Thomas JANNAUD

August 26th, 2010

Apple - Xcode - IDE Infrastructure Team

Introduction

I) Open Quickly

1. The Levenshtein Distance
2. Algorithm
3. Adaptation of the Levenshtein distance to our problem
4. Calibration of the parameters
5. Defuzzification

II) Working Sets

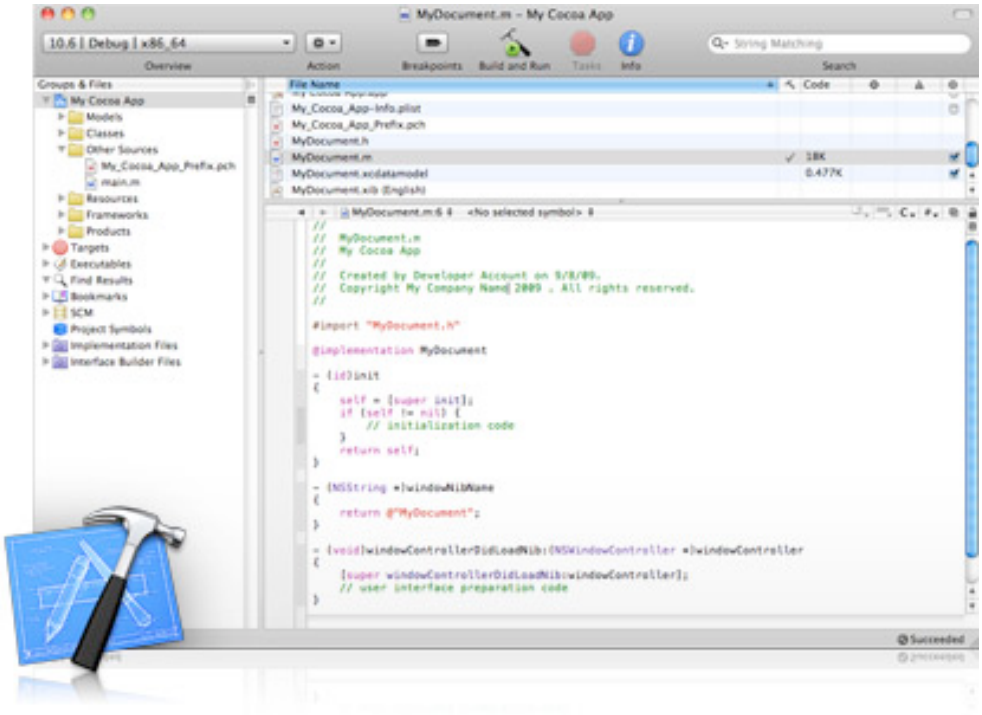
1. Problematic
    1.1. A step toward a solution
    1.2. Concept
    1.3. Incremental vs Static
2. Model
    2.1. Defining a distance in the files set - Files relevant to another
    2.2. Results
3. Performing clustering
    3.1. The cluster “Others”
    3.2. Size
4. Incremental Clustering
    4.1. Some useful metrics
    4.2. Incremental workflow

III) Further steps and Critics

1. Tests
2. Time
3. Memory Management
4. SVN

Conclusion

The centerpiece of the developer tools included with Mac OS X is the Xcode application. Xcode is a full-featured IDE for developing Mac applications and includes a world-class code editor, a graphical debugger, and integrated Objective-C, C, and C++ compilers. Xcode has intimate knowledge of the Cocoa frameworks that power Mac OS X, and it is even able to identify bugs by analyzing the code you write — without running the application.



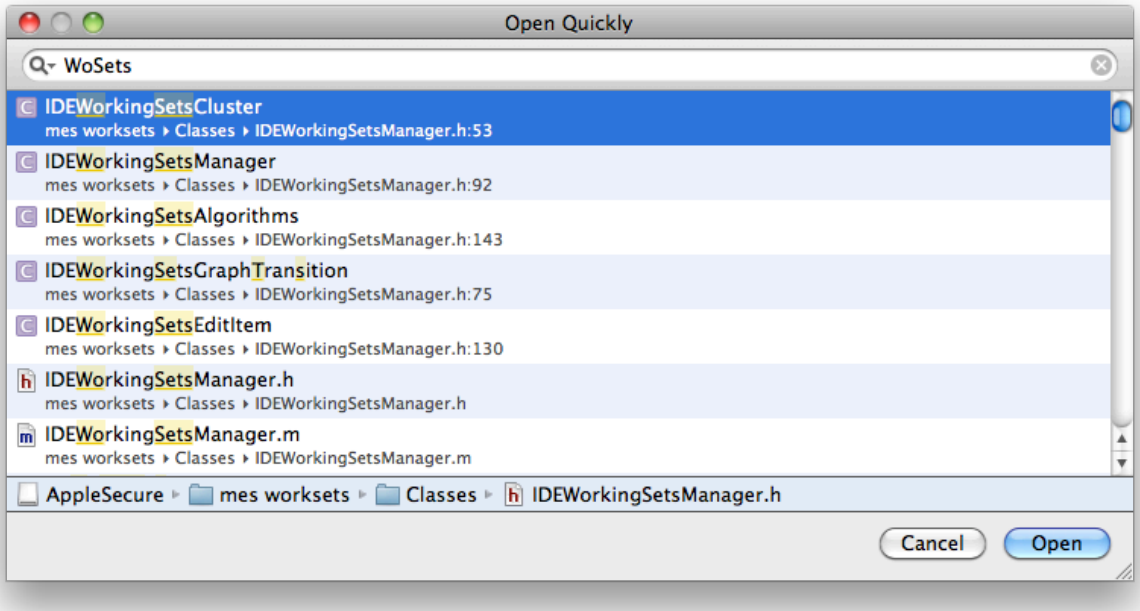
I did my internship in the infrastructure team. This team is at the origin of the development of the new version of Xcode. They are responsible for the architecture of the IDE (Integrated Development Environment) and providing other teams with an easy and robust way to integrate their work and features into the IDE.

My first project consisted in improving an existing feature, Open Quickly. With that being done, I explored the concept of working sets and developed a prototype to prove the feasibility of this project. Some work still needs to be done but I hope to see it shipped in the future.

Open Quickly

There are many tools to search for content in a project. In addition to the standard find and replace in the editor, Open Quickly works differently because it looks only for files and symbols (i.e classes and methods names) in your project. The purpose was make navigation easy throughout a project : instead of looking for the file you want to edit in the project navigator (the list of all your files in the left of the editor), you launch Open Quickly and you write the name of the file you want to open. This is very convenient when working on large projects where there are literally thousands of files.

To give a clear overview of the problem, we have the input of the user (a few letters, like ‘navMF’) and a list of all the symbols and files in the project, like ‘NavigatorMainFile.h’, ‘NavigatorOtherFile.h’, ‘Foo.h’, ‘main()’, ‘myFunction()’, ... We need to present to the user the list of symbols and files that match his input, and sort them by relevance. The user will then choose from the list the one he was looking for and it will open in the editor. Of course, we try to show at the top of the list what we think the user will click on. The rank of the result selected by the user is a good indicator of performance.



In the previous version of Open Quickly, the search was done by prefix. An input like ‘navMF’ would not have brought any results. So if you were looking for the file “NavigatorMainFile.h” you would typically have had to type in “NavigatorMa” to bring this result to the first position in the list (because of NavigatorOtherFile.h that could be in the first position too if you just input ‘Navigator’). Open Quickly doesn’t parse the code to know the symbols and method names in the code, it takes advantage of many existing tools (integrated debugger, ...) instead.

My work consisted in implementing fuzzy matching in Open Quickly, so that you need to write a fewer number of letters. In that case for instance, “navMF” would work to find the file that you want. I used a version of the Levenshtein distance that I modified to sort the symbols and filenames.

The Levenshtein Distance

The Levenshtein distance (LD) is a measure of the similarity between two strings, which we will refer to as the source string (s) and the target string (t). It was named after the Russian scientist Vladimir Levenshtein, who devised the algorithm in 1965. The distance is the number of deletions, insertions, or substitutions required to transform s into t. For example, if s is "test" and t is "test", then LD(s,t) = 0, because no transformations are needed. The strings are already identical. The greater the Levenshtein distance, the more different the strings are.

Example : If s is "test" and t is "tent", then LD(s,t) = 1, because one substitution (change "s" into "n") is sufficient to transform s into t.

The Levenshtein distance algorithm has been used in: spell checking, speech recognition, DNA analysis, plagiarism detection, ... to mention just a few of them.

Algorithm

The idea of the algorithm is that the Levenshtein distance can be computed recursively. For the ease of the explanation, let’s denote by s[0..i] the substring of s starting at index 0 with i+1 characters (= characters 0 to i) and by s[i] the

character at index i. Let n be the length of s and m be the length of t. Computing the Levenshtein distance of 2 strings is done with dynamic programming, and runs in time O(nm).

Changing s[0..i] into t[0..j] can be done

- either by deleting s[i] (cost : 1 operation, the deletion) and then recursively change s[0..i−1] into t[0..j]

- or by adding t[j] at the end of s[0..i] and recursively change s[0..i] into t [0..j−1] (cost : 1 operation, the addition)

- or by substituting s[i] by t[j] and recursively change s[0..i−1] into t[0..j−1] (cost : 0 or 1, the substitution, depending if s[i] = t[j])

The operation chosen is the one that minimizes the total number of operations.

We will have a matrix d of size m+1 * n+1 where at the end of the algorithm, d[i, j] = LD(s[0..i], t[0..j]) (this can actually be the loop invariant)

For i = 0 to m and j from 0 to n,
d(i+1, j+1) = min(d[i+1, j] + 1, d[i, j+1] +1, d(i, j) + cost)
where cost = 1 if s[i+1] ≠ t[j+1] else 0

When the algorithm terminates, d(m, n) contains the Levenshtein distance between s and t.

Adaptation of the Levenshtein distance to our problem

The algorithm described in the previous section is robust and pretty fast, but the Levenshtein distance is not well suited to our problem. Let us look at our example to see why : the target is NavigatorMainFile.h, we also have Foo.h. If we write nmf.h, we are at distance 3 of Foo.h (making 3 substitutions) but we are at distance 14 of what we have in mind. We understand that substitutions must be more costly than addition of letters. Furthermore the capital letters are more representative than the other letters so we want them more expensive to add than the other letters.

The Levenshtein distance works very well in spell checking because in that case you get a word almost right and you need to make one or two changes, not

really more. Here we try to find a way to get the user to write as few letters as possible to get him to what he has in mind.

We came with a handful of criteria and we changed the +1 cost of the Levenshtein algorithm by different parameters, among which

- In the case where the target is constituted of upper and lower characters, we want to penalize the capital case characters in the input that we have to substitute or delete more than if the target was only in lower case.

    for the target ‘afile.h’, the input ‘aBc’ would be less relevant than ‘abc’. By doing that, we want to say that when the user writes a capital letter, he really knows that the letter in the target will be capitalized. Filenames are usually made of many words and stringed together as in “OpenQuicklyDataSource.h”.

- when the user writes consecutive letters we want to give a bonus to targets that have matching letters in consecutive order too.

    for input ‘foo’, ‘fooa.h’ would be more relevant than ‘foao.h’ (though they all need one addition)

There are about 15 parameters based on capital letters, prefixes, punctuation, ...

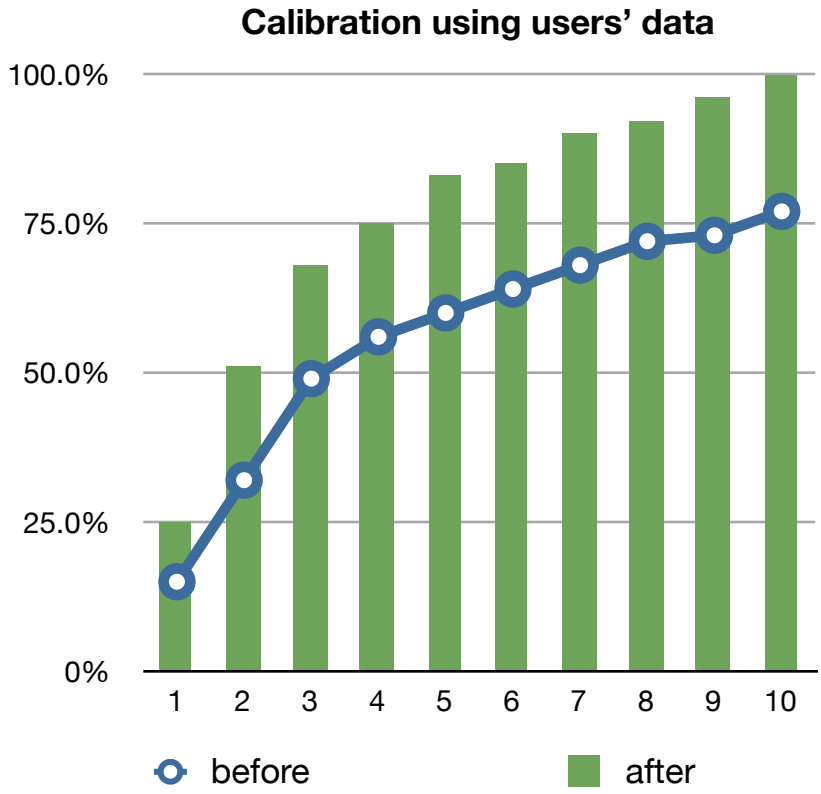
Another feature I have introduced is the memorization of the previous entries. The system keeps an history of the inputs and it will give a better score to entries that you were looking for. The memorization works in two ways, it records what are the inputs that lead to the choices of the user, and what are the most common choices.

Calibration of the parameters

Because there are many parameters (almost 20) and that the score function used to rank the results is non-linear (mainly because the Levenshtein distance is calculated recursively by taking the minimum of Levenshtein distances, and that the min() function is non-linear) it is not easy to find the right parameters. It is not so easy neither to write some tests. We would like to say “if we have this input, then this result must come before this one”, but because there are many parameters writing enough test cases is difficult.

The approach I took was to collect data from users in conjunction with writing a first draft of the new open Quickly version. Everybody in Xcode uses Open Quickly many times a day, so I have first set default parameters to the Levenshtein function, then released it internally with a little piece of code that dumped together with the input what were the first 50 results by relevance and on which one the user clicked, i.e which one was of interest. I asked everybody after a few weeks to send me their data file and it helped me calibrate Open Quickly much better.

That being done, I developed a tool that performs some statistical analysis, for a given vector of parameters, of the data I get. I can see how many times the most relevant result was the one that the user selected, what is the average rank of the selection of the user, ... I first tweaked the parameters manually to obtain better performance. I have also implemented a genetic algorithm to find a better solution but it took some time to make it work and the results were no better than the ones I had achieved manually. Having a tool that makes it possible to change the parameters easily and see at the same time what are the statistics we get out of these parameters was a great asset.



This chart shows, in percentage, how many results are in the n first results given back. For instance, 75% of the time, the choice of the user appears in our top 4 (after calibration), but it was in our top 9 before using our calibration tool.

Defuzzification

When the search in Open Quickly was done by prefix, it would show only words that start with the user's input and sort the results by increasing length. With the work I have done, all the symbols in a project are given a score and we show the first 50 results. But if the user enters a long input, many of our results will be totally irrelevant. We decided to show only results that contain all the letters in the input, in the same order.

Before defuzzification, writing 'date' would bring up 'NSDate' and 'NSData' in the results, but after it shows only NSDate. This is a bit regrettable because it would allow the user to make some mistakes in the input and yet find what he wants, but the cases like NSDate/NSData are rare after all and if the user makes some mistakes (which is rare with a keyboard) then he is likely to correct them. Plus if showing NSData when the user writes NSDate is a pretty cool feature, showing things that are too different in the list is not a good idea, it makes the list of results dirty. And because the defuzzification (in time O(n+m)) invalidates a lot of results before computing the modified Levenshtein distance (in time O(n * m)), the algorithm is faster. We could have said that we keep results where we don't need more than 1 suppression/substitution of the user's input letters but making the feature fast was more important.

This is how I improved Open Quickly. I got a lot of positive feedback on this. Let's now move on to my long term project this summer, which is exploring the concept of Working Sets.

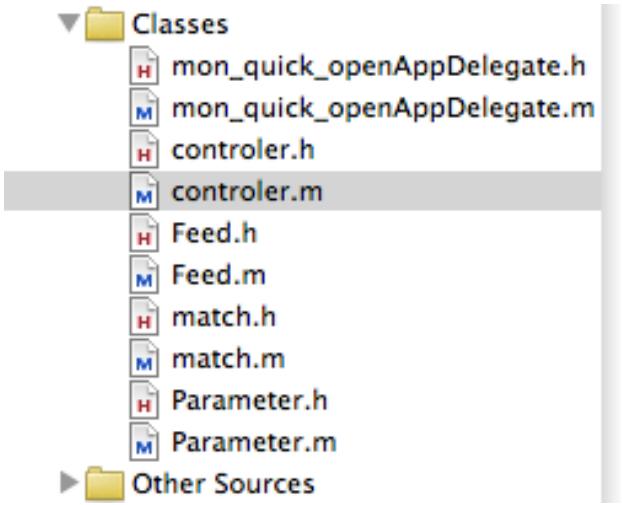
Working Sets

Let's talk about the concept of working sets, from what problematic it stems, go over some solutions we first thought of before dwelling on the answer we came up with to solve the problem. This will show how the thought process we went through.

Problematic

At any given time you are working on something and generally you have a specific goal. You may be fixing a bug or implementing a new feature or refactoring a bit of code to clean things up. The set of stuff relevant to what you're working on is specific to the task at hand. As you work you are finding the stuff you need to change by browsing your project, navigating to files through various relationships or doing batch searches.

Sometimes you get distracted from the task at hand and work on something else for a while, and later you go back to what you're working on. While you're working on other things, you're looking at different stuff and making changes to other things. When you get back to the first task, you may have lost your context. Sometimes the stack of discrete things you're working on gets deeper. You may work on one task on and off for a long time while many shorter-lived tasks come and go. Working Sets give you a way to let Xcode know what you're working on and to allow it to keep track of the context for each separate task.



In Xcode, files are accessible via the “project navigator”, which is a list of files and folders commonly used in most of the IDEs. This structure is very convenient to use most of the time when you work on small projects, but a bigger project can be made of hundreds of source files. The code is made of “bricks”, so there is one folder per feature and the source files related to the feature are placed in this folder.

However, it is not uncommon to work on files that are used in different features. For instance, there are some “global” files, that are used almost everywhere (the editor, the file that knows about the status of the main window and of all the files in the project, ...). If your two files are far apart from each other in the project navigator, you will need to scroll every time you navigate from one file to the other. So Working Sets keep grouped the files you want to work on.

A step toward a solution

Many attempts have been made to implement this concept, in Xcode 3 or in other IDEs though none of them was efficient and easy to use. Xcode allows this sort of thing today by explicitly creating and managing bookmarks. But people don't generally use this feature. Instead they find the stuff they need potentially over and over each time they need it. But a smart system for managing working sets can do a lot of the tracking automatically, and can therefore provide an easy way to get to the stuff for a given task that you've already found while working on it.

A working set is basically a collection of files. You can have as many working sets as you have ongoing tasks. One solution would be that whenever you start a new task, you can create a working set, then you add files to it manually. But we think that this would not work because all manual approaches were not used by the user. A manual approach involves some extra work for the user, so we decided to have something completely automatic. Working sets will be useful to the extent that the user does not usually have to manually manage their content. We believe that Xcode can watch what you are doing as you work and can automatically populate the Working Sets with the things you are working on.

Concept

When you are working on a small task A, you usually find yourself playing with a handful of files, the most recently opened ones. Changing task to task B suddenly doesn't change this list of most recent files, though if task B is not

new to you, the system should recognize it and retrieve the files you were working on at that time.

Recording the history of navigation will gradually populate a graph of navigation between files (the vertices), the edges being the probability of transition from one file from another. Based on these probabilities plus the recent history, the system must be able to suggest a reasonable list of "next" files to the user.

Based on further statistical analysis, the system can discover clusters of files that are all very likely to be used together. We call these clusters working sets.

Incremental vs Static

Let's say that we came up with a system that is able to understand and separate what tasks you work on, i.e that can do some pretty good clustering on the transitions probabilities graph. This is somehow a static way of thinking. It works well but is not ensured to work well "on the fly", dynamically. To understand why, we need to have the typical workflow of a user in mind. The user will work on one task, switch to another, switch back, ... Barring that a static approach can be very time consuming, it doesn't ensure any stability over time. If the system recomputes the list of clusters every time you navigate from one file to another, very likely there are some files that were in your working sets and that will disappear after a few steps of navigation, then reappear, ... The behavior is not predictable, though we can think of algorithms to smoothen this. Instead we chose to use an incremental approach, built on our algorithms to compute the list of clusters right from scratch.

Model

In most of the research papers I have read before starting my project, clustering is done by using matrices tricks, such as SVD (Single Value Decomposition). Most of the time the size or the number of the clusters is known beforehand, and an element cannot be in more than one cluster. This doesn't hold here. We need to device a system where a file can be part of many different Working Sets, and that is easily mutable into a system that allows for incremental changes.

As I said previously, we started to design an algorithm that shows what are the files you are most likely to go next, knowing what file you are currently on. We

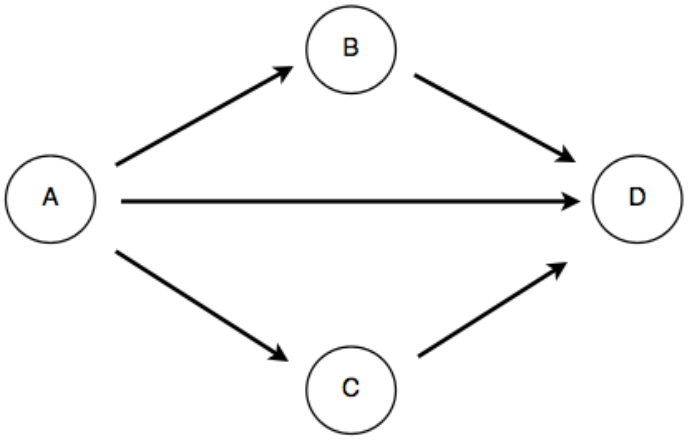
did this at a time where we didn't know yet what our final idea would be. There is a menu in Xcode that says what are the recent files, what are the superclasses and children of the classes in a file, ... and we first wanted to have a "most relevant files" menu, or something similar to that. This feature and the algorithms involved finally appeared to be of paramount importance in the clustering process. Let us describe how we compute the list of files relevant to another one _in other terms we will define here the distance we use in the files set_ before talking about how our clustering method works.

Defining a distance in the files set - Files relevant to another

In the source code of Xcode, we made some changes to record all the transitions from one file to another. It is not hard to create a graph where the vertices are the files and the edges are valued with the probability to go from one file to another. Actually, in the real world, nothing is as easy as it is to say it, for example, after finding where the navigation occurs in the source code and where you have to add your lines, you should pay attention to what the user does, like renaming a file.

For each vertex, we have a table of all the connected vertices. If we were to associate each vertex with a float number (the probability), every time there is a new transition from the file to another you have to update the probabilities of all associated vertices. This is not a good model (time : O(n) to update all the probabilities, where n is the number of files in your project). Instead, we associated every edge with an integer value, the number of times you go from one file to another. Every file also has a 'total' value that sums all the edges values. When you record a new transition, you add one to the total and to the edge. If you want to know the probability you just need to divide one by the other, in time O(1).

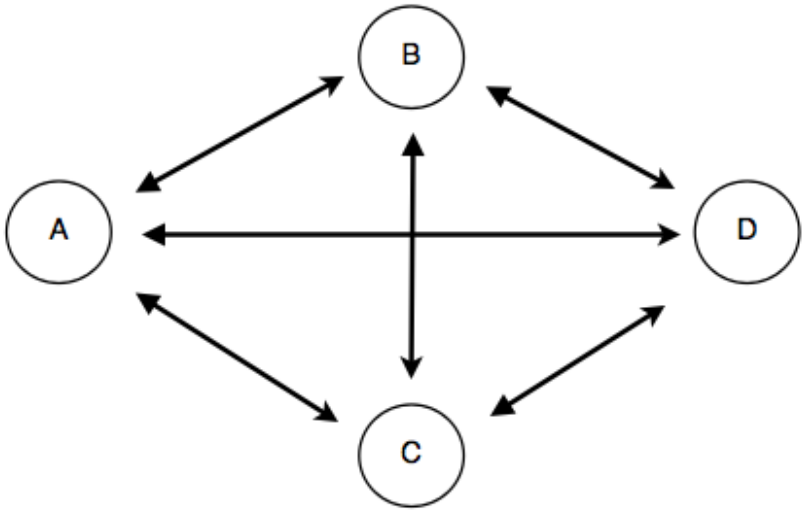
There are many ways, given the matrix of all the probabilities of transition from one file to another, to define the distance between 2 files.



Let's consider this graph and see how we can define the distance between A and D. The first idea was to add all the probabilities of all the possible paths from A to D. This would result in adding all the probabilities between the arrows in the graph above : D can be relevant to A because B is relevant to A and D is relevant to B for example, so this idea is pretty intuitive.

We would like to say something like : score = Pa,d + Pa,b * Pb,d + Pa,c * Pc,d

and the distance will be any decreasing function of the score. But in reality, the graph of navigation can contain loops, and it is more likely to look like



In that case,

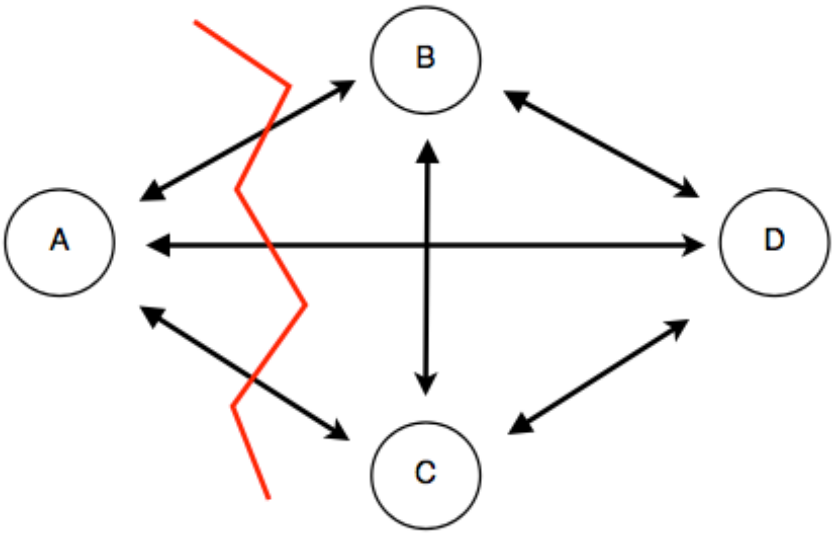
score = Pa,d + Pa,b * Pb,d + Pa,c * Pc,d + Pa,b * Pb,a * score + Pa,d * Pd,a * score + Pa,c * Pc,b * (Pb,d + Pb,a * score) + ...

We end up with a bigger equation, and yet we consider here only 4 vertices.

If we call M the matrix of probabilities of transitions, then the score we are looking for is (I + M + M^2 + ... + M^n + ...) at row A and column D.

Indeed, we have by definition Ma,d = Pa,d, and (M^2)a,d = probability to be on file D, starting from file A, with a path of length 2. Because on every line the numbers in M are between 0 and 1 and their sum is 1, M has an (infinite) norm of 1 and we know the sum will not converge : M(1) = (**1**) (a vector with only ones) so (I + M + M^2 + ... + M^n) = (n+1) * (**1**) and thus the sum doesn't converge globally but may converge on certain coefficients. A solution would be to calculate partial sums and to take the result as granted for the distance, it should give a relevant score function.

Another way to view the problem was to see what is the biggest cut in the graph that we can make, with A as the source and D as the sink. A cut would somehow naturally add the weights of certain edges, hence considering all the paths from A to D all at once.



Both of these algorithms could be developed but it seems that it would take too much time to compute all the distances between A and X for all X using these methods. Instead we use something faster and yet intuitive enough.

Somehow, the bigger the probability between two files, the smaller the distance. We will use a "shortest path approach" to give the distance. The

distance between two files will be the distance of the shortest path. It doesn't take fully into account the "addition" property of the probabilities, but this algorithm is fast and robust and still makes sense : the more you have some connections, the easiest it is to find a path and the shortest the distance in a certain way.

Firstly, let's define the distance between a and b by $D_{a,b} = -\log(0.05 + 0.9 * (0.6 * P_{a,b} + 0.4 * P_{b,a}))$ if they are connected by an edge, else, the sum of the distances in between the edges of a shortest path connecting A and B.

The formula seems far-fetched at first sight so let's give a brief explanation :

- at the beginning, the formula was $D_{a,b} = -\log(P_{a,b})$. The minus sign (-) is here so that the distance is a decreasing function of the probability. The log is here because probabilities are objects that you tend to multiply along a path whereas you add distances. If the probability to go from A to B is $P_{a,b}$ and the probability to go from B to C is $P_{b,c}$ then $P_{a,b} * P_{b,c}$ represents the probability of the path A, B, C. If we had defined $D_{a,b} = -\log(P_{a,b})$ then $D_{a,c} = -\log(P_{a,b}) - \log(P_{b,c}) = -\log(P_{a,b} * P_{b,c}) = -\log(P_{a,b,c})$ and this formula would work.

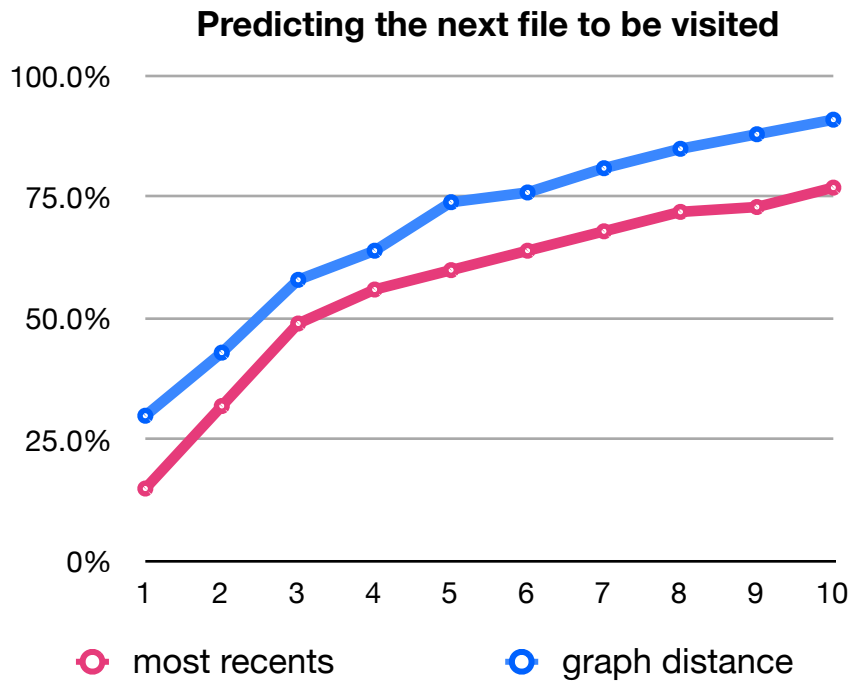
- $P_{a,b}$ can be null, in that case $D_{a,b} = +\infty$, which is fine in theory, but in practice we want to avoid that so we take $D_{a,b} = -\log(0.05 + 0.95 * P_{a,b})$

- somehow we want something decreasing with the number of steps in the path. If $P_{a,b} = 1$ and $P_{b,c} = 1$, then $P_{a,c} = 1$. But it means that in reality, starting form file A, we always go first onto B then onto C. Thus C must be further away than B from A. This is done by having 0.9 instead of 0.95. It has the same effect than saying $D_{a,b} = -\log(0.05 + 0.95 * P_{a,b}) + cste$.

- We choose $0.6 * P_{a,b} + 0.4 * P_{b,a}$ instead of $P_{a,b}$ to take into account the symmetric of the graph. Somehow if you go very often from file A to file B, then it makes sense to suggest A as a file close to B, though you might prefer files X that have a high $P_{b,x}$ probability.

Results

I recorded the history of navigation from people in my team, on the Xcode project (thousands of files). Then I compared at every time the relevance of the most recent files and the relevance of the files closest to the file being edited ("closest" with respect to the distance in the graph of transition, that I have just described).



This chart shows, in percentage, how many times the next file visited by the user will be in the top n files in the respective lists. For instance, 75% of the time, the next file visited by the user appears in our top 5 (with out distance function), but to get the same percentage we need to look in our top 9 with the recent files list.

Performing clustering

The cluster “Others”

The most important rule in our clustering strategy is that we want all the files to be in at least one cluster. This makes it easier when it comes to choose for a file what is the cluster it fits into best for example. The problem with this approach is that certain files can’t belong to a working set for many reasons : either the file is very “new” and the system doesn’t have enough information on what are the related files, or a file is part of a cluster that grows and grows and at some point we need to “clean” the working set and so we need to assign another working set to the file, ...

For this reason we imagined the working set “Others”. This working set will be treated differently than every other clusters. For instance it will not have any constraints on its size. At some point certain clusters can merge, so one of the

two disappears, but the working set “Others” is different and will always exist. In a certain way, “Others” acts like a trash. When a file is added to another working set, it is usually taken out of “Others”.

How our clustering method works : we use the distance function we defined earlier in our set of files. The 10 files closest to a file (relative to this distance) are the 10 most relevant files with respect to this first file. While doing a static clustering, we compute for all the files what are the 10 most relevant files. For all these 10 relevant files we do the same thing. We now have 10 lists of 10 files each. If we find more than 7 files that are in 80 % of these lists then they form a cluster. If the cluster found is “similar” to another cluster, then we merge them.

relevant files for A1	for A2	for A3	...	for B1
A1	A1	A1		
A2	A2	A3		B2
A3	A3	A4		B3
A4	A4	A5		B4
A5	A6	A6		B6
A6	A7	A8		B8
A7	A9	A9		B9
A8	B2	B1		A2
B1	C2	B4		A8
C1	C3	C2		C2

In this example, files A1, A3, A4, A6 are part of most of the lists, so we decide to aggregate them.

That being done, some files are not part of any cluster. We compute the list of the 10 most relevant files for each of them and treat it as a cluster. We then assign the file to the cluster that is the most similar to this latter, if the measure of similarity is bigger than a certain threshold. If not, we assign the file to a special cluster, the “Others” cluster. Certain measures are described further on this report.

Size

Clustering has nothing to do with magic. We could have said there is just one big cluster that contains all the files in a project. This would be true and yet totally useless. Given the context we are working on, that is to say to find the clusters of files that are likely to be used together in a software project, we

stipulate that the size of a cluster will be of about 10 files. Making this assumption, we will try to make clusters with a size between 6 and 15 files. This means that we will not accept clusters containing more than 16 files or less than 5 files. Every time we add a file to a cluster, merge two clusters, separate a cluster into two, ... we will check that this size rule is respected. The working set “Others” is the only one that has no size constraints.

This explains for the “static” clustering. The incremental clustering is somehow based on it, but with additional rules made to address problems that arise during user’s navigation, in contrast to static analysis.

Incremental Clustering

One of the main problems is to ensure stability. If we were to recompute all the clusters / working sets every time an element of navigation appears (i.e going from one file to another) then firstly it can take a lot of computational time, and secondly we are not able to ensure stability. When the user goes from one file to another within a working sets, we do not wish to recompute all the working sets. The primordial question is “What happens when you are going from a file ‘a’ in a working set A to a file ‘b’ in a working set B ?”

This question, among several other fundamental ones, is answered by our rules and the set of functions we have defined to work with the clusters.

Every time we navigate from one working set to another, we could merge the working sets together, add the file we come from to the working set we go to or reversely add the file we go to to the working set we come from. The choice we make is purely based on certain metrics we developed.

<u>Some useful metrics</u>

- measure of similarity of two clusters C1 and C2 = $|C1 \cap C2| / |C1 \cup C2|$

  When creating a new cluster, we can wonder if there isn’t already a cluster that exists and is similar to it. If so, we might merge them if the size rules stay valid.

- score of a file ‘f’ in a cluster C = $|C \cap L| / |L|$ where L is the list of files relevant to ‘f’.

When we consider adding or removing a file to or from a cluster, we consider both the scores of the file in the ‘new’ cluster and in the ‘old’ cluster.

- integrity of a cluster C = #{ file f / L(f) ∩ C > .7 * #C } / #C

L(f) is the list of files relevant to file f. Before adding or after removing a file from a cluster, we can consider whether the cluster is still valid or if it will be, and it depends on the score of this function.

Incremental workflow

Another thing worth paying attention to : there are several ways to open a new file, one using our working set navigator, and others. When using our working sets navigator, the user, without knowing it, gives us an important piece of information : he will open file ‘a’ in working set ‘W’. While using other navigators, we just know that he is know editing file ‘a’. In case ‘a‘ is part of only one working sets, there is no real issue, else we need to think about it more thoroughly. So, if there is no explicit information, if ‘a’ is part of the current working set, we assume the user is doing intra cluster navigation. Else we look for the last working set ‘a’ was used in. If there isn’t any, we look for the cluster that fit ‘a’ the best.

Last but not least, because the working set “Others” is special, there are some special cases everywhere in the algorithm. So going from working set A to the working set Others doesn’t try to take the file you are coming from into the working set Others. When we visit a new file, i.e a file that we have never visited before, this new file is included in the working set Others. More navigation steps from this file to files from another cluster will eventually reinforce the probability to go from one

This section is voluntarily brief on the exact description of the algorithm and above all on the different thresholds used, in a non disclosure agreement purpose.

Further steps and Critics

Tests

There are many clustering algorithms in the street, and it would be a good idea to take some time and test many of them. Also, the algorithm we chose is made of plenty of functions with many parameters. Choosing a different distance function, changing the threshold parameters, ... all these tweaks that we could do would change the results of our algorithm.

One of the most important thing to do would be to record enough data to test the algorithms. This is a burning issue. It is not possible to say that an algorithm works well when it is not massively tested. The framework that records all the transitions from one file to another is already there, but people on the team didn't have much time to use it, so there is very few data. Furthermore, we are the team building the new version of Xcode thanks to Xcode, so there are new versions everyday and people delete their personal data before committing their changes to the trunk. What need to be done is sit with someone, look at his at least 2 weeks data history, determine what are the working sets manually and test various algorithms on them. And do that for more than one person.

Time

Let's imagine that you work a lot and for a long time on certain files. Then you add a new file that needs to be in this working set. Because there is so much navigation between the other files, it will take some time for the new file to have strong links with the preexisting files. It would be interesting to relax the probabilities over time, as if every time you navigate from one file to another, it adds +1 to the edge, but at the same time multiply all some other edges (all the others or just all the others in the working set) by 0.98 for instance, or find a way that follows that intuition.

Another thing that we record is the time that you spend on a file. So there are some times you will spend 2 seconds before going to another file, and some times where you will do lots of edits and spend 20 minutes on a file before going to another one. We don't use this parameter in our current model but adding +1 or +.2 or anything else regarding the time we spend on a file is a point of interest.

Memory Management

The graph of probabilities is currently implemented as a sparse matrix, meaning that instead of using a matrix we use hash tables that for a given file return another hash table, consisting of the destination file and a number.

This graph can become huge and in order to reduce the footprint of user data for each project, it could be interesting to save just the cluster information. One way to do that would be to save the list of working sets when the user quits Xcode or closes the project, and when he reopens it, it creates a new graph of probabilities, and populates it with edges with default values for all the pair of files belonging to the same working sets. This would address the first “time” problem.

SVN

It would be a great if working sets could be shared among users of a same project. There are scores of software engineers working on the development tools such as Xcode, and you could be debugging the code of someone else. In that case, you would like to know what are the files relevant to the files you are editing, and if working sets were shared it would make it easier.

Conclusion

I was very happy to work at Apple as a Software Engineer Intern. I have learnt a lot about the software development process in a leading company in this sector.

It was also thrilling to explore a domain completely new to me and probably to the other IDEs on the market. Clustering, predicting the next move of the user and try to help him with the navigation, are two very interesting and passioning challenges I had to face.

If the time was too short to make the project go all the way to its end, I hope my work will serve in the near future. Nothing is more exciting than seeing the features you have developed ship and use them in “real life”.