Improving performance of prototype recognition in Prefab

By

Orkhan Muradov

A senior thesis submitted in partial fulfillment of

The requirements for the degree of

Bachelor of Science

With Departmental Honors

Computer Science & Engineering

University of Washington

June 2011

Thesis and presentation approved by

_____

Date

_____

Abstract

Because most graphical user interfaces that we see nowadays consist of various buttons or window building blocks, Prefab helps to recognize them by looking only at raw pixels. Prefab works by reverse engineering an interface and building a tree structure representing that interface. This application has a great potential for identifying the structure of interfaces and customizing them in new ways. Most of the applications that we use on daily basis have many widgets and buttons, and whenever we use them we operate very quickly by scrolling and switching from one window to another. In order to identify all the parts of the interface correctly, the speed of this application is very important.

1. Introduction and Motivation

Using Prefab we can now interpret pixels of an existing application and modify them independently of their underlying implementation. This help researchers test and deploy their interaction in the context of real applications. Pixels are first copied from a source window and interpreted using Prefab. Then various enhancements can be added, with input then mapped back to the source window [Figure 1]. Papers presented at CHI 2010 [1] and CHI 2011 [2] have demonstrated many potential enhancements: dynamically expanding the motor space with bubble cursor, visualizing state changes with phosphor, parameter spectrum previews with side views, presenting software tutorials, adaptive GUI visualization, language translation, user interface customization, and more. All these interaction techniques recognize widget layout in real time by executing this pixel-based feature identification cycle multiple times.

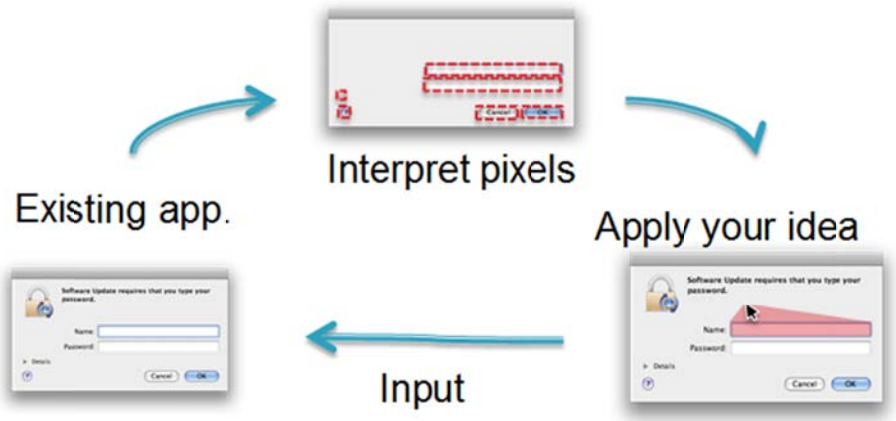


Figure 1

Prefab's pixel-based widget identification cycle

Successfully executing this cycle requires a method for rapidly finding features in an image. Prefab's original implementation uses a decision tree [Figure 2]. This Build Tree approach first chooses a non-transparent pixel hotspot pixel for each feature in the library. Then Prefab builds a decision tree in which every node stores an offset relative to the hotspot and every edge represents the color of the pixel at that offset.

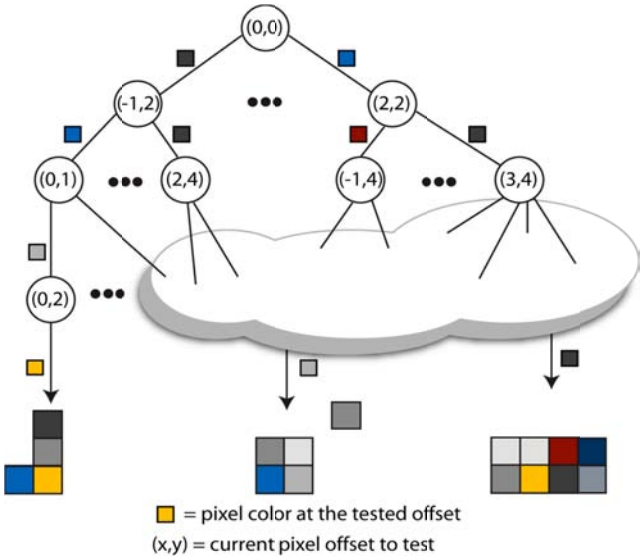
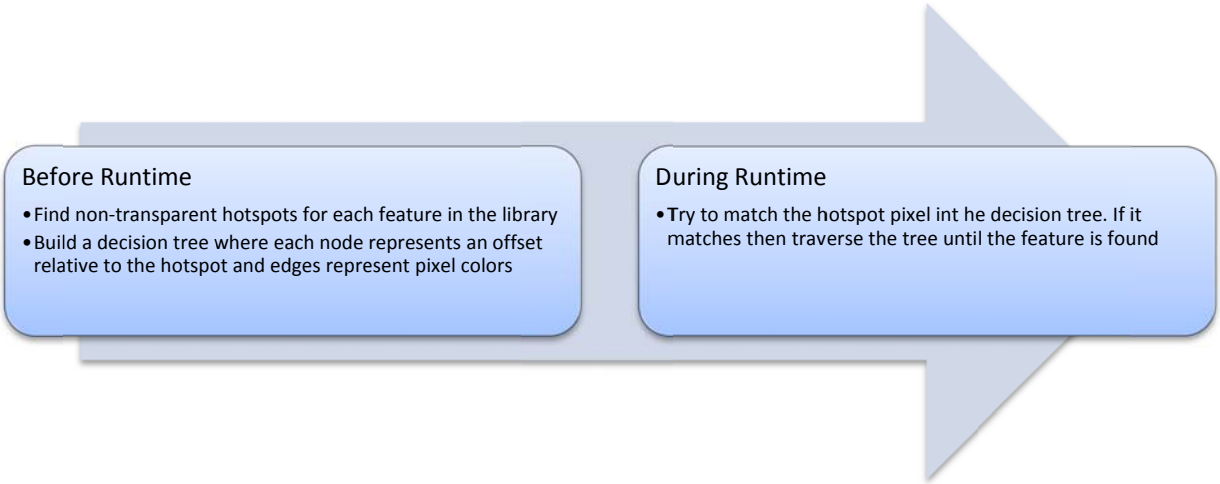


Figure 2

Decision tree of a Build Tree implementation

During runtime this implementation scans through the image only once by checking if the pixel is the hotspot pixel of a feature. It moves through the image and traverses the tree until it finds a matched feature. If the pixel doesn't match a pixel in the decision tree, then tree traversal ends. The multi-threaded version of this implementation divides a bitmap into segments and uses multiple threads to look for features.

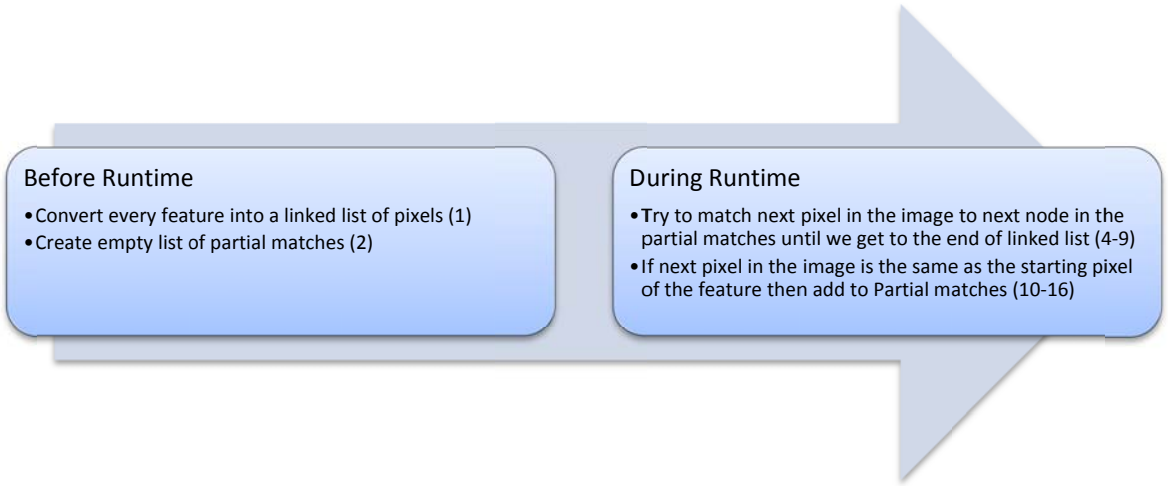


2. Examining Strategies for Finding Features

It is difficult to say how reliable the initial algorithm is because Prefab's current library of prototypes is relatively small and there are no other algorithms to compare with. Because performance of feature detection is very important for Prefab's functionality, we developed other algorithms, benchmarking software, and a larger library.

2.1 Partially Matching

Before the execution all the features are converted into a linked list of pixels. The last node of this structure is a match node that specifies that a feature was found. The program visits each pixel in the main image and if a pixel matches the first pixel of the feature in the data structure then this feature is added into the partial matches data structure. As we iterate through the image we add more features to the partial matches and advance corresponding nodes in the partial matches.



The goal of this approach was to visit each pixel in the image only once, therefore improving the performance. Because the initial collection of linked list features could become very large, we started using different data structures in order to improve retrieval time. Pseudo code below describes the algorithm:

```
1     Build a list of features named START
2     Initialize a list of features named PARTIAL
3     FOR each pixel in image
4         IF pixel matches the first pixel of feature in PARTIAL THEN
5             Move to the next node of linked list in PARTIAL
6             IF Feature.Next is a Match THEN
7                 We have a match
8             ENDIF
9         ENDIF
10        IF pixel matches the first pixel of feature in START THEN
11            IF Feature.Next is a Match THEN
12                We have a match
13            ELSE
```

```
14                          Add feature.Next from START to PARTIAL
15                      ENDIF
16                  ENDIF
17          ENDFOR
```

In order to improve the results, various data structures for START and PARTIAL were used:

List of list of features

This was initial structure that was used in the implementation but it appeared to behave very slowly with a larger library of features.

Hashtable

In the first implementation of hashtable, the key was a pixel and the value was a list of linked list of features that start with this pixel. This appeared to behave slowly because the partial matches data structure was getting very large. The next implementation of hashtable included column as a key and corresponding value was a linked list of features.

2D Array

This implementation was much faster than the hashtable because retrieval time was instant comparing to the hashcode calculation in hashtable. This matrix has the same height and width as the image. Each column/row corresponds to the column/row in the image and is linked to the list of features

Later look ahead hotspot was added to this implementation, which improved the results dramatically.

2.2 Integral Image

Integral image is an algorithm for quickly and efficiently generating the sum of values in a rectangular subset of a grid [Figure 2]. Every row/column in the integral image is calculated by:

$$\text{sum}(x, y) = i(x, y) + \text{sum}(x-1, y) + \text{sum}(x, y-1) - \text{sum}(x-1, y-1)$$

where each value in the image corresponds to the pixel value in the integer format.

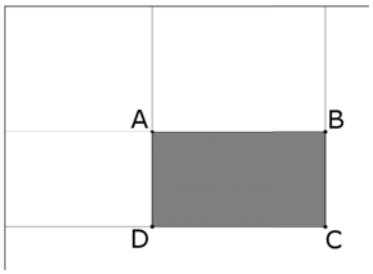
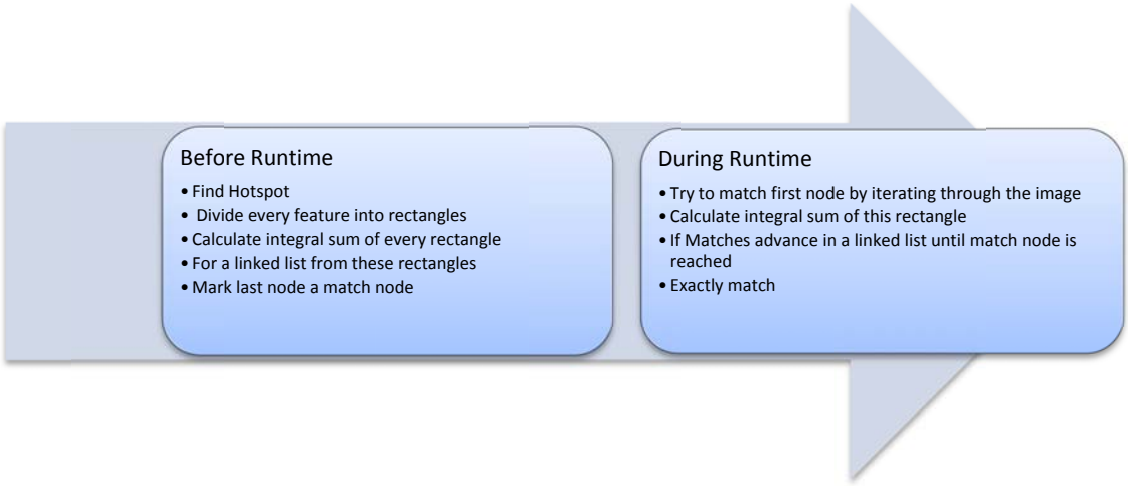


Figure 3
Corners of rectangle in the image for integral sum calculation

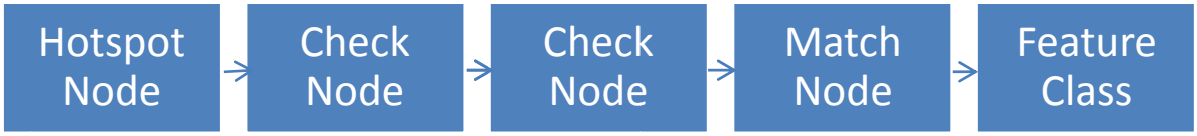
This algorithm was very promising because most of the work is done before the runtime execution, which includes calculating integral image from the image and keeping features in a linked list data structure. Many other features can be easily filtered out by calculating integral sum and comparing to the integral sum in the integral image.



During the preprocessing, stage every feature is divided into rectangles and then combined together as a linked list. In this linked list the first node is a hotspot which points to check nodes. The last node is a match node which links to the feature itself. The feature was divided into rectangles by recursing to the left, up, right, and then down of the transparent pixel which as a result gives us a minimal number of rectangles. Integral sum for each rectangle is calculated by

$$\sum_{\substack{A(x)<x'\leq C(x)\\ A(y)<y'\leq C(y)}} i(x',y') = \mathrm{sum}(A) + \mathrm{sum}(C) - \mathrm{sum}(B) - \mathrm{sum}(D).$$

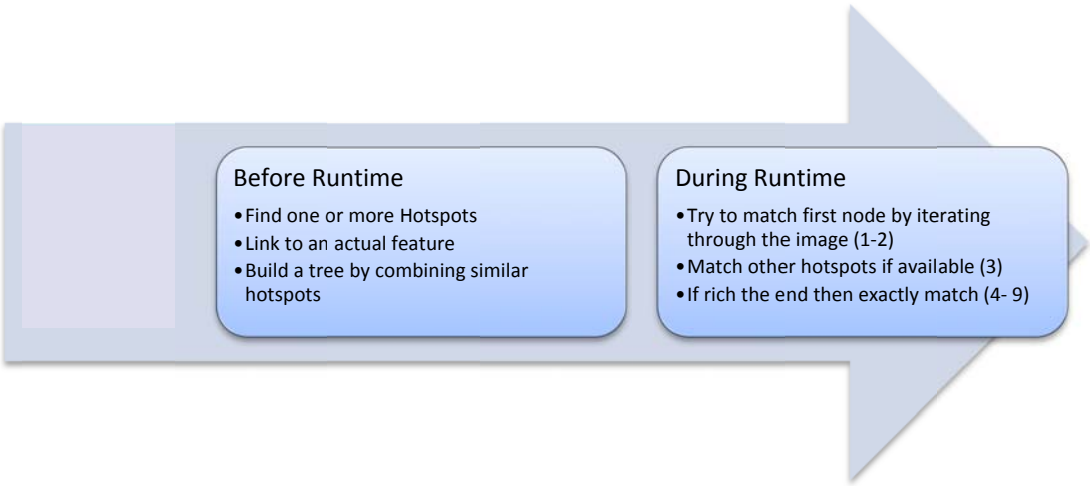
and then saved as a check node in the linked list. As a result our feature will look as follows:



During the runtime we go through the main image trying to match a hotspot pixel. If the hotspot pixel matches the image pixel, then the next step is to calculate integral sum for the rectangle in the image of the same area as the next check node. If integral sum of this rectangle matches the value in the check node, we advance in the linked list until we get to the match node. When we get to the match node we need to exactly match every pixel of the feature. This happens because pixels might be in different order in the rectangle but the sum can stay the same. In fact, throughout our experiment it was concluded that a large number of false matches are reported if the feature is not also exactly matched.

2.3 Finding Hotspot and Exactly Matching

After matching integral sum of rectangles in the image, another similar but very simple technique was used for finding features in the image. After matching hotspot pixel, instead of calculating the integral sum of the rectangle, we can exactly match the feature right away. This works because most of the features in the library are small and exactly matching works very fast on small features. During pre-processing stage, the features were saved in the hashtable where key corresponds to the hotspot pixel and the value is the list of features that start with this pixel.



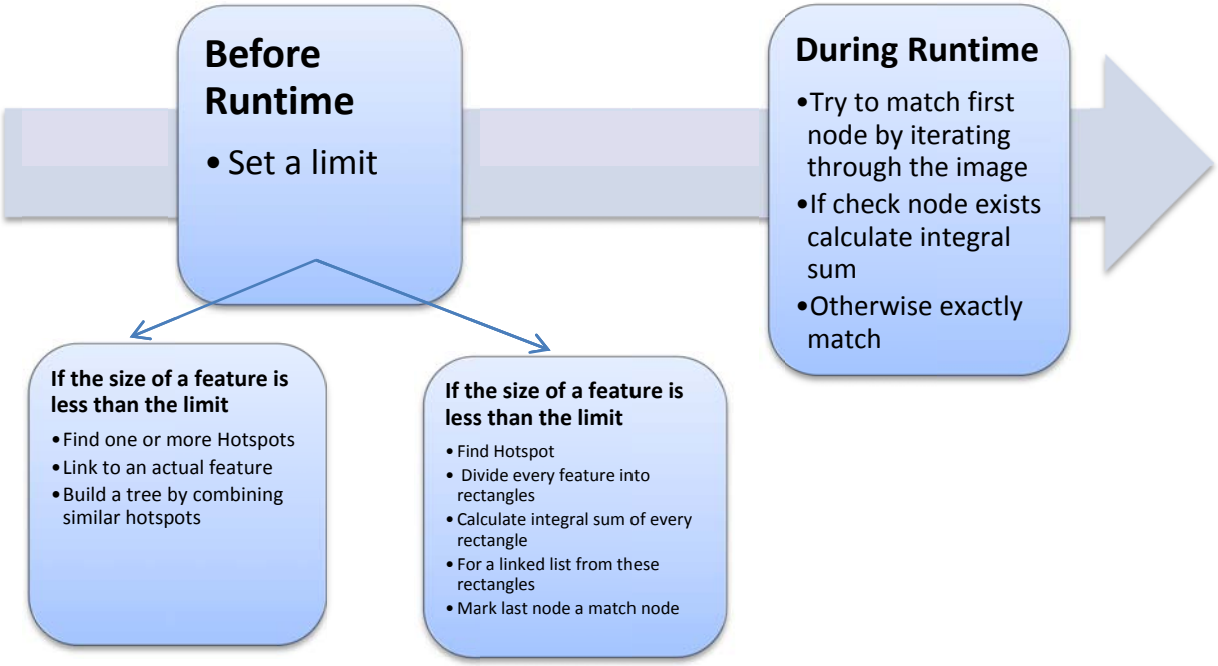
Matching more than one hotspot and then exactly matching the feature proved to be a much faster method. After running various tests it was concluded that it usually takes around 2 hotspot matches to filter out big number of features. Instead of using hashtable to contain all the features, in this case, tree behaves faster. Every node in the tree is a hotspot and leaves are list of features. This tree is shallow and has height of three.

Pseudo-code for the two hotspot and exactly matching algorithm is shown below:

```
1   IF tree contains pixel on the first level THEN
2           Save next level pointed from this node as NEW;
3           IF any of the saved nodes in LIST contain pixel THEN
4                   FOR every feature in the features
5                           IF exactly match THEN
6                                   Delete from saved
7                                   Match was found
8                           ENDIF
9                   ENDFOR
10          ENDIF
11          Add NEW to the saved pointers LIST;
12  ENDIF
```

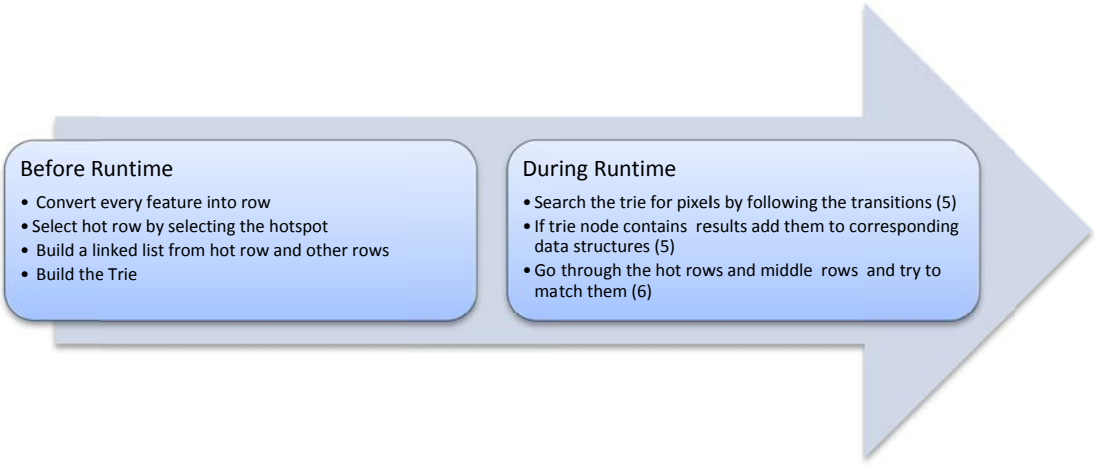
2.4 Threshold of Previous Two Techniques

This algorithm combines above described algorithms in such a way that it makes a decision whether to calculate and check integral sum or not. From the results integral image algorithm proved to behave very slowly with small features. Since the library of features mostly consists of small features this algorithm has an advantage by not calculating integral sum. If the feature's size is greater than the limit then we do calculate integral sum.



2.5 Aho - Corasick

The Aho–Corasick string matching algorithm is a dictionary-matching algorithm that finds elements of a finite set of strings within an input text and matches all patterns simultaneously. This algorithm constructs a finite state machine that look like a trie with additional links between the various internal nodes. These extra internal links allow fast transitions between failed pattern matches to other branches of the trie that share a common prefix [Figure 4]. This aids the automaton to transition between pattern matches without the need for backtracking.



This algorithm lets us visit every pixel in the image only once and the performance depends only on following the transitions in the trie. The trie and features are built before the runtime, so during the runtime we just need to follow the pointers in the trie. If we see a result in the node then we found a feature. Because features most of the time have height bigger than 1, they are divided in to the linked list of 1D features. At first this algorithm was tested on artificially created features of height one.

For instance let's build a trie from the following features of height one: {blue, red}, {blue, red, green, green}, {blue, green, green}, {green, blue, red}.

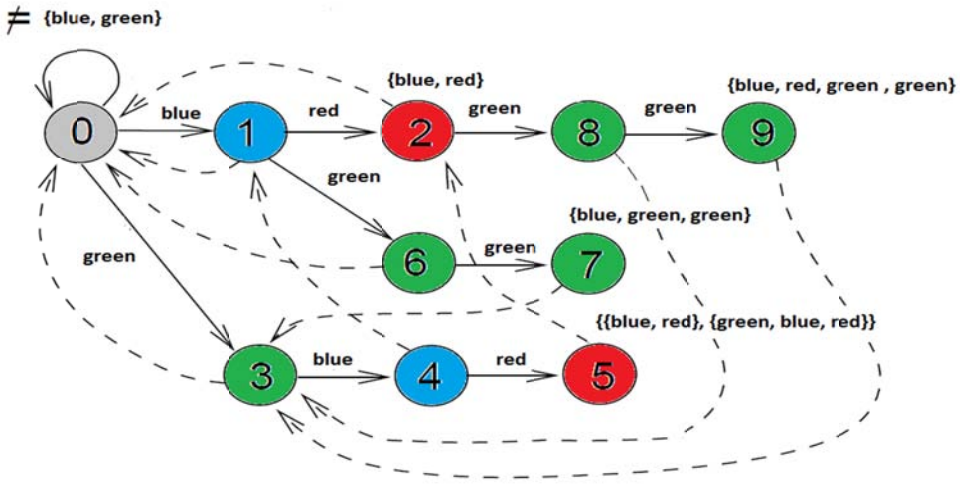


Figure 4

Aho-Corasick Trie with transitions and fail-transitions

Trie includes the following features: {blue, red}, {blue, red, green, green}, {blue, green, green}, {green, blue, red}.

The whole process can be represented as a pipeline process:

1) Divide each feature into 1D "mini-features".

2) Put mini-features into the linked list where head node contains hotspot pixel.

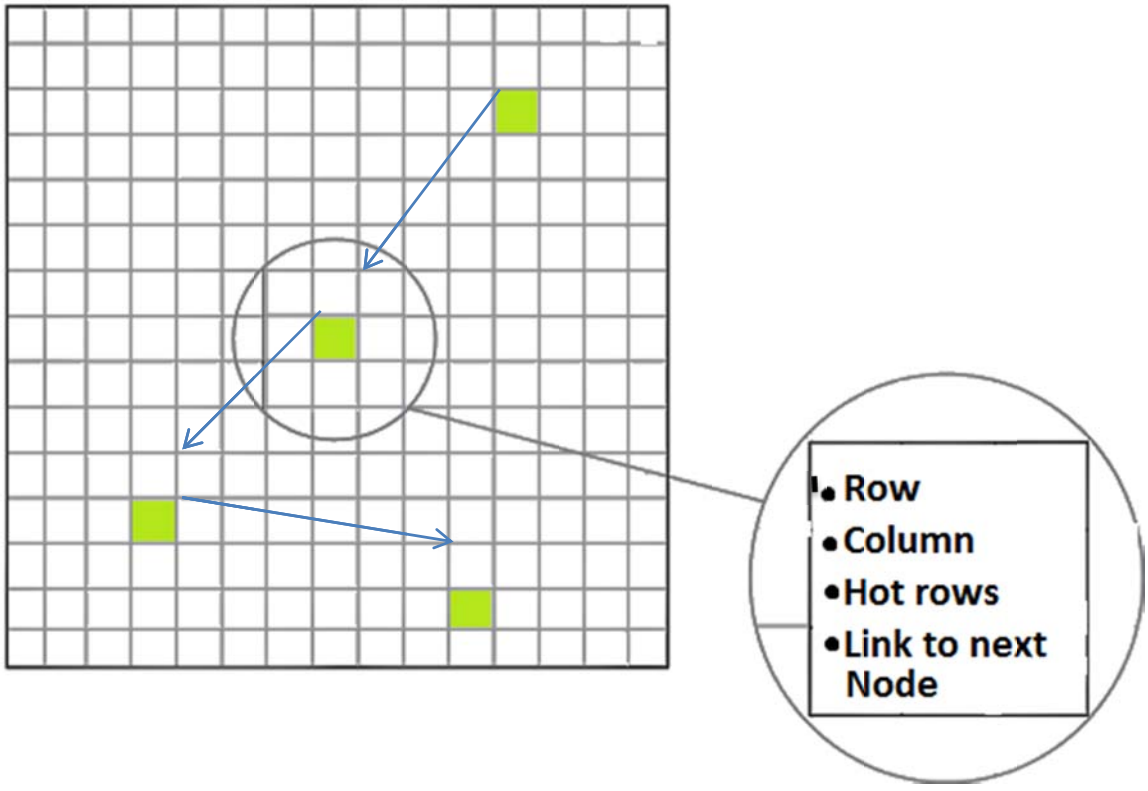
3) Build Trie

5) Run Algorithm on the image by marking middle nodes in the 2D array and putting head nodes into the queue. After running a few tests, one transition is made on average. Using a binary search tree instead of hashtable gave faster results.

6) Go through the queue and calculate number of features that were found

In order to improve the performance during runtime the following data structure for hot row results has been used instead of a queue. In the beginning when a hot row was found, a new instance of result class has been created and enqueued into the queue. This was slow because the program was creating many result classes in order to save row and column of the image. The new data structure is much faster because it uses only pointers. Before the runtime a 2D array is created where every row/column is represent as a linked list node. During runtime, when the hot row is found, the corresponding entry in the 2D array is updated to point to the found hot rows.

Image representation of this data structure is shown below:



3. Benchmarking Tool

In order to measure the performance of different algorithms, we needed a benchmarking tool that captures the average times over several trials. Datasets that were selected included different programs: YouTube video in Firefox, Microsoft Visual Studio, Calculator, Microsoft Word and its dialog boxes, Microsoft Excel, Microsoft PowerPoint, Skype, Windows Live Messenger, cs.washington.edu website in Internet Explorer, iTunes, PDF document in Acrobat Reader, and the Solitaire game. People use these programs every day and we needed to confirm that algorithms behave fast.

The benchmarking tool can be divided into two parts: a first phase where the datasets are created and the second phase for benchmarking the time it takes to run given functions over the image. During the first phase this tool starts capturing screen shot images after the user specifies the title of the window to be captured and name of the output xml file. User can stop the screenshot capturing at any time by clicking stop button or wait until the time exceeds the limit. The second phase uses a Wrapper Function class that contains all the function definitions that are used to benchmark the image. Each function is run over the image according to its priority.

For instance, let's say we have two functions in our wrapper class where the first one represents Aho-Corasick algorithm with the highest priority and the second one is the Build Tree algorithm. Then when we run the benchmarking tool, it first uses the Aho-Corasick algorithm over the image and then the Build Tree algorithm is run over the same image. As an input of the second phase the program uses the same xml file for the image dataset which was created during the first phase. Each xml file for each trial consists of average times for every function executed during the trial and time spent on every function run on every image. Summary file includes average times in milliseconds for every function execution over all trials.

Benchmarking tool was used on various find features functions such as single-threaded and multi-threaded Build Tree algorithm, Integral Sum with one hotspot match algorithm, Integral Sum with one/two/three hotspot match and then exactly match algorithm and one hotspot match with or without integral sum using threshold of previous algorithms. Please see appendix for the table.

4. Building Bigger Library

In order to improve the functionality of existing software for library building, I have added the following features:

- Software automatically detects whether a new prototype goes to positive or negative set of examples.
- Scroller feature that lets the user browse images as a gallery view. This tool shows found prototypes by running find features algorithm on every image shown.

In order to get accurate results from our benchmark tool, Prefab needed a larger library of prototypes. Most algorithms brought satisfactory results when using our existing small library. But we were not sure if it is always the case. Having a larger library of prototypes helped identify bugs and determine disadvantages of new algorithms. The larger library I created contains around 700 prototypes from various frequently used programs, such as Mozilla Firefox, Google Chrome, Microsoft Office, YouTube, Calculator, and Microsoft Visual Studio.

5. Results

At first, the initial Build Tree algorithm was tested on various datasets of a size of 500-4000 images with the interval of 10ms. These benchmark tests were tested using initial large library of around 100 prototypes. The results showed that it takes much longer to find feature on YouTube webpage and Visual Studio datasets. In our understanding, images from Visual Studio dataset contain many buttons and widgets. On the other hand, an image's pixels from YouTube video are more different than the image in the previous frame. [Table 1]

Dataset name	Single-threaded Build Tree algorithm	Multi-threaded Build Tree algorithm
Calculator	83ms	33ms
Visual Studio	265ms	159ms
YouTube Video in Firefox Browser	237ms	138ms
Microsoft PowerPoint 2007	67ms	40ms
Microsoft Word 2007	110ms	63ms

Table 1

The first algorithm that was tested and compared to Build Tree approach was the Partially-matching algorithm. But the results received were much slower than single threaded build tree implementation. The bottle neck for this implementation is the number of comparisons we make in order to get a match.

In order to reduce the number of pixel comparisons we started using Integral Image algorithm which helped to filter out many features by calculating integral sum of a prototype. We also had to exactly match because some features might have same pixels but in different order, thus resulting with the same integral sum. Unfortunately we noticed that many features are alike and have the same integral sum. This approach proved to be fast with the old large library consisting of around 115 prototypes [Table 2]. But after building bigger library of prototypes probability of having features with same integral sum increased as number of prototypes in the library approached to 700 [Table 3].

Image name / implementation Over 3 trials in ms	Calculator	Firefox	Mac-firefox	Notepad	Google talk	Chrome-settings-dialog	Notepad-init
Singlethreaded BuildTree	53ms	74ms	83ms	130ms	80ms	247ms	128ms
MultiThreaded BuildTree	37ms	54ms	51ms	87ms	50ms	137ms	85ms
1 hotspot match + IntegralSum match + exactly match	47ms	65ms	53ms	141ms	95ms	291ms	142ms
1 hotspot match + exactly match	35ms	50ms	42ms	108ms	71ms	215ms	107ms
2 hotspot match + exactly match	32ms	44ms	47ms	75ms	49ms	147ms	75ms
3 hotspot match + exactly match	36ms	48ms	44ms	77ms	50ms	153ms	77ms
Threshold hotspot match with/without IntegralSum match + exactly match	42ms	57ms	52ms	125ms	82ms	260ms	123ms

Table 2

Benchmark results from running on quad-core machine using a library of size 115

Image name / implementation Over 3 trials in ms	Calculator	Firefox	Mac-firefox	Notepad	Google Talk	Chrome-settings-dialog	Notepad -init
Singlethreaded BuildTree	61ms	93ms	83ms	46ms	27ms	93ms	44ms
MultiThreaded BuildTree	37ms	50ms	36ms	25ms	17ms	52ms	25ms
1 hotspot match + IntegralSum match + exactly match	87ms	202ms	65ms	45ms	21ms	181ms	142ms
1 hotspot match + exactly match	63ms	161ms	62ms	42ms	19ms	146ms	40ms
2 hotspot match + exactly match	48ms	44ms	47ms	41ms	22ms	124ms	38ms
Threshold hotspot match with/without IntegralSum match + exactly match	78ms	183ms	67ms	47ms	24ms	172ms	77ms
Aho- Corasick	55ms	87ms	88ms	43ms	30ms	97ms	40

Table 3

Benchmark results from running on six-core machine using a library with a size of 676 prototypes

Our next approaches without checking integral sum gave very decent results. Results were surprisingly fast compared to the integral sum implementation. Later two hotspots prior to exactly matching approach was tested and the results were even faster than multithreaded build tree match implementation [Table 2]. As it was expected results became slower with 3 hotspot matching. Matching hotspot requires 3 times more time than matching pixel one by one. This is because we need to calculate offset and then match the pixel. We came to conclusion that this approach is not very reliable because of the difficulty of optimizing how many hotspots we need to check before exactly matching the whole feature. As we can see from Table 2, one hotspot matching is faster with Firefox settings windows for OS X that two-hotspot matching algorithm.

Integral Image threshold approach was developed by combining above described two approaches. It is not very efficient to calculate integral sum for very small features and it is much faster to exactly match right away. So, this algorithm decides whether integral sum needs to be calculated or not. We see some improvements from just plain integral image algorithm but it was a little slower than exactly matching the image after one or two hotspot matches. This happens because majority of the features in the library are very small, of a size less than 10 pixels.

So far we have tried to reduce number of pixel visits in the image, reduce number of pixel comparisons, and decide if we really need to compare every pixel for very small features. Aho-Corasick implementation is based only on following transitions in the trie and going through each pixel in the image only once. This approach combines our previous goals in order to improve efficiency. Aho-Corasick algorithm is also very stable in a way that we do not need to guess how many hotspots we need to check before checking other pixels. Because most of the work is done during pre-processing time, Aho-Corasick approach proved to be really fast. In the Table 3, all of the above algorithms were re-run on a six-core machine using library that contains 676 prototypes. The results of this approach do not depend on the size of the library as much as above described algorithms.

6. Discussion and Conclusion

So far we have tried creating various algorithms that are very different from each other but have the same goal: improve the performance of Prefab feature recognition. At first, we tried to minimize number of pixel visits in the image by putting partial matches into various data structures. As a result we had to make a large number of comparisons, which slowed the whole process. In order to reduce number of comparisons we started using so called integral image. It helped to filter out many features and we were left only with features that contained the same number of pixels but in different order. This implementation was not as fast as we expected because the library consists mostly of small features. Calculating integral image for small features was significantly slower than matching pixel by pixel. Next we tried eliminating integral image calculation and using threshold algorithm with it. This gave us decent results but the approach was not very reliable because we cannot really decide which feature is small or big. The next strategy was to use Aho-Corasick algorithm, which uses a special trie structure with pointers and back-pointers. This helped us to achieve our goals: visit each pixel in the image only once and decrease number of comparisons.

Given what I have done I think it remains unseen which algorithm will prove to be faster at large scale. This work will help to understand if Prefab can find prototypes instantaneously in real time using a large library of features.

There are other approaches to be explored, such as reusing found features from previous frame in a new frame. This would significantly improve the performance because we do not need to recalculate the same features over the same parts of the image. Since Aho-Corasick implementation showed really good and stable results, we could try running it in parallel. The image could be divided into segments and each thread can look for features in each segment using the same trie. Also, if scaling is an issue then we could run Aho-Corasick algorithm in parallel with different libraries. In other words, a large library can be divided into many small ones and each thread would build its own trie. This might eliminate a large number of transitions while searching for features. At the end results from all the threads can be added together. All this work and new modifications would significantly help to get instantaneous widget recognition in applications using Prefab.

Acknowledgements

I want to thank James Fogarty, Morgan Dixon and Daniel Leventhal for all the help and discussions related to this work. This work was generously supported by the National Science Foundation under an REU supplement to award IIS-0812590.

References

[1] Dixon, M. and Fogarty, J. (2010). Prefab: Implementing Advanced Behaviors Using Pixel-Based Reverse Engineering of Interface Structure. *Proceedings of the ACM Conference on Human Factors in Computing Systems* (CHI 2010), pp. 1525-1534.

[2] Dixon, M., Daniel Leventhal and Fogarty, J. (2011). Content and Hierarchy in Pixel-Based Methods for Reverse-Engineering Interface Structure. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* (CHI 2011), pp. 969-978.

[3] Wikipedia, Summed Area Table, http://en.wikipedia.org/wiki/Summed_area_table

[4]Wikipedia, Aho-Corasick String Matching Algorithm, http://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_string_matching_algorithm

[5] Kilpeläinen P. (2005 Spring). Set matching and Aho-Corasick Algorithm, http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf