Queens College, CUNY, Department of Computer Science

Software Engineering

CSCI 370

Fall 2019

Instructor: Dr. Sateesh Mane

due Friday Sep. 27, 2019 11:59 pm

1 Project 1

- **All students must submit solutions individually for this project.**
  See below about grading policy.
- This project does not require a GUI or a database.
- This document requires a calculation involving a lot of computation.
- This project requires statistical sampling using pseudorandom numbers.
- To reduce the overall computation time, the application should perform parallel processing.
- This can be accomplished on the Venus server by writing a multi-threaded program.
- You are responsible to design your program code to perform the computations in parallel.
- **Read the project description and think carefully before you start coding.**
  1. Think carefully about what information you really require to answer the questions.
  2. Then plan the calculations you have to do.
  3. Significant optimizations may be possible.
  4. *It will help to draw some pictures of random walks and/or calculate a few cases by hand, to give you an idea how to design your program.*

1.1 Project report & grading policy

- **Please submit your solution via email, as a file attachment, to `Sateesh.Mane@qc.cuny.edu`.**
  1. The file attachment is a zip archive with the following naming format:

     `StudentId_first_last_CS370_project1_Fall2019.zip`
  2. Important: **do not encrypt the zip.** Other formats such as RAR archive or OneDrive or Google Drive, etc. will not be accepted.
  3. You may submit update solutions up to the due date.
     (a) Use the same name for the zip archive: your previous submission will be replaced.
     (b) Only your final submission will be graded.
  4. Late submissions will receive a reduced grade, *possibly including a failing grade.*
- **All students must submit solutions individually for this project.**
  1. **Students who work together on this project must inform me of the names of all collaborators in advance of the due date.**
  2. Otherwise, if I receive identical solutions from multiple students, they will be classified as cheating and will all receive a failing grade.
  3. Every student must submit a zip archive, even if he/she worked with others.
- **Your project zip archive must contain all your program source code.**
  1. Your program code must be written in either C++ or Java.
  2. Your code will be tested by compiling and running it on the Venus server.
  3. *A failing grade will be awarded if your program does not compile and run successfully.*
  4. Non-fatal warnings, e.g. use of deprecated features, will be excused.
- **Your project zip archive must also include a cover document.**
  1. Acceptable formats are pdf and docx ("cover.pdf" or "cover.docx").
  2. **The cover document must contain your answers to all the questions/problems you solve in Secs. 1.6, 1.7, 1.8 and 1.9.**
  3. There are instructions in each of those sections how to format the answer for that section.
  4. The more sections you answer correctly, the higher your grade.

1.2 Introduction

- This project involves **self-avoiding random walks** (commonly called "SAW").
- **A self-avoiding random walk is not allowed to intersect itself.**
- To explain, consider a two-dimensional integer grid.
    1. The random walk begins at the origin $(0, 0)$.
    2. At each increment, the walk takes a step horizontally ± 1 unit else vertically ± 1 unit.
    3. Diagonal steps are not allowed: each step is either horizontal else vertical.
    4. Suppose that after n steps, the path of the random walk is given by the set of points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\}$, where $(x_0, y_0) = (0, 0)$.
    5. At the next step, the random walk goes to the point (x_{n+1}, y_{n+1}), which is one of the four possible points $(x_n + 1, y_n)$, $(x_n - 1, y_n)$, $(x_n, y_n + 1)$ or $(x_n, y_n - 1)$.
    6. *However, it is forbidden for the point* (x_{n+1}, y_{n+1}) *to equal one of the points* $(x_0, y_0), \ldots, (x_n, y_n)$.
    7. **Once a self-avoiding random walk passes through a point, it cannot pass through the same point again.**
- Two or more points in the random walk are not allowed to coincide.

$$(x_i, y_i) \neq (x_j, y_j) \qquad \text{if } i \neq j\,. \tag{1.2.1}$$

- An example of a self-avoiding random walk is displayed in Fig. 1.
    1. The walk starts at the origin $(0, 0)$.
    2. Twelve steps are shown, labelled 1 through 12.
    3. After step 12, three possible steps labelled $13a$, $13b$ and $13c$ are shown,
    4. The step labelled $13a$ is forbidden because the random walk would intersect itself.
    5. The steps labelled $13b$ and $13c$ are allowed.
        (a) Note that if the next step is $13b$, then the random walk reaches a dead end, because all possible steps after that will self-intersect the path of the random walk.
        (b) *It is possible for a self-avoiding random walk to reach a dead end.*
    6. There is one additional step after step 12, which is not displayed in Fig. 1.
        (a) That is to reverse the direction of step 12 and go from $(-1, 1)$ to $(-2, 1)$.
        (b) **A "reverse step" is forbidden because it also counts as a self-intersection.**
- In this project, we shall only consider self-avoiding random walks on integer grids.
- Every random walk always starts at the origin.

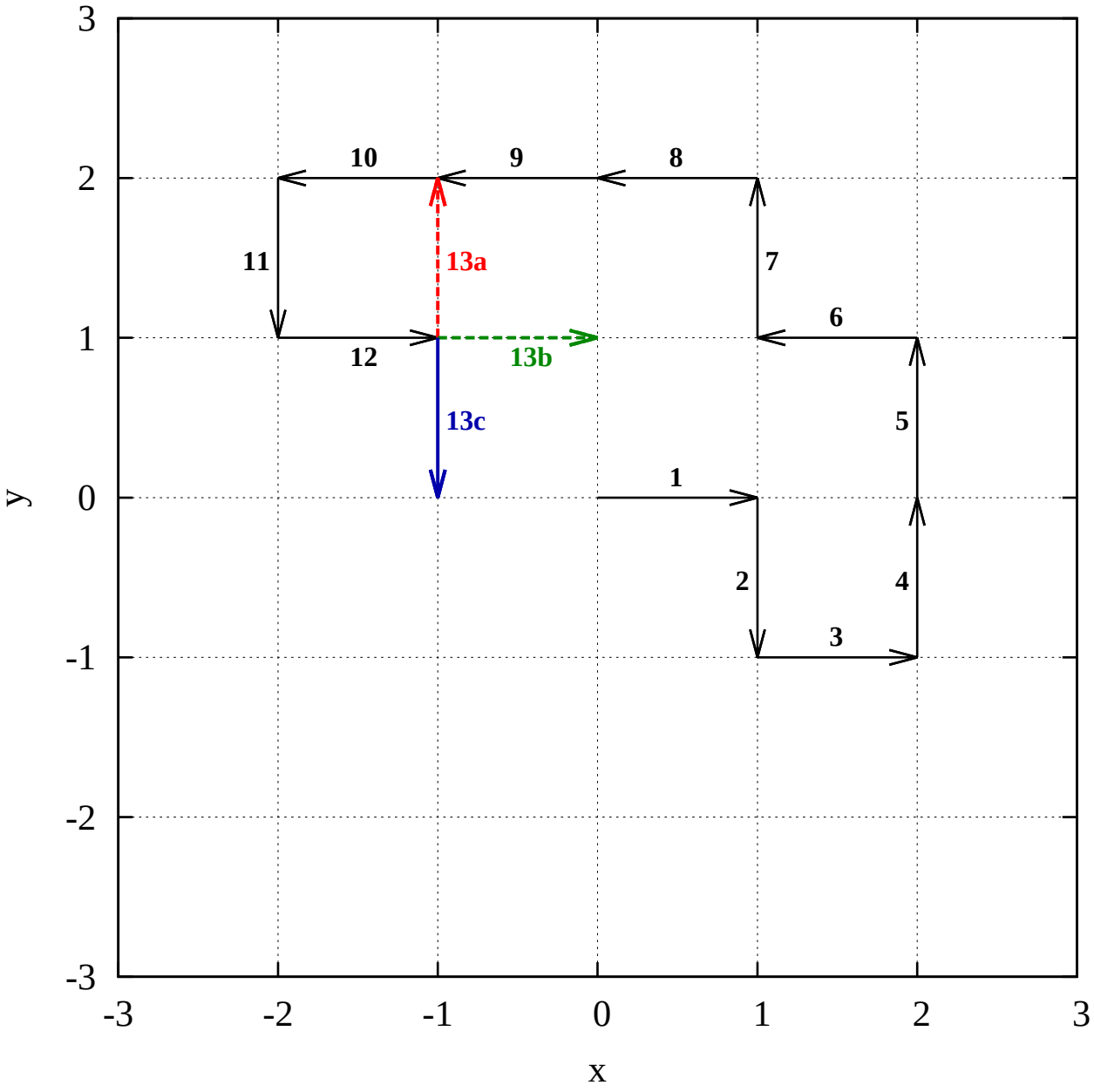

Figure 1: Example of self-avoiding random walk (SAW) on a two-dimensional integer grid. The step labelled 13a is forbidden because the path would intersect itself.

1.3 Simple cases

- Here are a few results which you should be able to verify for yourself.
- In one dimension, there are only two self-avoiding random walks with n steps.
    1. All positive steps, going from $x_0 = 0$ to $x_n = n$.
    2. All negative steps, going from $x_0 = 0$ to $x_n = -n$.
- In two dimensions, there are four self-avoiding random walks of length 1, from $(0, 0)$ to $(1, 0)$, $(-1, 0)$, $(0, 1)$ and $(0, -1)$, respectively.
- In two dimensions, there are twelve self-avoiding random walks of length 2.
    1. Each of the random walks of length 1 has three possible choices for the next step.
    2. You should draw a graph and plot the random walks for yourself.
- In three dimensions, there are six self-avoiding random walks of length 1, from $(0, 0, 0)$ to $(1, 0, 0)$, $(-1, 0, 0)$, $(0, 1, 0)$, $(0, -1, 0)$, $(0, 0, 1)$ and $(0, -0, 1)$, respectively.
- In three dimensions, there are thirty self-avoiding random walks of length 2.
    1. Each of the random walks of length 1 has five possible choices for the next step.
    2. You should also verify this for yourself.

1.4 End-to-end squared distance

- The **end-to-end squared distance** is the square of the distance between the end point and the start of a random walk.
- For a random walk with n steps, the end-to-end squared distance R_n^2 is defined as follows.

$$R_n^2 = (x_n - x_0)^2 + (y_n - y_0)^2 \qquad \text{(2 dimensions)}\,, \tag{1.4.1}$$

$$R_n^2 = (x_n - x_0)^2 + (y_n - y_0)^2 + (z_n - z_0)^2 \qquad \text{(3 dimensions)}\,. \tag{1.4.2}$$

- Because we always begin the random walk at the origin, we can simplify:

$$R_n^2 = x_n^2 + y_n^2 \qquad \text{(2 dimensions)}\,, \tag{1.4.3}$$

$$R_n^2 = x_n^2 + y_n^2 + z_n^2 \qquad \text{(3 dimensions)}\,. \tag{1.4.4}$$

- For the self-avoiding random walk in Fig. 1, after 12 steps the end-to-end squared distance is $(-1)^2 + 1^2 = 2$.
- We shall employ pseudorandom numbers to generate a sample of N_{SAW} self-avoiding random walks with n steps each.
- The mean (or average) of the end-to-end squared distance is given by the average:

$$\langle R_n^2 \rangle = \frac{1}{N_{\text{SAW}}} \sum_{i=0}^{N_{\text{SAW}}-1} R_n^2[i]\,. \tag{1.4.5}$$

- *Note: because the random walks are on an integer grid, the value of R_n^2 is an integer. However, when summing over a large number of walks, the total sum may encounter integer overflow.*
  1. Hence in a computer program, it may be better to declare the end-to-end distance as `double` not `int`.
  2. It may also be better to declare some other variables as `double` not `int` to avoid overflow, even though their values are integers.

1.5 Programming details

- **In this project, we shall estimate the value of $\langle R_n^2 \rangle$ for self-avoiding walks with n steps, for various values of n.**
- In one dimension, there are totally 2^n random walks with n steps, but only two are self-avoiding.
- In two dimensions, there are totally 4^n random walks with n steps, but only a small fraction are self-avoiding.
- For a value of n such as $n = 30$ or $n = 40$, it is impractical to count all the self-avoiding random walks with n steps.
- Hence, to generate a sufficiently large set of self-avoiding random walks with n steps, we need to perform statistical sampling.
- This will require the use of pseudorandom numbers to generate a sample set of self-avoiding random walks.
- The Venus server has multiple cores, so computations can be performed in parallel.
  1. Hence write a **multi-threaded program** to perform calculations in parallel.
  2. I have found that using $N_T = 1000$ threads works well on the Venus server.
- *What to do in each thread?*
  1. In each thread, run an outer loop of N_W passes.
  2. Because of CPU time limits on student accounts, you may be limited to $N_W \lesssim 10^6$.
  3. You should experiment to see how fast your program runs.
  4. Let the total number of steps in a random walk be n.
  5. Then each thread executes nested loops as in the following pseudocode.

```
for (int iwalk = 0; iwalk < N_W; iwalk++) {
  // initialize a random walk, etc
  for (int istep = 0; istep < n; istep++) {
    // ....
  }
  // ...
}
```

- On each pass in the inner loop, we generate a pseudorandom number.
- I have found (in Java) that the most efficient pseudorandom number generator is `ThreadLocalRandom`.
- **`ThreadLocalRandom` is thread safe.** Math.random() is not thread safe.
- **The C++ pseudorandom number generators in `std::random` are thread safe.**

- For a random walk in two dimensions, you can generate a pseudorandom number as follows.

```
int r = ThreadLocalRandom.current().nextInt(4);                 // Java
std::uniform_int_distribution<int> r_distribution(0, 3);        // C++
int r = r_distribution(generator);                              // C++
```

- This will give you a set of four possible choices for the next step:

$$\begin{aligned} x[\text{istep}+1] &= x[\text{istep}]+1 \qquad (r == 0)\,, \\ x[\text{istep}+1] &= x[\text{istep}]-1 \qquad (r == 1)\,, \\ y[\text{istep}+1] &= y[\text{istep}]+1 \qquad (r == 2)\,, \\ y[\text{istep}+1] &= y[\text{istep}]-1 \qquad (r == 3)\,. \end{aligned} \tag{1.5.1}$$

- **You must test for self-intersection before updating the random walk.**
- You can create a data structure to store the path of the random walk $\{(x_0, y_0),\, (x_1, y_1), \dots\}$.
- Else you can design some other algorithm to test for self-intersection.
- **If a random walk self-intersects, reject it. Break out of the inner loop.**
- *Hence not all of the N_W passes through the outer loop will yield a self-avoiding random walk with n steps. Many of the passes will fail.*
    1. Suppose the loops yield `N_SAW` self-avoiding random walks with n steps.
    2. Compute the end-to-end squared distance of each self-avoiding walk with n steps.
- Return the data for the end-to-end squared distance and `N_SAW` to the calling application.
- The calling application may be `main()`, or a function called by `main()`.
- The calling function collects all the data returned by all the threads.
    1. The total number of self-avoiding random walks (for walks with n steps) from all the threads, say `N_SAW_tot`, is the sum of the values of `N_SAW` from each thread.
    2. The sum of all the end-to-end squared distances (for walks with n steps), say `end_sq_dist_tot`, is the sum of all the values from all the self-avoiding walks in all the threads.
- The mean end-to-end squared distance is given by (pay attention to integer division):

$$\langle R_n^2 \rangle = \frac{\texttt{end\_sq\_dist\_tot}}{\texttt{N\_SAW\_tot}}\,. \tag{1.5.2}$$

- Also define a "fraction" parameter $f_{\text{SAW}}(n)$ as follows *(beware of integer overflow).*

$$f_{\text{SAW}}(n) = \frac{\texttt{N\_SAW\_tot}}{N_T N_W}\,. \tag{1.5.3}$$

- *Note: for large values of n, the numbers may become large and overflow, hence it may be better to declare the end-to-end distance and* `N_SAW`*, etc. as* `double` *not* `int`*.*

1.6 Graphs & analysis

- **Write a section "Graphs and analysis" in your cover document.**
- Run your program and create a table of values of $\langle R_n^2 \rangle$ and $f_{\text{SAW}}(n)$ for $n = 10, 11, \ldots, 40$.
- Skip small values of $n < 10$ because we want the "asymptotic behavior" for large values of n.
- **Perform your calculations for self-avoiding random walks in two dimensions.**

n	$\langle R_n^2 \rangle$	$f_{\text{SAW}}(n)$
10	...	...
11	...	...
⋮	⋮	⋮
$n_{\max} = 40$	...	...

- If a maximum value of $n_{\max} = 40$ is too much for your program, you can go up to $n_{\max} = 30$ ***(for reduced credit).***
- **Fill the above table with values and print it in your cover document.**
- Print two charts in your cover document.
  1. Plot the charts using Excel or some other graphing tool.
  2. **Plot a chart of $\langle R_n^2 \rangle$ vs. n.**
  3. **Plot a chart of $f_{\text{SAW}}(n)$ vs. n.**
     (a) *Use a logarithmic scale on the vertical axis of the chart for $f_{\text{SAW}}(n)$.*
     (b) If you have done your work correctly, the points will lie approximately on a straight line in a logarithmic chart.
- If you have done your work correctly, you will find that $\langle R_n^2 \rangle$ is proportional to a power of n and $f_{\text{SAW}}(n)$ is approximately exponential in n. *Note the minus sign in the exponent for β.*

$$\langle R_n^2 \rangle \propto n^{\alpha}\,, \tag{1.6.1}$$

$$f_{\text{SAW}}(n) \propto e^{-\beta n}\,. \tag{1.6.2}$$

- Use Excel (or any other method you wish) to fit a trendline through the data in your chart for $\langle R_n^2 \rangle$ and compute the value of the exponent α.
- Use Excel (or any other method you wish) to fit a trendline through the data in your chart for $f_{\text{SAW}}(n)$ and compute the value of the exponent β.
- **Write the values of α and β in your cover document.**
- If you have done your work correctly, you should obtain $0 < \alpha < 2$ and $0 < \beta < 1$.

1.7 Optimizations

- **Write a section "Software design and optimizations" in your cover document.**
- *Think carefully about what you are being asked to calculate.*
- Do not code the above project description blindly.
- It is possible to make significant optimizations which will greatly speed up your calculations, *or even reduce the calculations you need to do.*
- The questions in the later sections in this document (Sec. 1.8 and after) require more computation and you may need to optimize your code to answer them, so that your program executes in a reasonable time.
- ***A higher grade will be awarded for programs which run faster.***
- This project is as much *(or more)* about software design as about crunching numbers.
- **Write down your optimizations in your cover document.**
- Actually, you must write down your software design in your cover document, before you can explain the "optimizations" that you implemented.

1.8 Self-avoiding random walks in higher dimensions

- **Write a section "Higher dimensions" in your cover document.**
- Successfully programming the calculations for self-avoiding random walks in two dimensions and correctly answering the questions listed in Sec. 1.6 will score a passing grade of C (possibly B if you figure out significant optimizations in Sec. 1.7).
- To score a higher grade you must answer the questions in this section.
- **Perform the calculations listed in Sec. 1.6 for self-avoiding random walks in 3 dimensions.**
- **Perform the calculations listed in Sec. 1.6 for self-avoiding random walks in 4 dimensions.**
- Write a table, draw charts and write down the values of the exponents α and β in 3 and 4 dimensions.
- Denote the values by α_d and β_d for $d = 2, 3, 4$ dimensions, respectively.
- **Your software design to handle higher dimensions will be examined, not just your number crunching results.**

1.9 Self-avoiding polygons

- **Write a section "Self avoiding polygons" in your cover document.**
- To score A+ you must answer the questions in this section correctly.
- A **self-avoiding polygon (SAP)** is a self-avoiding random walk in two dimensions with the special condition that the last point equals the starting point.

$$(x_n, y_n) = (x_0, y_0) = (0, 0)\,. \tag{1.9.1}$$

- Other points in the random walk are not allowed to coincide.
- Write a multi-threaded program with N_T threads.
- In each thread, write an outer loop with N_W passes and calculate the number of self-avoiding polygons with n steps, using pseudorandom numbers as in Sec. 1.5.
- Define a "fraction" parameter $f_{\rm SAP}(n)$ as follows *(beware of integer overflow).*

$$f_{\rm SAP}(n) = \frac{\texttt{total number of SAP with n steps, in all threads}}{N_T N_W}\,. \tag{1.9.2}$$

- **Tabulate the value of $f_{\rm SAP}(n)$ as a function of n, for self-avoiding random walks in two dimensions in your cover document.**

1. If you think about it, n must be an even number because a random walk with an odd number of steps cannot close on itself.
2. Once again, skip small values $n < 10$ and begin your table with $n = 10$.

n	$f_{\rm SAP}(n)$
10	...
12	...
⋮	⋮
$n_{\rm max} = 40$	...

3. **Plot a chart of the above table using Excel (or any other charting tool) in your cover document.**
4. *Use a logarithmic scale on the vertical axis of your chart.*
5. You should find that $f_{\rm SAP}(n)$ decreases approximately exponentially with n (note the minus sign in the exponent).

$$f_{\rm SAP}(n) \propto e^{-\gamma n}\,. \tag{1.9.3}$$

6. **Write the value of the exponent γ in your cover document.**

- **Repeat the calculations for self-avoiding polygons in 3 and 4 dimensions.**
- Technically, they are not "polygons" but "self-avoiding random loops" but never mind.
- Print tables, plot charts and write the values of the exponent γ_d in $d = 2, 3, 4$ dimensions.