Upcoming:

Saturday December 10th – Continuous Integration and Coverage assignment due (submit in time for deadline, extended two days, do **not** wait until showcase demo)

Monday December 12th – short "showcase" demos for judges, signup form posted on canvas calendar and piazza (three simultaneous slots between 10 and 3), it's ok if only part of the team shows the demo if other team members have outside obligations

Friday December 16th – Final Delivery with final complete demo for your mentor – whole team needs to participate, remotely if necessary (submit *after* demo)

Tuesday December 20th – Third 360-degree feedback due

NO EXAM DURING FINALS WEEK

Today: A few advanced topics that have become increasingly mainstream in recent years (used in industry) but there's still plenty of research to do

Similar Code:

"Code clones" is a well-known concept – code has been copy/pasted from one part of the system to another

Considered a code smell, remove by refactoring or at least "simultaneously edit"

Why are code clones "bad"? Make codebase larger and less maintainable. Tend to diverge over time: some have had bugs fixed and others have not, some have been extended with new features that others haven't

Harder to recognize over time, rarely remain textually identical, various detection techniques based on tokens, abstract syntax trees, program dependence graphs

Many IDEs have plugins to automatically detect and help refactor code clones

Code clones are static, detected by analyzing the source code

Other notions of similar code are dynamic, and can only be detected by executing the code

For example functional clones, also called “simions”, produce the same or very similar output for the same input

Example: bubble sort vs. merge sort

Why are simions “bad”? When there are multiple different implementations of the same functionality in the same organization, work is being duplicated – software organizations seek to reduce or eliminate repeated work

But simions might use different algorithms to produce the same results, and there are sometimes good reasons for different algorithms in the same codebase – e.g., different time/space tradeoffs, different data structures shared with other parts of the code – but these could all be combined behind a single API with configuration options, to ease maintenance

Simions are harder to detect automatically than code clones: they require execution with a good/large sample of representative inputs, and there may be false positives (all you know is they match on the inputs you checked, not other inputs - in general its undecidable whether two pieces of code compute the same function)

Generating test inputs and comparing test outputs is much more challenging for objects (e.g., Java) than for primitive values (e.g., C)

Researchers have suggested a variety of other kinds of code similarities

Similar executions at the instruction level could be useful for leveraging hardware accelerators, and along with simions useful for program understanding – if you don’t understand what the code at hand does or how it works, find dynamically similar code (behaves alike) that is not statically similar (looks alike), which might help you understand how your code works

Example similar execution but not also simions: Eigen Value Decomposition vs. Singular Value Decomposition

Similar path conditions (a concept from symbolic execution) could be useful for automatic code repair – your code has a bug, similar code that passes the test cases your code fails while following analogous execution paths might be a source of code snippets to copy into your code to repair the bug

Similar code might be useful in code search:

Google provides an open-source code search tool https://github.com/google/codesearch

Github supports code search https://github.com/search

These and other code search tools primarily search metadata (e.g., author, issues) or code identifiers and comments

But could potentially be combined with code similarity detection tools, e.g., the code matching your search query terms isn't what you want, but maybe some other similar code (with different identifiers so the query doesn't find it) is exactly what you want

Examples of similar code:

```
int magic1(int[] ar){
	int sum = 0;
	for(int i=0;i<ar.length;i++)
		sum += ar[i];
	return sum;
}
```

```
int magic2(int[] ar){
	int sum = 0;
	for(int i=0;i<ar.length;i++)
		sum += @2 *@ ar[i];
	return sum;
}
```

```
int magic3(int[] ar){
	int sum = 0;
	int i = 0;
	try{
		while(true)
			sum += ar[i++];
	}
	catch(ArrayIndexOutOfBoundsException ex){}
	return sum;
}
```

```
int magic4(int[] ar){
	int sum = 0;
	int i = 0;
	try{
		while(true)
			sum += @2 *@ ar[i++];
	}
	catch(ArrayIndexOutOfBoundsException ex){}
	return sum;
}
```

Hint: Listings 1 and 3 calculate the sum of an array, Listings 2 and 4 calculate twice the sum of an array. Listings 1 and 2 use a traditional for loop, while listings 3 and 4 iterate without explicit bounds checking.

Record/Replay:

Useful for recording bugs detected in user environment so they can be reproduced (and fixed) in developer environment

Useful for organizations to check whether a “critical security patch” breaks the functionality they depend on, before deploying on production servers

Useful for running the same GUI test suite over and over on different browsers or different handheld devices, which might otherwise have to be done manually

Let’s focus on the bug reproduction problem:

Users are notoriously bad at reporting bugs, they will report “I tried to do X and it didn’t work” without precise details of what exactly they did in what order while trying to do X and what they mean by it didn’t work, plus users usually cannot know what is going on internally in the system while they try to do X

The goal of record/replay is to record “everything” that happens in the application, so exactly the same the same sequence of activities can be replayed later

But recording “everything” would add far too much overhead, immensely slowing down the application, so users would turn it off

Instead, only sources of non-determinism are recorded, because by definition all the deterministic activities will repeat exactly the same way

Non-deterministic sources include all inputs from the user, the network, the file system, the database, devices, return values from library or system calls (e.g., random number generator, time of day), etc. that could be different during different executions

During replay, instead of obtaining new inputs, the system retrieves the previous values from the recorded log - note inputs here are the return values from the library and system calls that actually obtain the inputs from the user, the database, etc.

So recording can be implemented by intercepting all library/system calls in the user environment and storing their return values

Replay can be implemented by intercepting all library/system calls in the developer environment and feed results from the log rather than actually running those calls, or indeed running those calls but wrapping/forcing them to produce the same results

In theory, everything should then run exactly the same way – except the operating system can still schedule multi-threaded applications in different orders, so interleavings that impact inter-thread dependencies have to be recorded too and forced during replay

Example: the user runs the "hello world" application, which always outputs "hello world" every time it's run, in this case the recording log can be empty because the application is already deterministic

Another example: the user runs "hello world its <time of day>", so the recording log has to save the time of day that was actually produced from the system clock in the user environment to get the exact same execution in the developer environment later on instead of "hello world its <new time of day>"

A more realistic example: the application queries a database, gets the query results from the database, and then crashes in the middle of a long complicated computation – but since this is a huge real-world database we do not want to send the whole thing to the developer (there's also a privacy issue when the database contains sensitive data)

Or someone else may be using the database concurrently, and update the data records in between when they query results were sent to the user application and when the crash occurred - the recording log doesn't snapshot the whole database on every query, but only has to save the exact original response from the database (where there is still a privacy issue but smaller)

Or there could have been a long series of database queries and database updates, where an update after one query affects the data returned by the next query – the recording log reproduces this series without needing the actual database

This works for non-database applications, of course, could be user I/O, network I/O, file I/O, etc., recording any "inputs" that need to be replicated later

All this recording is still very expensive, so there is ongoing research in further lowering performance overhead – what is the minimal recording overhead so that we can still reliably reproduce the execution, maybe some of the non-determinism doesn't really need to be recorded because we can determine (e.g., through static program analysis) a priori that it is unlikely to impact further progress of the execution (or a limited set of "executions of interest")

Also research on how to anonymize the data sent from user to developer so that the exact same low-level execution paths are forced by fake data to still reproduce the bug - here the fake data can be substituted in between recording in the user environment and transmitting to the developer, which can be done offline without runtime overhead

Replay as described here can only work when the code that ran in the user environment is exactly the same as the code that runs in the developer environment, what if we'd like to replay after inserting debugging instrumentation or even after an attempted bug fix? In other words, record with one version of an application and replay with another version. This is called "mutable replay", many open research questions about when/how this can be achieved

For example, we can apply static analysis to the before and after versions of the code to determine exactly what read/write dependencies are affected by the change, so we can determine whether or not the same input can be used from the log or we have to "go live" to obtain a new input

Example read/write dependency:

We recorded the log with this code

x = read(something); some random code that doesn't change x; y = f(x);

y depends on x, we can use the logged value from read(something) as the parameter to f

Then the code is changed to

x = read(something); other random code; x = read(something else); other random code; y = f(x);

We cannot use the logged value from read(something) as the parameter to f, we need to obtain a new value by setting up and running read(something else).

Check time…

Metamorphic Testing:

Machine learning, data mining and search applications are difficult to test and debug because the correct answers are often not known in advance for all inputs - applications that answer questions where no one knows the answer, if we already knew the answers there would be no reason to implement the system

Recall that a typical test case provides inputs and then checks the results, to see if the test case passes or fails – but how do we do this if we don't know what the result is supposed to be? (no "test oracle")

Machine learning uses the term "testing" to check that a model derived from training data "works" with other data not included in the training set, and uses various statistical prediction metrics to determine quality

But if an application predicts correctly most of the time but not all of the time, how do we know whether the prediction failures are due to inherent limitations of the machine learning algorithm or a bug in the code implementing the algorithm? We still need what software engineering calls "testing"

Metamorphic testing is a technique that can detect (some) bugs with no test oracle

The basic idea is to derive a followup test from a source test in such a way that the result of the second test should have a known relationship to the results of the first test, and then compare the actual outcomes of the two tests

If the relationship between the two outcomes is not as expected, then there is probably a bug in the code – even though we may not know whether either outcome is correct (they could both be wrong)

Test case 1: Executing function (input1) produces output1

Test case 2: Executing function (input-transformation(input1)) should produce output-transformation(output1), did it?

Example:

The input is a table of many data samples and the output is a clustering of the data samples to find those data samples that are most similar to each other

If we permute the order of the input table, we should still get the same output clusters

If we add one new data sample to otherwise the same input table, we should still get the same output clusters with the new data sample in one of the clusters, did we?

Another example:

The input to a search engine is a phrase like "x y z" and the output is a set of documents from our corpus that contain the phrase "x y z"

If we change the input to the phrase “x y”, the output should contain all the same documents plus additional documents that contain “x y” but not “x y z”, does it?

Metamorphic testing is not a complete solution: someone has to devise what are the metamorphic relations to check, how do we know what the metamorphic relations should be?

Some machine learning algorithms employ randomness to intentionally get different results on different executions with the exact same inputs, so we need a notion of “close enough” for even the same inputs to be properly in a metamorphic relationship with themselves

Metamorphic testing can’t find all bugs, only those bugs that impact a known metamorphic relation - so there’s still much more research to do on finding and fixing bugs in big data applications

Metamorphic testing is also often used with conventional applications that do have test oracles, where the test oracle is a fallible and tired human, or just to produce more test cases to find bugs (analogous to adding more test cases to increase coverage)

This is a very biased sample of advanced topics in software engineering, since these are the topics recently or currently investigated by my lab. We are also conducting research on applying “gameful” techniques to engaging 6-8 graders in learning about computational thinking.

We are seeking undergraduate and MS level project students for all of these efforts. I also seek to admit one or two new PhD students to work specifically on the mutable record replay problem (where there is funding for new GRAs).

If you’re interested in the similar code problems, contain my PhD student Mike Su (mikefhsu@cs.columbia.edu).

If you’re interested in computational thinking education, contact my DES student Jeff Bender (jeffrey.bender@columbia.edu).

If you’re interested in record/replay or metamorphic testing, contact me (use kaiser+4156@cs.columbia.edu so I’ll see it).