Heaps

https://cs.pomona.edu/classes/cs140/

Outline

Topics and Learning Objectives

- Discuss data structure operations
- Cover heap sort
- Discuss heaps

Exercise

• Heap practice

Extra Resources

• Introduction to Algorithms, 3rd, chapter 6

Data Structures

Used in essentially every single programming task that you can think of

- What are some examples of data structures?
- What are some example programs?

What do they do?

- They organize data so that it can be effectively accessed.
- A data structure is not necessarily a method of laying out data in memory
- It is a way of logically thinking about your data.

The Heap Data Structure (not heap memory)

A container for objects that have key values (Sometimes called a "Priority Queue")

Operations:

- Insertion : O(lg n)
- Extract-min (or max) : O(lg n)
- Heapify : O(n) for batched insertions
- Arbitrary Deletion : O(lg n)
- Good for continually getting a minimum (or maximum) value

Selection sort

- Continually look for the smallest element
- The element currently being considered is in blue
- The current smallest element is in red
- Sorted elements are in yellow



Selection sort

- Continually look for the smallest element
- The element currently being considered is in blue
- The current smallest element is in red
- Sorted elements are in yellow



Selection sort

- Continually look for the smallest element
- The element currently being considered is in blue
- The current smallest element is in red
- Sorted elements are in yellow



Selection sort

Continually look for the smallest element

What is the runtime of selection sort? How can we make it faster with a heap? (Breakout)

With a heap: $O(n^2) \rightarrow O(n \lg n)$

- Insert all elements into a heap: n
- Extract each element: n * lg n



Example : Event Manager

Uses a priority queue (synonym for Heap)

Example: simulation or game

- play sounds
- render animation
- detect collisions
- register input

Heap Implementation

Conceptually you should think of a Heap as a binary tree But it is implemented using an array (why?)

Heap Property: for any given node x,

- 1. $key[x] \le key[x's left child]$, and
- 2. $key[x] \le key[x's right child]$

Where is the minimum key?

No pointer following Fewer heap-allocations No pointer storing

Heap Implementation

Note: Heaps are **not** unique

You can have multiple different configurations that hold the same data







How do you calculate the indices of a node's children?

parent_index = (node_index - 1) // 2

left_child_index = 2 * node_index + 1

right_child_index = 2 * (node_index + 1)

Level	3

Value	4	4	8	9	4	12	9	11	13
Index	0	1	2	3	4	5	6	7	8
Parent	-	0	0	1	1	2	2	3	3
Left Child	1	3	5	7	9	11	13	15	17
Right Child	2	4	6	8	10	12	14	16	18

Exercise


















```
FUNCTION Dijkstra(G, start_vertex)
   found = \{\}
   lengths = {v: INFINITY FOR v IN G.vertices}
   found.add(start_vertex)
   lengths[start_vertex] = 0
   WHILE found.length != G.vertices.length -
      FOR v IN found
         FOR vOther, weight IN G.edges[v]
            IF vOther NOT IN found
               vOther length = lengths v + weight
               IF vOther length < min length
                  min_length = vOther_length
                  vMin = vOther
      found.add(vMin)
      lengths[vMin] = min_length
```

RETURN lengths

What is the running time?

How many times does the outer loop run?

O(n)

How many times do the inner two loops run?

O(m)

```
FUNCTION Dijkstra(G, start vertex)
   found = \{\}
   lengths = {v: INFINITY FOR v IN G.vertices}
   found.add(start_vertex)
   lengths[start_vertex] = 0
   WHILE found.length != G.vertices.length
      FOR v IN found
         FOR vOther, weight IN G.edges[v]
            IF vOther NOT IN found
               vOther_length = lengths[v] + weight
               IF vOther_length < min_length
                  min length = vOther length
                  vMin = vOther
      found.add(vMin)
      lengths[vMin] = min_length
```

RETURN lengths

What is the running time?

We can bring this down to O(m lg m) with a simple change.

State of the art of Dijkstra's: O(m + n lg n) (uses Fibonacci heap)

```
def dijkstras heap(adjacency list, start vertex):
    """Dijkstra's Algorithm implemented with all vertices placed in a heap.
    This version of Dijkstra's Algorithm has a running time of O(m lg m).
    11 11 11
    n = len(adjacency list)
    path lengths = {v: inf for v in adjacency list}
    predecessors = {v: None for v in adjacency list}
    path lengths[start vertex] = 0
    predecessors[start vertex] = None
    found = set()
    vertex min heap = [(path lengths[start vertex], start vertex)]
    while len(found) != n:
        vfrom length, vfrom = heappop(vertex min heap)
        found.add(vfrom)
        for vto, weight in adjacency list[vfrom]:
            path length = vfrom length + weight
            if path length < path lengths[vto]:</pre>
                path lengths[vto] = path length
                predecessors[vto] = vfrom
                heappush(vertex min heap, (path lengths[vto], vto))
    return path lengths, predecessors
```

```
while len(found) != n:
    vfrom_length, vfrom = heappop(vertex_min_heap)
    found.add(vfrom)
    for vto, weight in adjacency_list[vfrom]:
        path_length = vfrom_length + weight
        if path_length < path_lengths[vto]:
            path_lengths[vto] = path_length
            predecessors[vto] = vfrom
```

heappush(vertex_min_heap, (path_lengths[vto], vto))

def print path(end vertex, predecessors):

```
path = [end_vertex]
pred = predecessors[end vertex]
```

```
while pred is not None:
```

```
path.append(pred)
pred = predecessors[pred]
```

print(" -> ".join([str(v) for v in reversed(path)]))

Dijkstra's Algorithm Correctness

Theorem:

• Dijkstra's algorithm will find the shortest path from the start vertex to every other vertex on any graph with non-negative weights.

Proof using a loop invariant. Loop predicate:

• At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set

Initialization

• Initially, the found set is empty. So, the invariant is trivially true. Loop predicate/invariant: At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set

```
found = set()
while len(found) != n:
    vfrom length, vfrom = heappop(vertex min heap)
    found.add(vfrom)
    for vto, weight in adjacency list[vfrom]:
        path length = vfrom length + weight
        if path length < path lengths[vto]:
            path lengths[vto] = path length
            predecessors[vto] = vfrom
            heappush (vertex min heap,
                     (path lengths[vto], vto))
```

Maintenance (1)

- Assume all previous iterations have produced the correct shortest path for all vertices in the found set.
- For purposes of a contradiction, assume that when a vertex u is added to the found set its path length is <u>not</u> optimal.
- At the time u is found we must have some path to u

Loop predicate/invariant: At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set

```
while len(found) != n:
    vfrom_length, vfrom = heappop(vertex_min_heap)
    found.add(vfrom)
```

```
for vto, weight in adjacency_list[vfrom]:
    path_length = vfrom_length + weight
    if path_length < path_lengths[vto]:
        path_lengths[vto] = path_length
        predecessors[vto] = vfrom</pre>
```

Maintenance (1)

- Assume all previous iterations have produced the correct shortest path for all vertices in the found set.
- For purposes of a contradiction, assume that when a vertex u is added to the found set its path length is <u>not</u> optimal.
- At the time u is found we must have some path to u

Loop predicate/invariant: At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set

while len(found) != n:
 vfrom_length, vfrom = heappop(vertex_min_heap)

found.add(vfrom)

```
for vto, weight in adjacency_list[vfrom]:
    path_length = vfrom_length + weight
    if path_length < path_lengths[vto]:
        path_lengths[vto] = path_length
        predecessors[vto] = vfrom</pre>
```

Maintenance (1)

- For purposes of a contradiction, assume that when a vertex u is added to the found set its path length is <u>not</u> optimal.
- At the time u is found we must have some path to u
- To have a shorter path to u, it must go through some vertex k not in found.
- But since we only have positive edges, a shorter path going through k, means that k must have been chosen before u. Contradiction.

Loop predicate/invariant: At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set

while len(found) != n:

vfrom_length, vfrom = heappop(vertex_min_heap)
found.add(vfrom)

```
for vto, weight in adjacency_list[vfrom]:
    path_length = vfrom_length + weight
    if path_length < path_lengths[vto]:
        path_lengths[vto] = path_length
        predecessors[vto] = vfrom</pre>
```

Termination

found set.

Loop predicate/invariant: At the start of each iteration of the while loop, the shortest path has been found for every vertex in the found set

• Given the loop invariant the shortest path to all vertices have been calculated.

• The loop terminates when all

vertices have been added to the

while len(found) != n: vfrom_length, vfrom = heappop(vertex_min_heap) found.add(vfrom)

```
for vto, weight in adjacency_list[vfrom]:
    path_length = vfrom_length + weight
    if path_length < path_lengths[vto]:
        path_lengths[vto] = path_length
        predecessors[vto] = vfrom</pre>
```