Simulation Algorithms

Simulations are immensely useful, both for profit (design validation) and fun (games such as Angry Birds). Simulations can be optimized with the aid of data structures such as heap-backed priority queues, and by sorting algorithms.

Most simulations have a discrete clock, which means that the simulation timeline consists of small, equally-sized time quanta, and the universe changes at the beginning of each quanta. For example, a physical simulation of the balls on a billiard table would update the ball positions each 10 milliseconds. As long the time quanta are small, users will not realize the discrete nature of the simulation.

Discrete simulations must assume that world changes can be modeled as happening instantaneously, because the changes are computed at the beginning of each quanta. Under this assumption, we can speed up the simulation by figuring out the times when changes happen, instead of simulating every time quantum.

Events

The circuit simulation model in Problem Set 2 conveniently specified that gate output values change instantaneously (in real life, a gate's output voltage transitions continuously between the minimum voltage and the maximum voltage, over a period of time). This allowed our simulator to work by computing the effects caused by changes in gate output values, instead of simulating every time quantum. From now on, we will call these changes **events**.

For every event (gate output change), we looked at the gates whose inputs are connected to the gate's output and, if the input changes caused a change in the gates' output values, we created new events to account for these changes. In order to get the correct results during simulation, the simulator needed to consider the events in chronological order, so it used a priority queue to manage events. Events were added to the queue as they were created, and the simulation code popped the top event (with the minimum step) off the queue at each step.

Line-Sweep Algorithms

The sweep-line algorithm in Problem Set 3 involves simulating a vertical line that sweeps left-to-right through the wire layout, so we can treat it as a simulation problem. This time around, all the events can be computed directly from the input data, before starting the simulation. So we populate a Python list (array) with all the events.

```
class CrossVerifier(object):
  def _events_from_layer(self, layer):
    """Populates the sweep line events from the wire layer."""
    left_edge = min([wire.x1 for wire in layer.wires.values()])
    for wire in layer.wires.values():
      if wire.is_horizontal():
        self.events.append([left_edge, 0, wire.object_id, 'add', wire])
      else:
        self.events.append([wire.x1, 1, wire.object_id, 'query', wire])
```

The events are sorted according to their "time" (the x coordinate of the sweep line) in the `__init__` method of `CrossVerifier`, and then `_compute_crossings` iterates over the sorted list and processes the events in "chronological" order (lines 9-10).

```
 1 class CrossVerifier(object):
 2   def _compute_crossings(self, count_only):
 3     """Implements count_crossings and wire_crossings."""
 4     if count_only:
 5       result = 0
 6     else:
 7       result = self.result_set
 8 
 9     for event in self.events:
10       event_x, event_type, wire = event[0], event[3], event[4]
11 
12       if event_type == 'add':
13         self.index.add(KeyWirePair(wire.y1, wire))
14       elif event_type == 'query':
15         self.trace_sweep_line(event_x)
16         cross_wires = []
17         for kwp in self.index.list(KeyWirePairL(wire.y1),
18                                    KeyWirePairH(wire.y2)):
19           if wire.intersects(kwp.wire):
20             cross_wires.append(kwp.wire)
21         if count_only:
22           result += len(cross_wires)
23         else:
24           for cross_wire in cross_wires:
25             result.add_crossing(wire, cross_wire)
26 
27     return result
```

The "hacked-together" code that ships with Problem Set 3 has two types of events: *add* and *query*. First off, all the horizontal wires are *added* to the range index, keyed according to their y coordinates. Then the sweep line moves across the wire layout from left to right and, every time it "hits" a vertical wire, a *query* event occurs. During a *query*, the range index is asked to provide all the horizontal wires whose y coordinates are between the y coordinates of the vertical wire's endpoints. This list of wires may contain false positives – wires that are completely to the left or completely to the right of the vertical wire, but it will definitely contain all the wires that intersect the vertical wire. Convince yourself that none of the other horizontal wires could possibly intersect the vertical wire.

Comparison Model I

We have learned in lecture that using the comparison model means we can't do searching in better than $\Omega(\log n)$ time, and we can't sort faster than $\Omega(N \log N)$. The advantage of the comparison model is that it can encompass arbitrarily complicated objects, as long as we can define an ordering relationship on them.

The events in Problem Set 3 are Python lists (arrays), because Python conveniently implements an ordering relationship on list, based on lexicographical ordering[1]. We'll call the lists used to represent events *vectors*, to distinguish them from the array that holds all the events.

The first element in an event's vector (main comparison criterion) is the event's x coordinate, so that events are processed left-to-right, modeling a vertical line that sweeps across the entire plane left-to-right.

The next two elements in an event's vector are used to break ties among events that happen at the same sweep line position. They are

1. a small integer representing the event type's priority; this ensures that *add* events at an x coordinate are processed before *query* events at that coordinate

2. a wire ID, which is a unique number; this ensures that the lexicographical comparison algorithm will never go past this element, so it will not attempt to compare `Wire` instance

Last, the inefficient sweep-line algorithm supplied in Problem Set 3 wants to process all the *add* events before any *query* event, so it sets the x coordinate for all the *add* events to be the minimum among the x coordinates of all wire endpoints. Together with setting the priority of *add* events set to 0, this ensures that *add* events come ahead of *query* events during comparisons.

```
class CrossVerifier(object):
  def _compute_crossings(self, count_only):
    """Implements count_crossings and wire_crossings."""
    if count_only:
      result = 0
    else:
      result = self.result_set

    for event in self.events:
      event_x, event_type, wire = event[0], event[3], event[4]

      if event_type == 'add':
        self.index.add(KeyWirePair(wire.y1, wire))
      elif event_type == 'query':
        self.trace_sweep_line(event_x)
        cross_wires = []
        for kwp in self.index.list(KeyWirePairL(wire.y1),
                                   KeyWirePairH(wire.y2)):
          if wire.intersects(kwp.wire):
            cross_wires.append(kwp.wire)
        if count_only:
          result += len(cross_wires)
        else:
          for cross_wire in cross_wires:
            result.add_crossing(wire, cross_wire)

    return result
```

[1]the same ordering used for words in a dictionary, e.g. Alyssa < Andrew

Range Queries

The wire crossing algorithm supplied in Problem Set 3 adds all horizontal wires to a range index, which is later queried to find all the wires that can potentially intersect a vertical wire.

Range Index API

The range index concept is outlined in Problem 1 of Problem Set 3.

```python
class RangeIndex(object):
  """Array-based range index implementation."""

  def __init__(self):
    """Initially empty range index."""
    self.data = []

  def add(self, key):
    """Inserts a key in the range index."""
    if key is None:
        raise ValueError('Cannot insert nil in the index')
    self.data.append(key)

  def remove(self, key):
    """Removes a key from the range index."""
    self.data.remove(key)

  def list(self, first_key, last_key):
    """List of values for the keys that fall within [first_key, last_key]."""
    return [key for key in self.data if first_key <= key <= last_key]

  def count(self, first_key, last_key):
    """Number of keys that fall within [first_key, last_key]."""
    result = 0
    for key in self.data:
      if first_key <= key <= last_key:
        result += 1
    return result
```

A range index supports ADD(x) and REMOVE(x) updates to build up a set of keys, and offers range queries over the keys. COUNT(l, h) returns the number of keys in the range index that belong to the $[l, h]$ range ($l \leq key \leq h$) and LIST(l, h) returns a list of the above-mentioned keys.

The code supplied in `circuit2.py` is optimized for size, not for speed, to convey the concept of a range index. Convince yourself that ADD runs in $O(1)$ time, but REMOVE, COUNT runs in $O(N)$ time, and LIST run in $O(N + K)$ time, where K is the number of keys in the list returned by LIST.

A first cut at optimizing range indexes would use a sorted array.

```python
class BlitRangeIndex(object):
  """Sorted array-based range index implementation."""

  def __init__(self):
    """Initially empty range index."""
    self.data = []

  def add(self, key):
    """Inserts a key in the range index."""
    if key is None:
        raise ValueError('Cannot insert None in the index')
    self.data.insert(self._binary_search(key), key)

  def remove(self, key):
    """Removes a key from the range index."""
    index = self._binary_search(key)
    if index < len(self.data) and self.data[index] == key:
      self.data.pop(index)

  def list(self, low_key, high_key):
    """List of values for the keys that fall within [low_key, high_key]."""
    low_index = self._binary_search(low_key)
    high_index = self._binary_search(high_key)
    return self.data[low_index:high_index]

  def count(self, low_key, high_key):
    """Number of keys that fall within [low_key, high_key]."""
    low_index = self._binary_search(low_key)
    high_index = self._binary_search(high_key)
    return high_index - low_index

  def _binary_search(self, key):
    """Binary search for the given key in the sorted array."""
    low, high = 0, len(self.data) - 1
    while low <= high:
      mid = (low + high) // 2
      mid_key = self.data[mid]
      if key < mid_key:
        high = mid - 1
      elif key > mid_key:
        low = mid + 1
      else:
        return mid
    return high + 1
```

This implementation is a bit more complicated, but runs a lot faster. COUNT runs in $O(\log N)$ time, and LIST runs in $O(\log(N) + K)$ time, but in exchange ADD runs in $O(N)$ time. REMOVE still runs in $O(N)$ time, like before. In a query-heavy workload, `BlitRangeIndex` will vastly outperform `RangeIndex`. If there are a lot of updates, the performance won't be too bad, because the $O(N)$ operations in ADD and REMOVE, as well as the $O(K)$ operation in LIST, are built-in Python functions, which are most likely implemented in C, and therefore have very small constant factors. For many practical cases, this implementation will do very well. However, on really large data sets, this code will be out-performed by an implementation of an algorithm with better asymptotic performance.

Comparison Model II

We can use the comparison model to introduce an ordering relationship between all the wires in the range index – horizontal wires should be ordered by their y coordinate.

The first instinct when implementing this ordering relationship would be to augment the `Wire` class with the right magic methods, so `Wire` instances can be compared using the standard Python operators <, <=, > and >=. However, this approach is "dirty" – What if we later want to order vertical wires by their x coordinate? How about ordering all the wires by their names? Moreover, the approach falls apart when it comes time to do a range query – how would we create keys representing the y coordinates of a vertical wire's endpoints? "Fake" `Wire` instances come to mind, but that's also quite hackish.

Instead, we create a new kind of objects, `KeyWirePair`, which are used as the range index keys.

```python
class KeyWirePair(object):
  """Wraps a wire and the key representing it in the range index.

  Once created, a key-wire pair is immutable."""

  def __init__(self, key, wire):
    """Creates a new key for insertion in the range index."""
    self.key = key
    self.wire = wire
    self.wire_id = wire.object_id
```

A `KeyWirePair` instance that is inserted into the range index has its `key` set to the wires' y coordinate. Comparisons use the `key` field as the primary criterion, and then use `wire_id` (a wire's unique ID) to break ties. Two keys with the same `wire_id` represent the same wire, so they are equal. It doesn't make sense to insert the same wire into the range index twice, so the data structure used to implement the range index doesn't have to deal with equal keys.

```python
class KeyWirePair(object):
  def __lt__(self, other):
    return (self.key < other.key or
            (self.key == other.key and self.wire_id < other.wire_id))

  def __le__(self, other):
    return (self.key < other.key or
            (self.key == other.key and self.wire_id <= other.wire_id))

  def __gt__(self, other):
    return (self.key > other.key or
            (self.key == other.key and self.wire_id > other.wire_id))

  def __ge__(self, other):
    return (self.key > other.key or
            (self.key == other.key and self.wire_id >= other.wire_id))

  def __eq__(self, other):
    return self.key == other.key and self.wire_id == other.wire_id

  def __ne__(self, other):
    return self.key == other.key and self.wire_id == other.wire_id
```

Note the special method names (e.g., `__le__` means *less than or equal to* and is used to implement the <= operator).

`KeyWirePair` instances used for range queries have the `wire` field set to `None`. To facilitate range queries, we subclass `KeyWirePair` with `KeyWirePairL` and `KeyWirePairH`, which are meant to be used as the low and high keys in range queries.

```
class KeyWirePairL(KeyWirePair):
  def __init__(self, key):
    self.key = key
    self.wire = None
    self.wire_id = -1000000000

class KeyWirePairH(KeyWirePair):
  def __init__(self, key):
    self.key = key
    self.wire = None
    self.wire_id = 1000000000
```

Asides from setting `wire` to `None`, the special classes set the `wire_id` field to special values that are guaranteed to be smaller / greater than all the IDs of real wires. This means that a `KeyWirePairL` with a certain key (y coordinate) will be considered lower than all the keys for wires with the same y coordinate and, conversely, a `KeyWirePairH` will be higher than the keys of wires with the same y coordinate. This implementation trick eliminates some boundary cases for range queries, because the keys in the index are guaranteed to not be equal to the keys used in queries.

The need of being able to bound wire IDs in `KeyWirePairL` and `KeyWirePairH` pushed us away from Python's built-in `id()` function, which provides unique IDs for all Python objects. Instead, the verifier implements its own ID scheme for `Wire` instances.

```
class Wire(object):
  def __init__(self, name, x1, y1, x2, y2):
    # Normalize the coordinates.
    if x1 > x2:
      x1, x2 = x2, x1
    if y1 > y2:
      y1, y2 = y2, y1

    self.name = name
    self.x1, self.y1 = x1, y1
    self.x2, self.y2 = x2, y2
    self.object_id = Wire.next_object_id()

  # Next number handed out by Wire.next_object_id()
  _next_id = 0

  @staticmethod
  def next_object_id():
    """Returns a unique numerical ID to be used as a Wire's object_id."""
    id = Wire._next_id
    Wire._next_id += 1
    return id
```

Wire IDs are consecutive non-negative integers, so it's reasonable to assume they will be bounded by 1,000,000,000 (a billion), because storing a billion wires would require a lot of RAM.

LIST Range Queries in BSTs

Given a binary search tree (BST), we want to implement LIST$(tree, l, h)$ which produces a list of all the keys between l and h ($l \leq key \leq h$), in $O(\log(N) + K)$ time, where N is the number of keys in the tree, and K is the number of keys output by LIST.

We will convince ourselves that the pseudo-code below will do the trick.

```
LIST($tree, l, h$)
1  $lca = $ LCA$(tree, l, h)$
2  $result = []$
3  NODE-LIST$(lca, l, h, result)$
4  return $result$
```

First off we identify the smallest subtree that contains all the keys between l and h. Intuitively, LCA computes the root of that subtree. Formally, LCA computes the lowest-common ancestor of nodes with keys l and h. If l and h do not exist in the tree, LCA returns the lowest-common ancestor of the two nodes that would be created by inserting l and h.

```
LCA($tree, l, h$)
1  $node = tree.root$
2  until $node$ == NIL or ($l \leq node.key$ and $h \geq node.key$)
3      if $l < node.key$
4          $node = node.left$
5      else
6          $node = node.right$
7  return $node$
```

Once the subtree's root is found by LCA, NODE-LIST performs an in-order traversal of the subtree, but prunes the subtrees that are guaranteed to contain keys outside the $[l, h]$ range.

```
NODE-LIST($node, l, h, result$)
1  if $node$ == NIL
2      return
3  if $node.key \geq l$
4      NODE-LIST($node.left, l, h, result$)
5  if $l \leq node.key$ and $node.key \leq h$
6      ADD-KEY($result, node.key$)
7  if $node.key \leq h$
8      NODE-LIST($node.right, l, h, result$)
```