollarsPerRide;
                double totalCostFromT = timeCostFromT + moneyCostFromT;
                double totalItinCost = totalCostFromT + toGetToT.getCost() +
toGetFromT.getCost();

                toTOrigin.closeMe();
                toTDest.closeMe();
                toGetToT.addToEnd(new Segment(closestTOrigin, closestTDest, this,
totalCostFromT));
                toGetToT.addToEnd(toGetFromT);
                return toGetToT;
        }

        public String description() {
                return "riding the T";
        }
}
```