Stable Marriage R/C++

March 30, 2022

Task 1

See rest of files in Github Repo.

Task 2

To compare times between R/C++ and Python, consistency and accuracy in method comparison is a must. As we deal with larger preference tables and the need to auto-generate these tables, for an accurate comparison we should compare running times using the same preference lists. Another consideration is the preference table sizes and how many to compare. These issues are addressed as follows in the simulation study:

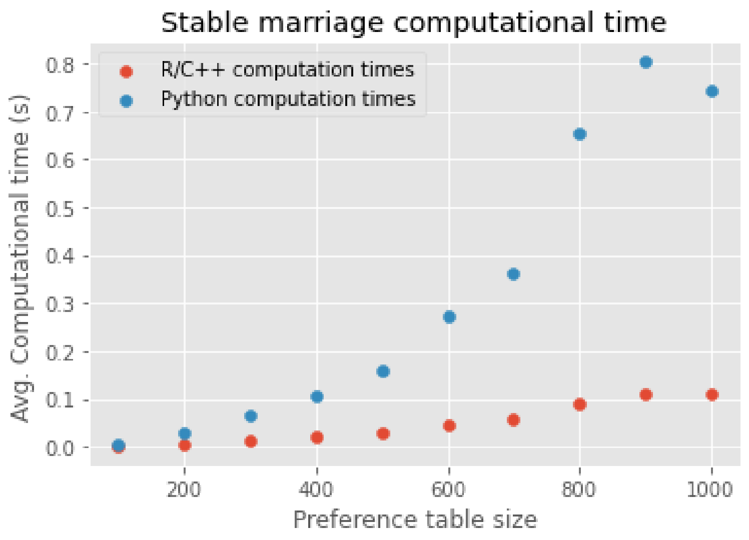
1. Preference tables for $n = 100$ to $n = 1000$ at regular 100 intervals are generated using the `GeneratePref` python function made in the first assignment. For each preference table size n, the simulation is ran 4 times.
2. For Python the original code from the simulation study in the first assignment is ran after each preference generation, and the times are averaged over the 4 simulations for each preference table size.
3. For R/C++, the preference tables are exported to a CSV file then read in as a R `dataframe` object. The simulation then follows that of python, with the times saved and averaged over 4 simulations.

Code for the R/C++ and Python simulation study are given below. The python code includes both the simulation and csv file generation.

```python
# Python: Simulation study
nums=[i*100 for i in range(1,11)]
sims = 4
times_py = []
for num in nums:
    men_tot = []
    wom_tot = []
    for sim in range(sims):
        times_py_temp = []
        # generate preference tables and carry out algo
        men = ['M'+str(i) for i in range(num)]
        women = ['W'+str(i) for i in range(num)]

        # Generate a separate copy of the preference table (use same seed) since variables get changed in
        # python implementation.
        random.seed(42)
        m_r, w_r = GeneratePref(men, women)

        random.seed(42)
        m, w = GeneratePref(men, women)

        men_tot.append(m_r)
        wom_tot.append(w_r)

        # Python implementation
        clear_output(wait=True)
        display('Iteration ' + str(num))
        start = time.perf_counter()
        # Find stable matchings
        match = StableMarriageAlgo(m, w)
        end = time.perf_counter()
        times_py_temp.append(end-start)

    # Append mean time
    times_py.append(np.mean(times_py_temp))

    # Save to csv file for R processing
    with open("times/men_{:d}.csv".format(num), "w") as outfile:
        keys=men_tot[0].keys()
        for item in men_tot:
            writer = csv.writer(outfile, delimiter = ",")
            writer.writerow(keys)
            writer.writerows(zip(*[item[key] for key in keys]))
    with open("times/wom_{:d}.csv".format(num), "w") as outfile:
        keys=wom_tot[0].keys()
        for item in wom_tot:
            writer = csv.writer(outfile, delimiter = ",")
            writer.writerow(keys)
            writer.writerows(zip(*[item[key] for key in keys]))
```

```r
library(StableMarriage)
prefSizes<-c(100,200,300,400,500,600, 700, 800, 900, 1000)
times <- c()
for(size in prefSizes) {
  temp_times<-c()
  # Open csv and read in as dataframe
  file_w<-sprintf("times/wom_%s.csv", size)
  file_m<-sprintf("times/men_%s.csv", size)
  women <- read.csv(file_w, header = FALSE)
  men <- read.csv(file_m, header = FALSE)
  colnames(men) = men[1,]
  colnames(women)=women[1,]
  for(i in seq(0,3)) {
    # Split between iterations in CSV
    test_w<- women[(i*(size+1)+2):(i*(size+1)+size+1),]
    test_m<-men[(i*(size+1)+2):(i*(size+1)+size+1),]
    print('Iteration')
    # Carry out algo.
    ptm <- proc.time()
    res<-StableMarriageAlgo(test_m, test_w)
    temp_times<- c(temp_times, proc.time() - ptm)
  }
  # Append times
  times<-c(times, mean(temp_times))
}
```

Results of both are seen in the following Figure . It can be clearly seen that the R/C++ greatly outperforms the Python implementation. This is the expected result as C++ has long been regarded as a much faster and more efficient language.



Task 3

1. Re-runnable

    - Python (4/5): All packages used are either standard libraries, or well maintained packages that are widely used. All functions within these packages are commonly used and thus likely to stay stable and not immediately obvious to be depricated and cause issues if the code was to be run in the far future. Some work could have been done to reduce dependency on packages
    - C++ (3/5): The C++ implementation uses standard libraries available in modern C++ versions. However there are drawbacks with its reliance on C++17 and this requires computers to have this newer version installed. Areas which use C++17 are small and these could be changed to allow backwards compatibility in future.
    - R/C++ (2/5): See above issues, plus a further drawback in RCPP being a bit finicky about using C++17 for compiling, this ended up requiring some changes to internal file when compiling the project (specifically a `makevars` file was needed before compilation of package).

2. Repeatable

    - Python (5/5): All randomness used is fixed to a seed as that deterministic output is given, even when random numbers are involved. If the notebook was re-run, on the same computer set up, the same output should be given. Python was all run using the same version of python (3.9.7).
    - C++ (5/5): No randomness was used in this problem or implementation so there would be no issue with whatever software the program was run on, it should still give the same output
    - R/C++ (5/5): This implementation did not have any randomness in the problems, except for Task 2, which used randomly generated preference tables that were created in Python and exported anyhow. So as above with the C++ implementation, there should be no issue with repeatable results given the deterministic nature of the algorithm.

3. Reproducible

- Python (5/5): The only data required for reproduciblility is for randomness to come from the same source, this can be done by setting an equivalent seed to generate the preference tables (in our case random.seed(42)) to attain the same results. The tests given at the end each task to check the output is what we expect can also help garner reproducible results.
- C++ (5/5): Like repeatable, this implementation shouldn't suffer any reproducible problems as no randomness is involved
- R/C++ (5/5): Same as above.

4. Reusable

- Python (5/5): By commenting liberally, we have the means for future developers to easily digest, adapt, and explore the code base. As is the tests to check if a more efficient way to carry out tasks is found, we can compare this to the previous results with the old way and verify consistency.
- C++ (2/5): Despite significant commenting, reusing and understanding the code base in C++ is a tricky task, in future some documentation could have been included to better explain what was going on. A few tests have been provided to help consistency and expected results if one was to change and re-use portions of the code and check the output is expected.
- R/C++ (3/5): The addition of documentation for the R package gives this a better score for reusable than the pure C++ implementation. the wrapper functions are also well explained and commented to clearly let a developer know what is going on.

5. Replicable

- Python (4/5): The code does not fully deviate from the algorithm description in the stable marriage given by Knuth and Goldstein, and follows along as broadly as possible. Perhaps use of list comprehension and pythonesque language might cause some differences from the algoritmh's pseudo-code
- C++ (5/5): The code follows along broadly with the algorithm description, with additional elaboration and is more verbose in certain sections. This could be perceived to add to replicable nature of the problem as it means less shortcuts and optimisation workarounds from the fundamental algorithm.
- R/C++ (3/5): While the code follows broadly, the additional requirements of Wrapper functions to convert o R and C++ means we deviate slightly from the main algorithm to accommodate these changes.