Course
CS-E3190
Course materials
Your points

« Programming Exercise 1 - Stable Matching Course materials 4.2 The stable matching problem »
CS-E3190 / Programming Exercise 1 - Stable Matching / 4.1 Implementing iterative algorithms

Implementing iterative algorithms

Our first programming exercise at this course asks you to write a solver subroutine that computes a stable matching between n applicants and n positions. (Click here to fast-forward to the problem statement and the submission dialog.)

One possible solution to the exercise is to prepare an implementation of the Gale–Shapley "Proposal Algorithm" studied at the first lecture.

The Gale–Shapley algorithm is an example of an *iterative* algorithm that proceeds in successive steps, each of which brings us closer and closer to a solution as measured by, for example, a potential function.

Before we proceed with the formal problem statement, here are some general observations and guidance that you may find useful in implementing iterative algorithms:

- **First prepare a correct implementation, then optimize for performance**. As with any implementation task, it is advisable to first prepare an implementation that is *correct*, and only then think about how to optimize the implementation for scalability to large inputs. Indeed, a correct implementation can in many cases be optimized one part at a time, which tends to be easier than starting from scratch and trying to achieve both goals at the same time. Also, from the concrete standpoint of the course grading, a correct implementation will already award you points.
- **The importance of testing**. As discussed with the warm-up exercise, each exercise has available *unit tests* that help you to prepare a correct implementation. When a test fails, you should understand *why* the test fails and correct your implementation accordingly. With iterative algorithms, you can always have your implementation print out what it is doing in each step before failure. In most cases this enables you to identify the first step where your implementation starts to err, and accordingly find the error and correct your implementation.
- **Have each step do as little work as possible**. An iterative algorithm consists of successive steps—maybe literally billions or trillions of such steps—so it pays off to think carefully how to implement each step so that it gets executed with as little work as possible. In many cases you can reduce work by *precomputing* and by careful use of supporting *data structures*. For example, to implement the Gale–Shapley algorithm, you will need to compare preferences in a preference list to determine which applicants get rejected. Could you perhaps avoid scanning a preference list by appropriate precomputation? Also, you will need to track which applicants are applying for which particular positions at each step. What kinds of data structures do you already know that could support such tracking? Again such tracking should be fast and avoid redundant scanning.
- **Explicitly list the work that remains**. Each step of an iteration can often be optimized to do as little work as possible by explicitly listing the work that remains, and *only* the work that remains. *Stacks*, *queues*, and *linked lists* are often useful data structures for tracking the work that remains and to update quickly the state of an iteration from one step to the next step. For example, to implement the Gale–Shapley algorithm, it may be a good idea to track the positions, and *only* the positions, that have received more than one application. Or conversely, track the applicants who have no tentative match yet. It may also be a good idea to track the tentative matches. It is all too easy to turn an $O(n^2)$-time-implementable algorithm design into a careless $O(n^3)$-time implementation by doing redundant work, for example, in scanning for work to be done.
- **Iterative algorithms are difficult to parallelize**. In an iterative algorithm, the steps of the iteration naturally occur in succession, where the previous step needs to be complete before the next step can be started. That is, the next step can be started only if we have available the information that results from the previous step. This property makes it difficult to parallelize iterative algorithm designs, although in some cases it is possible to identify some steps of the iteration, or extensive computations inside each step, that can be executed independently of each other in parallel. In the worst case, however, it may be that the steps depend on each other in succession, and thus parallelization becomes difficult. For the present exercise, it suffices to implement the Gale–Shapley algorithm without parallelization.
- **The scaling tests tell about scalability**. Recall our discussion of the scaling tests in connection with the warm-up exercise. Here is an often useful rule of thumb for analyzing the practical scalability of your algorithm implementation. Recall that the scaling tests proceed with increasing values of n following a geometric (exponentially growing) progression. If we write $T(n)$ for the running time of a scaling test with input size n, and assume that $T(n)$ grows as a polynomial function of n, say, $T(n) = Cn^d$ for constants C and d independent of n, we observe that $T(2n) = C(2n)^d = 2^d \cdot Cn^d = 2^d T(n)$. That is, with each doubling of the input size parameter from n to $2n$, we should observe the running time $T(2n)$ grow to 2^d times the running time $T(n)$. For example, for an algorithm implementation with quadratic scalability ($d = 2$), we should observe the running time $T(n)$ multiply with $4 = 2^2$ with each doubling of the input size n. For an implementation with cubic scalability ($d = 3$), we should observe the running time $T(n)$ multiply with $8 = 2^3$ with each doubling of the input size n. In practice one of course does not get an exact polynomial fit of the running times, for example, due to effects from the memory hierarchy and other microarchitectural reasons as well as thermal effects, but running a geometric scalability test can provide a diagnostic to identify e.g. a careless $O(n^3)$-time implementation from an actual $O(n^2)$-time implementation.
- **The worst case may be really bad**. The input instance to an iterative algorithm may result in essentially the maximum possible number of steps permitted by the potential-function analysis. For example, for the Gale–Shapley algorithm, there exist input instances where every applicant gets matched with either the *least or second-least* preferred position by that applicant. Furthermore, it will take at least $(n-1)(n-2)$ steps to converge. (You can find such instances in the unit tests.)

Now we are ready for the problem statement and the submission dialog.

« Programming Exercise 1 - Stable Matching Course materials 4.2 The stable matching problem »