Dynaplex: Analyzing Program Complexity using Dynamically Inferred Recurrence Relations

This artifact contains an implementation of the dynaplex algorithm and benchmark programs, as described in the paper: *Dynaplex: Analyzing Program Complexity using Dynamically Inferred Recurrence Relations*.

Overview

The development and experiment environment is provided as a single Docker image at `unsatx/dynaplex:oopsla21`. The following is a guide to set up and use this artifact:

1. Prerequisites: - Linux-based OS (tested on Ubuntu 20.04 and Debian 10.7). - Docker (tested with Docker 20.10.7). - Make sure you can run `docker run hello-world` successfully on the host machine.

2. Pull Docker image:

```
docker pull unsatx/dynaplex:oopsla21
```

3. Run container:

```
docker run -it unsatx/dynaplex:oopsla21
```

4. Test Dynaplex (inside container):

The evaluation benchmark is divided into 3 folders according the programming languages. To reproduce the results of Table 1 use the following command:

For python programs. Results take approxiamtely 15 minutes

```
cd /dynaplex/recursive_programs/python
./bm.sh
```

For ocaml programs. Results take approxiamtely 5 minutes

```
cd /dynaplex/recursive_programs/ocaml
./bm.sh
```

For C++ programs. Results take approxiamtely 5 minutes

```
cd /dynaplex/recursive_programs/cpp
./scripts/bm.sh
```

`bm.sh` analyzes each program five times and collect the results into the folder `res`. The results for all programs in the benchmark will be displayed on the standard output. The following explains an example of the result for analyzing `subset_sum.py`:

```
 Computing the recurrence relation
T(n-1)
T(n-1)
Computing polynomial relations
Command:  /home/dishimwe/dishimwe/dynaplex/dig.py -trace ./subset_sum/traces -maxdeg 5 -r -nlog
b'models before applying heuristics
1.5O
0.003968 x + 1.478
-0.01706 x^2 + 0.1977 x + 1.047
0.002292 x^3 - 0.05754 x^2 + 0.4087 x + 0.7425
0.001036 x^4 - 0.02227 x^3 + 0.1437 x^2 - 0.251 x + 1.453
0.000421 x^5 - 0.01148 x^4 + 0.1178 x^3 - 0.5861 x^2 + 1.493 x - 0.05704
b'Models after applying Heuristics
1.5
Analysis complete in 0.002336740493774414 seconds
Polynomial relation:  n^0
k=0 p=0
'
T(n) = T(n-1) +  T(n-1) +  (n^0(logn)^0)
Solving the recurrence relation
Analysis complete in 0.774 seconds
```

The output shows the recurrence relation calculated and the time it took for the analysis. The results produced by running the benchmark reproduce the results shown in Table 1 of the paper and column 3 (dynaplex) of Table 2. The rest of Table 2 is not reproducible from this artifact as those results were taken from Chora [1] and NPWCARP [2]. The results from the paper were produced on an AMD Ryzen 16-core machine with 32 GB of RAM running Debian 10 thus the time it takes to run experiments may be different on other machines.

To reproduce the graphs in Figure 5: For `closest_pair`

```
cd /dynaplex/graphs
python3 -W ignore closest.py
python3 -W ignore /dynaplex/dig.py -trace closest/traces -maxdeg 5 -plot
```

For `strassen`

```
cd /dynaplex/graphs
python3 -W ignore closest.py
python3 -W ignore /dynaplex/dig.py -trace strassen/traces -maxdeg 5 -plot
```

The graphs similar to those in Figure 5 of the paper will be saved in `fig.png`. For more detailed explanation of the evaluation results refer to the evaluation section of the paper.

Dynaplex architecture

1. Trace collection: a. code instrumentation b. input generation
2. Complexity analysis: a. Generate recurrence relations b. Generate polynomial relation c. Solve recurrence relation for closed form complexity

Trace collection

1. We instrument the recursive function to collect the problem size and depth of recursion at each iteration
2. We also instrument the recursive function and helper functions (if available) to collect the size and number of iterations for **one** recursive step.
3. We store traces in files with following naming convention a. output- `S` for recursion traces for input of size `S` b. traces for the iterations

Complexity analysis

To compute the complexity from collected traces we use the following command: /path/to/analyzer.py -trace /path/to/trace/folder

Descriptive example

```
def bubble_sort(arr, n, depth, file):
    with open(file, 'a') as f:
        print("{};{}".format(depth, n), file=f)
    if n==1:
        return arr
    for i in range(n-1):
        if arr[i]>arr[i+1]:
            if depth == 0: #we only count iterations for one recursion step.
                global counter
                counter = counter + 1
            arr[i], arr[i+1] = arr[i+1], arr[i]
        return bubble_sort(arr, n-1, depth+1, file)

#Driver to collect all traces
counter = 0
def main():
    global counter
    counter = 0
    size = random.randint(1, 500)
    arr = random_list(size)
    depth = 0
    path = "./bubble_sort"
    try:
        os.mkdir(path)
    except OSError as error:
        pass
    file = "./bubble_sort/output-{}".format(size) #naming convention
    bubble_sort(arr, size, depth, file)
    with open("./bubble_sort/traces", 'a') as f:
        print("{};{}".format(size, counter), file=f)
    counter = 0


if __name__ == '__main__':
    for i in range(100): #run the bubblesort a few times to collect enough traces
       main()
```

Analysis results

Command: dynaplex/analyzer.py -trace dynaplex/bubblesort **results** recurrence: T(n) = T(n-1) + f(n)
polynomial relation: f(n) = n^1
recurrence relation: T(n) = T(n-1) + n
complexity: O(n) **Safe warnings to ignore**: dynaplex algorithm uses learning algorithms (sklearn) which splits data into *train_set* and *test_set*. If the trace files are too small this can raise a value error such as: `ValueError: With n_samples=1, test_size=0.25 and train_size=None, the resulting train set will be empty. Adjust any of the aforementioned parameters.` If this happens collecting traces with larger input will generate more traces and avoid the above error.

Note:

The code instrumentation does not need to follow the same guidelines shown in the example. For example, check the recursive_programs/cpp folder for more effective code instrumentation. The traces need to follow the same guidelines. For recursive traces in files named output-size there should be `depth;size` to indicate `depth` of recursion and `size` of the problem at that depth. For iteration traces in file named traces there should be `size;counter` to indicate the size of the problem and the number of iteration one recursive step takes.

References

[1] Jason Breck, John Cyphert, Zachary Kincaid, and Thomas Reps. 2020. Templates and Recurrences: Better Together. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (London, UK) (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 688–702.

[2] Krishnendu Chatterjee, Hongfei Fu, and Amir Kafshdar Goharshady. 2019. Non-polynomial worst-case analysis of recursive programs. *ACM Transactions on Programming Languages and Systems* 41, 4 (2019), 1–52