Project SP2 Feedback

- Grading completed by tomorrow
- Write your names in all files
- More documentation.
- Do not include class files, large inputs
- In your source file /* Sample input: } small input.
  Sample output:
  :
  */

Zip file

Gxx_.... folder

Files, subfolders

Partition with 3 pivots

start:

$x_1\ x_2$ [unprocessed] x_3

$x_1 \le x_2 \le x_3$

Invariant:

x_1 [$< x_1$] [$x_1 - x_2$] [→ ? ←] [$x_2 - x_3$] [$> x_3$] x_3

Selection problem

Selection problem Select(A, p, r, k) - finding k largest elements of A[p..r]

External version - used a PQ (Minheap) to hold k largest seen so far.

Internal version - memory size is enough to hold entire array.

Idea: q ← Partition (A, p, r)

[$\le x$ | x | $> x$]

p ... q-1, q, q+1 ... r

size of small set $= q - p$; 1; size of big set $= r - q$

if $r - q \ge k$ then
  return Select (A, q+1, r, k)
else if $r - q + 1 = k$ then
  return A[q..r]
else //
  return Select(A, p, q-1, k-(r-q+1))
  ∪ A[q..r]

return Select (A, p, q-1, k - (r-q+1)) ∪ A[q..r]

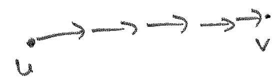
Expected RT of select = O(n) — take CS 6363.

Shortest Paths

Framework: Directed graph $G = (V, E)$, source node $s \in V$,

Weights (lengths) $w: E \to \mathbb{R}$.

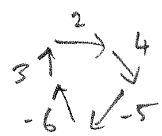
Given a path P from u to v,



$$w(P) = \sum_{e \in P} w(e)$$

weight of P, length of P

Definition: A cycle C is called a negative cycle if $w(C) < 0$ $\left(\sum_{e \in C} w(e) < 0 \right)$

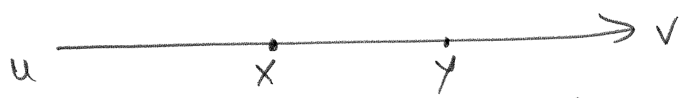


$w(C) = -2$

shortest path problem: For each $u \in V$, find a simple path from s to u whose weight is a minimum (among all simple paths from s to u)

Basis of s.p. ~~problem~~ algorithms

Theorem: Let P_{uv} be a shortest path from u to v



that has P_{xy} as a subpath. Then P_{xy} is a shortest path from x to y. — true ~~only~~ in graphs with no negative cycles.

$\Rightarrow$ All shortest path algorithms (that run in polynomial time) work only in graphs without negative cycles.

↓ G has no negative cycles — assumption.

(a) $w(e) = 1$ for all $e \in E$

↳ BFS (breadth first search) — $O(E)$

(b) Dynamic program — Bellman-Ford algorithm — $O(V \cdot E)$

(c) $w(e) > 0$ for all $e \in E$ — Dijkstra's algorithm — $O(E \log V)$

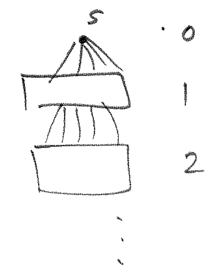
↳ similar to Prim's

(d) G is a DAG : DAG-shortest-paths – $O(E)$

Breadth-first search (BFS)

Idea: Queue to hold vertices that have been seen so far
When processing a node, add its unseen neighbors
to the Queue.

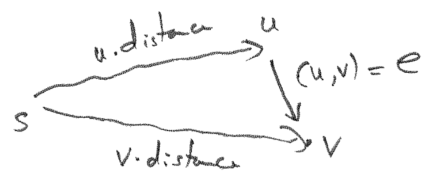
Layered graph




```
// Initialization
for each u ∈ V do { u.distance ← ∞
                    u.parent ← null }
                    u.seen ← false
Create a queue of vertices Q ← {s}
~~s.seen ←~~ s.distance ← 0     s.seen ← true
while Q is not empty do
      u ← Q.remove()
      for each edge e = (u,v) ∈ u.Adj do
            if !v.seen then
                  v.distance ← u.distance + 1
                  v.parent ← u
                  v.seen ← true
                  Q.add(v)
```

Utility functions for all shortest path algorithms where $w(e) \neq 1$ for all e.

```
Initialize(s)
    for u ∈ V do { u.distance ← ∞ ; u.parent ← null; u.seen ← false }
    s.distance ← 0
```




```
boolean Relax(u, v, e)
  // Seek a shorter path to v
     by going through u
  if v.distance > u.distance + e.Weight then
        v.distance ← u.distance + e.Weight
        v.parent ← u
        If there is a PQ based on u.distance as
         priority of u, then
             PQ.decreaseKey(v)
```

```
DualPivotPartition(A, p, r)
// use only when r-p+1 > 2

Select 2 elements of A[p..r] at random, and exchange
smaller element with A[p] and larger element with A[r].
x1 <-- A[p];    x2 <-- A[r]
//Precondition: x1 <= x2
i <-- p+1;      l <-- p+1;      j <-- r-1
unproc <-- r-p-1  // (r-p+1) - 2

//Loop invariant: A[p] = x1,   A[p+1..l-1] < x1,
// x1 <= A[l..i-1] <= x2,   A[i..j] - unprocessed,
// A[j+1..r-1] > x2,   A[r] = x2,  unproc = j-i+1

while unproc > 0 do
    while A[i] <= x2 and unproc > 0 do
        if A[i] >= x1 then  // Case 1
            i++;   unproc--
        else  // Case 2
            Exchange A[l] <--> A[i]
            l++;   i++;   unproc--
    if unproc <= 0 then break
    while A[j] > x2 and unproc > 0 do  // Case 3
        j--; unproc--
    if unproc <= 0 then break
    if A[j] < x1 then // Case 4
        if l != i then
            tmp <-- A[l]
            A[l] <-- A[j]
            A[j] <-- A[i]
            A[i] <-- tmp
        else  // Edge case when S_2 is empty
            Exchange A[i] <--> A[j]
        l++
    else // Case 5
        Exchange A[i] <--> A[j]
    i++;   j--;   unproc -= 2
Exchange A[p] <--> A[l-1]
Exchange A[j+1] <--> A[r]
Return {l-1,j+1}
```



```
K-largest (A,k)   // Find the k largest elements of an array A
    q ← Select (A, 0, A.length - 1, k)
    Return A[q .. A.length - 1]

Select (A, p, r, k)   // Find index of the kth largest
                         element of A[p..r]
    q ← Partition (A, p, r)
    // A[p..r] has been rearranged such that
    // A[p..q-1] ≤ A[q] < A[q+1..r]
    // Size of big side = r - q
    if r - q ≥ k then
        return Select (A, q+1, r, k)
    else if r - q + 1 = k then
        return q
    else return Select (A, p, q-1, k - (r - q + 1))
```