Computing efficient bisection commits

Last Updated: 2015/02/11

Background

Git bisect is an efficient way to find a faulty commit in the git history. For a completely linear history this would produce a binary search on these commits. The problem however arises when we have many branches and merges in the history. Then the question on how to pick the next commit to test becomes harder.

Overview

The algorithm proposed here works like this:

1) Get a deterministic list of commits in the desired range. (best is topological sorted)

2) Assign a bit pattern `B(k) = 0{k} 1 0{n-k}` where `B` is the bit pattern, `k` is the k-th commit in the list from 1) and `n` is the number of commits in the desired range.
   This bit pattern B is called the first order bit pattern. So the bit pattern consists of n-1 zeros and a 1 set at the same position as in the deterministic list computed in 1)
   `Example: 000100000 for n=9, k=3`

3) Derive a bit 2nd order pattern for each commit which is the logical OR of the first order bit patterns of all ancestors and the commit itself.
   `C(k) = B(k) | B(all ancestors of k)`

4) The bit count of the second order bit patterns give a hint on how good this commit is to test. Find the commit whose second order bit pattern is closest to n/2. This is the best commit to test.

Detailed Design

This will write out the details of the four steps of the overview.

Getting a deterministic list of commits in the desired range.

The best way to create this list would be a topological sorted list, because this will make it easy in later steps to use run length encoding. This could be done with
`git rev-list --topo-order A..B`

Assigning the first order bit pattern

The first order bit pattern is unique for each commit and is guaranteed to be orthogonal to each other bit pattern in the desired range.

Run time

The run time is O(n) as you can just walk through the list generated in the first step and then compute the bit pattern in O(1).

Memory consumption

As the bit pattern for each commit is n, the memory consumption for all commits is O(n^2).
If the bit patterns are run length encoded, the memory consumption will drop to O(n) again, as there will be a fixed number of runs of bits in each bit pattern. First there will be a number of zeros, one 1, and then zeros again, so the run length encoding is expected to shrink the memory consumption back to O(1) per commit and O(n) for all commits.

Deriving the second order bit pattern

Computing `C(k) = B(k) | B (ancestors of k)` can be rewritten as `C(k) = B(k) | C (direct ancestors of k)` because the second order bit pattern is transitive. So the easiest way to implement this would be a recursive function

```
set = {} # we start with the empty set.
```

```
comp2ndBS(Commit c):
      ret = B(c)
      for a in direct-ancestors(c):
            if not a in set:
                  set[a] = comp2ndBS(a)
            ret |= set[a]

set[tip] = comp2ndBS(tip)
```

Run time

Each commit is only computed once, so the run time is the number of edges in the graph of the desired range which for typical git histories is O(n)

Memory consumption

The worst case of the heap memory consumption would be O(n^2) again, the best case with linear history creates a O(n) if we use run length encoding for the bit patterns This is similar to the memory consumption to the first order bit pattern. Because of the topological ordering of commits, we'd expect a good run length compressible bit pattern. For each commit the number of runs at worst case equals the number of branches which are not merged in yet, but will be merged in later. This is a low constant number in practise, so we can estimate the memory consumption of this step close to O(n).

As this is a recursive algorithm, we may be concerned with the stack memory consumption, which in the worst case is O(n) for a linear history and O(log(n)) for a balanced tree.

Finding the bit count close to n/2 of the second order bitmaps

To find the best resulting commit using the second order bit patterns, we need to first count the ones per pattern, which can be done while the pattern is run length encoded. There should be no need to decompress it to a full bitmap. Once each commit is assigned a the counted bits in their second order pattern, sort them by number of bits set and pick the middle.

Run time

expected average run time O(n log n)

Memory consumption

Worst case is O(n^2) again, but in practice we'd expect an average of O(n) because of good run length coding. No extra memory is needed for sorting and finding the right commit as that can happen in place.