CSC5021 -Software Engineering for Smart Devices

Project Report and Analysis - http://github.com/csotiriou/gcd-proj

Christos Sotiriou
MSc SAI

Introduction

This document serves as a supplementary deliverable for the final project for “Software Engineering for Smart Devices”. The goal of this exercise is described in “http://www-public.it-sudparis.eu/~gibson/Teaching/CSC5021/Project-CSC5021.pdf“.

The deliverables of this project are publicly available in http://github.com/csotiriou/gcd-proj .

NOTE:
Although this project is intended to be delivered for the University assignment, it is intended to be expanded with more classes and code. It is part of a much larger public project. As such, there are some features / classes that were not part of the assignment, but were added because they would facilitate development and personal testing. These classes are generally not well-tested. For a list of these parts of the deliverables, see the “What has been left out” section.

Project organisation

The deliverables are comprised of an Xcode workspace consisting of two Xcode projects.

- **LatticeCreator** - Builds the CSMatrixFramework framework, which is a library for the project data manipulation and classes that facilitate loading, like the basic Lattice class, and importers / exporters. It comes with its own tests.
- **CocoaGCDTests** - Builds a cocoa application that exploits the main classes, and pattern matchers, multithreaded and sequential.

Frameworks / Technologies used

- Xcode 5.0 (With XCTest framework)
- Expecta Framework (https://github.com/specta/expecta) for having elegant expressions in unit testing
- TRVSMonitor (https://github.com/travisjeffery/TRVSMonitor) for efficient asynchronous unit testing
- Cocoa Lumberjack (https://github.com/CocoaLumberjack/CocoaLumberjack) for efficient logging (faster than Apple's implementation)
- Doxygen is used for documentation. A DOXYFILE is included with the source code, although it may need some alterations to generate code, depending on the target machine (especially for creating the Class diagrams, for which an external tool called 'dot' is used, part of the 'graphviz' package. Easiest install: Use MacPorts)

Hardware

- Testing machine 2: Quad Core i7, OS X Mavericks, 16 Gbytes RAM
- RAM Requirements: 1,2 GBytes Maximum, for a 1000x1000x1000 cube and 100 words loaded in RAM for searching

Pattern Matching process

Pattern matchers are responsible for finding patterns in a cube and returning results / calling callback methods in the application. The general methodology used by the pattern matchers in the application is to construct strings for each line found, and for each line the matcher searches if the strings that are to be matched exist in the line. Each line is constructed by accessing the elements of the cube in certain direction (e.g. directions, X, Y, Z, with their contraries, and also diagonal directions).

All pattern matchers can be used both with the delegate pattern and GCD Blocks for notifying when the job is finished.

Optimisations that were deliberately not done

There are some optimisations that were not done during the pattern matching process. Those were deliberately not done, since the goal of the project is to analyse different aspects of multithreading, and not be as fast as possible. Some of the optimisations that were left out from the development progress would cause non-linear completion time and results of the pattern matchers, and are considered to be out of the scope of this project. Such optimisations include:

- Smart line extractor (LatticeLineExtractorSmart). A smart line extractor that would not return lines in which it would be impossible to find the strings that we are willing to search, by taking into account the length of the strings, and performing various sorting techniques.
- Force thread number that are active at any given moment. Grand Central Dispatch relies in the XNU kernel to decide how many threads will run. That means that in different machines with different capabilities the number of active threads for this application will be different. The only way to cope with these limitations is to use NSOperations and / or NSThread objects directly.

Process structure

The basic process of the pattern matchers for this project considers the following steps:

- For each direction searched in the cube, create a lines of strings comprised of the characters found in the cube, to be processed.
- For each line, compare each of the words inside the dictionary with the line, to find if they exist.
- For every word that is found, put a flag concerning this word to ‘true’.

The same idea is followed by all three pattern matchers. What differentiates each pattern matcher is how the jobs are distributed between threads.

- **The sequential pattern matcher** uses one thread, which sequentially makes all operations in the main thread.
- **The first asynchronous pattern matcher** assigns each thread a different direction for searching, and compares all words to search with each line found in this direction

- The **second asynchronous pattern matcher** divides among threads the words to be found, so each thread searches only a specific subset of the words to be found, in all directions. The threads that are scheduled are equal to the number of processors / cores of the machine.

When a word is found, a dictionary holding the results (with <String, NSNumber/BOOL> as key-value pairs) is updated.

For ensuring thread safety, two Grand Central Dispatch queues are used.

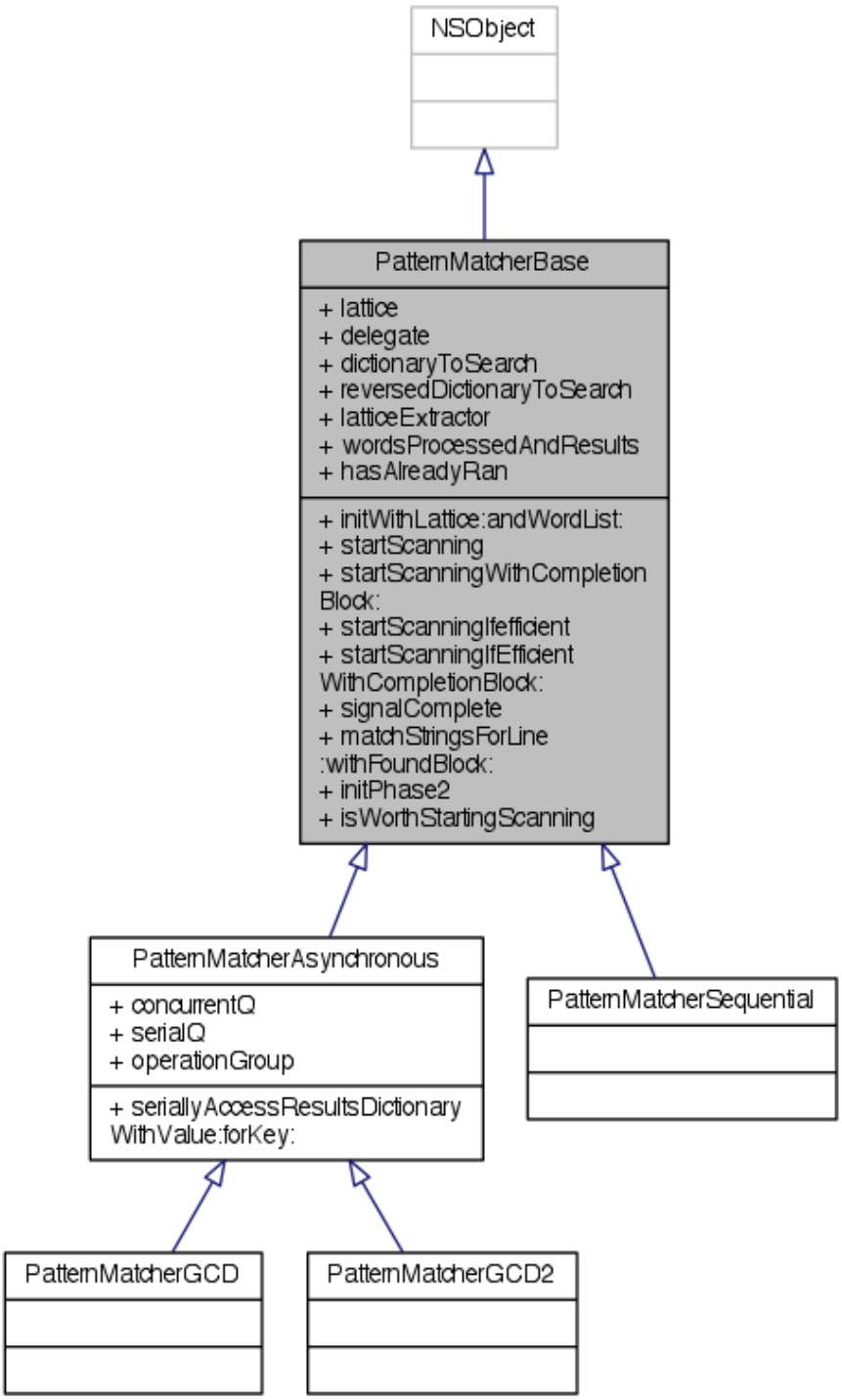
- **Background Process Queue**. A concurrent queue handling background threads. The results of each thread are handled by the serial queue, as they involve accessing and modifying of sensitive resources.
- **Serial queue**. For ensuring access and changing resources that are prone to collisions between threads (like the dictionary that holds the results).

Pattern Matcher Class Structure

As previously stated, the goal of this architecture was to be as modular as possible, in order to add and remove more pattern matchers after the delivery of the project.

The PatternMatcherBase class is defining the base functions that other pattern matchers will use for this project,. Such functions include the call to any delegates to notify for completion, the basic search iteration when a line is found and the check if it is meaningful to start a scanning process.

Below is a diagram showing the inheritance tree of the basic patten matcher abstract class.



Acceptable inputs

CSWordlist is responsible for ensuring validity of the words inserted for searching into a pattern matcher. As of the time of this writing, it is configured to accept strings that contain valid characters from the following string:

```
!\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
\]^_`abcdefghijklmnopqrstuvwxyz{|}~©≥¢±£≤
```

During creation of a pattern matcher, the word list object is passed as the argument. The handling of the available inputs was assigned in a separate class, in order to assign more responsibilities and flexibility. More specifically:

- Eliminate duplicates. Each wordlist cannot contain more than one instance of each word to be searched.
- Eliminate the possibility of providing a string with length more than the other words. All words should have the same length
- Import a wordlist from a file (.wdl), as seen in many examples during testing. Also, to import specific number of words from a file (also useful in unit testing).

Each pattern matcher has the same exact initialisation, setup and functionality from an API standpoint. The difference between each one is what jobs do they assign to each thread and how many threads they use. For each pattern matcher, the only basic difference is the “-startScanning” method of each class, and how it is implemented.

Performance Analysis

NOTE 1: The fact that the testing machines have 4 processors, **does not guarantee anything regarding the number of threads operational at any time**. The asynchronous pattern matchers are allowed to have 4 threads operational in the i5 machine, while in the i7 machine they are allowed to have 7 threads operational. The following tests are performed in the i7 machine. However, for the second asynchronous pattern matcher, the number of threads cannot exceed the number of cores / processors for this machine (this is by design). Grand Central Dispatch uses the XNU kernel to decide how many threads should run at a specific time. The criteria for such decisions are not clearly defined to developers. (see: https://developer.apple.com/library/mac/documentation/general/conceptual/concurrencyprogrammingguide/OperationQueues/OperationQueues.html#//apple_ref/doc/uid/TP40008091-CH102-SW6

NOTE 2: **The timing results are taken depending on how long it takes for a single test operation to finish (based on the XCTest framework by Apple)**. There are some macros showing how much it takes to complete some sub-operations, but these are only for developer testing (to identify bottlenecks) and they are based in the C function 'clock()', which is not totally accurate.

Lattice of size: 100x100x100, word length = 10, word count: variable, time measured in seconds.

	Sequential	Async 1 (7 active threads)	Async 2 (4 active threads)
10 words	2.446	1.210	1.211
100 words	13.554	9.445	8.843
500 words	77.679	50.320	50.707
1000 words	125.384	87.253	82.511

The bottleneck is clearly in the word processing, at the point where we have a line, and we iterate through all the available words to search in order to find if a word exists. The performance of the asynchronous pattern matcher of type 2 is a result of both the repeating work needed to be done by each thread (each thread creates all possible combinations of lines in all directions, and then processes some of the total words), and also because of the fact that there are less threads operational (4 in the case of the testing machine, when compared to 7 with the asynchronous matcher of type 1).

Lattice of size: variable, word length = 10, word count: 10, time measured in seconds.

	Sequential	Async 1 (7 active threads)	Async 2 (4 active threads)
10x10x10	0.013	0.142	0.137
100x100x100	2.630	1.313	1.323
500x500x500	255.277	75.351	80.157
1000x1000x1000	1811.937	531.001	527.443

Other bottlenecks observed:

Constructing strings representing the diagonal lines is a too expensive operation, and takes considerable more time than creating strings from the straight lines. This happens because of the algorithm complexity difference between the two approaches. While constructing strings with straight lines, one loop is required, whereas in constructing strings for diagonal lines, 2 loops are used, 1 for each half of the square lattice processed, thus doubling the algorithm complexity.

Lattice of size: 100x100x100, word length = variable, word count: 40, time measured in seconds.

	Sequential	Async 1 (7 active threads)	Async 2 (4 active threads)
10	2.381	1.414	1.211
30	2.640	1.521	1.613
70	2.477	1.216	1.214
100	1.984	1.210	1.013

By increasing the number of characters in strings to be searched and matched we see that there is a decrease in the pattern matching process. This happens because for the string searching process the '-rangeOfString:options' function is used in the NSString class. The algorithm performs better when searching larger strings inside another string of constant size, because the larger the value searched for, the less distinct possibilities exist in the string to be searched. For example, in an string with 10 characters, there are 9 different possible positions of a string with length 2 to be found, but if the string to be found had 10 characters, there would only exist one possibility (the whole string).

The amount of RAM need when processing a 1000x1000x1000 cube is 1.256 GBytes, with minor differences between Clang compiler versions and size of wordlist input. The memory never goes above this point, since AutoReleasePools are used to deallocate memory allocated for temporary objects that are used for pattern matching.

Conclusion

In total, the most expensive operation and seems to be the line processing, and constructing strings from each line. While changing other aspects of the problem seem to have linear effects on the time taken to complete the operation, changing the lattice size seems to grow the problem size and the time taken to complete the pattern matching exponentially.

To this seem to contribute a lot of factors. First of all, the internal representation of the cube, as seen in DNALattice1D is using a 1-dimensional C array, in which values are mapped as if they are coming from three dimensions. Although this seems to simplify operations like copying the lattice and its contents (it's simply a matter of a call to memcpy()), it also poses an extra overhead in storing and retrieving the data, since it must calculate each time where the data to be retrieved is stored, by making some multiplications fro accessing each element. Moreover, due to the dynamic binding of Objective C operations in memory, there is also an extra overhead when calling functions to retrieve, when compared to calling functions in C or C++.

Also, one would think that since the multi-processing pattern matchers use more than 2 threads, it's logical to have much increased speed (for example 6-7 times faster than the sequential matcher), however this is not true. Since all pattern threads are using a structure and a class that is in the same point in memory, Grand Central Dispatch needs some time to prioritise in sequence access to these resources ("queue" them), and that produces some overhead.

Possible improvements:

- Create a Lattice class that is represented by a 3-dimensional array in memory. This will have as a result the ability to grab large consequent chunks of memory at once in order to form strings to search, and construct strings easier, without many function calls. Some amount of effort would still have to be done for getting the diagonal lines, however.
- Use C++ for representing a 3D cube. Although, without measurements and good care, it is not sure how faster than Objective C this approach would prove to be.
- Allow access to the internal memory of the 3D cube, so that pattern matching can use direct calls to the memory, thus avoiding function calls to the encapsulating class. This would break encapsulation and would be more error prone, however.
- Change the pattern matching process to use a custom string search method, so that constructing a large strings representing lines of the cube in a certain direction would not be necessary. That would require a large optimisation effort however, since Apple's string searching methods are a mix of common high-performant algorithms that are utilised depending on the arguments (needle and haystack) and their properties.
- Pre-process the cube created by pre-creating the lines to be processed. This, however, would reduce the actual pattern matching operation overhead, but it would transfer it to the pre-processing stage. Also, it would require almost 7 times the amount of RAM needed by the current implementation (because it would store the lines of 7 distinct directions to memory), and that would be catastrophical, since the current implementation uses 1GByte of RAM for a lattice 1000x1000x1000.

What has been left out / Disclaimers

- **CSMatrixImporter** - The importer class was not in the original requirements, and unit tests are not provided, due to lack of time. It is a helper class, for facilitating development.
- **CSMatrixExporter** - The exporter class was not in the original requirements, and unit tests are not provided, due to lack of time. It is a helper class, for facilitating development.
- **DNALattice3D** - The DNA Lattice in 3 dimensional representation was not necessary in order to construct the application and the tests. It is not even completed, and not used at the time of delivery. Thus, it is not tested. It exists only because it will be implemented later, as an extra component.
- **LatticeLineExtractorSmart** - The smart extractor has not been implemented for this project. Use the LatticeLineExtractor only. Having the smart version would possibly produce smarter but non-linear results in terms of performance, which would make the project difficult to asses.
- **The Cocoa user interface provided IS INTENDED FOR PERSONAL USE**, and was created as a demonstration of how the classes and the pattern matchers can be used in a Model-View-Controller hierarchy, with callbacks.

Directions that are NOT searched

The diagonal direction that spans in the Z and Y axis is not taken into account. Due to heavy load, the testing procedures were not completed, and thus, I did not include the search in this direction. This excludes 2 directions from the search, and their opposites (total 4). The number of total directions that are included is 14.

Compromises with the requirements

The minimum constructible cube size is 3x3x3, and not 4x4x4. This is intentional, because at the point of development it was much easier testing the extraction of the lines and identify problems when using 27 characters, than when using 64 characters (which would include special characters).