Appendix C

Appendix to Design Heuristics (Chapter IV)

C-1 GOMS Evaluation for Airline-Scheduling Task in Chapter IV-1

> *Given an origin and a destination city, the user "attempts to locate the two flights arriving in and departing from a layover city that offer the minimum amount of `down time' between the flight times and the beginning and ending time of a scheduled meeting (in the layover city)".*

In the following example designs, suppose that the origin and destination cities are *Los Angeles (LAX)* and *Boston (BOS)* and that the layover city is *Chicago (ORD)*. Further, suppose that the meeting time is from 2 p.m. to 4 p.m. Our airline scheduling data set has a total of 135 flights. 47 flights originate from Los Angeles (LAX) and 18 of those go to Boston (BOS). Of those 18 flights, only 10 fulfill the meeting time constraints.

C-1.1 GOMS Evaluation for the Cognitive Solution of the Airline Scheduling Task

Flight number	Origin city	Destination city	Flight arrival time	Flight departure time
United_652	ORD	EWR	5:32 a.m.	2:30 a.m.
NW_826	MEM	ATL	10:13 a.m.	8:00 a.m.
Trans_235	MDW	LGA	9:00 a.m.	5:55 a.m.
United_244	LAX	ORD	3:37 p.m.	9:45 a.m.
United_342	ORD	MCO	12:19 a.m.	8:55 a.m.
American_494	ORD	EWR	7:48 a.m.	4:30 a.m.
American_491	BOS	DFW	7:29 a.m.	4:15 a.m.
		• • •		
United_7282	JFK	BOS	8:31 a.m.	7:00 a.m.
NW_5875	MEM	MOB	9:45 a.m.	8:20 a.m.
Delta_725	DFW	SFO	10:54 a.m.	9:00 a.m.
TWA_7762	PHL	JFK	7:53 a.m.	7:04 a.m.
American_346	ORD	LGA	10:04 a.m.	7:00 a.m.
American_340	ORD	LGA	11:36 a.m.	8:30 a.m.
Continental_1943	AH	LAS	9:46 a.m.	8:39 a.m.

Figure C-1: Cognitive design for the airline-scheduling task
(Note that the flights are not all shown here because the table is very large)

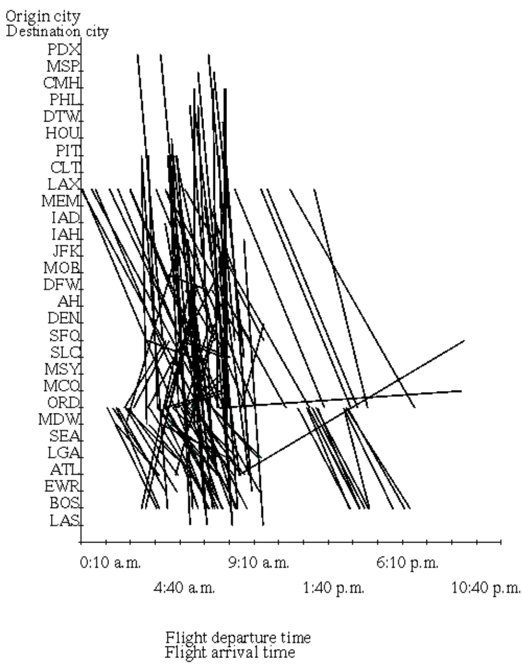
General goal	Cognitive, perceptual, or articulatory step taken by user	Time taken (msec)	**Sub total (msec)**	**Total time (msec)**	Target objects (data or graphics)
Find flight originating from Los Angeles (LAX) and point left hand at flight	Attend find flight origin	50			
	Initiate eye movement	50			
	Eye movement	30			
	Read origin city name	290			
	Initiate comparison	50			
	Compare origin city name with LAX	50			
	Verify comparison	50			
	Sub-total		570		
Find destination of flight and determine whether it is Chicago (ORD)	Attend find flight destination	50			
	Initiate eye movement	50			
	Eye movement	30			
	Read destination city name	290			
	Initiate comparison	50			
	Compare destination city name with ORD	50			
	Verify comparison	50			
	Sub-total		570		
Get flight arrival time	Attend get flight arrival time	50			
	Initiate eye movement	50			
	Eye movement	30			
	Read arrival time	290			
	Attend compare time with start of meeting	50			
	Compare if time is before meeting start time	50			
	Verify compare	50			
	Sub-total		570		
Compute time before meeting	Attend compute time before meeting	50			
	Subtract flight arrival time from 4 p.m. (3 significant figures) 150 + (*n*-1) * 100	350			
	Verify time before meeting	50			
	Sub-total		450		
Determine if current prior meeting downtime is the minimum	Attend compare with current min downtime	50			
	Compare with current min downtime (Assume an average to 2 significant figure comparison = 50 msec * 2 = 100 msec)	100			
	Verify results	50			
	Sub-total		200		

<table>
<tr><td colspan="6">Time taken to move finger in the calculation below we estimate as follows:
We assume that the visualization is enlarged so that each label entry is at least the width of a finger. Or else, finger pointing would be very difficult. Thus,
Height of widest finger = Height of each entry = 0.5 inches
For this particular design and data set, we assume that 5 out of the 10 flights that fulfill all time and city constraints require min changes. If they are evenly spread out, the movement per min number is 135/5 steps = 27 * 0.5 inches = 13.5 inches
Thus using Fitts Law, the estimated time for movement is 100 * ($\log_2$ (13.5/0.5)+.5) = 525 msec

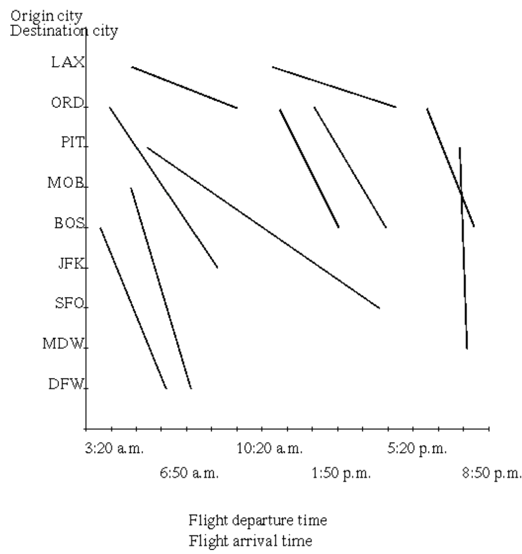
Note that for simplicity we assume that all the flights fit within 1 screen. This is highly unlikely since there are 135 flights. However even with this simplifying assumption the estimated time for this cognitive solution is still very significant.</td></tr>
<tr><td rowspan="8">If so, point to current row with right hand and keep current min downtime in STM</td><td>Attend change finger positions</td><td>50</td><td></td><td></td><td></td></tr>
<tr><td>Initiate finger lift</td><td>50</td><td></td><td></td><td></td></tr>
<tr><td>Lift finger</td><td>60</td><td></td><td></td><td></td></tr>
<tr><td>Initiate finger move</td><td>50</td><td></td><td></td><td></td></tr>
<tr><td>Move finger
100 * ($\log_2$ (13.5/0.5)+.5)</td><td>525</td><td></td><td></td><td></td></tr>
<tr><td>Initiate finger drop</td><td>50</td><td></td><td></td><td></td></tr>
<tr><td>Drop finger</td><td>60</td><td></td><td></td><td></td></tr>
<tr><td>Sub-total</td><td></td><td>845</td><td></td><td></td></tr>
<tr><td colspan="6">Total time for processing all rows for flight before meeting include:
Time taken to process origin of all rows = 135 * 570 msec. = 76950 msec.
Time taken to process destination of all rows. This only applies to flights with origin from LAX of which there are 18 = 18 * 570 msec. = 10260 msec.
Time taken to process total downtime before meeting. This only applies to flights from LAX to ORD of which there are 18 = 18 * 570 msec. = 10260 msec.
Time taken to process min-downtime. This only applies to flights from LAX to ORD that arrive before the meeting of which there are 10 = 10 * (450 + 200) msec = 6500 msec.
Time taken for finger movement. This depends on the number of times we have to change the min entry. Since there are 10 flights which fulfills both city and time constraints, we assume half of these require min changes thus total time = 5 * 845 msec. = 4225 msec.</td></tr>
<tr><td>Total time taken for processing flight with minimum total before meeting downtime</td><td>76950 + 10260 + 10260 + 6500 + 4225</td><td></td><td></td><td>108195</td><td></td></tr>
<tr><td colspan="6">Repeat for getting total downtime after meeting. Total time for processing all rows for flight after meeting include:
Time taken to process origin of all rows = 135 * 570 msec. = 76950 msec.
Time taken to process destination of all rows. This only applies to flights with origin from ORD of which there are 47 = 47 * 570 msec. = 26790 msec.
Time taken to process total downtime before meeting. This only applies to flights from ORD to BOS of which there are 19 = 19 * 570 msec. = 10830 msec.
Time taken to process min-downtime. This only applies to flights from ORD to BOS that depart after the meeting of which there are 10 = 10 * (450 + 200) msec = 6500 msec.
Time taken for finger movement. This depends on the number of times we have to change the min entry. Since there are 10 flights which fulfills both city and time constraints, we assume half of these require min changes thus total time = 5 * 845 msec. = 4225 msec.</td></tr>
<tr><td>Total time taken for processing flight with minimum total after meeting downtime</td><td>76950 + 26790 + 10830 + 6500 + 4225</td><td></td><td></td><td>125295</td><td></td></tr>
<tr><td rowspan="4">Add both downtimes
(We assume that both downtimes can be stored in STM)</td><td>Attend add downtimes</td><td>50</td><td></td><td></td><td></td></tr>
<tr><td>Mental add
(Assume 3 figures)
(150 + (n-1) * 100)</td><td>350</td><td></td><td></td><td></td></tr>
<tr><td>Verify results</td><td>50</td><td></td><td></td><td></td></tr>
<tr><td>Sub-total</td><td>450</td><td></td><td></td><td></td></tr>
<tr><td>Total time</td><td>108195 + 125295 + 450</td><td></td><td></td><td>233940</td><td></td></tr>
</table>

Total time taken to solve the airline-scheduling task using Figure C-1 is 234 seconds or approximately 4 minutes.

C-1.2 GOMS Evaluation for the Pure Perceptual Solution of the Airline Scheduling Task



(a) Full data set
This visualization shows all the elements in the data set (i.e. all 135 flights).



(b) Truncated data set.
This example visualization shows the ideal case where there is little occlusion among the different flight lines. This data set was chosen so that it contains some flights that fulfill the task constraints as well as some other random flights that do not occlude one another.

Figure C-2: Perceptual design for the airline-scheduling task
Each line represents a flight with origin and destination city mapped onto the *y-axis* and arrival and departure time mapped onto the *x-axis*. This is the best design that gets generated when ONLY mapping operations are considered by the automatic system. I.e. this is the best possible design from current state of the art systems.

General goal	Cognitive, perceptual, or articulatory step taken by user	Time taken (msec)	**Sub total (msec)**	**Total time** (msec)	Target objects (data or graphics)
Scan on *y-axis* to find origin city (LAX). We assume that cities are ordered alphabetically, thus a binary type search can be applied. Since there are 29 cities, a total of $\log_2(29) = 5$ searches are necessary to get to the desired city	Attend find origin city on *y-axis*	50			
	Initiate eye movement	50			
	Eye movement	30			
	Read city name	290			
	Initiate comparison	50			
	Compare origin city name with LAX	50			
	Verify comparison	50			
	Sub-total		570		
	Total time (5 * 570)		2850		
Scan to the right from *origin* city position and process next line that start from this *y-position*. Specifically scan to end point of line and determine if it is from ORD as well as before the meeting start time (we assume that the start and end meeting times as well as the origin, layover, and destination cities are marked on the display or stored in STM by the user).	Attend process line end-point	50			
	Initiate eye movement	50			
	Eye movement to end of line	30			
	Perceive point	100			
	Is point arriving at destination city?	50			
	Verify point arriving at destination city	50			
	Sub-total		330		
	Is point before meeting start time?	50			
	Verify that point is before meeting start time	50			
	Sub-total		100		
Compare end-point with current best candidate flight (here we determine if the current point is before or after our finger position which is at the current best flight candidate).	Attend compare *x-distance*	50			
	Is point after finger position?	50			
	Verify results	50			
	Sub-total		150		
The distance moved here depends on the x-distance between different flights before the meeting. This distance is NOT dependent on number of elements as was in the previous example but is instead dependent upon the x-axis scale. Assuming that the visualization fills the entire screen, the maximum x-axis length would be 14 inches in a 21 inch CRT display screen. Since 5 moves are necessary, we assume an average of 2.8 inches per move					
If so, point to current row with left hand	Attend change finger positions	50			
	Initiate finger lift	50			
	Lift finger	60			
	Initiate finger move	50			
	Move finger $100 * (\log_2 (2.8/0.5)+.5) = 299$ = approx 300	300			
	Initiate finger drop	50			
	Drop finger	60			
	Sub-total		620		

Total time for processing all rows include: Time taken to get to the proper origin city position on the *y-axis* = 2850 msec. Time taken to process destination of all lines that start from the origin city (LAX) of which there are 18 = 18 * 330 msec. = 5940 msec. Time taken to process total downtime before meeting. This only applies to flights from LAX to ORD of which there are 18 = 18 * 100 msec. = 1800 msec. Time taken to determine if flight is best candidate. This only applies to flights from LAX to ORD that depart after the meeting of which there are 10 = 10 * 100 msec = 1000 msec. Time taken for finger movement. This depends on the number of times we have to change the min entry. Since there are 10 flights which fulfills both city and time constraints, we assume half of these require min changes thus total time = 5 * 620 msec. = 3100 msec.					
Total time taken for processing flight with minimum total before meeting downtime	2850 + 5940 + 1800 + 1000 + 3100			**14690**	
Repeat for getting total downtime after meeting. Total time for processing all rows for flight after meeting include: Time taken to get to the proper origin city position on the *y-axis* = 2850 msec. Time taken to process destination of all lines that start from the layover city (ORD) of which there are 47 = 47 * 330 msec. = 15510 msec. Time taken to process total downtime before meeting. This only applies to flights from LAX to ORD of which there are 19 = 19 * 100 msec. = 1900 msec. Time taken to determine if flight is best candidate. This only applies to flights from LAX to ORD that depart after the meeting of which there are 10 = 10 * 100 msec = 1000 msec. Time taken for finger movement. This depends on the number of times we have to change the min entry. Since there are 10 flights which fulfills both city and time constraints, we assume half of these require min changes thus total time = 5 * 620 msec. = 3100 msec.					
Total time taken for processing flight with minimum total after meeting downtime	2850 + 15510 + 1900 + 1000 + 3100			**24360**	
Get exact downtime before meeting	Attend get exact downtime				
	Attend lookup arrival time from *x-axis*	1135			
	Mental subtract arrival time from meeting start time (150 + (n-1) * 100)	350			
	Verify downtime	50			
	Sub-total		1535		
Get exact downtime after meeting		1535			
Add both downtimes	Attend add	50			
	Mental add	350			
	Verify results	50			
	Sub-total	450			
Total time	14690 + 24360 + 1535 + 1535 + 450			**42570**	

Total time taken to solve the airline-scheduling task using Figure C-2 is approximately 43 seconds.

C-1.3 GOMS Evaluation for the Perceptual + Data Computation Solution of the Airline Scheduling Task

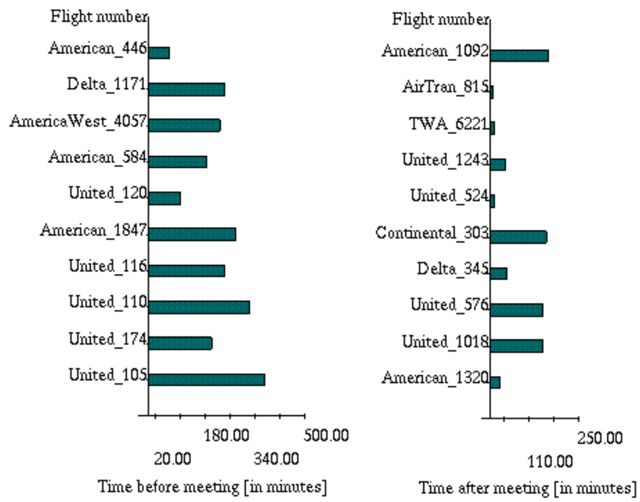


Figure C-3: Design generated when data processing operations are integrated into the automatic visualization system. The full data set is considered here but data transforms are applied by the automatic system to filter the data set so that only relevant flights are shown. The total downtime before the meeting for the flights from LAX to ORD is shown on the left chart and the total downtime after the meeting for the flights from ORD to BOS is shown on the right chart.

General goal	Cognitive, perceptual, or articulatory step taken by user	Time taken (msec)	**Sub total (msec)**	**Total time** (msec)	Target objects (data or graphics)
Find shortest bar in first chart	Attend find shortest bar	50			
	Initiate eye movement	50			
	Eye movement	30			
	Perceive shortest bar	100			
	Verify shortest bar	50			
	Sub-total		280		
Lookup total downtime for that bar	This figure was computed from Design 4, Task 2 in appendix E-2.4	1135			
Find shortest bar in second chart		280			
Lookup total downtime for that bar		1135			
Add both downtimes (assume 3 significant figures)	150 + (n – 1) * 100	350			
Total time	280 + 1135 + 280 + 1135 + 350			**3180**	

Total time taken to solve the airline-scheduling task using Figure C-3 is approximately 3 seconds.

C-2 Airline-Scheduling Task Design Alternatives

In chapter IV-1 we discussed several design alternatives for the airline-scheduling task that consists of different blends of data and mapping transforms. In this section we present several more design alternatives and discuss their strengths and weaknesses to highlight some of the issues that arise when we are trying to make decisions about whether to use data or mapping transforms to solve user goals. In particular we want to illustrate that it is not always best to fully pre-compute all tasks and the best design often is one which consists of a combination of both data computation and perceptual mapping operations.

In chapter IV-1 we focussed on design examples that assumed complete task specification (i.e. all the task parameters, e.g. *origin* and *destination cities*, *start* and *end meeting times*, are known before the analysis). In these cases, the data computation solution performs very well. However when tasks cannot be fully captured at the outset, it becomes necessary to map more of the data to graphics (i.e. it becomes more difficult to use data computation solutions) as was shown in Figure IV-5.Another way to deal with low task specificity (particularly imprecise task parameters) is to add input devices into the design and let users specify the task requirements during the data analysis process. In Figure C-4, we use the same data computation design as in Figure IV-3, however here, we assume that specific knowledge of the task parameters (e.g. the *layover city* and *meeting time* information) is unavailable. To enable data computation we integrate the graphic design with an option button for specifying the *layover city* and two sliders for specifying the start and end meeting times.

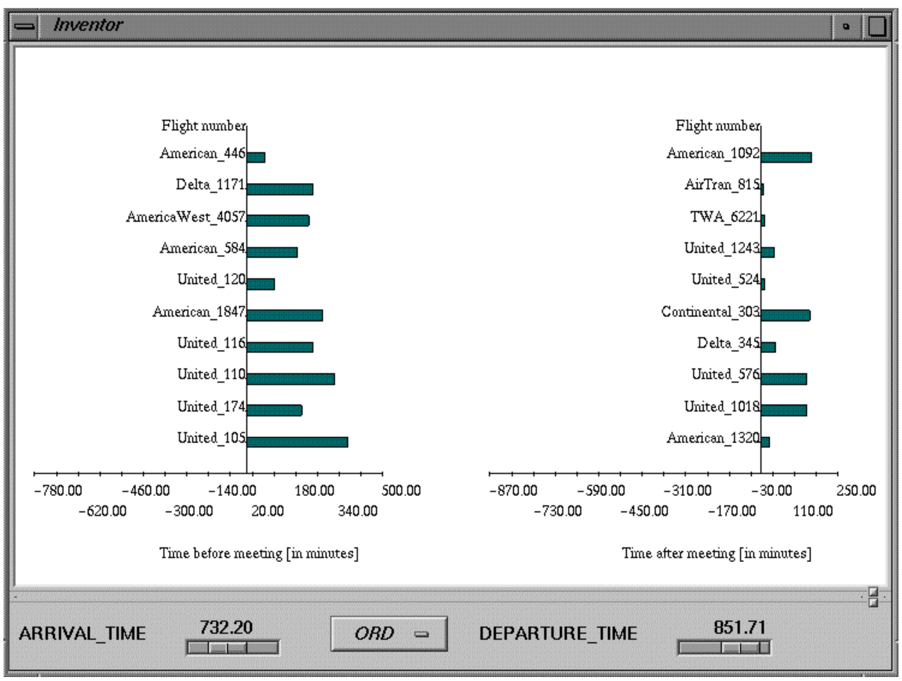


Figure C-4: Solving the airline schedule task with input devices. Here we assume that some of the airline-scheduling task constraints are unknown (i.e. the *layover-city* is not known and the *beginning* and *ending* meeting times are also not known). These constraints can be entered into the system through *sliders* and *option-buttons*.

Through these input devices, users may test different sets of input task parameters. Each different test, however, requires manipulation of some or all of the devices. In addition, only results from a single test are shown at any one time, thus it is highly probable that users may forget results from previous tests and need to repeat a test several times. These factors increase the difficulty and time required for solving the task. Therefore if the user is really unsure of the task parameters, or if the user foresees that the particular task step may be repeated many times, it is more effective to encode the information by mapping all the data to graphics so that the task can be performed perceptually rather than through data computations and summarizations coupled with input devices. For example Figure IV-1 encodes all of the airline scheduling data with mapping techniques, thus it does not require the system to know any of the task parameters (i.e. no user input is required even if we are unsure of the task parameters). Figure C-5 filters out all the flights based on *origin, destination,* and *layover* cities, but show the *arrival* and *departure* times of the relevant flights, so that there is less clutter compared to Figure IV-1, but some flexibility in our task meeting time constraints.

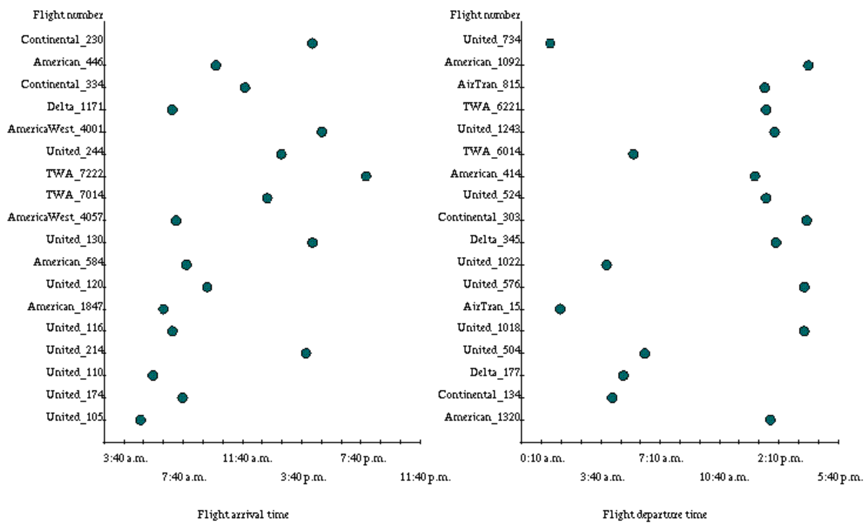


Figure C-5: Design alternative for airline-scheduling task where flights are filtered based on origin, destination, and layover city information. Arrival and departure times are shown however to allow flexibility in our task meeting time constraint. *Flight_arrival_time* is mapped to the *x-axis* of the left chart and *flight_departure_time* is mapped to the *x-axis* of the right chart.

Another design alternative for our airline-scheduling task is to combine the results of both the flights to and from the *layover city* and compute the total downtime for all flight pairs as in Figure C-6. The advantages of this display are that users need not compute the *total downtime* perceptually and that the *total downtime* can be determined with better accuracy compared to Figure IV-3. This is because in Figure C-6 the user only needs to look up the bar ends, whereas in Figure IV-3 the user is required to estimate the sum lengths of two spatially separated bars.

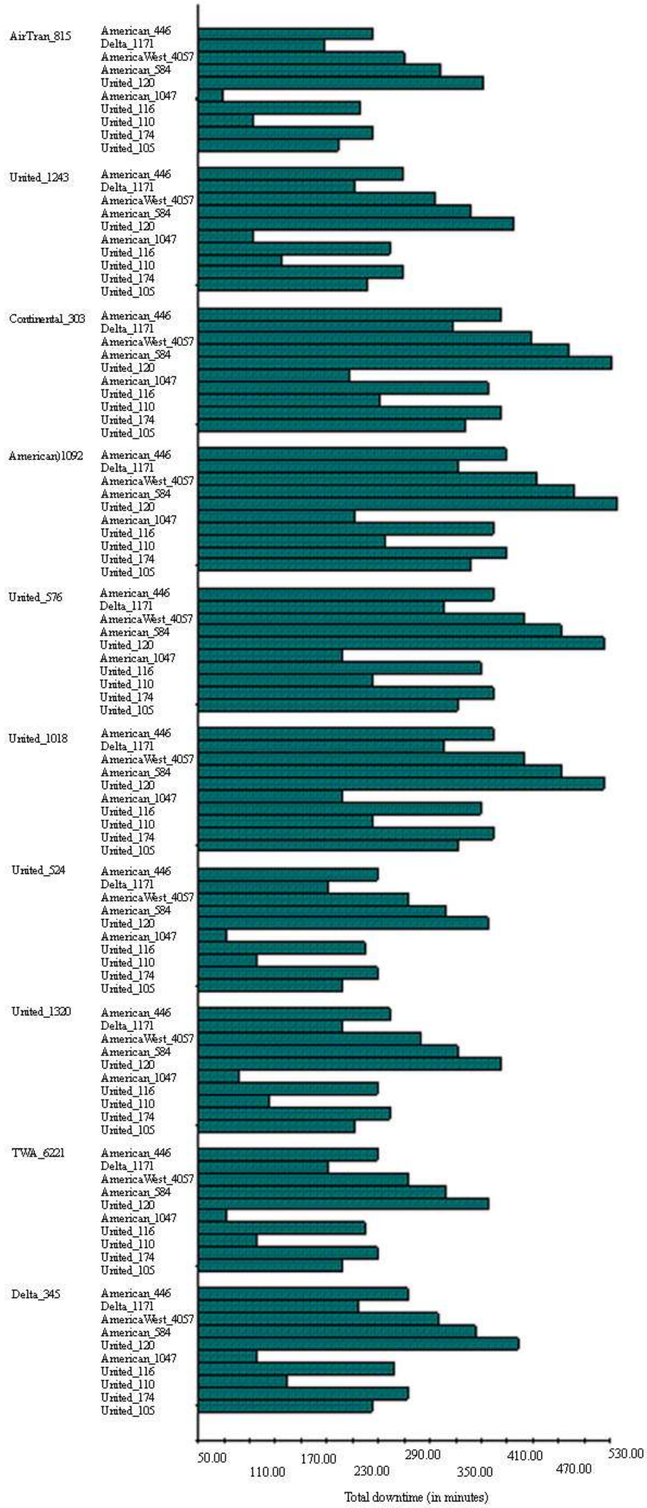


Figure C-6: Solving the airline-scheduling task where all searches as well as the total-downtime computations are performed with data transform techniques. *Total_downtime* is pre-computed and mapped to the *x-axis*. Note that in this design there are many more graphical elements than the other airline-scheduling task designs because all possible pairs of flights must be considered and shown.
[Note: the indented labels for this design was manually generated]

A weakness of Figure C-6 however, is that all flight pairs must be listed. Consequently if there are *n* flights before the meeting and *m* flights after the meeting we would need to show $n * m$ data points, whereas in Figure IV-3 we would only need to show $n + m$ data points. These extra data points make Figure C-6 much larger than Figure IV-3 and thus it requires more screen space (Figure IV-3 only shows 20 bars while Figure C-6 shows 100 bars). For larger data sets, the visualization in Figure C-6 might not even fit within a CRT screen. In such cases, we can only view a subsection of the visualization at a time and must navigate, using input devices, to different sections of the visualization. The manipulation cost involved in navigating and the cognitive cost involved in managing the different information slices make finding the two shortest bars in Figure IV-3 more efficient than Figure C-6 for any reasonably sized data set (i.e. > 30 elements).

C-3 Task Model

In this thesis, we focus on tasks in exploratory data analysis (EDA). EDA was first introduced by Tukey in his book entitled *Exploratory Data Analysis* [*Tukey, 1977*], where he described a series of logical and paper/pencil techniques for processing data. Since then several task models have been developed that try to characterize and define the operations involved in EDA. Some of these task models break high-level data analysis operations (e.g. cross-examine, discover shortage) down to simpler logical operators (e.g. find, compute, look-up). A designer or an automatic visualization system can then offload some of these logical operators onto the human perceptual system by generating a suitable external representation. Another alternative is to off-load some of the logical operations onto the computer system by using data computation functions (Figure C-7).

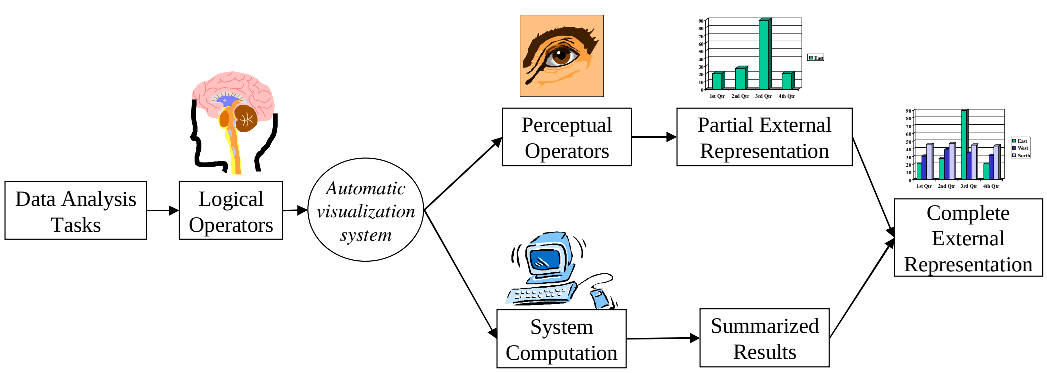


Figure C-7: This diagram shows how data analysis tasks can be broken down into perceptual operators (mapping transforms) and system computation operators (data transforms) and how these operators ultimately combine to produce a visualization design (external representation).

Our task model most closely resembles the task models presented by Casner and Senay et. Al [*Casner, 1991; Senay, 1994*]. Casner presented two logical task classes: *search* and *computation*. Search operators include *search*, *lookup*, *search and lookup*, as well as *lookup and verify*. Computation operators include *equal*, *less than*, *greater than*, *addition*, *difference*, *multiplication* and *quotient*. Senay and Ignatius presented three basic logical operations: *search*, *look-up*, and *compare* that are very similar to Casner's logical tasks. However, they extended the task specification to deal with groups of objects.

Our task model has four logical task operators similar to the ones in Casner and Senay et al.'s task language, *look-up, compute, find* and *compare*. We also expand our task language from previous approaches in three primary areas: 1) allowing task embeddings, 2) introducing precision declarations within the task, and 3) providing mechanisms for capturing task iterations.

C-3.1 Logical Tasks (Logical Operators)

Table C-1 summarizes our four simple logical operators: *lookup*, *compute*, *compare*, and *find,* and shows their input and output arguments. These tasks can be applied to single data objects and values or to sets of data objects and values. Tasks may also be applied to unknown arguments (indicated by a '?') if the user is unsure of an input argument during specification.

	Input arguments	Output arguments
Lookup	Data object(s), Data Attribute	Data value(s) or Data object(s)
Compute	Compute operator (addition, difference, etc), Data value(s), Data value(s).	Data value(s)
Find	Relationship, Data value(s), Data value(s).	Data object(s)
Compare	Compare operator (>, <, =, >=, <=), Data value(s), Data value(s).	Relationship

Table C-1: Tasks and their input and output arguments

By basing our task model on previous work in exploratory data analysis, we ensure that our operators are expressive of a relatively wide range of EDA goals. In the next sections we describe the four logical operators in Table C-1 in greater detail.

C-3.1.1 Lookup

The lookup task gets attribute information on one or more data concepts. It has two input arguments: the data concepts to perform the lookup on, and the data attribute of interest. For example, getting the *selling price* of a house is a lookup task. In this scenario, the lookup object is *house-1* and the lookup attribute is *selling-price*. The result of this lookup task is a data value indicating the selling price of *house-1*.

```
(Lookup      house-1, selling-price )
```

We can also look up attribute values for a set of data concepts. For example, in the task specification given below we have applied the lookup task to a set of three houses. In this case, the lookup task will return a set of three data values corresponding to the selling prices of the three input house concepts.

```
(Lookup     {house-1, house-2, house-3}, selling-price )
```

In addition to data values, we may also look up data relationships. For example, we may look up the *owner* relationship for a set of houses. This relationship points to a set of *person* concepts that represent the people who own the input houses. In this case the lookup task returns a set of data concepts rather than a set of data values as was in the previous two cases.

```
(Lookup     { house-1, house-2, house-3 }, owner  )
```

The lookup task is the most basic of all the logical operators. It is often embedded within one of the other logical tasks because they operate on data values, which must first be extracted or looked-up from data concepts.

C-3.1.2 Compute

The compute task generates new data values based on existing information. For example, the *difference* compute operator can be used to derive *gross-profit* values from *total-sales* and *total-cost*, as is shown below.

```
(Compute    Difference, (Lookup company-1, total-sales),
                        (Lookup company-1, total-cost) )
```

A compute task was also used in the airline-scheduling example presented in chapter IV-1 to calculate the *total downtime* in the *layover city*. In this case, the *addition* operator was applied to the *time before meeting* and the *time after meeting* data values.

We can also use computes to summarize a set of data values by determining their *minimum, maximum, mean,* or *median*. In this case the input arguments include the summarize operator and the set of values to be summarized. The output will be one data value (irrespective of the input data set size). For example we may use the specification below to determine the average *selling price* for a set of houses.

```
(Compute    Mean,
            (Lookup { house-1, house-2, house-3 },
                    selling-price ))
```

The type of computation operators we consider in our work include all the *data transform techniques* presented in chapter III.

C-3.1.3 Find

Find refers to the task of looking for a set of data concepts that fulfills certain data constraints. Some common constraints include upper and lower bound constraints (<, >), equality constraints (=) or both (<=,

>=). For example we may want to find all houses whose *selling price* exceeds 100k (lower bound constraint).

```
(Find   > ,
        (Lookup { house-1, house-2, house-3 } , selling-price ),
        100k )
```

Note that find is a logical task and it does not necessarily mean visual search. The find task *can* be solved through visual search by mapping the find attribute, (e.g. *selling price*) to a visual property (e.g. *x-position*). The find task can also be accomplished through data transform techniques (without the need for any visual search), in which case it would take the form of a data query. In the airline-scheduling example described earlier, there were several find tasks within the task sequence such as "finding all the flights originating from *Los Angeles*". To fulfill this task we can visually search for all such flights using Figure IV-1. Alternatively in Figure IV-3, the find task is performed by the system and the user is only presented with the flights that fulfill all of the find constraints.

Another common find task is to compare two sets of data values. For example, we may want to find all the months in which the price of rice exceeded the price of wheat. In this case we are comparing pairs of values for each month concept and returning all the concepts (i.e. months) whose value pair fulfills our constraint.

```
(Find   > ,
        (Lookup    { month-1, month-2, …, month-n } ,
                   rice-selling-price ),
        (Lookup    { month-1, month-2, …, month-n } ,
                   wheat-selling-price )  )
```

The find tasks that we have discussed so far are simple find tasks involving the comparison of value pairs. More advanced find tasks may look for complex group relationships among a set of values. For example, we may want to find all the months where the price of rice is increasing. In this case, we are looking for a set of values, related so that each value is greater than the previous one. The increasing trend is just one of many possible trends such as cycles, bell-curves, etc. Finding data trends is often difficult to achieve with data computation functions because the rules for capturing trend behavior are complex (i.e. low task specificity). For example, consider Figure C-8, which shows a linearly increasing trend with some outliers. A strict greater-than rule would not work here because the outlier points do not fulfill that rule. To capture this trend, we need a more complex rule allowing for outlier points and other similar exceptions. In addition, just specifying the trend as increasing may be insufficient. We may need to specify other trend properties such as its rate of increase, allowable error rate, number of outliers permitted, etc. Thus, there is a great deal involved in conveying to the system the exact user requirements for finding trends, thereby making them more appropriate for perceptual analysis (i.e. mapping techniques).

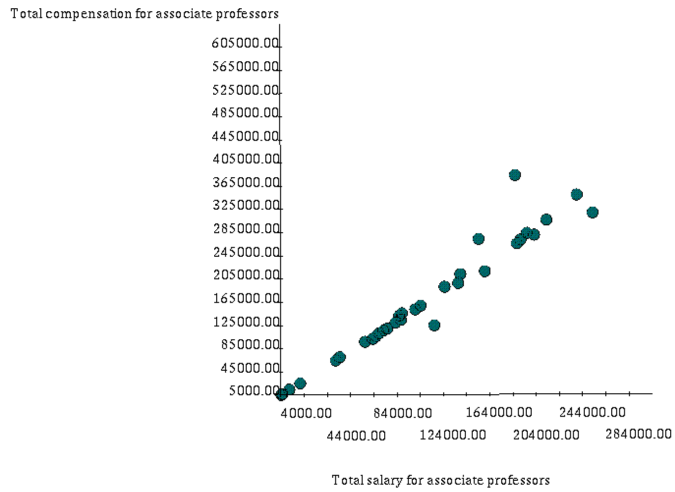


Figure C-8: Increasing trend

Another advanced find task is identifying group similarity, i.e. identifying a set of "similar" objects. The similarity attributes and the range of similarity, (i.e. what constitutes similar values) are often poorly defined, domain dependent, and cannot be easily captured in a task specification. The high articulatory distance therefore also makes these "find cluster" tasks more appropriate for perceptual processing.

In summary, advanced find tasks can be more effectively performed by mapping the source data to graphics because of the high articulatory load involved when using data computation functions. The role of data computation in this case is in directing the user's attention to objects that may be relevant. The system can make a best approximation of the find results and increase the saliency of those objects. Unlike the other data computation examples, however, we need to show all of the source data as well. This allows users to verify the system's results and to perceptually solve the task if the system approximation is incorrect or incomplete. For example in Figure C-8, the system may use data computation to approximate a line to the scatter points but still show the original data points so that users have the flexibility of verifying the system approximated line. Currently we do not deal with advanced find tasks in our work because of the high level of reasoning involved. Our work however does provide a basis that can be later expanded to deal with such tasks.

C-3.1.4 Comparison

Comparison tasks are used to determine relationships among data concepts. In this way, they are the complement of find tasks. In find tasks, the user knows the relationship to look for and is interested in the data concepts that participate in that relationship. On the other hand, in compare tasks, the user knows the set of data concepts to compare, but is unsure of the relationship(s) that exist among them. By comparing or analyzing different data attributes of the concepts, the user hopes to reveal the structure or relationships within the data.

We have found it useful to divide comparison tasks into three types: 1) comparing different data attributes within a single object set, 2) comparing the same data attribute across multiple different object sets, and 3) comparing different data attributes across different object sets. All three cases can be achieved with different combinations of logical *compare* and *lookup* operators.

1. Comparing different data attributes within a single object set

An example task of this type is comparing the *price-of-rice* and the *price-of-wheat* for particular *month* data concepts.

```
(Compare    (Lookup { month-1, month-2, …, month-n } ,
                    rice-selling-price ),
            (Lookup { month-1, month-2, …, month-n } ,
                    wheat-selling-price )  )
```

In this compare task, we may be interested in several different aspects of the data. For example, we may be interested in value pair comparisons, i.e. seeing if the *rice-selling-price* is greater than the *wheat-selling-price* over *n* months or in a particular month. We may discover that "in *April*, the price of rice was much lower than the price of wheat due to excess production". Alternatively, we may be interested in group or trend comparisons. For example, we may want to compare the trend of rice prices over time with the trend of wheat prices over time. In this case we are more interested in gestalt results such as, "when the price of rice is increasing, the price of wheat tends to increase as well". Group comparisons are often used in this context to determine correlations between different data value sets.

2. Comparing the same data attribute across multiple different object sets

The second type of compare task reasons about the same data attribute over different object sets. For example we may want to compare the size of houses in the *Shadyside* area with the size of houses in the *Squirrel Hill* area. In this case there are two object sets, one with a membership of all *Shadyside* houses and the other with a membership of all *Squirrel Hill* houses. In the task specification below, these object sets are determined using the *find* task. Both sets of objects are compared based on the same data attribute, namely *house-size*.

```
(Compare    (Lookup     (Find   =, Lookup( all-houses, neighborhood ),
                                Shadyside ),
                        house-size ),
            (Lookup     (Find   =, Lookup( all-houses, neighborhood ),
                                Squirrel Hill ),
                        house-size ) )
```

One of the main differences between *type-2* compare tasks and *type-1* compare tasks is that *type-2* compare tasks may be used to compare value sets of different sizes. On the other hand, *type-1* compare tasks always operate on value sets that are of the same size because the values are extracted from the same data concepts. In addition, *type-1* compare tasks may be applied to attributes with different properties, for example to *house-price* which is a *quantitative* attribute and to *date-sold* which is a *temporal* attribute. This is however not true of *type-2* comparisons that are by definition applied to the same data attribute.

3. <u>Comparing different data attributes across different object sets</u>

The final type of comparison task involves the comparison of different data attributes across different data sets based on a common shared attribute. For example we may want to determine whether the standard of living (measured based on household income) in a particular neighborhood affects the selling price of houses. In this case, we are comparing the *income* attribute of *household* data concepts with the *selling-price* attribute of *house* data concepts based on a common attribute, *neighborhood*.

Type-3 comparisons can be transformed into *type-1* comparisons with some data adjustment. Specifically, we must reorganize the data so that the "*data object*" or data record represents the common comparison attribute. In the example presented above, we want to structure the data so that each data concept represents a different *neighborhood* (e.g. *Shadyside, Squirrel Hill, Pt.Breeze*) rather than an individual house (e.g. *634 Maryland Av.*). Each *neighborhood* data concept will then have an *average-household-income* attribute and an *average-house-price* attribute. The *type-3* comparison task will then reduce to a *type-1* comparison task as is shown below. These data restructuring operations always occur through data transformation functions.

```
(Compare    (Lookup { neighborhood-1, …, neighborhood-n } ,
                    average-household-income ) ,
            (Lookup { neighborhood-1, …, neighborhood-n } ,
                    average-house-price ) )
```

C-3.2 Task Extensions

The task model presented above is adapted from previous characterizations of data analysis tasks. Unfortunately, this task model is not sufficient to capture all the information necessary to effectively reason about data computation techniques and mapping techniques. In order to facilitate decision making between these two design alternatives we enrich our data analysis task model from previous approaches in three primary areas: task embeddings, task iteration, and task accuracy or precision.

C-3.2.1 Task Embeddings

Previous task models only allowed for `flat' task specifications. Tasks are declared in isolation and task dependencies can only be captured with conditionals (e.g. Task C-1). Below (Task C-1), we show how Casner used if-then conditionals to specify that *downTime* should only be computed for the flight if the arrival time of the flight (*arrival1*) is before the start of the meeting (**beginMEETING**). This level of task dependency is insufficient for our work.

```
if ( arrivesBeforeMeeting? arrival1  beginMEETING)
 then (computeDownTime  beginMEETING  arrival1  DOWNTIME)
```

Task C-1: Subtask extracted from Casner's airline-scheduling task [*Casner, 1991*]

In order to reason about data computation operations, we must identify *intermediate tasks*, which are tasks used for generating intermediate results **-** useful in servicing the end goal but not useful in themselves. For example, consider the task specification below:

```
(Ratio  (Difference (Lookup (  { all-houses } , asking-price ),
                    (Lookup (  { all-houses } , selling-price) ),
        (Difference (Lookup (  { all-houses } , date-sold  ),
                    (Lookup (  { all-houses } , date-on-market) )
        )
```

In this task, we are trying to determine how house prices change the longer they stay on the market. The two *difference* tasks are intermediate tasks and they are related in this case because they are both embedded within the same *ratio* task. This is an important relationship to capture, and it has implications for both data computation as well as mapping decisions. Intermediate tasks, for example, are very appropriate for data computation operations because their results need not be shown to users and this allows for significant data simplification (refer to chapter IV-3.2). In addition, task embeddedness indicates a closer relationship among all the data attributes involved and we can use this information to constrain all the data attributes so that they are mapped in closer proximity (e.g. to the same graphical object).

A task can be embedded within another if its output argument class corresponds to the input argument class of the other task. For example, we can embed a *compute* task within other *compute* tasks because the *compute* operation both accepts and produces *data values*. Similarly, we could also embed a *find* task within a *lookup* task, or a *compare* task within a *find* task. It is important to note that task `embeddedness' is just one of many possible relationships that can exist among tasks. Tasks may also be related because one is conditioned upon another (as in Task C-1), because they are applied to the same object sets or because one task is generated based on the processing of another task. For example, if we want to compute the *ratio* task shown above with data computation, we must generate a lookup task so users may view the results of the computed *ratio* operation.

C-3.2.2 Task iteration

Previous work dealt with task repetitions by constructing loops around the task structures [*Casner, 1991*]. Unfortunately, these loop structures are not interpreted by the automatic visualization designer. It is crucial to capture task iteration information in our work because it has significant implications for making data computation and mapping decisions. For example as was shown in chapter IV-3.4, *all-to-all* task repetitions (i.e. processing each and every value in a set with all values in a second set) behave very differently from *value-pair* task repetitions (i.e. processing each value in a set with its corresponding value in a second set). The first is usually more appropriately solved with mapping designs whereas the second is more appropriately solved with data computation.

We deal with task repetitions by applying a task to the entire set of objects that we want it to iterate over and then specifying the type of task repetition that we desire. There are currently three classes of task repetition options: *value-pair* (*one-to-one*), *all-to-all*, and *previous-pair*. To illustrate each repetition class we apply them in turn to the two sets of values shown below.

{ A, B, C, D, E }

{ 1, 2, 3, 4, 5 }

Value-pair tasks are repeated over each pair of corresponding values in the two data value sets (i.e. they are applied to (A, 1) , (B, 2), (C, 3), (D, 4), and (E, 5)). *All-to-all* tasks on the other hand require each value in the first set to be paired with all values in the second set (i.e. they are applied to (A, 1), (A, 2), … , (A, 5), (B, 1), (B, 2), …, (B, 5), (C, 1), (C, 2), …). Finally, *previous-pair* operations ranks the input value set based on an ordering attribute and an ordering function. The task is then applied to each consecutive value pair in the ranked value set. For example, suppose the second value set is the ordering attribute for the first value set. Further suppose that the ordering function is the *greater-than* (>) operator. In this case the first value set will be ordered as follows { E, D, C, B, A } and the pairwise comparisons will be between (E, D), (D, C), (C, B), and (B, A). *Previous-pair* iterations are commonly applied to determine changes in a data attribute (e.g. *amount-of-sales*) with respect to changes in an independent ordered attribute (e.g. *time*). For example, we can use the *previous-pair* iteration to determine how the sales of a particular company are changing with time.

C-3.2.3 Task precision

Previous automatic visualization design systems use task accuracy as the most important effectiveness criteria for making data to graphical mapping choices. However, accuracy or precision can sometimes only be attained at the cost of significant cognitive load as we had shown in chapter IV-3.1. Depending on the task, it is not always necessary to attain high accuracy levels. For example, in some cases it may be sufficient that we know the net profit for month 1 is approximately 2 or 3 rather than exactly 1.6. If a system knows how important *accuracy* is to the user, it can make better choices between the many possible design alternatives.

For simplicity reasons we currently allow for three levels of accuracy: *high-accuracy*, *normal-accuracy*, and *fuzzy-accuracy*. High accuracy requires exact precision. For example in the difference task above, we want the differences calculated to all possible decimal values. Normal accuracy allows for an approximate comparison judgement. Relative values may be compared with no guarantees on the maximum amount of error. Finally *fuzzy-accuracy* refers to the case where we explicitly do not want accuracy. Here, we are specifying to the system that the task constraints are not absolute and that they should be relaxed. The default accuracy is *normal-accuracy*. There is a much wider range of accuracy levels than the ones that we have provided. However, we believe that our three accuracy levels covers many of the accuracy issues that arise when deciding between data computation and mapping operations. A deeper treatment of task *accuracy* is left for future work. The idea of attaching precision levels to tasks is not entirely new. In AutoVisual [*Feiner, 1990*], the degree of precision may be specified for the object selection task, which corresponds on some levels to our *find* task.

In the next section, we consider how variations in the tasks described in this section can influence design choices between data computation and mapping operations. This analysis is accompanied with designs generated by our prototype automatic visualization design system as well as discussions on why certain designs were ranked higher based on the dimensions described in chapter IV-2.

C-4 Exploring the Space of Data Techniques and Mapping Techniques

In this section, we describe how data computation vs. mapping design decisions are made in our automatic design system for the range of goals in our task model. We generate the possible range of goals by varying the task specification in three primary ways:

1. *Task structure variation*: In section C-3, we described four task classes in our framework, *lookup, compute, compare* and *find*. We may change the task structure by using different task classes or different task operators within each class. For example, for a *compute* task, we may be interested in the *addition, difference, multiplication,* or *quotient* operators. The task structure can also be changed by reordering the operators in the task or by changing their embedding structure.
2. *Task input argument variation*: In section C-3.1 we show that tasks may be applied to attributes, objects, or values. Task arguments can also be left unspecified ("?") to indicate that an argument is unknown or that there are many possible argument alternatives.
3. *Task data sets*: Tasks may also be applied to different data sets. The data set size and distribution of values can also affect design decisions between data and mapping techniques.

In the following examples, we systematically generate a set of visualization designs for each goal using our automatic visualization design system (AVID). We then analyze the designs using the metrics in chapter IV-2 and discuss how they are ranked and which design guidelines are used in their ranking. In order to keep this section at a reasonable length, however, we will not present all of the visualization designs that are generated by our system. Instead, we only show those designs that have interesting differences. Note that the examples addressed in this section are purposely chosen for their simplicity so that we may highlight important design decision points and show them in isolation. In section C-5, we explore a more realistic task of purchasing a car.

C-4.1 Task Structure Variation

In this section, we vary the task structure but keep the task arguments and the data sets constant. There are three ways in which the task structure may be varied: 1) task class or operator variation, 2) task expansion, and 3) embedding structure variation. We first present an example task, which we will subsequently alter based on these three structural variations. For each variation, we consider its effects on data computation and mapping design decisions.

C-4.1.1 University Example

We are considering attending a university for undergraduate study, but we are concerned about financing issues. Thus, we want to view the combined *tuition* and *room & board costs* of our set of candidate universities. The specification for this task is shown below.

```
(Compute    Addition,
            ( Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                     out-of-state-tuition ),
            (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory}},
                     room-&-board-costs )  )
```

Task C-2: Computing the total cost for attending a university

Figure C-9 shows a data computation design and a mapping design for solving Task C-2. The data computation design (Figure C-9a) shows the pre-computed total costs (*out-of-state-tuition* + *room-&-board-cost*) for each university in a single column of figures and the mapping design (Figure C-9b) shows each of the cost figures separately in two different columns. In the data computation design users only need to look up a single total cost figure for each university, however, with the mapping design users must look up two cost figures and then perform the addition task cognitively. In this case, the data computation design is clearly much more effective than the mapping design because while the mapping design requires *2n* perceptual lookups and *n* cognitive operations, the data computation design only requires *n* perceptual lookups. Our designer ranks Figure C-9a above Figure C-9b because it recognizes that the *addition* task cannot be effectively performed with text labels. Encoding the data with labels do not allow us off-load the addition task onto our perceptual system and as a result we must perform the task performed cognitively which has a high observational distance.

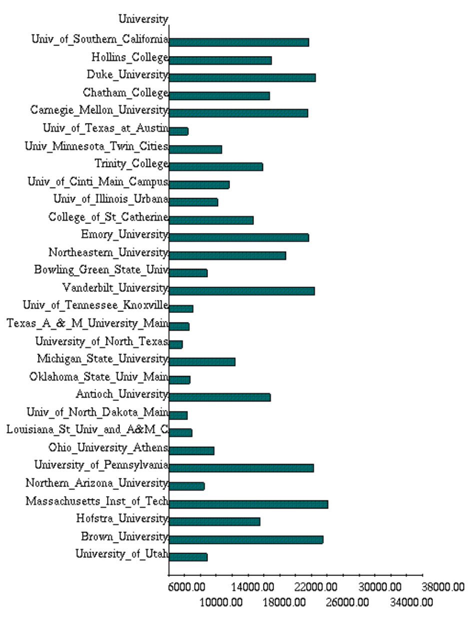
University	Total university cost [tuition + living]
Univ_of_Southern_California	23712.00
Hollins_College	18985.00
Duke_University	24540.00
Chatham_College	18730.00
Carnegie_Mellon_University	23590.00
Univ_of_Texas_at_Austin	8439.00
Univ_Minnesota_Twin_Cities	12693.00
Trinity_College	17842.00
Univ_of_Cinti_Main_Campus	13604.00
Univ_of_Illinois_Urbana	12134.00
College_of_St_Catherine	16664.00
Emory_University	23600.00
Northeastern_University	20805.00
Bowling_Green_State_Univ	10804.00
Vanderbilt_University	24390.00
Univ_of_Tennessee_Knoxville	9026.00
Texas_A_&_M_University_Main	8542.00
University_of_North_Texas	7683.00
Michigan_State_University	14392.00
Oklahoma_State_Univ_Main	8680.00
Antioch_University	18812.00
Univ_of_North_Dakota_Main	8337.00
Louisiana_St_Univ_and_A&M_C	8905.00
Ohio_University_Athens	11724.00
University_of_Pennsylvania	24290.00
Northern_Arizona_University	10474.00
Massachusetts_Inst_of_Tech	26075.00
Hofstra_University	17520.00
Brown_University	25454.00
University_of_Utah	10832.00

(a) Data computation design
***Out-of-state-tuition* and *room-&-board-costs* are pre-added with data computation operators and the results are shown as *text* in the table.**

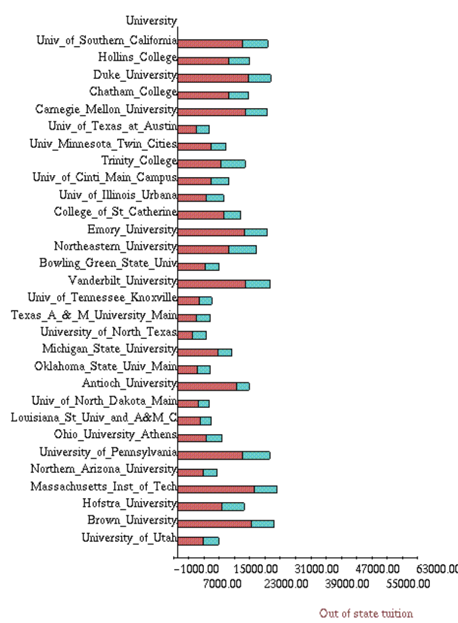
University	Room and board costs	Out of state tuition
Univ_of_Southern_California	6482.00	17230.00
Hollins_College	5515.00	13470.00
Duke_University	5950.00	18590.00
Chatham_College	5230.00	13500.00
Carnegie_Mellon_University	5690.00	17900.00
Univ_of_Texas_at_Austin	3309.00	5130.00
Univ_Minnesota_Twin_Cities	3744.00	8949.00
Trinity_College	6430.00	11412.00
Univ_of_Cinti_Main_Campus	4697.00	8907.00
Univ_of_Illinois_Urbana	4574.00	7560.00
College_of_St_Catherine	4440.00	12224.00
Emory_University	6000.00	17600.00
Northeastern_University	7425.00	13380.00
Bowling_Green_State_Univ	3352.00	7452.00
Vanderbilt_University	6525.00	17865.00
Univ_of_Tennessee_Knoxville	3262.00	5764.00
Texas_A_&_M_University_Main	3412.00	5130.00
University_of_North_Texas	3579.00	4104.00
Michigan_State_University	3734.00	10658.00
Oklahoma_State_Univ_Main	3344.00	5336.00
Antioch_University	3336.00	15476.00
Univ_of_North_Dakota_Main	2703.00	5634.00
Louisiana_St_Univ_and_A&M_C	2980.00	5925.00
Ohio_University_Athens	4095.00	7629.00
University_of_Pennsylvania	7270.00	17020.00
Northern_Arizona_University	3728.00	6746.00
Massachusetts_Inst_of_Tech	5975.00	20100.00
Hofstra_University	5920.00	11600.00
Brown_University	5926.00	19528.00
University_of_Utah	3975.00	6857.00

(b) Mapping design
***Room-&-board-costs* is shown as *text* in the left table and *out-of-state-tuition* is shown as *text* in the right chart. This design solution is very inefficient because users must perform the addition task cognitively.**

Figure C-9: Design solutions for total cost computation task (Task C-2).



(a) Data computation design
***Out-of-state-tuition* and *room-&-board-costs* are pre-added with data computation operators and the results are encoded on the *x-axis* of the bar-chart.**



(b) Mapping design
***Out-of-state-tuition* is mapped to the *x-length* of the red bars and *room-&-board-costs* is mapped to the *x-length* of the blue bars. Total cost can be easily determined perceptually by looking at the combined lengths of the stacked bars.**

Figure C-10: Encoding the same data as Figure C-9 but with bars instead of text

A more effective mapping design, however, would offload the addition cognitive operation onto the user's perceptual system by mapping the two cost figures to bar lengths instead of text. Figure C-10 shows the same data as Figure C-9 except that we use bars instead of text to encode the data. The data computation design shows the total cost on the *x-axis* and the mapping design shows *out-of-state-tuition-cost* and *room-&-board-cost* on the *x-axis* using two stacked bars. In this example, the mapping design is significantly improved over the textual design in Figure C-9b. In fact both data and mapping designs in Figure C-10 take the same number of perceptual lookups. I.e. Figure C-10b and Figure C-10a are ranked at the same level by our automatic system. This example shows that the difference in effectiveness between a mapping and data computation design can change significantly depending on the graphical properties used to encode the data. In the subsequent examples, we will only compare data computation and mapping designs that use the best possible graphical properties.

Our automatic visualization design system however must be able to reason about both graphical property effectiveness together with functional effectiveness (i.e. data or mapping technique effectiveness) and this can sometimes be difficult. For example it can be difficult to decide whether the textual data computation design (Figure C-9a) is better than the bar mapping design (Figure C-10b) because both designs utilize different graphical property encodings as well as different design functions (i.e. data computation vs. mapping functions). The more appropriate design here can be based on task requirements, user preferences, domain conventions, etc. Some of these issues (e.g. task requirements) are taken into account in our system. The beauty of automatic visualization design however, is that it is a cooperative process between the user and the system. The advantage of such systems as a design tool is that it can quickly generate both Figure C-9a and Figure C-10b and show them to the user who can then decide between similarly effective designs based on their preferences.

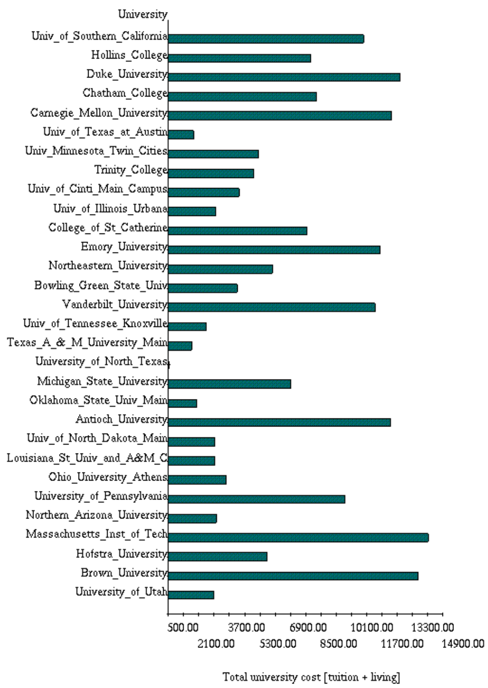
C-4.1.2 Task Operator Variation

Now suppose that instead of computing the total cost for each university, we also want to determine whether the university charges (i.e. tuition) are in line with the living costs in the area. Specifically, we want to view the differences between *tuition-cost* and *room-&-board-cost* for each university.

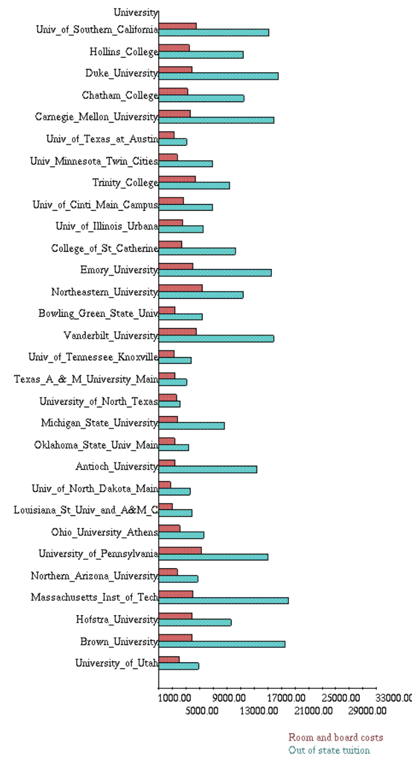
```
(Compute    Difference,
            (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                    out-of-state-tuition),
            (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                    room-&-board-costs)  )
```

Task C-3: Change in task operator from *addition* to *difference*

In this task, the observational distance for the data computation design (Figure C-11a) consists of *n* perceptual bar length lookups. The mapping design (Figure C-11b) however requires a comparison between the *tuition-cost* bar and the *room-&-board-cost* bar, which results in *2n* perceptual lookups. Thus unlike the previous *addition* example, the data computation design is ranked above the mapping design in this example. This is because the graphical language available for showing the *difference* task (bar pairs) is less effective than the graphical language available for the addition task (stacked bars) (*availability of perceptual operators* guideline in chapter IV-3.3).



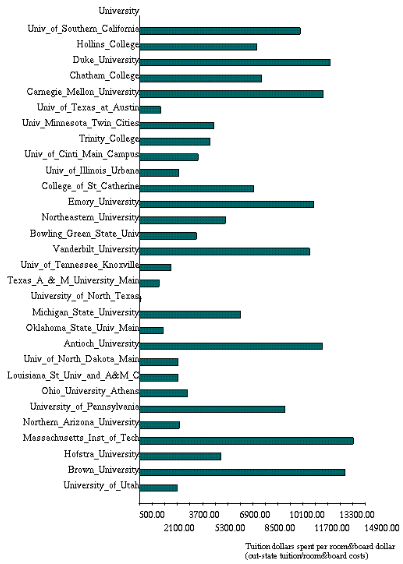
(a) Data computation design
The difference between *out-of-state-tuition-costs* and *room-&-board-costs* is pre-computed and encoded on the *x-axis* of the bar chart



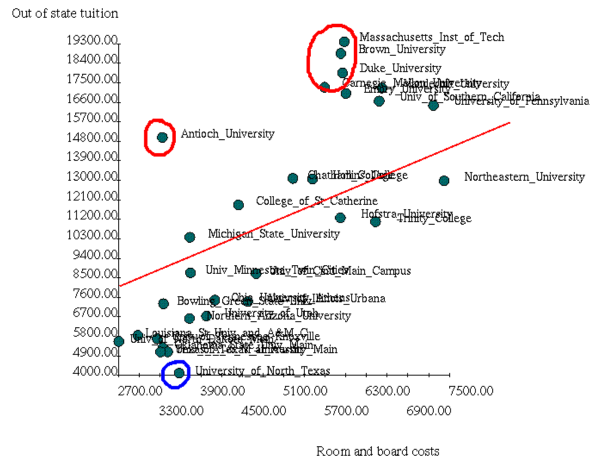
(b) Mapping design
***Room-&-board-costs* is mapped to the *x-length* of the red bars and *out-of-state-tuition* is mapped to the *x-length* of the blue bars. The cost differences can be perceptually determined by looking at the difference in lengths between each pair of bars. This perceptual design is less effective than the data computation design because the difference estimation operation is non-trivial unlike the addition operation in** Figure C-10(**b**).

Figure C-11: Computing the difference between *out-of-state-tuition* and *room-&-board-costs*

Now suppose that instead of computing the difference between *out-of-state-tuition-cost* and *room-&-board-cost*, we wanted to compute their ratio instead. Specifically, we want to see for each dollar spent on *room-&-board* how many dollars are spent on *tuition*. Figure C-12 shows the design alternatives for this task. In the data computation design the ratio values are pre-computed and mapped to the *x-lengths* of the bars. In the mapping design, the ratio values must be perceptual derived from the position of the marks which shows both *out-of-state-tuition-cost* on the *y-axis* and *room-&-board-cost* on the *x-axis*. To express the *ratio* task in the mapping design, our designer constrained both *cost* attributes to be mapped to *conjoint* properties on the same graphical object.



(a) Data computation design
The ratio of *out-of-state-tuition* dollars spent per *room-&-board-costs* dollar is pre-computed and shown on the *x-axis* of the bar chart. *University of North Texas* has a low ratio value while *MIT, Brown, Duke, CMU,* and *Antioch Universities* have high ratio values (i.e. tuition costs are high wrt. living costs).



(b) Mapping design
***Out-of-state-tuition* is mapped to the *y-axis* and *room-&-board-costs* is mapped to the *x-axis*. To estimate the ratio values between *out-of-state-tuition* and *room-&-board-costs* we look at the distance of the points from the average ratio line (shown in red). The points to the top that are circled red (*MIT, Brown, Duke, CMU, Antioch*) are the ones with high ratio values while the points to the bottom are the ones with below average ratio values (*University of North Texas*).**

Figure C-12: Computing the ratio between *out-of-state-tuition-costs* and *room-&-board-costs*

The mapping design (Figure C-12b) is not very effective if we want to determine exact ratio values. However, our intention is to find outlier universities, i.e. universities with either inordinately high or low *tuition-cost/room-&-board-cost* ratios then Figure C-12b is a useful solution. For this task, high accuracy is not required because we are only interested in relative ratio values. In fact, the perceptual load for both the data computation and mapping designs are similar (ignoring the occlusion problem). For example in both designs we can pre-attentively see that *Antioch University, MIT, Brown, Duke,* and *CMU* (circled in *red*) have unusually high *tuition/room-&-board-cost* ratio while *University of North Texas* (circled in *blue*) has an unusually low *tuition/room-&-board-cost* ratio. Our automatic designer ranked the data computation design slightly above the mapping design because of the greater perceptual complexity (i.e. graphical density) in the mapping design.

C-4.1.3 Task Expansion

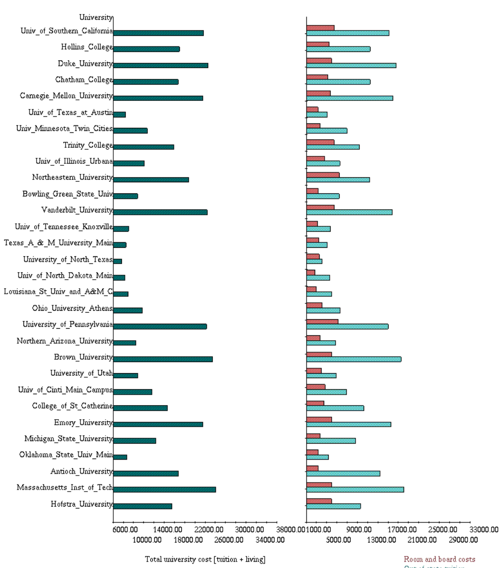
In addition to being interested in the total cost for attending each university, suppose we are also interested in the individual *tuition-cost* and *room-&-board-cost* figures.

```
(Compute    Addition,
            (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                    out-of-state-tuition),
            (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                    room-&-board-costs)   )
(Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
             out-of-state-tuition),
(Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
             room-&-board-costs)
```

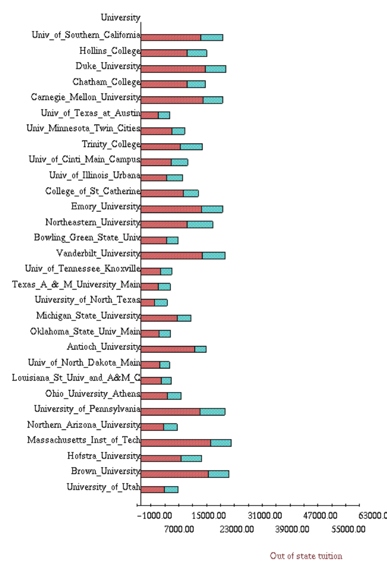
Task C-4: Expanded university total-cost task

Specifically, we want to see if higher cost universities are a result of additional *tuition-cost* or *room-&-board costs*. Thus, we have expanded the task specification from Task C-2 with additional *lookup* tasks.

Unlike the original task (Task C-2), our designer picked the mapping design (Figure C-13b) as the most effective for this expanded task. The mapping design is chosen based on the *task variation on attribute* design guideline in chapter IV-3.5 because it allows us to solve the *addition* and subsequent *look up* tasks by using the same set of graphical objects. All the task information is presented in a space efficient manner, and the observational distance for performing the *addition* task in the mapping design is negligible because *stacked bars* are very effective for addressing *addition* operations. On the other hand, the data computation design (Figure C-13a) has a greater number of graphical objects (two separate charts and three bars) because we must show both the computed *total cost* results as well as the original *tuition-cost* and *room-&-board-cost* attribute values.



(a) Data computation design
In this example we want to see both total costs (*out-of-state-tuition + room-&-board-costs*) as well as the individual costs. In this design total costs are pre-computed and shown on the *x-axis* of the left chart. The individual cost values are shown as blue and red bars respectively in the right chart.



(b) Mapping design
***Out-of-state-tuition* is mapped to the *x-length* of the red bar and *room-&-board-costs* is mapped to the *x-length* of the blue bar. This design is very effective because it allows us to easily derive the total cost values by looking at the combined lengths of the stacked bars. However, the two individual cost values are also readily accessible.**

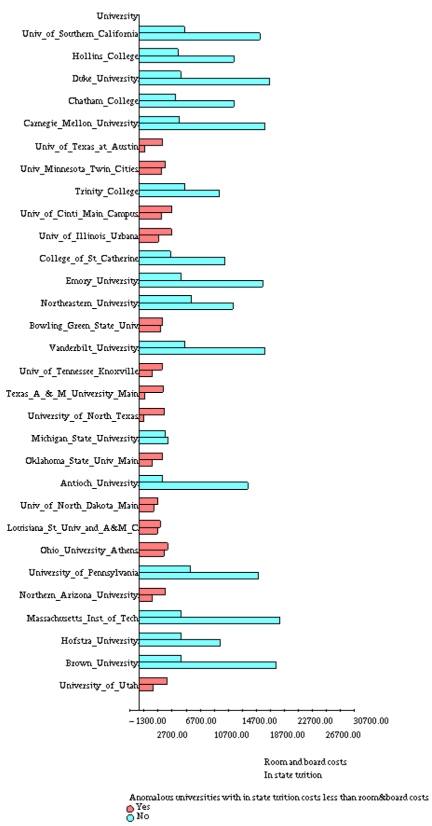
Figure C-13: Design solutions for expanded total cost computation task (Task C-4)

Now, instead of computing the total cost for each university, suppose we are curious to see if there are any universities with *tuition-cost* less than *room-&-board-cost*.

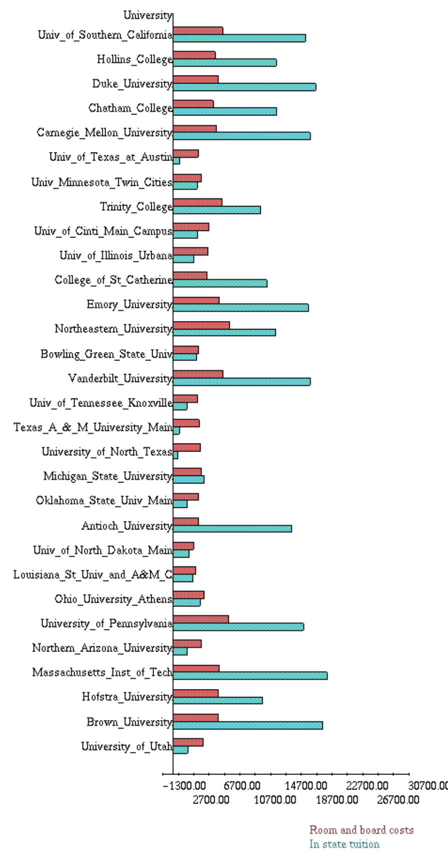
```
(Find   <,
        (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                in-state-tuition),
        (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                room-&-board-costs)  )
(Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
        in-state-tuition)
(Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
        room-&-board-costs)
```

Task C-5: Expanded tuition-cost and room-&-board-cost find task

Not surprisingly there are no universities with *out-of-state-tuition-cost* less than *room-&-board-cost* however there are some universities with *in-state-tuition-cost* less than *room-&-board-cost* as is shown in Figure C-14.



(a) Data computation design
In this design universities with *in-state-tuition* less then *room-&-board-costs* are shown in red. Not surprisingly these tend to be state universities. The individual *room-&-board-costs* (top bar) and *in-state-tuition* (bottom bar) values are encoded on the *x-axis*. Unlike Figure C-13**a, the computed results can be more effectively integrated into the design here, making it more effective compared to the mapping design.**



(b) Mapping design
***Room-&-board-costs* is mapped to the *x-length* of the red bars and *in-state-tuition* is mapped to the *x-length* of the blue bars. To find the universities with greater *room-&-board-costs* compared to *in-state-tuition* we must compare the lengths of each bar pair which is a fairly time consuming operation.**

Figure C-14: Design solutions for expanded find task (Task C-5)

Unlike the previous expanded task (Task C-4), the most effective design here is the data computation design (Figure C-14a). In Figure C-14a, the *find* task is performed with data computation functions and its results are shown using *color.* The blue hue bars show all the universities with *in-state-tuition-cost* less than *room-&-board-costs*. This design is effective because the *find* results can be introduced into the graphic without adding much visual complexity and without cluttering up the display space (in contrast to the data computation design in Figure C-13a). In addition, perceptual load has been reduced significantly because the find results can be read in a single pre-attentive perceptual operation (i.e. perceptual load = p). The mapping design, however, requires users to scan through all bars to visually search for all instances where *tuition-cost* is less than *room-&-board-cost* (i.e. perceptual load = np where n is the number of universities). The data computation design also presents the find results with great accuracy, leaving no room for perceptual errors.

This example illustrates the interaction between two different design guidelines, the *availability-of-perceptual-operator* guideline and the *task-variation* guideline. In this example, the available perceptual operator for performing the *find* task is not as effective as using data computation operations (i.e. preference for data computation design). However, there is *task-variation* on the *find* task and this usually results in greater perceptual complexity for the data computation design (i.e. preference for mapping design). While perceptual complexity in Figure C-14a is larger than in Figure C-14b because of the additional *color* encoding, the added complexity here is relatively small. Only a single graphical property is added here compared to the additional graphical object and region added in Figure C-13a. Consequently, the perceptual savings enabled for the *find* task through data computation outweighs the small added complexity from the use of *color*. Our automatic designer deals with interacting guidelines by adding costs to each design alternative based on the guideline being violated, and the depth of the violation. Specifically, a higher cost is added to Figure C-13a for the *task-variation* guideline violation compared to Figure C-14a, because in the former case, the added perceptual complexity is more significant.

C-4.1.4 Embedding Structure Variation

Since universities with *tuition-cost* less than the *room-&-board-cost* are such anomalies, we might want to examine the individual *tuition-cost* figures for only those universities to see which university has a highest and lowest costs within the set. This task is very similar to Task C-5 except that the embedding structure is changed. Specifically, the *find* task is now embedded within the two lookup tasks to indicate that we **only** want to lookup the cost values for the anomalous universities.

```
(Lookup (Find   <,
                (Lookup { … }, in-state-tuition),
                (Lookup { … }, room-&-board-costs)     )
        in-state-tuition)
```

Task C-6: Change in task embedding structure from Task C-5
(Note that we represent the university set with { … } here to make the task specification easier to read. However, the data set used is the same as all previous examples in this section).

In this case, the most effective design (Figure C-15) performs the find task with data computation, and only shows *in-state-tuition-cost* figures for those universities with greater *room-&-board-cost* compared to *in-state-tuition-cost*. All other universities are culled from the display based on our *intermediate task* design guideline (chapter IV-3.2) which calls for all intermediate results to be hidden in order to reduce display complexity.

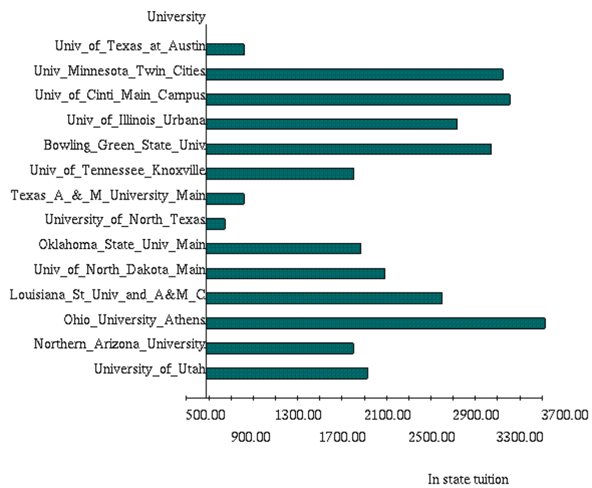


Figure C-15: Data computation design for Task C-6
In this design the automatic design system is able to utilize the embedding structure of the task to filter the design so that only those universities with *in-state-tuition* less than *room-&-board-costs* are shown. This cuts down on the number of elements that have to be shown significantly, producing a more easily interpretable display. *In-state-tuition* values are then shown on the *x-axis* of the bar chart.

We can also solve Task C-6 using the designs in Figure C-14 (i.e. our automatic designer will also generate those designs as alternatives for Task C-6). However, they are ranked lower because they both have greater perceptual clutter (i.e. a greater number of graphical objects), require more display space, and require more perceptual processing compared to Figure C-15. In particular, Figure C-14a requires an additional perceptual operation for getting all the purple bars, and Figure C-14b requires *n* additional perceptual operations for comparing each bar pair (where *n* = number of universities). In this example, our automatic system is able to exploit the additional task information captured by the embedding structure of the task to generate a more effective design than what was possible in Task C-5 which has a flat, unembedded structure.

C-4.2 Task Argument Variation

As was shown in section C-3.1, the data analysis tasks we consider in this thesis can be applied to three argument types: *values, attributes,* and *objects*. Tasks may also accept additional input arguments such as *iteration type* and *task type* as was described in section C-3.2. In this section, we consider how different input task arguments may result in different data and mapping choices. Specifically, there are three classes of input argument variations:

1. *Use of a data attribute value set vs. use of constants*: Tasks may be applied to single values or to sets of values. When tasks are applied to sets of values, we may iterate over these values in a variety of ways (*value-pair, all-to-all, previous-pair*).

2. *Use of known vs. unknown arguments*: According to the task model described in section C-3.1, task arguments may be specified or left as an unknown (?). In the latter case, we are indicating that there are many possible task arguments and the proper one(s) for the task can only be determined during exploration.

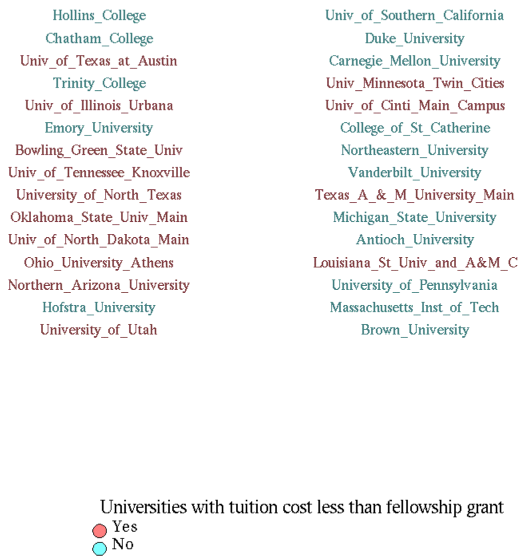
3. *Use of different classes of attribute types*: Data attributes are associated with a set of characterizations that describe their *set ordering* (quantitative, nominal, or ordinal), *domain of membership* (time, space, temperature, or mass), and relational structure [*Roth, 1990*; *Zhou, 1996*]. Previous work on automatic visualization has shown that these characterizations have significant impact on the choices made in mapping data to graphics. Here we consider some of the implications that these characterizations have on data vs. mapping decisions.

C-4.2.1 Use of constants vs. data attribute value sets

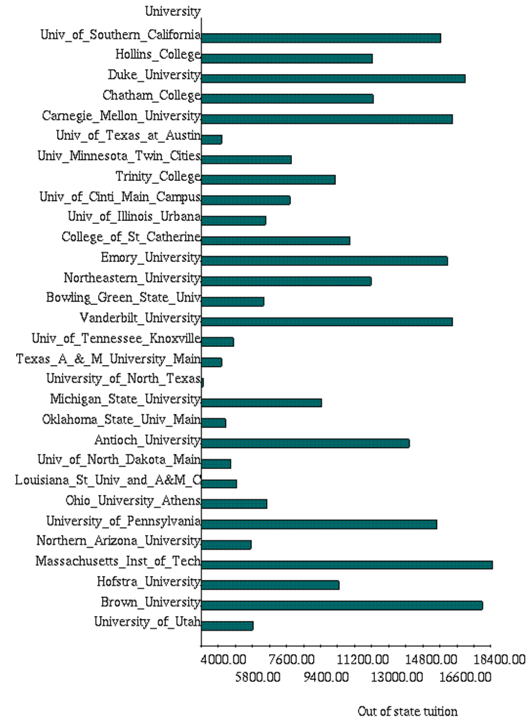
In this section, we continue our university-financing example. Suppose that we have received a fellowship that covers up to 10k worth of tuition cost. We might only want to attend universities with tuition costs below this figure so that we need not worry about paying any additional fees. The task specification is similar to Task C-5 except that instead of comparing two sets of attribute values (i.e. *tuition-cost* and *room-&-board-cost*), here we are comparing an attribute value set (*tuition-cost*) with a constant value (*10k*).

```
(Find   <,
        (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                out-of-state-tuition ),
        10k   )
```

Task C-7: Find task with a constant argument



(a) Data computation design
Universities with *out-of-state-tuition* less than 10k are shown in red.



(b) Mapping design
***Out-of-state-tuition* is mapped to the *x-axis*. To find universities with *out-of-state-tuition* less than 10k we must look on the *x-axis* for the 10k point and then compare the bar lengths to this point. Even though this operation is more difficult than finding the red labels in design (a), we still have access to the actual *tuition* figures here, so we may compute the amount saved from the 10k mark. This information is lost in design (a).**

Figure C-16: Design solutions for find task in Task C-7

The data computation design (Figure C-16a) pre-computes the *find* task and maps the results to *hue*. The *red* hued universities are the universities that pass the *find* task and the *blue* hued universities are the ones that did not pass the *find* task. The mapping design (Figure C-16b) maps the *out-of-state-tuition-cost* values to a set of *bar-lengths* and leaves it to the user to pick the bars that exceed *10k* in value. The mapping design for this example (Figure C-16b) is much more effective than the mapping design for Task C-5 (Figure C-14b). This is because the perceptual operation for finding all bars less than a *constant-value-line* (Figure C-16b) is pre-attentive while the operation for finding all bar pairs with *tuition-cost* less than *room-&-board-cost* (Figure C-14b) is not. Thus, our automatic designer assigns a lower cost to the *mapping-find* design when one of the *find* task arguments is a constant value.

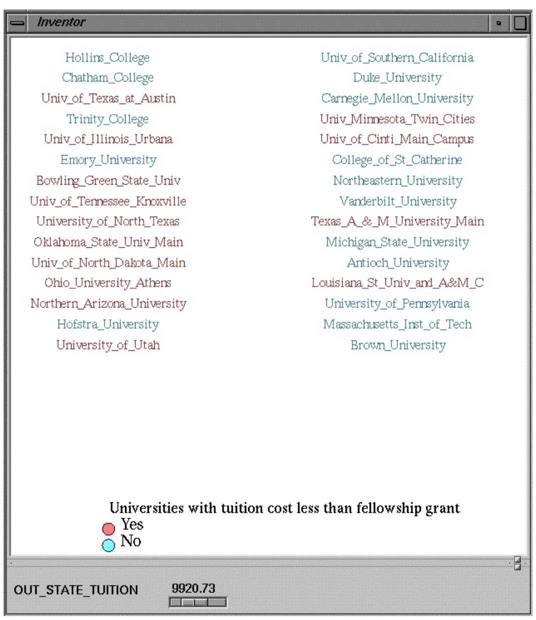
In this task the observational distance for both the data computation and mapping designs are comparable. Both solutions only require a single pre-attentive perceptual operation to find all universities that have greater *tuition-cost* than *10k*. However, our designer still ranks the data computation design ahead of the mapping design because it is more difficult to determine the *find* results with high *accuracy* using the mapping design. This is consistent with our *accuracy* heuristic in chapter IV-3.1.

C-4.2.2 Use of known vs. unknown arguments

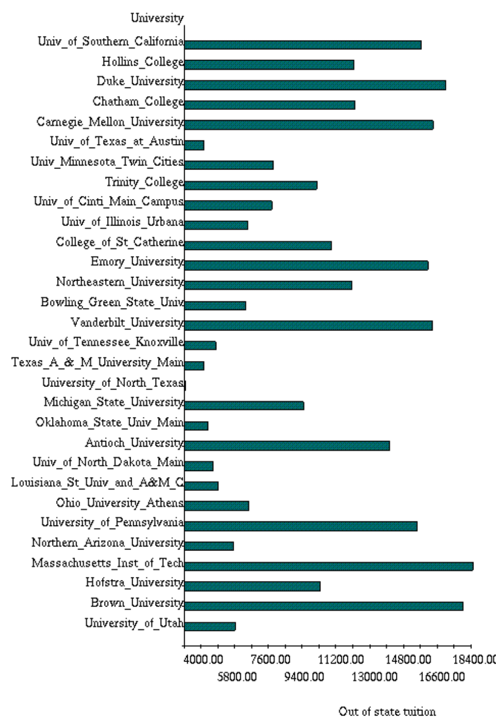
Now, suppose we are unsure of the amount covered by our fellowship because the figure is dependent on our expected *SAT* and *ACT* test scores. One way to perform this task is to leave the fellowship constant as an unknown ("?") in the specification.

```
(Find    <,
         (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                 out-of-state-tuition),
         ? )
(Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
         out-of-state-tuition)
```

Task C-8: Find task with an unknown task argument



(a) Data computation design
This design is similar to Figure C-16a except that here we must enter in the *out-of-state-tuition* threshold value we are interested in through a *slider* input device. This increases the overall load placed upon the user, making the mapping design alternative more attractive.



(b) Mapping design
Exactly the same design can be used here as was used in Figure C-16b even though we do not know the fellowship value in this case (i.e. task specificity is lower). This is because all of the data is presented to the user, thus there is greater flexibility in the range and types of questions that may be answered with this display.

Figure C-17: Design alternatives for Task C-8

For tasks with unknown arguments, a mapping design is generally preferred over a data computed one because the task specificity is low (*task-specificity* guideline in chapter IV-3.6) and the articulatory distance is higher if we use data computation. The mapping design in Figure C-17b shows the *tuition-cost* figures as bar lengths and leaves it up to the user to determine the fellowship threshold line perceptually. The observational distance for this design is *x2p* to perform the *find* task, where *x* represents the number of times the *find* task is repeated, and *p* represents the load of a simple perceptual operation. Each time the *find* task is repeated, a total of *2p* perceptual load is required; *1p* load to determine the threshold line on the *x-axis* and another *1p* to pre-attentively identify all bars below or above that threshold.

The data computation design (Figure C-17a) provides a *slider* input device that allows users to manually enter in different fellowship values. Universities with *tuition-cost* below this input value are then highlighted pink. In this case, the perceptual load is slightly smaller. Only a single perceptual operation is needed for each task iteration to pre-attentively identify all the pink university names (i.e. load = *xp* where *x* represents the number of times the *find* task is repeated). However, in addition to the perceptual load there is also an articulatory load. Each time the *find* task is repeated, the user must enter a new input value through the slider. The load for a single input is *2(m+k)* i.e. two mouse moves (*2m*) with one mouse click and one mouse release (*2k*). Thus, the total articulatory load is *2x(m+k)*. This articulatory load outweighs the additional perceptual load needed in the mapping design thus our designer ranks the mapping design above the data computation design.

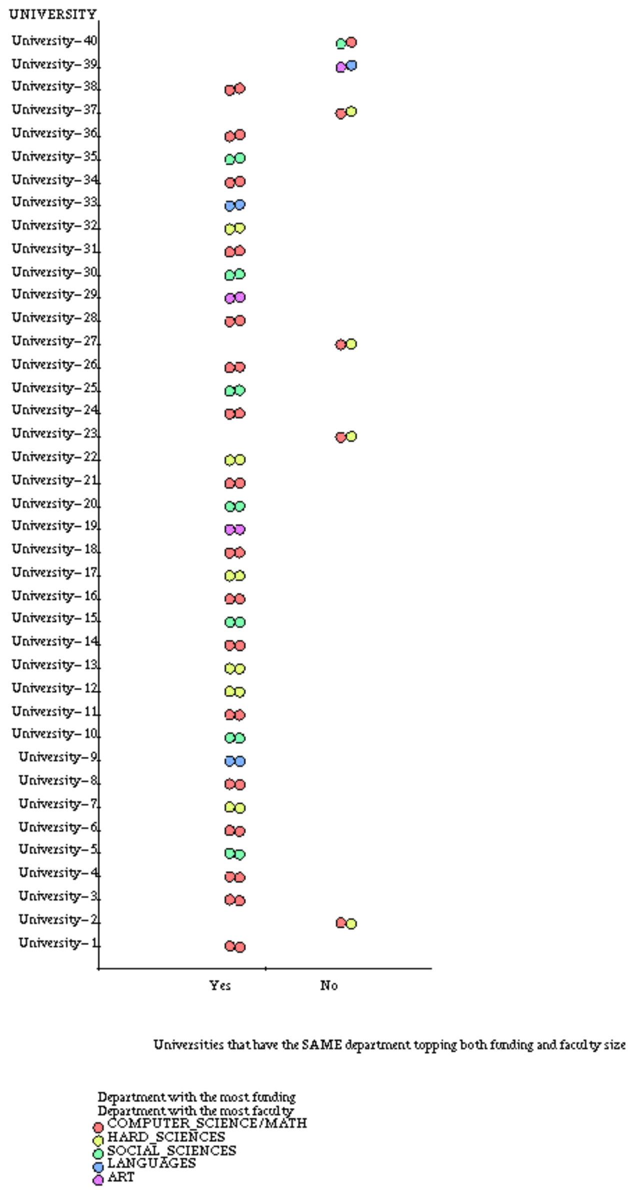
We want to point out however that accuracy is assumed to be less important for the *find* task here. When high accuracy is required, the data computation design becomes more effective because it can be difficult to get highly accurate find results from the mapping design. In particular, it may be difficult to determine which point on the *x-axis* corresponds to a desired fellowship figure and it may be difficult to perceptually project a straight line upwards from the *x-axis* to accurately process the *bar-lengths* especially for *tuition-cost* figures that are close in value to the fellowship. This is consistent with our *accuracy* design guideline in chapter IV-3.1.

C-4.2.3 Change in attribute or value type

In addition to financial costs constraints, suppose we are also only interested in finding out the departments in each school which are most prosperous (i.e. has the most funding as well as the most faculty). Of particular interest are the departments that top both the categories. Analyzing this data will help us determine which areas of study are the most popular and financially rewarding. Note that this task is identical to Task C-5 except that instead of comparing two quantitative attributes (*tuition-cost* and *room-&-board-cost*) we are comparing two nominal attributes (*department-with-most-funding* and *department-with-most-faculty*).

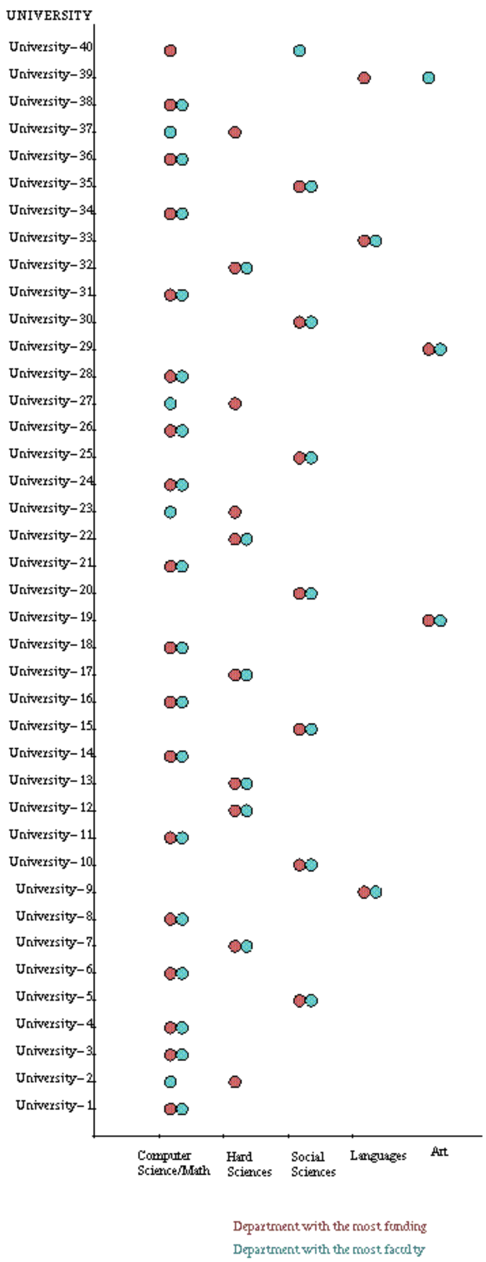
```
(Find   =,
        (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
                department-with-most-funding ),
        (Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory} ,
                department-with-most-faculty )     )
(Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory},
        department-with-most-funding ),
(Lookup {Carnegie_Mellon_University,MIT, Duke, …, Emory} ,
        department-with-most-faculty )
```

Task C-9: Finding the most prosperous department in each university based on funding and faculty size

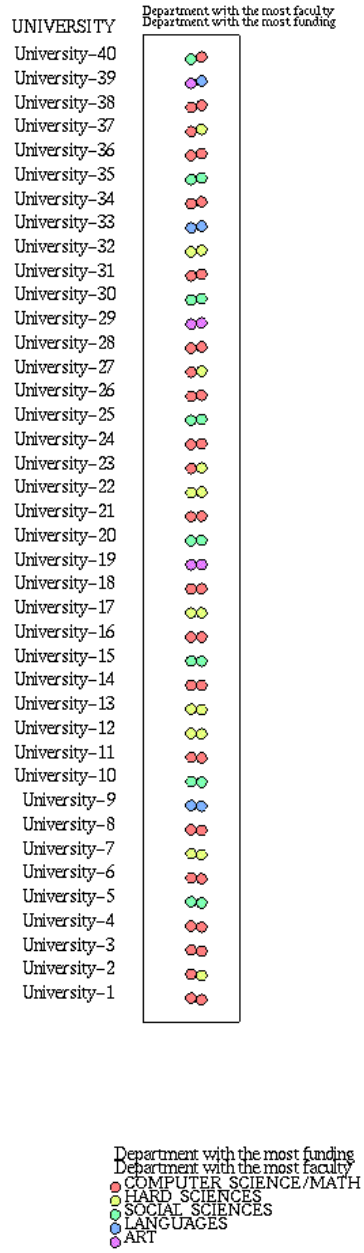


(a) Data computation design

Each university (*y-axis*) is associated with two marks, one representing the department with the most funding (left) and the other representing the department with the most faculty (right). The five main possible departments are represented using *hue*. In addition, universities with the same department topping both funding and faculty are pre-computed and shown in the left column.



(b) Mapping design with *position*
***Universities* are mapped to the *y-axis* and the five different departments are mapped to the *x-axis*. The department with the most funding is indicated with a red mark and the department with the most faculty is indicated with a blue mark. Universities with the same top department would have the two red/blue marks together. Finding such universities can be performed pre-attentively or semi pre-attentively with this display.**



(c) Mapping design with *hue*
***Universities* are mapped to the *y-axis* and the different department types are mapped to *hue*. Universities that have the same top faculty and funding departments are those with pair-marks that have the same color. This design is less effective than design (b) because there is a lot of *hue* noise here making it more difficult to identify identically colored marks.**

Figure C-18: Finding universities where the department with the most faculty is also the department with the most funding. The two mapping alternatives are less effective compared to the data computation solution because in both the mapping designs the *find* task may not be fully pre-attentive while in the data computation design it is very easy to identify the universities with the same top faculty and funding departments. In addition, this information is very well integrated into the design without increasing complexity by much.

In the data computation design, the universities that pass the *find* task are shown on the left and the universities that do not pass the *find* task are shown on the right. Each university is represented by a cluster of two marks, with the first mark representing the department with the most funding and the second mark representing the department with the most faculty. In this example, the data computation design is ranked higher than the mapping solutions because it allows pre-attentive performance of the *find* task, and the additional *find* results can be integrated with the two *lookups* very well, without significantly increasing graphical complexity.

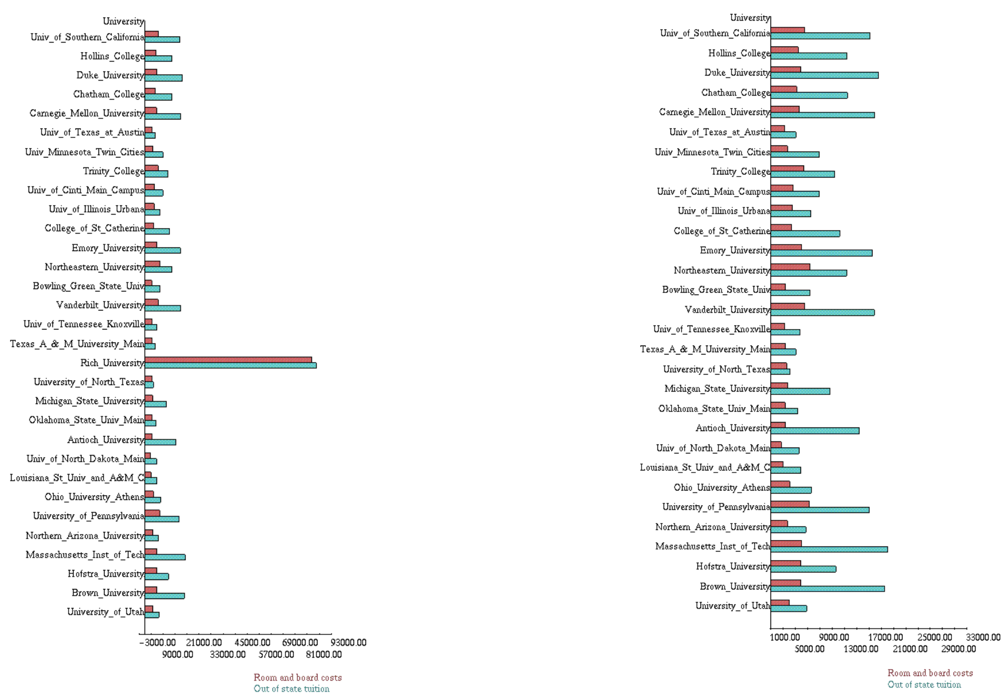
On the other hand, the mapping solution does not ensure pre-attentive performance of the *find* task. Figure C-18b shows a mapping design where the *department* attribute is mapped to *x-position* and the department with the most funding is represented by a *pink mark* while the department with the most faculty is represented by a *blue mark*. We can find the universities that fulfill our task constraint by looking for clusters of two marks (i.e. a two mark cluster indicates a university that has the same department topping both faculty and funding). Figure C-18c shows a different mapping design alternative where the *department* information is mapped to *hue* instead of *x-position*. The left *marks* in Figure C-18c represent the departments with the most funding and the right *marks* represent the departments with the most faculty. Unlike the data computation solution (Figure C-18a), pre-attentive perception is not possible here because there are too many colors in the display and the noise from these colors make it difficult to pre-attentively identify rows that have the same colored *marks*.

The important issue here, however, is that the mapping design for tasks with *nominal* attributes is more effective than the mapping design for tasks with *quantitative* attributes (e.g. in Task C-5). In the former case, the departments can be compared with 100% accuracy and the task can sometimes be accomplished pre-attentively. In the latter case, the mapping design is much less accurate and cannot be accomplished pre-attentively. Thus a lower cost should be assigned to mapping designs for tasks with *nominal* or *ordinal* input arguments compared to tasks with *quantitative* input arguments.

Nominal and ordinal data attributes are probably the easiest to process perceptually especially when they are well-bounded (i.e. they represent only a small number of different values or have a value membership set that is relatively small). Attribute values that are discrete (i.e. not continuous) and well-bounded can be to be encoded with a wide range of pre-attentive graphical properties that are maximally differentiated in value, which makes perceptual processing significantly easier and more accurate. Discrete attributes are less effective because although they are non-continuous, they are unbounded. Quantitative values are the most difficult to process perceptually because they are continuous and unbounded. This makes it difficult to accurately translate the perceptual values back into data values[1].

C-4.3 Data Set Variation

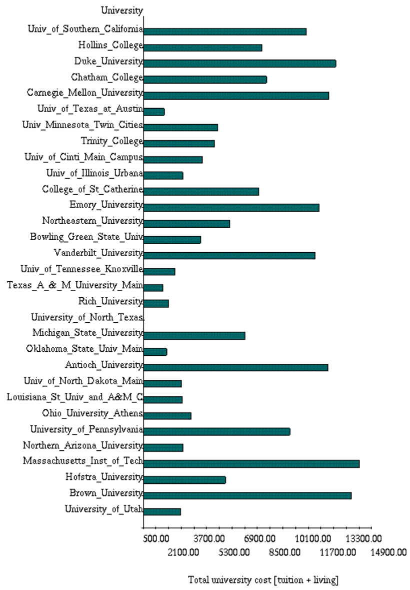
Variation in the data set mainly causes readability problems (e.g. *occlusion, display density, dwarfing*) which increases both the expressive and observational distances of a graphic. Some of these problems may be avoided using data computation designs that can hide and summarize data, consequently reducing perceptual complexity and increasing readability. Two of the primary properties of a data set that can affect readability include: 1) the distribution of values within the data set, and 2) the data set size.



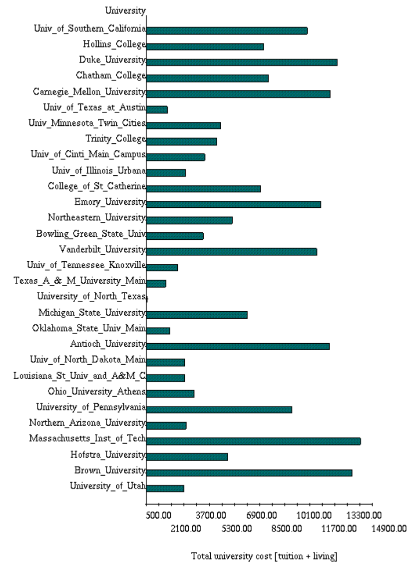
Data set with high cost *Rich-University* included **Data set with *Rich-University* removed**

Figure C-19: Mapping design where *out-of-state-tuition-cost* is mapped to the blue bar *x-lengths* and *room-&-board-cost* is mapped to the red bar *x-lengths* for a set of universities. In the data set with *Rich-University* there are severe dwarfing problems on the bar lengths making it difficult to accurately estimate cost values.

[1] This statement is true for all graphical property encodings except for when *text-labeling* is used. However, *text* is not perceptually pre-attentive, thus it is not a very effective graphical property choice.



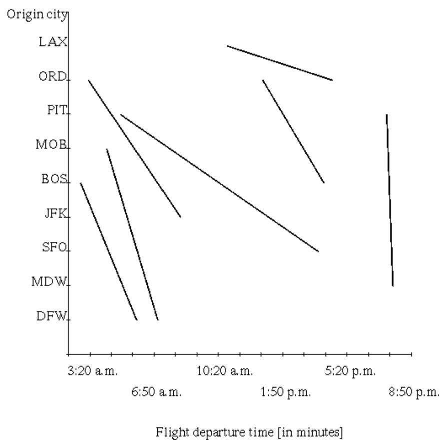
Data set with high cost *Rich-University* included (In this case *Rich-University* does not cause any dwarfing problems because we are only looking at the value differences rather than the actual cost values)



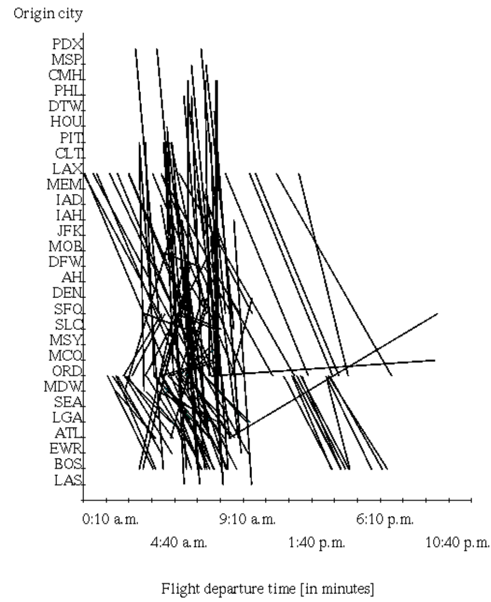
Data set with *Rich-University* removed

Figure C-20: Same data set as Figure C-19 but showing the pre-processed difference values between *out-of-state-tuition-cost* and *room-&-board-cost* instead of the original cost figures. Note that in this case it does not matter whether *Rich-university* is included or not, the difference distributions of the two data sets are comparable. I.e. the dwarfing problem is no longer an issue in the data transform design.

The distribution of values within a data set mainly affects the accuracy and complexity of achieving tasks. If a data set contains values that are widely set apart, then it is likely that the graphical scale used to encode the data will be much smaller than the data scale. This may cause some data values to be minimized or dwarfed so that it they are hard to access perceptually. For example, consider the *difference* task in Task C-2. Suppose that the costs of a particular university far exceed the others, as in Figure C-19-left. The long bars for *Rich-University,* cause the other cost bars to be greatly dwarfed. This makes perceptual judgements of bar length differences very difficult. On the other hand, if the distribution of values are close together, as in Figure C-19-right (which has the same data as the left figure but with the high cost university removed), the length differences can be determined with greater accuracy and with less probability for error. In cases where the data distribution results in intense data dwarfing, it is possible that using a data computation design will reduce the dwarfing effects. For example, Figure C-20 shows the pre-processed difference values of the data set in Figure C-19-left. In this design, the great dwarfing effect has been reduced because we are only showing the difference values between each bar pair instead of the values themselves.



(a) Smaller filtered airline data set that is much easier to process compared to design (b) because there is no occlusion here.



(b) Large unfiltered airline data set that is difficult to analyze because of the high degree of occlusion. With large data sets it is advantageous to do some pre-processing and pre-filtering to simplify the data set before mapping it to graphics or the resulting display will be difficult to analyze and process.

The size of the data set can also affect design choices. When data sets are large, data computation operators are usually preferable (to mapping operators) because they can summarize the data set and fewer elements need to be shown to users. For example, consider the airline-scheduling example shown below, where each flight is represented by a line. This graphic is appropriate when the flight database is small. However when there are many flights, we would quickly fill the visualization with so many lines that there would be too much occlusion to read anything useful from it. The data computation solution (Figure IV-3) however, does not have these problems because the data filtering possible significantly reduces the number of flights that need to be shown. The only exception is for *all-to-all* tasks (chapter IV-3.4) which should be performed with mapping techniques especially for larger data sets because the data computation solution significantly expands the amount of data in the visualization.

C-4.4 Summary

In this section we presented a set of simple examples and showed how variations to the 1) task, 2) task arguments, and 3) data sets, can result in different data computation and mapping designs. We intentionally used simpler examples so that the variations in design and user goals can be more clearly illustrated. We also briefly discussed how larger data sets and *all-to-all* task iterations can result in readability issues such as visualizations that are too dense, graphical elements that are too small or graphical elements that get dwarfed because of the wide distribution of values within the data set. As is shown in the examples here, some of these readability problems can be avoided through making careful choices between data and mapping operators. However, this alone cannot solve all the readability problems that may arise. In appendix F we discuss how interactivity, graphical, and rendering transforms can also be used to alleviate many of these readability problems.

The designs shown in this section are all generated by our prototype automatic designer. Our designer uses a *cost* structure that is based on the design guidelines described in chapter IV-2, chapter IV-3 and on previous heuristics used in automatic data graphic design [*Mackinlay, 1986a, 1986b; Casner, 1991*]. This cost structure is used to make choices between different graphical artifacts and data functions. Designs that simplify the data or that have a lower semantic distance get a lower score, while designs with high complexity or a great semantic distance get a higher score (i.e. a lower cost or score indicates a better design). Designs are then generated according to increasing cost (i.e. decreasing effectiveness). Implementation details on our designer are described in chapter V. In the next section, we explore a more realistic task of purchasing a car and show how our designer may be used in an iterative data analysis session.

C-5 Purchasing a Car

In this example, Bob is planning to purchase a car. In picking a car Bob is interested in the following properties: 1) low fuel consumption (i.e. high miles per gallon), 2) cheap price, 3) high performance engine (i.e. high engine capacity), 4) good speed and power (i.e. high *horsepower*). It is obvious that Bob cannot get a car that has all of these properties because they conflict with one another. Higher performance cars are usually more expensive and cars with good fuel consumption (high mpg) usually have lower performance. Thus Bob would need to trade-off these properties with one another and pick the car with the "best" balance between price and performance. There are several possible ways to accomplish this task. One way is to simply determine the best car in each category, and pick the car that tops the most categories. The task specification is shown below:

```
Find ( = , Lookup ( {all-cars},  price ),
          Compute ( Min, Lookup ( {all-cars},  price ) )
Find ( = , Lookup ( {all-cars},  mpg ),
          Compute ( Max, Lookup ( {all-cars},  mpg ) )
Find ( = , Lookup ( {all-cars},  engine-capacity ),
          Compute ( Max, Lookup ( {all-cars},  engine-capacity ) )
Find ( = , Lookup ( {all-cars},  horsepower ),
          Compute ( Max, Lookup ( {all-cars},  horsepower ) )
```

Task C-10: Car purchasing task A

The most effective design chosen computes the *max, min* and *find* tasks internally and the user is only shown the cars that top each category (Figure C-21). A problem with specifying the task in this manner arises when each category is topped by a different car. When this occurs there is no way to pick the best candidate. In addition, even though a car may top one or multiple categories (e.g. *Metro* in Figure C-21) it may not necessarily be a good choice for Bob if its ranking is low in the other categories. This is because Bob is looking for a balance between both price and performance.

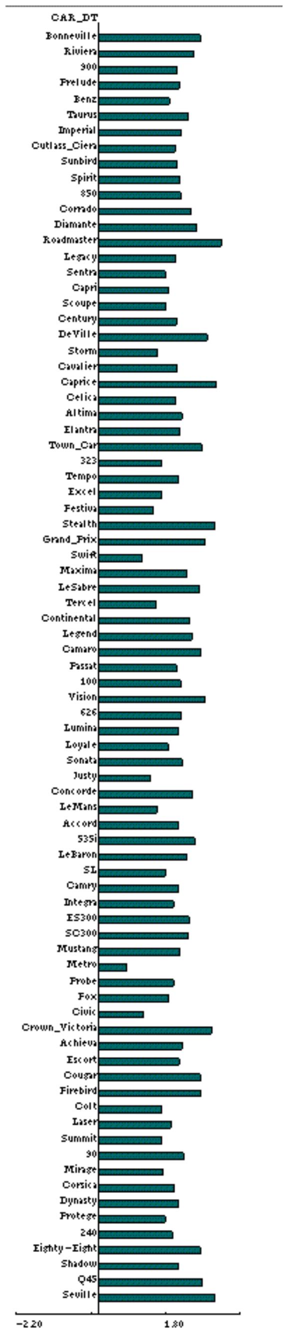
Horsepower	Engine-size	MPG in the city	Minimum price
Stealth	Roadmaster	Metro	Metro

Figure C-21: Data computation design for car purchasing task A (Task C-10)
The top cars are shown for each car picking category. While this design is very simple and easy to interpret it does not allow us to flexibly take all four car picking attributes into account simultaneously.

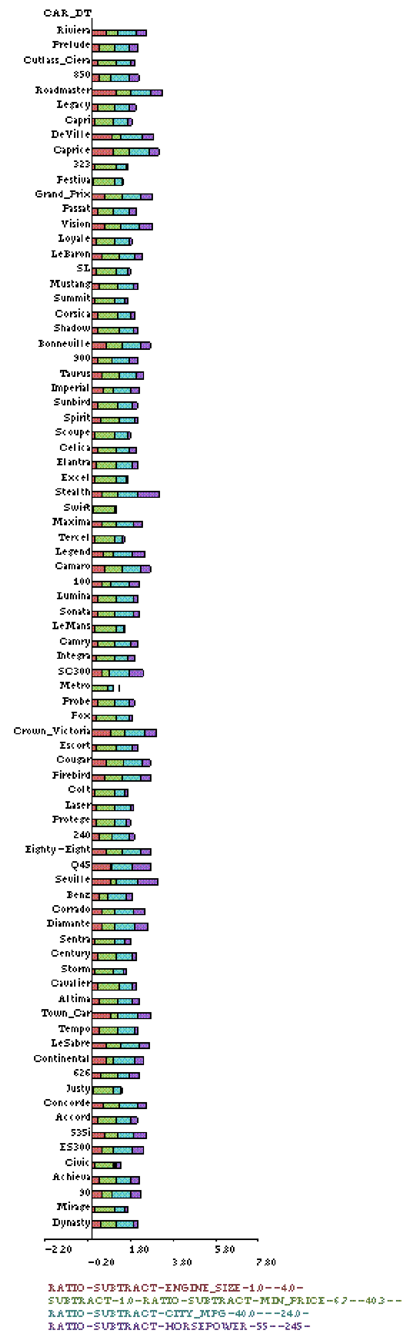
A better way to solve the task is to weigh each car based on all four attributes, and rank the cars based on the sum score of these weights (Task C-11). We will assume that all four categories are as important to Bob, so the weight is even for each category. To calculate the rank for each car, we score it from 0 to 1 for each attribute. The score for a particular attribute value is computed by determining where it falls between the minimum and maximum values for that attribute. 1 indicates the best value in the current data set and 0 indicates the worst value. Note that for the *price* attribute we need to perform an additional difference operation because the goodness of this attribute is inverse to its value size. The rank for a particular car is the summation of each individual attribute score. Since there are 4 attributes, the maximum score is 4.0.

```
Compute (  sum ,
    Compute (  Difference ,  1.0,
        Compute (  ratio,   Compute (   Difference ,
                                        Lookup (  {all-cars},  price ),
                                        min-price )
                            Compute (  Difference , max-price,  min-price ) )
             )
    Compute (  ratio,   Compute (   Difference ,
                                    Lookup (  {all-cars},  mpg ),
                                    min-mpg ),
                        Compute (   Difference , max- mpg,  min-mpg ) )
    Compute (  ratio,   Compute (   Difference ,
                                    Lookup (  {all-cars},  engine-capacity ),
                                    min-engine-capacity )
                        Compute (   Difference ,
                                    max-engine-capacity, min-engine-capacity )
                                 )
    Compute (  ratio,   Compute (   Difference ,
                                    Lookup (  {all-cars},  horsepower ),
                                    min-horsepower )
                        Compute (   Difference ,
                                    max-horsepower,  min-horsepower ) )
    )
```

Task C-11: Car purchasing task B
(Note that in this task specification we assume that the min and max values for each data attribute have been pre-calculated)



(a) A normalized score is computed for each of the car picking attributes and the sum scores are shown on the *x-axis* of the bar chart.



(b) A normalized score is computed for each of the car picking attributes and each score is represented by the *x-length* of a differently colored bar. The red bar represents the *engine-size* score, the green bar represents the *min-price* score, the blue bar represents the *mpg* score, and the purple bar represents the *horsepower* score. The sums can then be deduced by looking at the combined length of the stacked bars.

Figure C-22: Designs for car purchasing task B (Task C-11). Design (b) is more effective than design (a) because in design (b) it is easy to lookup the score sums and in addition, the individual scores are also given so that we may examine each criteria separately.

Because the task is computation intensive (i.e. many compute operators and task embeddings), the most effective design generated is a data computation design (Figure C-22a). In Figure C-22a, all the computations are pre-computed by the system and only the sum scores are shown for each car. While this method is much more effective than car purchasing task A (Task C-10), it requires that the user know what weightings to give to the different attributes. In addition, because the attributes are all rolled up into a single score, it is difficult to determine for a particular car which attributes contributed most to its final score. Figure C-22b shows an alternative hybrid design that is ranked second for this task. In this design, the score computation for each car attribute category is pre-computed, but the *addition* task for the four individual scores is achieved through a mapping transform. Each differently colored bar in Figure C-22b represents the score of a different car attribute, and the bars are stacked to facilitate the four score summations. This design solves the single score problem in Figure C-22a, however, the problem of identifying appropriate task weightings remain. In addition, both designs also suffer from having to display too many data elements, forcing users to navigate in the display to get to all the elements.

Note that a pure mapping design is much worse compared to the data computation designs in Figure C-22 because there are many embedded computations here and it is very difficult to graphically integrate all the information in a way that is consistent with the task constraints. In addition the added load from having to do all the intermediate tasks perceptually (i.e. embedded tasks) significantly increases the semantic distance of the design.

An issue related to task specificity is that of premature commitment. As we previously discussed, the more fully a task can be specified, the lower the articulatory load for the data computation design. However, the more that we need to commit at the outset of the task, the less flexibility we have later in the data analysis process. The importance of premature commitment depends on the task, the domain, and the user. We will show in the next section that our task specification language allows for less premature commitment by using the ('?') wild card character. This character indicates that the task parameter is unknown at the time of task specification. Building the task specifications themselves as well as deciding when it is appropriate to use wild card characters ('?'), however, is beyond the scope of this thesis.

Another way to perform the task is through a series of *finds*. We begin by finding a set of cars that have high mpg, and from there, we narrow down the set to those that have relatively low price and so on as in car purchasing task C (Task C-12), *case-1*. The problem with embedding tasks in this way is that it suggests an ordering to the tasks while in reality there is none. Thus in car purchasing task C (Task C-12), *case-2,* we use the *and* operator to group the *find* tasks together without embedding them.

Case 1:

```
Lookup
    (Find (  >  ,
           Lookup
                ( Find (  >  ,
                         Lookup
                          ( Find (  <  ,
                                    Lookup ( Find (  <  ,
                                                    Lookup (  all,  mpg ),  ?),
                                            price ),
                                    ? )
                          engine-capacity ),
                         ?),
                 max-speed ),
            ?)
    name )
```

Case 2:

```
Lookup (    And (   Find (  <  , Lookup (  {all-cars},  mpg ),  ?),
                    Find (  <  , Lookup (  {all-cars},  price ),  ? )
                    Find (  >  , Lookup (  {all-cars},  engine-capacity ),  ?)
                    Find (  >  , Lookup (  {all-cars},  max-speed ),  ?)
                    )
        )
```

Task C-12: Car purchasing task C

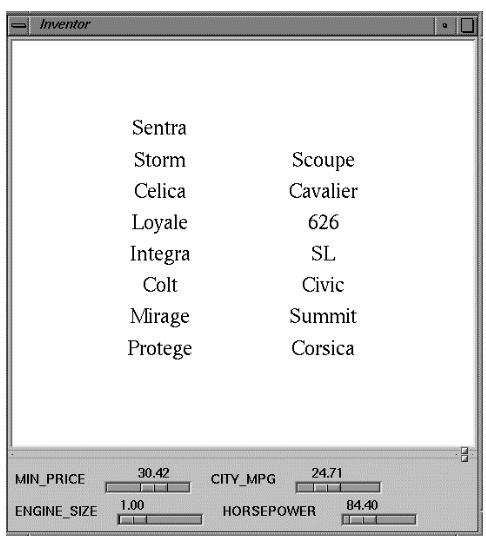


Figure C-23: Data computation design for car purchasing task 3
In this design users get the flexibility to enter in filtering thresholds for each of the four car picking attributes through *slider* input devices. The system then pre-computes all cars that fulfill those threshold conditions and then only displays those cars. This produces a much cleaner and effective design compared to the previous designs in Figure C-22. In addition, in this interface the user may weigh each of the four car picking attributes differently (e.g. place more importance on *horsepower* and less on *price*) by setting more or less stringent thresholds. In Figure C-22, each of the four car picking attributes are weighed equally and users are not given the ability to alter this weighting.

The best ranked design for car purchasing task C (Task C-12) is a data computation design (Figure C-23). In Figure C-23, four sliders are provided so that users may adjust the four car purchasing attribute thresholds. Only cars that fulfill the slider thresholds are shown as text in the display. Because of this data filtering, the number of cars shown at any one time is usually small. Even in the case where the original data set is large, a user can alter the threshold values so that most of the car concepts are filtered out except for the best choices. For this reason, the data computation design is ranked at the top even though car purchasing task C (Task C-12) has low task specificity. The data computation design has much less perceptual clutter compared to the mapping designs (Figure C-24 and Figure C-25) and it requires much less display space, thereby making it unnecessary to navigate through the visualization. In the mapping designs many more data concepts must be shown, and as a result users may need to scroll or zoom in and out to get to all the elements.

The top mapping design is shown in Figure C-24. In Figure C-24, the left bar chart encodes *engine-size* on the *x-axis* and *car-price* as *bar saturation*. The right chart encodes *miles-per-gallon* in the city on the *x-axis* and *horsepower* as *bar saturation.* The *car-names* are mapped to the *y-axis* of both charts. To perform car purchasing task C (Task C-12), we must identify bars in the first chart that are long (large *engine-size*) and unsaturated (low *car-price*). They must be paired with bars in the second chart that are long (high *miles per gallon*) and saturated (high *horsepower*). Compared to Figure C-23 many more graphical elements need to be shown here, thus users will most likely need to navigate within the visualization to get to all the elements. In addition, *saturation* does not allow value comparisons to be performed accurately. For these reasons, the data computation design in Figure C-23 is ranked higher than the pure mapping design in Figure C-24.

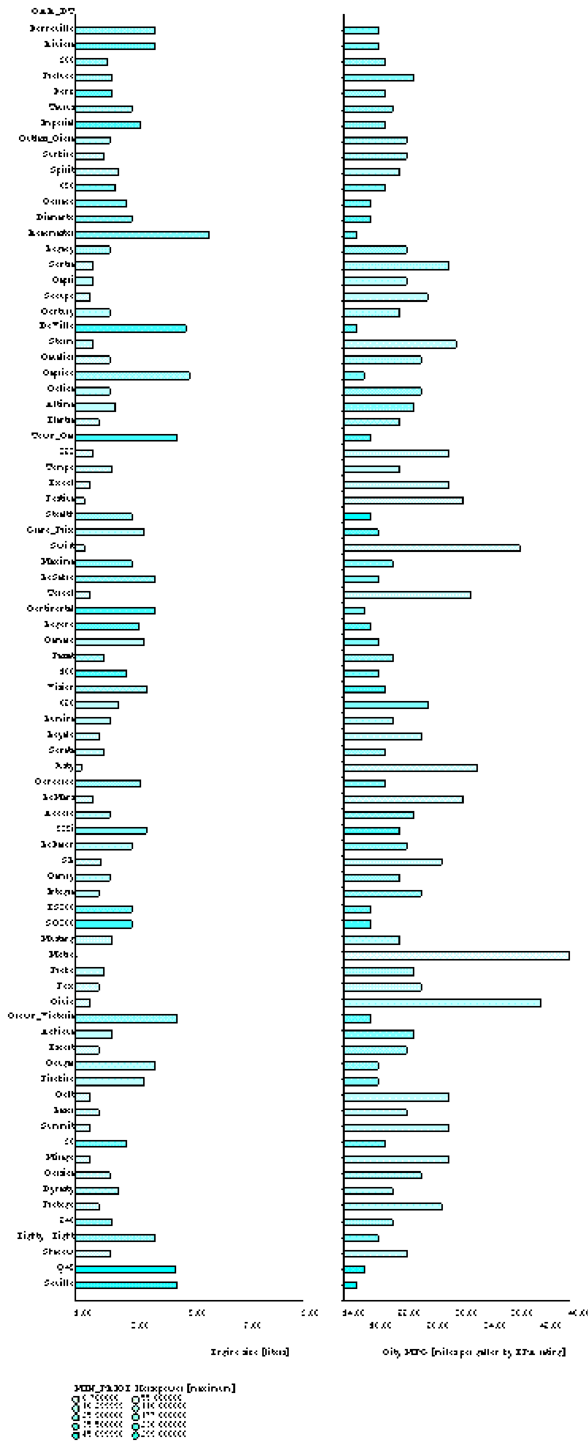


Figure C-24: Mapping design-1 for car purchasing task 3

Mapping design that shows the original four stock picking attributes to the user. *Engine-size* is mapped to the *x-axis* of the left chart and *min-price* is mapped to the *saturation*; *city-mpg* is mapped to the *x-axis* of the right chart and *horsepower* is mapped to *saturation*. Desirable cars are those with long (large *engine-size*), unsaturated (low *min-price*) bars in the left chart and short (low *city-mpg*), saturated (high *horsepower*) bars in the right chart. This design is significantly more complex in terms of number of elements and difficulty of interpretation compared to Figure C-23. In addition the *saturation* encodings does not allow for accurate value lookups or comparisons

An alternative design is to map each attribute to a set of aligned bars (Figure C-25). Figure C-25 is deemed less effective compared to Figure C-24 because it requires users to scan across a wide spatial distance to get to each aligned value. In addition, an even greater amount of navigation is probably needed here due to the larger visualization size. However, by using *position* to encode all of the attributes, we can perform value comparisons much more accurately than in Figure C-24.

Figure C-25: Mapping design-2 for car purchasing task 3
Mapping design that shows the original four stock picking attributes to the user. *min-price* is mapped to the *x-axis* of chart 1 (left-most chart), *city-mpg* is mapped to the *x-axis* of chart 2, *engine-size* is mapped to the *x-axis* of chart 3, and *horsepower* is mapped to the *x-axis* of chart 4 (right-most chart). Desirable cars are those with short bars in chart 1 (low *min-price*), short bars in chart 2 (low *city-mpg*), long bars in chart 3 (large *engine-size*), and long bars in chart 4 (high *horsepower*). This design is more accurate than Figure C-24 because all values are encoded on the *x-axis* (i.e. no *saturation* values are used). However this design is also less integrated and requires significantly more eye movement, display space, and display navigation.

A better mapping alternative is to integrate the attributes together within a single graphical object so that users need not read graphical properties across different objects across multiple spaces. In this case however, none of the graphical objects available to the designer is capable of expressing five attributes (*name, mpg, price, engine-capacity, max-speed*) in an integrated manner.

This example illustrates how a design that most suits the user's needs can be derived through iterative refinement between the user and the automatic design system. An automatic design system supports this process, by enabling users to rapidly test out different design ideas, and by presenting design alternatives that can serve as starting points for further refinements. As we had discussed previously, the ultimate goal is not for users to learn and specify the task constructs shown here, but rather to attach the automatic designer to a higher level domain specific system, that can generate these specifications based on verbal descriptions by the user or based on the design alternatives that are favored by the user.