Report for Assignment 8

Deep Kiran Shroti
Roll Number- 12EC35013

March 27, 2016

Simulation Using Heaps

Assignment Statement
You are required to write a program to simulate the collision of some balls on a 2D planar region bounded by straight walls.
The simulation is to be done efficiently using the application of the priority queue data structure, which is the objective of this experiment.

Input and Output Specification
Initialize your simulation with a pre-defined area (length and breadth specification), a set of five particles with initial velocities (as per your convenience) and radius. The user may provide a time horizon (100 sec say).Your output should be a plot exhibiting the trajectories of the five particles (in different colors) up to 100 sec.
In a seperate text file, you should log the velocity and position vectors of each particle at the moment of each collision.

1 Introduction

1.1 Heaps

- The (binary) heap data structure is an array that can be viewed as a complete binary tree.

- Each node of the tree corresponds to an element of the array that stores the record with key in the node. Being a complete binary tree, it is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

- The root of the tree is A[1], and given the index i of a node, the indices of its parent PARENT(i), left child LEFT(i), and right child RIGHT(i) can be computed simply as follows.

- PARENT[i] : $\{return \lfloor i/2 \rfloor\}$
- LEFT[i] : $return 2 * i$
- RIGHT[i] : $return 2 * i + 1$

Additionally, heaps also satisfy the **heap property:** for every node i other than the root, A[PARENT(i)] $\leq$ A[i] for a min-heap and A[PARENT(i)] $\geq$ A[i] for a max-heap.

1.2 Priority Queue

The heap data structure itself has enormous utility. A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A priority queue supports the following operations.

- INSERT(S, x) inserts the element x into the set S.
- MAXIMUM/MINIMUM(S) returns the element of S with the largest/smallest key.
- EXTRACT-MAX/MIN(S) removes and returns the element of S with the largest/smallest key.

One application of priority queues is to schedule jobs on a shared computer. The priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

A priority queue can also be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key. The events must be simulated in order of their time of occurrence, because the simulation of an event can cause other events to be simulated in the future. The simulation program uses EXTRACT-MIN at each step to choose the next event to simulate. As new events are produced, they are inserted into the priority queue using INSERT.

2 Implementation Details

1. **Priority Queue & Event Driven Simulation**

   In event driven simulations, Simulation is a collection of important events arranged in terms of there priority(Time of occurance). Unlike time driven simulation where after each (predecided) small time units we check to see which events may happen at the current time point, and handle those that do,

in event-driven simulation the next-event time advance approach is used. For the case of discrete systems this method consists of the following phases:

- Step 1: The simulation clock is initialised to zero and the times of occurrence of future events are determined.
- Step 2: The simulation clock is advanced to the time of the occurrence of the most imminent (i.e. first) of the future events.
- Step 3: The state of the system is updated to account for the fact that an event has occurred.
- Step 4: Knowledge of the times of occurrence of future events is updated and the first step is repeated.

A suitable data structure for handling event driven simulation is **Priority Queue**. Priority queues are a generalization of stacks and queues. Rather than inserting and deleting elements in a fixed order, each element is assigned a priority represented by an integer. We always remove an element with the highest priority, which is given by the minimal integer priority assigned.

Event Driven Simulation Using Priority Queue

1. **Pseudo Code :**

GET_COLLISION_TIME

```
/*
Computes the next possible collision with highest
priority between two balls or a ball and the four
walls

Parameters : PhysicsOject obj1,PhysicsOject obj2

COMPLEXITY : O(1)
*/
GET COLLISION TIME(PhysicsOject obj1,PhysicsOject obj2)
  if 'obj2 != NULL' //COLLISION WITH A BALL
    t = findNextCollisionTime(obj1,obj2)
    if t>0 //VALID COLLISION
      return newHeapNode(t, obj1, obj2)
    else
      error 'CANNOT COLLIDE'
  else //COLLISION WITH WALLS
    Direction = findDirectionofMovement(obj1)
    switch 'Direction'
    {
      case 'TOP/RIGHT'
        computeMinTimeToWalls()
        setFlag(obj1)
      case 'TOP/LEFT'
        computeMinTimeToWalls()
        setFlag(obj1)
      case 'BOTTOM/RIGHT'
        computeMinTimeToWalls()
        setFlag(obj1)
      case 'BOTTOM/LEFT'
        computeMinTimeToWalls()
        setFlag(obj1)
    }
```

SIMULATE_COLLISION

```
/*
Simulates a collision between
two balls or a ball and an wall

Parameters : HeapNode

COMPLEXITY: O(n)
*/
SIMULATE(HeapNode Collision):
  currentTime = Collision.timestamp
  if 'Collision.type != WallCollision'
  {
    Collision.obj1.state =
        computeState(obj1,currentTime)
    Collision.obj2.state =
        computeState(obj2,currentTime)
    updatePath(obj1)
    updatePath(obj2)
    obj1.lastCollision = currentTime
    obj1.lastCollision = currentTime

    //O(n) Time operation
    removeFutureEvents(obj1)
    removeFutureEvents(obj1)
  }
  else /*COLLISION WITH A WALL*/
  {
    if 'COLLISION WITH TOP OR BOTTOM WALL'
      obj1.state.vy = -obj1.state.vy
    else
      obj1.state.vx = -obj1.state.vx
  }
```

PLOT_PATH

```
/*Plotting Function

Parameters : 1

COMPLEXITY : O(n^{2})
In worst case scenario where all possible C(n,2)
    collision has occurred.
*/
PLOT_PATH(World w):
  for i=1:w.size
  {
    Plot the Path of w.Objects[i]
  }
```

MAIN

```
/*
Main function

Parameters : NIL
*/
MAIN():
  //INITIALIZATION PART
  PhysicsObject Objects[M]
  Rect mainWall
  mainWall = newWall(lnth,brdth)
  for i=1:M
  {
    objects[i].loc = getLocationInput()
    objects[i].vel = getVelocityInput()
  }
  for i=1:M
  {
    for j=i:M
    {
      //Statement 1
      t = getCollisionTime(objects[i],objects[j])
      //Statement 2
      heapInsert(HeapOfEvents,t)
    }
    //Statement 3
    t = getCollisionTime(objects[i],mainWall)
    //Statement 4
    heapInsert(HeapOfEvents,t)
  }

  //SIMULATION PART
```

```
while 'HeapOfEvents.heapSize > 0 && currentTime <
    TIME_LIMIT'
  //Statement 5
  c = extractMin(HeapOfEvents)
  SIMULATE_COLLISION(c) //O(n)
  for i=1:M
  {
    //Statements Set 1 -- O(1)
    c1 = getCollisionTime(c.obj1,objects[i])
    c2 = getCollisionTime(c.obj2,objects[i])

    c3 = getCollisionTime(c.obj1,mainWall)
    c4 = getCollisionTime(c.obj2,mainWall)

    //Statements Set 2 -- O(log(n))
    heapInsert(HeapOfEvents,c1)
    heapInsert(HeapOfEvents,c2)
    heapInsert(HeapOfEvents,c3)
    heapInsert(HeapOfEvents,c4)
  }

//Statements Set 3 -- O(n^2)
plotPath()
```

2. **Complexity Analysis:**

- Set (or initialize) the state of objects : $O(n)$
- *INITIALIZATION PART* - Find first set of collision
  - Statement 1 : $O(n)$
  - Statement 2 :

$$log(1) + log(2) + log(3) + ... + log(q) \quad (1)$$
$$= O(log(q!)) \quad (2)$$
$$= O(qlog(q)) \quad (3)$$
$$= O(n^2 log(n)) \quad (4)$$

    * $q = n^2$ because max size of heap in our case : $O(n^2)$
  - Statement 3 : $O(n)$
  - Statement 4 :

$$O(log(n)) + O(log(2n)) + O(log(3n)) + ... + O(log(n^2)) \quad (5)$$
$$= O(log(n!)) + O(nlog(n)) \quad (6)$$
$$= O(nlog(n)) \quad (7)$$

    Hence, the overall complexity of the program for calculating the first set of future events is $O(n^2 log(n))$

- *SIMULATION PART*
  - Statements 5 (SIMULATE_COLLISION) : $O(n)$
  - Statements Set 1 : $O(1)$
  - Statements Set 2 : $O(nlogn)$ (Repopulate EventsHeap)
  - Statements Set 3 (End of Simulation-Plot) : $O(n^2)$

  Hence, each event simulation takes O(n) to complete. At the end of the simulation plotting takes $O(n^2)$ time for smaller amount of simulation-time.