Contents

[Abstract 2](#_Toc477733315)

[Report Summary 3](#_Toc477733316)

[1. Introduction 4](#_Toc477733317)

[1.1. Aims and Objectives 6](#_Toc477733318)

[2. Related Work 7](#_Toc477733319)

[2.1. Background 7](#_Toc477733320)

[2.2. Cost Models 8](#_Toc477733321)

[2.3. State Diagrams 9](#_Toc477733322)

[2.4. Ranking of Suggestions 9](#_Toc477733323)

[2.5. Sampling 9](#_Toc477733324)

[3. Project Planning and Organisation 10](#_Toc477733325)

[3.1. Original Project Plan 10](#_Toc477733326)

[3.1.1 Research 10](#_Toc477733327)

[3.1.2 Implementation 10](#_Toc477733328)

[3.1.3 Testing 11](#_Toc477733329)

[3.1.4 Documentation 11](#_Toc477733330)

[3.2 Final Project Plan 11](#_Toc477733331)

[3.2.1 Research 11](#_Toc477733332)

[3.2.2 Design 11](#_Toc477733333)

[3.2.3 Development 11](#_Toc477733334)

[3.2.4 Testing 11](#_Toc477733335)

[3.2.5 Write-Up 11](#_Toc477733336)

[3.3 Organisational Tools 11](#_Toc477733337)

[3.3.2 Version Control 12](#_Toc477733338)

[3.3.3 Documentation 12](#_Toc477733339)

[Design 13](#_Toc477733340)

[Testing Strategies 14](#_Toc477733341)

[Performance Analysis and Evaluation 15](#_Toc477733342)

[Future Work 16](#_Toc477733343)

[Conclusions 17](#_Toc477733344)

[Appendices 18](#_Toc477733345)

[Appendix 1: Original Project Plan 18](#_Toc477733346)

[References 20](#_Toc477733347)

# Abstract

The cost model developed throughout the project can be found at:  
[www.github.com/samanthacoll22/PlacesProject](http://www.github.com/samanthacoll22/PlacesProject)

# Report Summary

Chapter 1: Introduction – A look into the history of query autocomplete and an outline of the project aims.

Chapter 2: Project Planning and Organisation – A discussion of the time management and general organisation of the project.

Chapter 3: Design – A technical look at the programmatic design of the application.

Chapter 4: Testing Strategies – A technical look at ensuring the application works as expected.

Chapter 5: Performance Analysis & Evaluation – A study into the continued reliability of the application as the simulation increases and an evaluation of the aims.

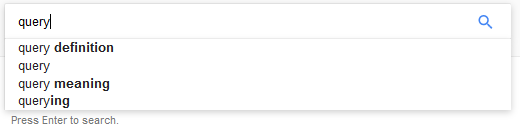
Chapter 6: Future Work – A reflection and discussion on what actions could be taken to further improve the simulation.

Chapter 7: Conclusions - A mirror to Chapter 1, covering which of the aims the project has met and which aims remain to be completed.

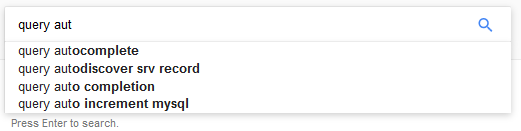
For an idea on how the application is used Appendix \*BLAH\* User Guide should be consulted.

# Introduction

Web searching is a tedious and omnipresent activity which is carried out by millions of people on a daily basis. However, this time-consuming and error-prone process of formulating and physically typing search queries is what can make query input laborious. In response to this problem, search engines have adopted the concept of query auto-complete (QAC) as a way or reducing the effort it takes to submit a query. QAC provides the user with a list of suggested queries as they begin to enter their query in the search box. The incomplete input of the user is often referred to as *query prefix* and the suggested queries are often called *query completions*. As shown in Figure 1, query completions often start with the characters that the user has entered into the search box (see Figure 1a) and are updated as the user continues to type (see Figure 1b). The primary objective for effective QAC is to provide the user with their intended query after the fewest possible keystrokes, and at the highest possible rank in the list of completion suggestions. QAC helps the user to formulate their query when they have an intent in mind but not a clear way of expressing it in a query. It helps to avoid possible spelling mistakes, especially on devices which have smaller screens. The most common approach to QAC is to extract past queries with each prefix from a query log, and rank them by their past popularity [1]. Although this approach provides satisfactory result on average, it is hardly most favourable as it fails to take in clues such as time or user context which quite often influences the queries most likely to be typed. This report will focus on the development of a cost model that shows how a user interacts with a ‘search bar’ on a website. It will then explore these different approaches of QAC and show how each keep cost to a minimum. In addition to a minimal cost, cutting down the search time duration of users implies a lower load on the search engine, which results in savings in machine resources and maintenance [2].



**a) *A list of four query completions for the prefix “query”.***



**b) *An updated list of four query completions for the prefix “query aut”.  
Figure 1: Examples of query auto completion.***

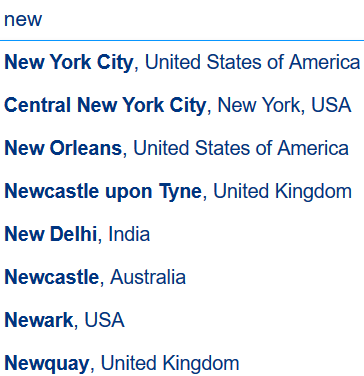
As the web continuously becomes a platform for real-time news and media, time consumption takes a vital role on the retrieval of information. Users turning to search engines for information about new or ongoing events, takes up the majority of daily queries [2]. However, importantly the use of QAC is not limited to the use or search engines, such as Bing, Google or Yahoo!. QAC has been integrated into other common services for search tasks. Facebook[[1]](#footnote-1) provides a QAC for finding friends by taking in mutual friends the user may share with others; it also assists users to formulate requests for searching through tweets on the social medium Twitter[[2]](#footnote-2); QAC is also used for product searches in online shopping stores, such as Amazon[[3]](#footnote-3) which will often provide up to 10 queries. In the concept of web search, research indicates that using the suggested queries can vastly improve user satisfaction, especially for informational queries [3]. The main focus for this report is how QAC is integrated into a travel metasearch engine like Booking[[4]](#footnote-4), where user’s search for the broadest selection of accommodation in every corner or the world.

## Aims and Objectives

The main aims and deliverable of the project are:

* To create a cost model of how people interact with the search box on a website like Booking, which can be seen in Figure 2, where people start to type in letters and the system predicts where they might want to go.
* To evaluate the impact of ranking on query autocomplete and how it influences the goal of finding the minimum cost.
* To evaluate the number of suggestions the user is presented with and the effect it has on the cost model.
* To evaluate the outcome of generating suggestions at different instances and how they assist to keep cost to a minimum.

An additional aim of the project is to create a java application that will display various destinations that the user will type in at their own speed. The time taken for the user to type the destinations will be noted and an average time for each letter will be generated. This will then be implemented into the cost model for each user, giving a more realistic, personal result.



***Figure 2: Output from user typing “new” into Booking.***

# Project Planning and Organisation

To guarantee the project could be completed by the final deadline, knowledge of the status of the project was key. To guarantee success, the current status of the project at all points was confirmed by the creation of a project plan. The project plan meant that if the project was destined to be incomplete for a particular reason, then this would become apparent with enough time to allow changes to rectify the problem.

## Original Project Plan

The original plan was created by splitting the project into four main sections: Research, Implementation, Testing and Documentation. Each of these sections were then sub-divided into smaller, more manageable tasks, which could be tackled individually. Two of these sub-tasks came with strict deadlines: Poster Creation (27th November 2016) and Report Write-up (31st March 2016), which dictated deadlines to support the sections. The creation of the poster required the research to have been exhausted, which confirmed that section complete. The report covers every aspect of the project, guaranteeing each section has been fully covered by that date. The full original plan is located in Appendix 1: Original Project Plan.

### Research

Research was undertaken into the way query auto completion works. A detailed look into how the user jumps between typing a letter, to looking at the list of suggestions, to then picking from the list, and the probability of each state occurring. A decision was made that ten days was a suitable amount of time to create the poster which assisted in creating a deadline for the end of December for the research section to be competed. The time available was then split amongst the sub tasks within research.

### Implementation

The Implementation of the system was to be completed by the end of February, thus allowing a carefully allocated, suitable amount of time for the final overall system to be tested and the report to be written before the final deadline. The time available between creating the poster and the deadline set for Implementation gave the overall time available. This time was taken and divided amongst these main tasks based on the perceived level of difficulty they carry.

### Testing

Four main aspects of testing were anticipated to be employed throughout the project: Unit Testing, Sub-System Testing, Full Testing and User Trials. Ideally, the testing portion of the report was to be carried out in parallel with the Implementation section. Unit Testing would be performed as each small section of the cost model was created, with Sub-Section testing being carried out at the end of each new feature and User Trials carried out just before writing the final report. Leaving only Full System Testing as the only part of testing carried out independently of the other sections. It was decided that two weeks would be a sufficient amount of time to satisfy this.

### Documentation

The documentation section consists solely on two set deliverables for the project. Strategically, the timescales that are allocated to these deliverables have been greatly exaggerated to ensure the project’s completion before the deadline. Creating a buffer like this allows expansion for the other sections, if necessary, whilst still ensuring the deliverables completion before their respective deadlines.

## Final Project Plan

As the project progressed the initial project plan became impossible to follow precisely. As problems occurred, and the aims of the project evolved, the project plan was continually updated to reflect the current state of the project. The final project plan is available in Appendix 2 Final Project Plan.

### Research

The research section was largely unchanged from the original plan. The only extra research that was carried out was to source a significantly larger amount of places/cities. This makes the cost model much more accurate, as before, when there was only 125 cities, after the third letter was type the user was frequently presented with their destination as the first suggestion. This resulted in them selecting it the majority of the time, when in reality they may have been forced type again or look further down the list. As previously stated, there are now around 3000 places/cities which result in far better simulations.

### Design

The design of the

### 2.2.3. Development

### 2.2.4. Testing

### 2.2.5. Write-Up

The time taken to complete both the report and poster was largely as anticipated during the original project plan. The only change occurring in this section was to allow other unfinished work to run alongside it. Therefore, stages of implementation and testing ran in parallel to writing the report. This was greatly expected when the original plan was drawn up and was the sole reason for awarding the write-up so much time.

## Organisational Tools

In order to complete the project, various organisations tools were employed to keep within the timescale outlined in Chapter 3.2 Final Project Plan.

### 2.3.2. Version Control

Version Control is an important aspect of any project with a heavily weighted software component. For this project specifically, Git was utilised. Git offers a variety of benefits to a project, including the ability to work on different features, or sections, in parallel whilst on separate branches. This ability allows the changes on an individual feature to have no negative consequences or effects on the other. However, the most important and most relied on feature of Git, if the key functionality breaks, is the ability to revert to a previous version of the software. Allowing access to a functional version at a later date, as a form of back-up copy.

Git operates on a central server, where work can be uploaded to and downloaded from. The Git server used for this project was provided by Github[[5]](#footnote-5), which holds a copy of all work on their servers. This not only grants access to be worked on by multiple machines, but as long as the software is being pushed on a regular basis to the server, creates a safety net as large amounts of work cannot be lost if something were to happen to the work station.

### 2.3.3. Documentation

When dealing with a project of significant size, it is important to maintain a detailed record of work completed throughout. As the project was carried out, design and implementation notes have been documented in a logbook. Giving an insight to the thought processes throughout. Additionally, when software is pushed to Git, it is a requirement to detail any changes that have been made. These messages are known as a Git log and they show what has been altered and more importantly, why.

# Project Context

Initially, before any work could begin to take place, several key questions had to be answered about how the project should proceed. To answers these questions, research into the history of query autocomplete and it’s advancements over the last few decades was initiated. Furthermore, investigations into several different aspects of the project were explored further.

## Background

The concept of Word Predictions is the process in which a writer writes the first letter or letters of a word and a word predictor generates a list of possible words or choices and if the desired word is listed, then this can be selected. It is from this concept of word prediction, where the most likely next words are listed, that the concept of query auto completion was adopted. The first instance dates back around half a century, where Longuent-Higgins and Ortony outline a method, where commands and identifiers were entered by users to decrease the amount of keystrokes necessary to complete a word [4]. Additionally, a true motivator for auto completion was to help individuals with physical disabilities increase their typing speed. As algorithms were developed, the deeper the research into the overall benefits were explored. Early studies found that a reduction in key strokes through word prediction was often over shadowed by having to scan the list of predictions for the desired word [5]. Therefore, it was discovered that displaying five suggested items to the user was found to provide a reasonable balance between keystroke saving and scanning the array of predicted words [6].

Throughout the 1970s and 1980s, the algorithmic work for word predictions was split up into three classes: character predictors, word completers and systems with a combination of both [7]. As assumed, character predictors make likely letters quicker and simpler to select. On the contrary, word completers take a supported text input and return words based on the initial prefix of one or more letters entered by the user. Combined systems can carry out both these tasks. The first instance of a combined approach was The Reactive Keyboard, which was introduced by Darragh in 1990 [8]. This approach included many familiar aspects of today’s autocomplete, incorporating the use of ‘most popular’ and ‘task relevant’ words.

At its stage in time, The Reactive Keyboard is a prototypical example as it portrays the concept of word completion and vastly accelerated typewritten communication with a computer. However, its vocabulary of words was based on a standard and not extensive variety of words, therefore, later advancements of the model showed word completion based on previously entered text. In a similar approach, a user’s search history over a long-period of time can be used for query auto complete [9], but this has yet to be integrated into a word completion system. Word completion is reliant on a specific tree structure which will take the input and match it with its completions, very similar to the data structure which is used in query auto completion.

Query auto completion, also sometimes referred to as incremental search or real-time suggestions, goes back to the Emacs first text editor [10], which output a single line of feedback compared to the list of suggestions familiar to us. It wasn’t until the millennium that Jef Raskin, introduced the use of query auto completion in which the user will get instantaneous feedback throughout the process of entering a query [11]. This process was called delimited search, which is a traditional search interface where the user will search in three steps: a user submits a query, the system will compute a result, to which the user will receive said result. Since this moment, QAC has been used by websites, desktop searches, operating systems, email clients, internet browsers and search engines. Leading to 2004, when Google Suggest was launched, which was the first instance of query auto completion in the setting of a search engine.

## Cost Models

Cost estimation models are mathematical or parametric equations used to estimate the cost of a product or project. These estimations will never be exact as there are many variables involved in calculating a cost estimate, such as technical or human. Moreover, any process that is significantly human based will never be exactly accurate as humans are far too unpredictable and complex to be exact. Cost models function through the input of parameters that describe the attributes of the product or project in question. In the case of this report, the cost model is formed by generating the sum of the places. The equation for the cost model can be seen in “(1)”.

*Total cost = ∑ [cost (place) \* popularity (place)]* (1)

### 3.2.1. Finding the Cost of a Place

To generate the cost of a place 3 main factors must be known: the cost it takes for the user to type a single letter, then the cost of a user to look at the list of suggestions, and finally, the cost for the user to pick a suggestion. However, when the user is looking at the list suggestions there is a degree of uncertainty of what they might do next. This leads to a probability that the user might type and a probability that they may pick from the list. For the purpose of the cost model, it has been decided the user will only be presented with a list of suggestions after typing in a minimum of three letters.

Therefore, if we take the example “Paris”, we know that it is necessary for the user to type “P”, “A”, “R”, which each carry a cost to type letter. So, right now our total cost for our destination is (3 \* typeCost). However, when the user is presented with the retrieved list of suggestions, will the user look at the list and pick? Or will they simply type another letter? This is where we have to prepare for all outcomes. The process for finding the fourth letter can be seen in “(2)”.

*(3 \* typeCost) + (probabilityType \* typeCost)   
 + ((1 – probabilityType) \* lookCost)* (2) *+ (lookCost + (probilityPick \* pickCost))*

**Going to go on and explain this more about the equation.**

### 3.2.2. Determining the Ranking Cost.

On the other hand, the popularity of a place can be determined as the United Nations Statistical Division (UNSD), publishing an annual book on world statistics helped us construct a more precise idea of countries popularity, based on its geographical size. A simple representation of the cost model can be seen in Figure 3.2.

Cost (Place)

Looking Cost

Typing Cost

Picking Cost

Ranking (Place)

Total Cost

Probability Pick

Probability Type

***Figure 3.2: A Simple Diagram of Key Parameters for Cost Model.***

## State Diagrams

A state diagram is used in computer science and fellow related fields to give an abstract description of the behaviour of a system. Specifically, a state diagram analyses the behaviour of a single object in response to a series of events occurring and takes into consideration all possible states that object could find itself in. The cost model will generate four states: a type state, a look state, a pick state and a search/end state. These will each be explore in detail in the design section of the report.

## Sampling

The process of sampling takes a predetermined number of observations from a larger population and carries out analysis on the smaller group. To take every city/place in the world and run the cost model could take days, or maybe even weeks to carry out. It is then far more time-efficient and economical to take a sample of the data. The simulation to run the cost model has around 3000 cities/places loaded into it. These are the cities in the world that have a population of over 100,000 people. Therefore, we are already sampling before generating any total cost. Originally, 319,000 cities/places were used for the simulations which would give more accurate results, but the waiting time to load was around 10 minutes before the simulator proceeded. The advantages and disadvantages of sampling can be seen in Table 1.

***Table 1: The Advantages and Disadvantages of Sampling***

|  |  |
| --- | --- |
| Advantages of Sampling | Disadvantages of Sampling |
| * Low cost of sampling. If data was being collected for the every place in the world, the cost would be extremely high. | * **Chance of bias.** Since we get to choose our own specific data, we could favour some places more than others. This could draw erroneous conclusions, e.g. more well-known places are perhaps more likely to get selected. |
| * Less time consuming. Could take days to run the simulation for all places. Sampling gives us an understanding, in a healthier amount of time. | * **Difficult to select truly representative sample.** For instance, we may favour so choose higher ranked places. Which will give a lower cost than lower ranked places. Meaning the lower ranked destinations aren’t represented truly and the total cost may be lower than should be. |
| * Organisation of convenience. By taking a selection of places, it makes it easier to handle and less chance of error along the way. |  |

## Ranking of Suggestions

In essence, query completions are ranked according to predetermined criterion. Therefore, when a prefix is provided, some of this criterion is then returned to the user. Normally, a precomputed auto-completion system is necessary for determining QAC which correspond to each individual prefix in advance; storing these associations in data structures, such as prefix trees. Figure 3.5 shows a simple QAC framework.



***Figure 3.5: Simple QAC Framework***

When a user enters a prefix in the search box, we can see a basic list of suggestions is retrieved. Based on further re-ranking using signals determined at query time, e.g. location, season, time, the user will then receive a finalised list of query completions. Mainly these completions that are returned have a limited length, 6 for Booking and 4 for Google. For this project a list of known cities/places will be searched and the ones that match the prefix of the desired place will be placed into a separate list. This list will have no form of structure and will simply be outputting the first suggestion it comes across. One of the parameters for the cost model will allow the user to decide who they would like their suggestions displayed. Either, alphabetically, reverse-alphabetically or by population, depending on their desired destination.

## Software Engineering

To develop and ensure an easy maintainable project it is vital to try conduct good software engineering practices throughout. By using well-structured software engineering practices, they should assist in making testing easier as the software should be more flexible and subject to change. The CS409 Software Architecture and Design as Strathclyde University helped pin point various designs and patterns used in the project.

### Architectural Style

The highest level programming choice to be made in a project is the Architectural Style. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems. They are thought of as sets of principles that shape an application and they determine how much data and functionality should be separated. Since the project to be developed using Java, the architectural design that best suits would be Object-Orientated architectural style, and this is the primary style used throughout.

#### 3.6.1.1 Object-Oriented

Object-oriented architecture style based on dividing up responsibilities a system may have into individual, independent, reusable objects, which hold the data and behaviour relevant to the object. This style interprets the system as a series of banding objects, instead of a set of routines or procedural calls. This approach evokes a high level of abstraction, which means if one section of code is altered then it won’t dramatically affect the programs other sections. Objects are discrete, self-sufficient and loosely coupled. They communicate with one another through interfaces, by calling methods or accessing properties in other objects. Creating an interface gives access to publically available method calls that additional classes may use to interconnect with any class implementing that interface. The key advantages of the object-oriented architectural style are:

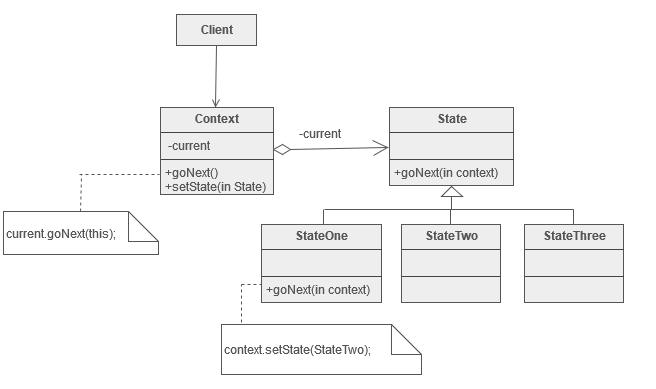
* + **Understandable**. It maps the application more closely to the real world objects, making it more understandable.
  + **Reusable**. It provides for reusability through polymorphism and abstraction.
  + **Testable**. It provides for improved testability through encapsulation.
  + **Extensible**. Encapsulation, polymorphism, and abstraction ensure that a change in the representation of data does not affect the interfaces that the object exposes, which would limit the capability to communicate and interact with other objects.
  + **Highly Cohesive**. By locating only related methods and features in an object, and using different objects for different sets of features, you can achieve a high level of cohesion

### 3.6.2 Design Patterns

Design patterns are one of the most valuable tools for developers. They illustrate the best design solutions that others have encountered, and allow you to apply the same principle to your own designs. More importantly, knowing design patterns gives a common vocabulary for software developers to use when talking about their designs. Design Patterns are essentially proven solutions to commonly recurring problems. The use of design patterns provides the means to deal efficiently with such problems. Below is an overview of each of the design patterns which have been used throughout the project.

#### 3.6.2.1 State Pattern

The state pattern is a behavioural software design pattern that implements a state machine in an object-oriented way. With the state pattern, a state machine is implemented by implementing each individual state as a derived class of the state pattern interface, and implementing state transitions by invoking methods defined by the pattern's superclass. The state pattern can be interpreted as a strategy pattern which is able to switch the current strategy through invocations of methods defined in the pattern's interface. This pattern is used in computer programming to encapsulate varying behaviour for the same object based on its internal state. This can be a cleaner way for an object to change its behaviour at runtime without resorting to large monolithic conditional statementsand thus improve maintainability. The UML diagram of a state pattern cab be seen in Figure 3.6.2.1.



***Figure 3.6.2.1: UML Diagram of State Pattern***

The main advantage in the State design pattern, an object's behaviour is the result of the function of its state, and the behaviour gets changed at runtime depending on the state. This removes the dependency on the if/else or switch/case conditional logic. With State pattern, the benefits of implementing polymorphic behavior are evident, and it is also easier to add states to support additional behavior. The State design pattern also improves Cohesion since state-specific behaviors are aggregated into ConcreteState classes, which are placed in one location in the code. With the State design pattern, it is very easy to add a behavior by just adding one more ConcreteState class. State pattern thus improves the flexibility to extend the behavior of the application and overall improves code maintenance.

#### 3.6.2.2 Observer Pattern

The Observer Design Pattern is essentially a lower level version of the same principles the Event-Based Implicit Invocation Architectural Style was based on. Essentially, one component allows others implementing a specific interface to register for updates. Whenever this first object recognises a change of some sort it notifies all of those listening for updates. Using the Observer Pattern allows for two components which hold similar data to remain consistent while also staying loosely coupled. This design pattern can be seen most clearly in the sample UML diagram shown in Figure 3.6.2.2.

Observer

*Update()***Update(State)**

Concrete Subject

*getState()*

Subject

addObserver(Observer)  
notify()

Concrete Observer

***Figure 3.6.2.2: Observer Design Pattern UML***

The main advantage to this design pattern is that it very strongly decouples the concrete subject from the concrete observer allowing any changes to the concrete observer to be made easily. The main disadvantage, however, is that debugging potential problems may become more difficult as the flow of code becomes less obvious. There are two versions of the Observer pattern: the push model and the pull model. They both follow the same basic structure however with one major difference. The push model gives the Observer the information it requires when it is notified. By comparison the pull model simply alerts the Observer that something has happened. This means the Concrete Observer requires a reference to the Concrete Subject in order to determine what has changed and react accordingly. These differences can be seen clearly in Figure 3.6.2.2 with the aspects only in the push model shown in **blue** and those only in the pull model shown in *red*. The push model has the added advantage of also cleanly decoupling the Concrete Observer from the Concrete Subject.

The Observer pattern is used in the creation of the JavaApp that allows the user to login and find their typing speed. This application depends greatly on the use of listeners, which the Observer pattern is based on. The event listener was used a great deal, where an ActionLister was registered to a button UI control. In this case the ActionListener is the observer and the button is the subject. As the button changes state, the next method would be called.

# Design

# Testing Strategies

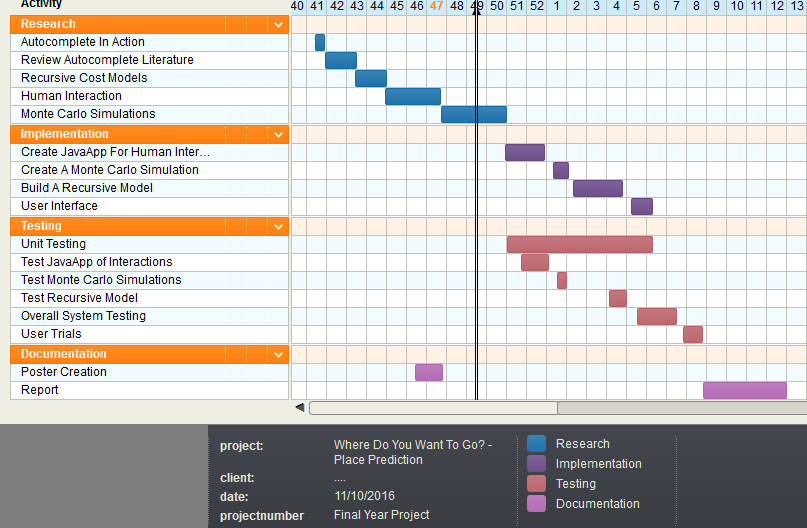
# Performance Analysis and Evaluation

# Future Work

# Conclusions

# Appendices

## Appendix 1: Original Project Plan



Design…. The figure shows like user types – types again – looks etc

State diagram. The cost changes the further down they look.

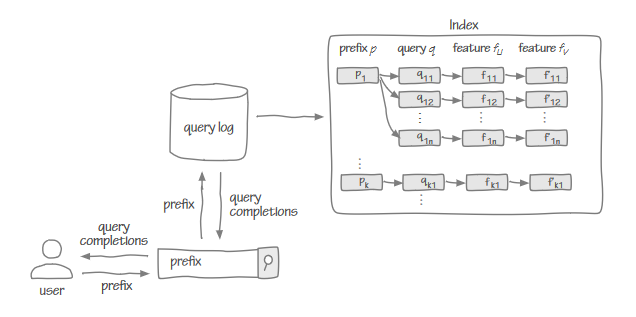
Graphs of probability.

If someone is a slower typer but fast looker and vice versa

At what point does the suggestions become irrelevant?

Latitude, cities that are closer are more likely to be recommended… etc

Sampling, take 1000 cities. Fix them. Only going to take 1000s for the cities, take the average and that what it is to do. Make sure cities don’t change.



# References

[1] Z. Bar-Yossef and N. Kraus.  
Context-sensitive query auto-completion.   
In WWW ’11, pages 107–116, 2011

[2] S. R. Kairam, M. R. Morris, J. Teevan, D. J. Liebling, and S. T. Dumais.   
Towards supporting search over trending events with social media.   
In ICWSM ’13, 2013.

[3] Yang Song, Dengyong Zhou, and Li-wei He.

Post-ranking query suggestion by diversifying search results. In Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval,

SIGIR ’11, pages 815–824, New York, NY, USA, 2011b. ACM.

[4] H. Christopher Longuet-Higgins and Andrew Ortony.

The adaptive memorization of sequences.

In Proceedings of the 3rd Annual Machine Intelligence Workshop, Machine Intelligence 3, pages 311–322. Edinburgh University Press, 1968.

[5] Gregg Vanderheiden and David Kelso.   
Comparative analysis of ﬁxed-vocabulary communication acceleration techniques. Augmentative and Alternative Communication, 3:196–206, 1987.

[6] Andrew Swifﬁn, John Arnott, J. Adrian Pickering, and Alan Newell.   
Adaptive and predictive techniques in a communication prosthesis.   
Augmentative and Alternative Communication, 3:181–191, 1987.

[7] John J. Darragh.  
Adaptive predictive text generation and there reactive keyboard.   
Master’s thesis, Computer Science Department, University of Calgary, 1988.

[8] John J. Darragh, Ian H. Witten, and Mark L. James.  
The reactive keyboard: A predictive typing aid.   
Computer, 23(11):41–49, 1990.

[9] Fei Cai, Shangsong Liang, and Maarten de Rijke.  
Preﬁx-adaptive and time-sensitive personalized query auto completion.   
IEEE Transactions on Knowledge and Data Engineering, 28(9):2452–2466, September 2016.

[10] Eugene Ciccarelli.   
An introduction to the Emacs editor.   
TechnicalReportAIM-447, MIT Artiﬁcial Intelligence Laboratory, Cambridge, Massachusetts, 1978.

[11] Jef Raskin.   
The Humane Interface.   
Addison-Wesley Professional, 2000.

1. https://www.facebook.com [↑](#footnote-ref-1)
2. https://www.twitter.com [↑](#footnote-ref-2)
3. https://www.twitter.com [↑](#footnote-ref-3)
4. https://www.booking.com [↑](#footnote-ref-4)
5. www.github.com [↑](#footnote-ref-5)