Static binding, dynamic binding, and performance

Software design can have a significant impact on the performance of an algorithm. We illustrate this point using the variants of union find, implemented with both dynamic binding and static binding. For a further discussion of static and dynamic binding, see pp 737–745 in Prata, and pp 722–737 for a discussion of runtime polymorphism.

Each of the files `union_find0.{hpp,cpp}`, `union_find1.{hpp,cpp}`, `union_find2.{hpp,cpp}`, implements a family of the following variants of union find, but in different ways:

1. Quick find
2. Quick union
3. Weighted quick union
4. Weighted quick union with path compression

Each `.cpp` file contains a test driver which prompts the user for the number of sites (e.g., vertices in a graph) and connections (e.g., edges between vertices). It then generates a random set of connections, and checks the time required to apply union to incorporate the connections as well as repeated checks on whether sites are connected.

Questions for you to answer

0. Why does the function `test()` behave differently in `union_find0.cpp` and `union_find1.cpp`?

The first set of experiments concerns the effect of algorithmic tweaks inside each family of implementations.

1. Based on experiment, how do the algorithmic improvements from quick find to weighted quick union with path compression affect runtimes for the implementation in `union_find0.{hpp,cpp}`? Your answer should include run time data from your experiments, including the number of sites and connections.
2. Same as question 1 for the implementation in `union_find1.{hpp,cpp}`?
3. Same as question 2 for the implementation in `union_find2.{hpp,cpp}`?

The second set of experiments concerns the effect of differences between the different families of implementations. Restrict attention to weighted quick union and weighted quick union with path compression since they are the best versions of union-find.

4. Based on experiment, rank the three implementations of each the two algorithms from fastest to slowest. Your answer should include run time data from your experiments, including the number of sites and connections.
5. Explain the results you obtained in the previous question.

Be sure to note which machine you used to obtain your test results, and what compiler options you used to build the executables.

Tips

- You should large enough instances of your tests to get accurate timing results.
- You should small enough instances of your tests that an inefficient algorithm won't run forever.
- In other words, you'll need to experiment with the problem sizes.
- In the makefile I use `-O3`, the highest level of optimization. This gives good results on `g++` 4.8.2 on the CS machines; on other machines YMMV.

What you should submit

Submit your written report in the Project04 directory of the Google Drive folder.