Final work analysis: sparse based BFS random testing

Junjie Pan

1.Introduction

In this work, I propose a new method aiming to find testing bugs which is very sparely spread in codes. In general case, for the software to be testing, bugs should be also very sparsely distributed in the software codes, that's why bugs sometimes is very hard to be found. However, it always be able to find a bug if tester spend enough time on it, such as examing codes line by line, but this kind of time consuming method is impractical when the program scale become too large. However, since bugs are very sparsely distributed, it reminds us many sparse method used in information theory. Therefore, my method is apply the sparse method from information theory to try to solve the bugs finding problem based on BFS(breadth first search method).

However using sparse method, ideally, can only make sure to find high potential bugs. In worst case, it will miss bugs that can be easily find by some very directive method. Sometime it would also miss the bugs that occur in very obvious place. To make up this shortage, I also add a piece of non-sparse structure random testing codes as part of testing procedure. However, our emphasis is still heavily rely on sparse based BFS random testing.

In this work, I choose block sparsity model [1] as my sparse method. I use 90% of time budget for this model, and use 10% time budget to run general random testing. Hopefully this combining scheme can make up the shortcoming of each testing method.

The flowing contents are organized as three parts. In the first part, I illustrate the sparse model in great detailed since it is most important part of my proposed method. In the second part, I will briefly introduce the structure of the program and its result analysis. In the last section, I will give a brief conclusion.

2. block-sparse random based BFS testing model and improvement

The concept of block-sparse originated from signal processing research area[1], which facilitate many real-world applications especially for dimension deduction problem. One simple case of block-sparsity is structure-less block sparsity. That is, for example, given an array arr(which can be a list, a vector, or a tuple) of n length with sparse-level k (which means only n*k elements of arr store the useful information we want) where $0<k<= 0.5$ in general, when we divide the array into b blocks, then the n*k useful information would also sparsely diffuse in u blocks of totally b blocks, where $u/b<= 0.5$.

Take bug testing as an example, suppose there is a tuple consist of both safely actions and unsafely actions, and the unsafely actions are just small part of all actions, then the unsafely actions(we use bug to denote unsafe action in the remain contents) can be viewed as very sparse information of the tuple. Then, based on the block-sparsity theory, when we divide tuple into several blocks with the block size satisfying some criteria (actually this is a bit of complicate, so I temporally don't consider it and simply set it as fix number in this work), then the bugs would diffused sparsely in this blocks, which means the number of blocks that would contain the bugs would sparse compared to the total number of blocks, with high probability.

Algorithm:

```
Define n as length of queue # queue is as the same definition of bfsrandom
Define block_num # number of blocks
Define sparse_level
Define valid_blks =    n*sparse_level # the part of blocks that would contain bugs
Define ref_tb # reference table with n length, ref_tb[i] = 1 indicate there would be bug
               # in the i-th actions and test the i-th bundle of actions

If    n >= 2*block_num
    Block_size = int ()

    Valid_ids = indexes of randomly chosen valid_blks from block_num of blocks

    For    i    in valid_ids
        start_id = i*block
        end_id = i*(block+1)
        ref_tb[start: end] = inner_rand(end_id- star_id, in_spar_level, min)

else:
   ref_tb = inner_rand(n, 0.5, 1)


 inner_rand(len, s_level, min_num):

    arr = [0]*len
    sparsity = int(len*s_level);
    if sparsity< min_num:
        sparsity = min_num;

    a = range(len);
    random.shuffle(a);
    sparse_ids = a[0:sparsity]#[random.randint(0, len-1) for i in range(sparsity)]
    print sparse_ids
    for i in sparse_ids:
        arr[i] = 1;
    return arr
```

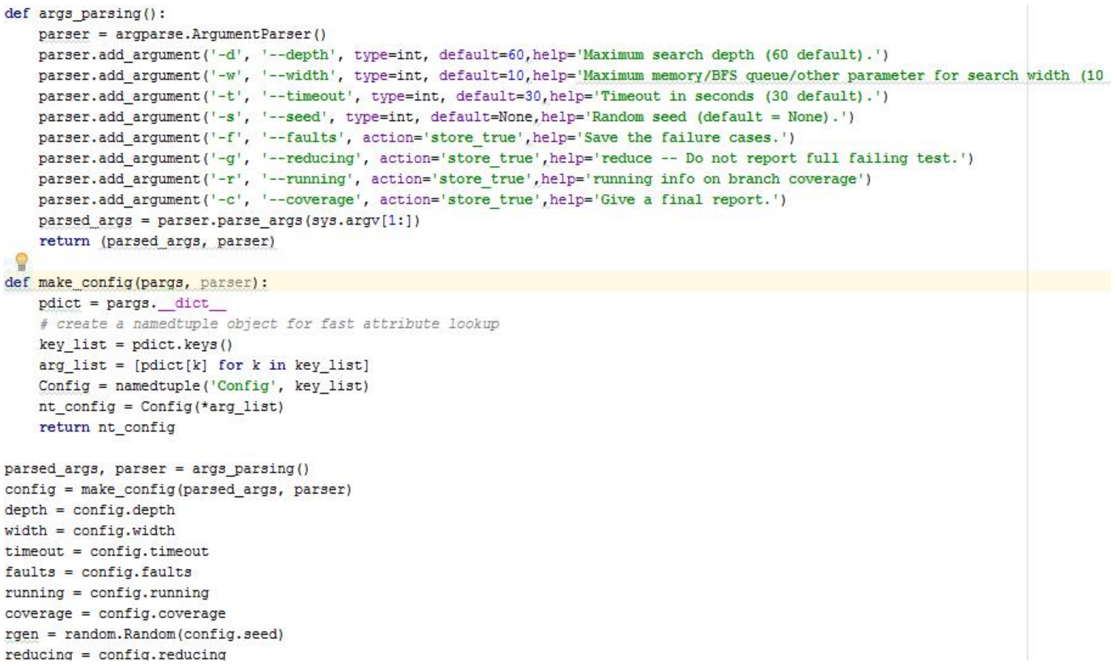
Remark: Since the sparse properties would be not so good when the length of queue is small, I adjust the sparsity level based on different length. Actually sparsity covering only work when the useful information is sparse enough. However, we assume that the high sparsity of bug can be achieved only when the sequence is longer enough, especially when apply block sparsity. Therefore, I will make some very minor change that set the sparsity based on different sequence size.

Improvement

Though sparse model is really favor in find useful information from extreme sparse environment since it always work well, it sometimes would fail to find simple bugs that can be easily detect by general method. In order to make up this shortcoming, I only assign 90% time budget for the sparse testing, and assign the remaining 10% time budget for general random testing.

3. Experiment result

In this program, I use argparse to parse the arguments from command line to program. The part of argparse code is as below:

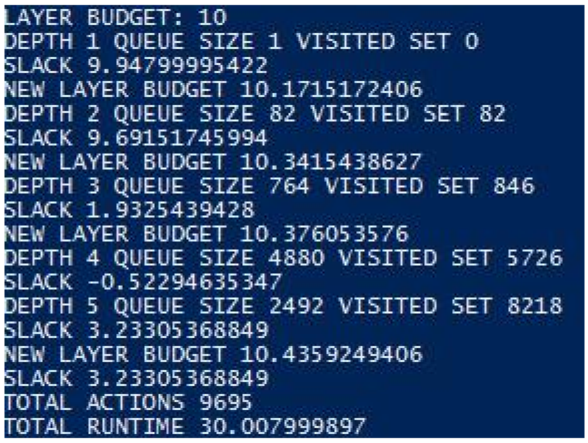


```
def args_parsing():
    parser = argparse.ArgumentParser()
    parser.add_argument('-d', '--depth', type=int, default=60,help='Maximum search depth (60 default).')
    parser.add_argument('-w', '--width', type=int, default=10,help='Maximum memory/BFS queue/other parameter for search width (10
    parser.add_argument('-t', '--timeout', type=int, default=30,help='Timeout in seconds (30 default).')
    parser.add_argument('-s', '--seed', type=int, default=None,help='Random seed (default = None).')
    parser.add_argument('-f', '--faults', action='store_true',help='Save the failure cases.')
    parser.add_argument('-g', '--reducing', action='store_true',help='reduce -- Do not report full failing test.')
    parser.add_argument('-r', '--running', action='store_true',help='running info on branch coverage')
    parser.add_argument('-c', '--coverage', action='store_true',help='Give a final report.')
    parsed_args = parser.parse_args(sys.argv[1:])
    return (parsed_args, parser)

def make_config(pargs, parser):
    pdict = pargs.__dict__
    # create a namedtuple object for fast attribute lookup
    key_list = pdict.keys()
    arg_list = [pdict[k] for k in key_list]
    Config = namedtuple('Config', key_list)
    nt_config = Config(*arg_list)
    return nt_config

parsed_args, parser = args_parsing()
config = make_config(parsed_args, parser)
depth = config.depth
width = config.width
timeout = config.timeout
faults = config.faults
running = config.running
coverage = config.coverage
rgen = random.Random(config.seed)
reducing = config.reducing
```

Remark: since this part is for extra credit, I past the screenshot here.

Where the default values are as follow: depth = 60; width = 10, timeout = 30, seed = none, and faults, reducing, running and coverage are defaulted to true.

In my experiment I set `python tester2.py -d 60 -w 10 -t 30 -s 1` and let all other values to be default. The result is:



```
LAYER BUDGET: 10
DEPTH 1 QUEUE SIZE 1 VISITED SET 0
SLACK 9.94799995422
NEW LAYER BUDGET 10.1715172406
DEPTH 2 QUEUE SIZE 82 VISITED SET 82
SLACK 9.69151745994
NEW LAYER BUDGET 10.3415438627
DEPTH 3 QUEUE SIZE 764 VISITED SET 846
SLACK 1.9325439428
NEW LAYER BUDGET 10.376053576
DEPTH 4 QUEUE SIZE 4880 VISITED SET 5726
SLACK -0.52294635347
DEPTH 5 QUEUE SIZE 2492 VISITED SET 8218
SLACK 3.23305368849
NEW LAYER BUDGET 10.4359249406
SLACK 3.23305368849
TOTAL ACTIONS 9695
TOTAL RUNTIME 30.007999897
```

As we can see, the total runtime is 30.0079 which is very closed to our time budget

4.Conclusion

Actually, there are many things need to be improved, such as block the actions from the actions level instead of from the queue level, which is much harder. In the future, I will try to erase the timing methods for testing, instead, I will try to make use of block-sparsity properties to determined when to stop testing. What's more, how to choose the time-budget for different testing model still need to seriously study. Actually, the location to add the random test should also take into account. In my case, I add it in the end of program. Actually, maybe it can be added in both the beginning of the program and end of program. All of these need to further work.