CSCI 136: Data Structures and Advanced Programming
Lecture 33
Priority Queues / Dijkstra’s Algorithm

Instructor: Dan Barowy Williams

Topics
Student Course Surveys
Priority Queues
Dijkstra’s algorithm

Your to-dos
1. **Review readings** from Bailey.
2. **Study** for the final exam.
   a. **Pro tip:** **review quizzes**.
   b. **Do problems** in study guide/practice exam.
   c. **Don’t stress out!** Just be methodical and do your best.
3. **Work on resubmissions** you plan to submit.

Announcements
1. **No lab** this week.
2. **No colloquium** this week.
3. Instead: **end of year ice cream social** on Friday.
Evaluation Forms

(all of these are anonymous)

We care a lot about what you say in these forms. Please take your time and write thoughtful responses.

Your feedback is very valuable to us!

Purpose of SCS Forms

“[T]he SCS provides instructors with feedback regarding their courses and teaching. The faculty legislation governing the SCS provides that SCS results are made available to the appropriate department chair, the Dean of the Faculty, and at appropriate times, to members of the Committee on Appointments and Promotions (CAP). The results are considered in matters of faculty reappointment, tenure, and promotion.”

—Office of the Provost, Williams College

Purpose of “Blue Sheets”

Student comments on the blue sheets [...] are solely for your benefit. They are not made available to department or program chairs, the Dean of the Faculty, or the CAP for evaluation purposes.

—Office of the Provost, Williams College
Blue sheet prompts:

* What course topic did you **enjoy the most**?

* What course topic did you **least enjoy**? Do you think that it was valuable to learn anyway?

* Are there other aspects of the course that you **liked** or **disliked**? (E.g., office hours, TAs, assignments, course structure, meeting times, etc.) Feel free to suggest an alternative approach.

* Did you **look forward to coming to class**?

(Binary) max heap

Max heap property: for any given node \( n \), if \( p \) is a parent node of \( n \), then the key of \( p \) is \( \geq \) the key of \( n \).

A **binary heap** is usually implemented as an **always-complete binary tree**.

Suppose we want to insert a new node,
First, **insert** the new node at the first available position in the tree that **maintains completeness**.

Next, **compare** the new node with its parent.

If the **max heap property** is violated, **swap**.

**Continue swapping** the new node with parents until the **max heap property is satisfied**.
**Insertion**

Continue swapping the new node with parents until the max heap property is satisfied (parent ≥ node or no parents remain).

**Find-max**

To find the maximum element in a max heap, simply return the root.

---

**Insertion**

The swapping procedure performed on insert is often referred to as heap-up or percolate-up.

**Extract**

To remove and return the maximum element in a max heap, first perform find-max.
Extract

Temporarily store the max element.

Replace the root with the last element in the complete tree.

Extract

Replace the root with the last element in the complete tree.

Extract

Compare the root with its children. Swap the root with the largest element.
Compare the root with its children. Swap the root with the largest element.

23 ≥ 42 ?
No.

Continue swapping until the max heap property is satisfied (parent ≥ node or no parents remain).

23 ≥ -1 ?
Yes.

Return the saved maximum element.

The swapping procedure performed on extract is often referred to as heap-down or percolate-down.
A binary heap is often implemented using an implicit binary tree data structure. In other words, heap nodes are actually stored in an array or vector.

leftChild(i) = 2 × i + 1
rightChild(i) = 2 × i + 2
parent(i) = ⌊(i − 1) / 2⌋
Max heap in action

left child
right child

Max heap in action

left child
right child

Max heap in action

left child
right child

Max heap in action

left child
right child

Done!
Advantages:
- **find-max**: $O(1)$
- **insert**: $O(\log n)$
- **extract**: $O(\log n)$

Recall the example from our first class

---

Lots of interesting variants on heaps!

From Wikipedia: *priority queue* page.
Graphs: shortest paths

Shortest path problem

The **shortest path problem** is the problem of finding a *path between two vertices* in a graph such that the sum of the weights of its constituent edges is minimized.
Dijkstra’s algorithm

- **Invented by Edsger Dijkstra in 1959.**
- The original version used a min-priority queue.
- Designed using pencil and paper; algorithm was intended to demonstrate to non-technical people how computers could be useful.
Looking for path from A to F.

```
function Dijkstra(graph, source):
    create vertex set Q
    for each vertex v in graph:
        dist[v] = INFINITY
        prev[v] = UNDEFINED
    add v to Q
    dist[source] = 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each v neighbor of u:
            alt = dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u
        return dist[], prev[]
```

Looking for path from A to F.

```
function Dijkstra(graph, source):
    create vertex set Q
    for each vertex v in graph:
        dist[v] = INFINITY
        prev[v] = UNDEFINED
    add v to Q
    dist[source] = 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each v neighbor of u:
            alt = dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] = alt
                prev[v] = u
        return dist[], prev[]
```
1. function Dijkstra(graph, source):
2.  create vertex set Q
3.  for each vertex v in graph:
4.    dist[v] = INFINITY
5.    prev[v] = undefined
6.  add v to Q
7.  dist[source] = 0
8.  while Q is not empty:
9.    u = vertex in Q with min dist[u]
10.   remove u from Q
11.   for each neighbor v of u:
12.      alt = dist[u] + length(u, v)
13.      if alt < dist[v]:
14.        dist[v] = alt
15.        prev[v] = u
16.   return dist[], prev[]

Looking for path from A to F.
Looking for path from A to F.
```python
function Dijkstra(Graph, source):
    create vertex set Q
    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0
    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q
        for each neighbor v of u:
            // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u
    return dist[], prev[]
```

Looking for path from A to F.

Done!

Read prev backward from F and reverse.

Applications

Delivery routes.

Graphs: traveling salesperson
Applications
Optimal 49,687-stop pub crawl

Recap & Next Class

Today:
- Priority queues
- Heaps

Next class:
- Dijkstra’s algorithm

http://www.math.uwaterloo.ca/tsp/