香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Further Brute Force

Isaac Chan {snowysecret}, Bosco Wang {happypotato}

2025-07-02

Table of Contents

Table of Contents

Today's Lecture

- Today's lecture will not be about Brute Force. Instead we will look into *"fracturing search"* in an exponential state space.
- We will go through a few example problems, then you will have some time to attempt a few problems as a group!
- Questions will come up throughout the lecture, and cards (up to 8) will be awarded for correct answers. Only questions enclosed in a red box similar to the one below will be counted.

Example Question:
Is P = NP?

Review: Dijkstra's Algorithm

- Single-source shortest path algorithm
- Requires edge weights to be non-negative

```
REPEAT {
  choose an unfinalized node with minimum distance
  mark it as finalized
  update the neighbours' distance
} UNTIL (all nodes are finalized)
```

Table of Contents

Motivation: Tree K Smallest Values

Problem 1:
Suppose that you have a (possibly infinite) rooted tree where each vertex i has a value v[i]. Also, if i is not the root then i has a parent p[i] satisfying $v[p[i]] \leq v[i]$. Given that each vertex has at most D children, find the K smallest values in the tree.

Motivation: Tree K Smallest Values

Problem 1:
Suppose that you have a (possibly infinite) rooted tree where each vertex i has a value v[i]. Also, if i is not the root then i has a parent p[i] satisfying $v[p[i]] \leq v[i]$. Given that each vertex has at most D children, find the K smallest values in the tree.

Question:
Describe clearly an algorithm to solve this problem when K = 2.

Motivation: Tree K Smallest Values

Problem 1:
Suppose that you have a (possibly infinite) rooted tree where each vertex i has a value v[i]. Also, if i is not the root then i has a parent p[i] satisfying $v[p[i]] \leq v[i]$. Given that each vertex has at most D children, find the K smallest values in the tree.

Answer:
Firstly, the root of the tree must be the smallest value. Then, we can check all the direct children of the root, the smallest among them is the 2nd smallest value.

Motivation: Tree K Smallest Values

Problem 1:
Suppose that you have a (possibly infinite) rooted tree where each vertex i has a value v[i]. Also, if i is not the root then i has a parent p[i] satisfying $v[p[i]] \leq v[i]$. Given that each vertex has at most D children, find the K smallest values in the tree.

Question:
Describe clearly an algorithm to solve this problem when K = 3.

Motivation: Tree K Smallest Values

Problem 1:
Suppose that you have a (possibly infinite) rooted tree where each vertex i has a value v[i]. Also, if i is not the root then i has a parent p[i] satisfying $v[p[i]] \leq v[i]$. Given that each vertex has at most D children, find the K smallest values in the tree.

Answer:
To find the third smallest value, the possible candidates are: All children of the root which are not the 2nd maximum, **and** all children of the 2nd maximum itself! (Finding 1st and 2nd smallest is the same.)

Motivation: Tree K Smallest Values

Can we generalize the solution to general K?

Motivation: Tree K Smallest Values

Can we generalize the solution to general K?

- Consider maintaining a priority queue consisting of the *candidates* for the next smallest value in the tree.
- Initially, the priority queue should only consist of the root, as it is the only (obvious) candidate for the 1st minimum.

Motivation: Tree K Smallest Values

Can we generalize the solution to general K?

- Consider maintaining a priority queue consisting of the *candidates* for the next smallest value in the tree.
- Initially, the priority queue should only consist of the root, as it is the only (obvious) candidate for the 1st minimum.
- For each iteration (i = 1, 2, ..., K),
  - Extract the minimum from the priority queue
  - That value is regarded as the i-th minimum in the tree
  - Now, insert all its children to the priority queue as they are now candidates for the next minimum
- Kind of like “Dijkstra on a tree”!

Motivation: Tree K Smallest Values

- Now, why do we care about this specific problem so much?

Problem 1:
Suppose that you have a (possibly infinite) rooted tree where each vertex i has a value v[i]. Also, if i is not the root then i has a parent p[i] satisfying $v[p[i]] \leq v[i]$. Given that each vertex has at most D children, find the K smallest values in the tree.

Fracturing Search: Generalization of Problem 1

Let's generalize Problem 1 as follows: Suppose that you want to find the **K smallest (or largest) objects** in some (potentially very large) search space.

- For example, K "largest/smallest" **combinations** of things is something that this generalization can help a lot with
  - E.g. K-th Minimum Spanning Tree: unordered combinations of edges
  - E.g. K-th Shortest Walk: ordered combinations of edges

Fracturing Search: Generalization of Problem 1

- First, we need to impose a **tree structure** on the search space.
- Start with the entire search space (the “root”) and then we fracture it into **disjoint subspaces** based on its children.
- Use a priority queue or any heap structure to **keep track of candidates**.
- ```
  REPEAT {
    1. choose the smallest node in the priority queue
    2. pick this node
    3. add all its children to the priority queue
  } UNTIL (picked K values)
  ```
  ```

Fracturing Search vs Dijkstra's Algorithm

```
REPEAT {
  choose the smallest node in
  the priority queue
  pick this node
  add all its children to the
  priority queue
} UNTIL (picked K values)
```

```
REPEAT {
  choose an unfinalized node
  with minimum distance
  mark it as finalized
  update the neighbours'
  distance
} UNTIL (all nodes finalized)
```

(... vs Prim's Algorithm?)

Fracturing Search: Generalization of Problem 1

It turns out that whilst Problem 1 is easy to solve, it is not that true for general Fracturing Search problems.

Some key questions:

1. **What are the states themselves?** (Might not be as simple as “values in a tree node”)
2. How to define the **parent-child relationships** between states in the search space? (In problem 1, the tree is given; generally, build your own tree)
3. How to make sure the **structure is a tree**, not a DAG or anything else?
4. How to make sure **all candidates are considered**?

K Smallest Subset Sums

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

K Smallest Subset Sums

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

Example:
For the input N = 4, A = [1, 2, 3, 6], what are the K = 5 smallest subset sums?
1st smallest: 0 ([])
2nd smallest: 1 ([1])
3rd smallest: 2 ([2])
4th smallest: 3 ([3])
5th smallest: 3 ([1, 2])

K Smallest Subset Sums

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

Consider a Fracturing Search approach.

What are the states?

- The state should store enough info to (1) give us the answer (2) find the immediate children (3) ensure no subset is visited twice
- Sum of subset? → Given a sum S, we can hardly tell what its children would be.
- The subset itself? → Sure, but we don't want to store O(N) things in a state.

K Smallest Subset Sums: Idea 1

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

Consider the state (**s** = sum of subset, **i** = first *undecided* index), which means we have a subset of sum s and we have already *decided* whether or not to include [1, i – 1] in the subset. We will then consider index i.

Initial state: (0, 1)

State transitions: upon extracting (s, i) from priority queue, consider (s, i + 1) or (s + A[i], i + 1).

K Smallest Subset Sums: Idea 1

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

```
pq = empty-min-pq
pq.push(0, 1)
repeat K times:
    sum, i = pq.top()
    pq.pop()
    output sum   // the next minimum
    if i ≤ N:
        pq.push((sum, i+1))
        pq.push((sum+a[i], i+1))
```

... but is this correct?

K Smallest Subset Sums: Idea 1

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

In fact, it is **wrong** because some subset get visited multiple times.

For example, when the states (0, 1), (0, 2), (0, 3), ..., (0, N+1) all gets visited, and they are all referring to the empty subset {}.

This code fails because **the state search structure is a DAG instead of a tree, hence double counting**.

K Smallest Subset Sums: Idea 2

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

Let's try to fix this issue of double counting.

- The basic idea is to **prevent a transition that purely advance that state**, but do not change the underlying subset.
- Each transition should bring us to new subset, so we can handle O(K) transition to find the K smallest subset sum.
- Try to use the following state: (**s** = sum of subset, **i** = last ***taken*** index)

K Smallest Subset Sums: Idea 2

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

The state (**s =** sum of subset, **i =** last ***taken*** index)means we have a subset of sum s and we have already *decided* whether or not to include [1, i – 1] in the subset **and includes index i-1**. We will then consider index i.

Initial state: (0, 1)

State transitions:

Go from (s, i) to (s + a[i+1], i+1), (s + a[i+2], i+2) up to (s + a[N], N)

K Smallest Subset Sums: Idea 2

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

```
pq = empty-min-pq
pq.push(0, 1)
repeat K times:
    sum, i = pq.top()
    pq.pop()
    output sum   // the next minimum
    loop j from i+1 to N:
        pq.push((sum+a[j], j))
```

Time Complexity:
O(NK log NK) <- terrible!

K Smallest Subset Sums: Idea 2

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

Why is it so **slow**?

- It is because each node visited a lot of other nodes, i.e. the **outdegree is too large.**
- We would want **every node** to have its **outdegree constant-bounded**.
- That way, O(K) state transition actually means O(K) nodes considered, won't be dragged down by the long tail of immediate unvisited nodes.

This seems quite hard, but it is actually possible for this task!

K Smallest Subset Sums: Idea 3

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

Let's try to fix the out-degree issue.

We still keep the state (**s** = sum of subset, **i** = last *taken* index), but do something different for the transitions.

Initial state: (0, 1)

State transitions (assumed **a is sorted in ascending order**):

Go from (s, i) to (s + a[i+1], i+1) and (s - a[i] + a[i+1], i+1).

K Smallest Subset Sums: Idea 3

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

State transitions (assumed **a is sorted in ascending order**):

Go from (s, i) to (s + a[i+1], i+1) and (s - a[i] + a[i+1], i+1).

Sorting a in ascending order is important:

- It ensures that -a[i] + a[i + 1] is non-negative
- Each state transition should result in non-negative addition to answer
- Else the K state visited == K-th smallest promise will not hold

K Smallest Subset Sums: Idea 3

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

Now we can follow the pseudo-code below to solve this task:

```
pq = empty-min-pq
pq.push(0, 1)
repeat K times:
    sum, i = pq.top()
    pq.pop()
    output sum   // the next minimum
    if i ≤ N:
        pq.push((sum-a[i]+a[i+1], i+1))    // drop a[i], take a[i+1]
        pq.push((sum+a[i+1], i+1))         // take a[i+1]
```

Time Complexity:
O(K log K)

K Smallest Subset Sums: Idea 3

Problem 2:
Given an array of non-negative integers A[1], A[2], ..., A[N].
Find the K smallest subset sums of this array.

It obviously has a **constant-bounded outdegree**, and try to convince yourself that **it forms a tree** and **covers the whole search space**.

These are the main key points for a good state

- Constant number of info to be stored
- Constant-bounded outdegree
- Non-negative edge weight
- Forms a tree and cover whole search space

IOI 2023 Practice Task 1 – A Plus B

Problem 3 (IOI 2023 Practice T1):
Given two sorted arrays A[i] and B[j], both with length N, let C[i, j] = A[i] + B[j]. Find the N smallest elements in C.

Example:
Let A = [0, 2, 2], B = [3, 5, 6].

(flattened) C = [3, 5, 6, 5, 7, 8, 5, 7, 8] → sort → [**3, 5, 5**, 5, 6, 7, 7, 8, 8]

1st minimum: 3 = 0 + 3 = A[1] + B[1]

2nd minimum: 5 = 0 + 5 = A[1] + B[2]

3rd minimum: 5 = 2 + 3 = A[2] + B[1]

IOI 2023 Practice Task 1 – A Plus B

Problem 3 (IOI 2023 Practice T1):
Given two sorted arrays A[i] and B[j], both with length N, let C[i, j] = A[i] + B[j]. Find the N smallest elements in C.

- There is an easy binary search solution to this problem!
- But we want to try solving it using fracturing search.

IOI 2023 Practice Task 1 – A Plus B

Problem 3 (IOI 2023 Practice T1):
Given two sorted arrays A[i] and B[j], both with length N, let C[i, j] = A[i] + B[j]. Find the N smallest elements in C.

When trying to solve with fracturing search, we must first ask the following questions:

- What are the states?
- What are the parent-child relationships?
- Does our modelling above cover all possible candidates?

IOI 2023 Practice Task 1 – A Plus B

Problem 3 (IOI 2023 Practice T1):
Given two sorted arrays A[i] and B[j], both with length N, let C[i, j] = A[i] + B[j]. Find the N smallest elements in C.

Your turn!
Give an appropriate state modelling to solve this problem.

IOI 2023 Practice Task 1 – A Plus B

Problem 3 (IOI 2023 Practice T1):
Given two sorted arrays A[i] and B[j], both with length N, let C[i, j] = A[i] + B[j]. Find the N smallest elements in C.

First guess: (i, j) is our state. Would that work?

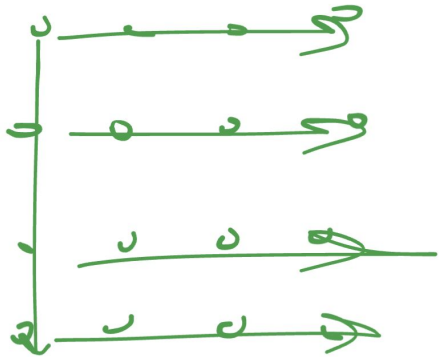
- (2, 2) → (2, 3) → (3, 3)
- (2, 2) → (3, 2) → (3, 3)
- Easy to make the tree not a DAG...

IOI 2023 Practice Task 1 – A Plus B

Problem 3 (IOI 2023 Practice T1):
Given two sorted arrays A[i] and B[j], both with length N, let C[i, j] = A[i] + B[j]. Find the N smallest elements in C.

First guess: (i, j) is our state. Would that work?

- (2, 2) → (2, 3) → (3, 3)
- (2, 2) → (3, 2) → (3, 3)
- Easy to make the tree not a DAG... **unless we restrict state transitions as follows:** Only allow (i, 0) → (i + 1, 0) and (i, j) → (i, j + 1)



IOI 2023 Practice Task 1 – A Plus B

Problem 3 (IOI 2023 Practice T1):
Given two sorted arrays A[i] and B[j], both with length N, let C[i, j] = A[i] + B[j]. Find the N smallest elements in C.

Overall algorithm:

- Maintain priority queue of ((A[i] + A[j]), (i, j)) states
- Children of (i, j): (i, j + 1)
- If j = 0 then also (i + 1, 0)
- Obviously, this covers all possible candidates
- Time complexity: O(N log N)

K-th Minimum Spanning Tree

Problem 4:
Given a graph with $N \leq 50$ vertices and at most $N(N-1)/2$ edges, find the K-th ($K \leq 10^4$) minimum spanning tree.

- What would be a sensible state to store?

K-th Minimum Spanning Tree

Problem 4:
Given a graph with $N \leq 50$ vertices and at most $N(N-1)/2$ edges, find the K-th ($K \leq 10^4$) minimum spanning tree.

- What would be a sensible state to store?
- Maybe (weight of spanning tree, spanning tree itself), sorted in ascending order of weight of spanning tree.
- What is the root state? → The minimum spanning tree itself.

K-th Minimum Spanning Tree

Problem 4:
Given a graph with $N \leq 50$ vertices and at most $N(N-1)/2$ edges, find the K-th ($K \leq 10^4$) minimum spanning tree.

- What would be a sensible state to store?
- Maybe (weight of spanning tree, spanning tree itself), sorted in ascending order of weight of spanning tree.
- What is the root state? → The minimum spanning tree itself.
- Now, we will need to figure out the parent/child relationships of states.

K-th Minimum Spanning Tree

Problem 4:
Given a graph with $N \leq 50$ vertices and at most $N(N-1)/2$ edges, find the K-th ($K \leq 10^4$) minimum spanning tree.

Let's think of what the first layer of the search structure should be.

- Remove one edge from the MST, find the smallest possible replacement for it
- One idea is to let the i-th child subspace of the root to consist of all spanning trees not including edge i of the current spanning tree for each i in [1, N – 1]. Does that work?

K-th Minimum Spanning Tree

Problem 4:
Given a graph with $N \le 50$ vertices and at most $N(N-1)/2$ edges, find the K-th ($K \le 10^4$) minimum spanning tree.

Let's think of what the first layer of the search structure should be.

- Remove one edge from the MST, find the smallest possible replacement for it
- One idea is to let the i-th child subspace of the root to consist of all spanning trees not including edge i of the current spanning tree for each i in [1, N – 1]. Does that work? **No; the subspaces overlap, e.g. a spanning tree that excludes two MST edges is counted twice.**

K-th Minimum Spanning Tree

Problem 4:
Given a graph with $N \le 50$ vertices and at most $N(N-1)/2$ edges, find the K-th ($K \le 10^4$) minimum spanning tree.

- We can instead let the i-th child subspace contain all spanning trees that
  (1) includes the first $i-1$ edges of the MST
  (2) does not include the i-th edge of the MST

  for $1 \le i \le N - 1$
- Now the subspaces no longer overlap!

K-th Minimum Spanning Tree

Problem 4:
Given a graph with $N \leq 50$ vertices and at most $N(N-1)/2$ edges, find the K-th ($K \leq 10^4$) minimum spanning tree.

- For depth 2+: this case requires more thought. Denote a string of length M representing the state of edges in the graph: '1' means used in MST, '0' means not used in MST, '?' means maybe used in MST
- Root: MST itself
- Depth = 1: by definition we change a bit from the root solution, i.e. change a '1' to '0'. We will find the **first** 1 that is changed to a 0.
- Run Kruskal on the new graph (with sure 1 edges and unsure ? edges)
- Depth = 2: do the same thing

Robotic Cow Herd (USACO 2016)

Problem 5 (USACO 2016):
Given N arrays ($1 \le N \le 10^5$), where the i-th array has size M[i] ($1 \le M[i] \le 10$) and has elements P[i][1], P[i][2], ..., P[i][M[i]].
Pick one element per array and sum them up to obtain the total cost.
Among all the ways to pick elements in each array, find the sum of total cost of the top K ($1 \le K \le 10^5$) *least* costly ways.

Robotic Cow Herd (USACO 2016)

Problem 5 (USACO 2016):
Given N arrays ($1 \le N \le 10^5$), where the i-th array has size M[i] ($1 \le M[i] \le 10$) and has elements P[i][1], P[i][2], ..., P[i][M[i]].
Pick one element per array and sum them up to obtain the total cost.
Among all the ways to pick elements in each array, find the sum of total cost of the top K ($1 \le K \le 10^5$) *least* costly ways.

- For each array P[i], sort its elements in ascending order.
- The state here *seems* obvious – the 1st least costly way is (1, 1, ..., 1)
- Then transitions could be in the form of "add 1 to one of the entries"
- But we can't make sure the parent-child relationship structure is a **tree**.

Robotic Cow Herd (USACO 2016)

Consider this state definition

- Each state will be a tuple: (cost, i, j)
- cost: The total sum for the combination this state represents.
- i: The index of the last array that is using an element other than its first one.
- j: The index of the element being used from array i (where j >= 2).

Every combination (except the one using all first elements) is uniquely defined by the last array that deviates from the baseline. This prevents ambiguity and forms our search tree.

Robotic Cow Herd (USACO 2016)

The first and smallest sum is found by taking the first element from every array.

BaseCost = sum(P[i][1] for i in 1..N).

This is our 1st answer.

The simplest "next best" solutions are those where exactly one array deviates from the baseline.

e.g. one deviation is to use P[1][2], the state will be

(BaseCost - P[1][1] + P[1][2], 1, 2).

Robotic Cow Herd (USACO 2016)

For each state (cost, i, j), the state represents the combination with P[i][j] as the deviation in the last deviating array i. We can generate its 3 unique children by the following:

Transition 1 (Deeper Deviation): create a new combination by choosing the next element from the same array i.

- If j+1 is a valid index in P[i]: (cost - P[i][j] + P[i][j+1], i, j+1) is a valid next state

Transition 2 (Next Array Deviation): create a new combination by adding a deviation to next array, i+1.

- If i+1 is a valid array index: (cost - P[i+1][1] + P[i+1][2], i+1, 2) is a valid next state

Transition 3 (The Undo Transition):

- If there are only Transition 2, we could never advance to newer array but keep choosing index 1, as that have no deviation, i.e. it is impossible to pick 1 in array x and pick 2+ in array x+1
- This transition provide a way to revert that deviation and move forward.
- If j == 2 and i+1 is a valid array index:
- (cost - (P[i][2]-P[i][1]) + (P[i+1][2]-P[i+1][1]), i+1, 2) is a valid next state.

Robotic Cow Herd (USACO 2016)

Transition 3 (The Undo Transition):

- If there are only Transition 2, we could never advance to newer array but keep choosing index 1, as that have no deviation.
- This transition provide a way to revert that deviation and move forward.
- If j == 2 and i+1 is a valid array index:
- (cost - (P[i][2]-P[i][1]) + (P[i+1][2]-P[i+1][1]), i+1, 2) is a valid next state.

To make this transition valid, - (P[i][2]-P[i][1]) + (P[i+1][2]-P[i+1][1]) should be non-negative.

Hence, we should sort all array by (P[i][2] - P[i][1]) beforehand.

Team Problem Set

Team Problem Set

Rules

- You are given 5 problems.
- Try your best to solve as many of the given problems as possible!
- Work with others sitting on your table.
- Teams will be invited to share their solutions afterwards.
- Time limit: 30 minutes

Solutions

K Smallest Sums I

This is basically equal to problem 5 from before. Basically just make sure your solution maintains the fact that each distinct state is visited once.

K Smallest Sums II

This is a hybrid of problem 2 and problem 5.
Using a similar method to solving problem 2, find the K smallest sums for **each** array independently (using different L_i and R_i). Treat the answer to this sub-problem for array i as B_i.
Then use the method from problem 5 on the arrays B_1, B_2, ..., B_K.

Solutions

Queueing

This is basically equal to problem 5 from before. Basically just make sure your solution maintains the fact that each distinct state is visited once.

Solutions

Olympiad

This is basically equal to problem 4 from before. We use the partitioning

(1) Force to use the first i-1 contestants

(2) Force to **not** use the i-th contestant

for $1 \le i \le N - 1$

Table of Contents

K-th Shortest Walk

Problem:
Given a weighted, directed graph with N nodes and M edges and a pair of nodes (s, t), find the lengths of the K-th shortest walk from s to t.

- A walk from s to t is defined as a sequence of nodes $(u_0, u_1, u_2, ..., u_k)$ where $u_0 = s$, $u_k = t$ and there exists directed edges from u_i to u_{i+1}.
- The length of a walk is defined to be the sum of the edges.

K-th Shortest Walk

Problem:
Given a weighted, directed graph with N nodes and M edges and a pair of nodes (s, t), find the lengths of the K-th shortest walk from s to t.

- Eppstein's algorithm solves this in O(M + N log N + K).
- Let's see how this algorithm works!

K-th Shortest Walk: Algorithm 1

- We can modify Dijkstra's algorithm to find the K shortest walks.
- Instead of using a visited array, maintain a `count[]` array for each node, and don't visit when a node is visited K times.

Question
Why is it sufficient to visit each node K times only?

K-th Shortest Walk: Algorithm 1

- We can modify Dijkstra's algorithm to find the K shortest walks.
- Instead of using a visited array, maintain a `count[]` array for each node, and don't visit when a node is visited K times.

Answer

Note that the walk is distinct every time you visit a node. Then since we only need K shortest walks, we only need at most K from each node.

- Since each node is visited K times, the time complexity is O(KM log KM).

K-th Shortest Walk: Algorithm 1

```
q = empty min heap
count = 0-filled array
push (0, s) on q
while count[t] < k:
    (l, u) = pop q
    if count[u] == k:
        continue
    count[u] += 1
    if u == t:
        found a path of length l
    for each outgoing edge (u, v, w) from u:
        push (l + w, v) on q
```

K-th Shortest Walk: Algorithm 2

- Can we do better?
- We think of it from another perspective: instead of finding the actual shortest walk, we find the “minimum deviation” from the shortest path.

K-th Shortest Walk: Algorithm 2

- Can we do better?
- We think of it from another perspective: instead of finding the actual shortest walk, we find the “minimum deviation” from the shortest path.

- First, find any shortest path from all nodes to the ending node t in the graph. This is equivalent to reversing all edges and finding the shortest path from t to all nodes. This also has a nice property that it forms a tree.
- Let d_u be the length of the shortest path from u to t, this can be viewed as potential of node u.

K-th Shortest Walk: Algorithm 2

- Let d_u be the length of the shortest path from u to t, this can be viewed as potential of node u.
- Any walk from s to t can be viewed as following the shortest path tree, but occasionally taking a 'sidetrack' via an edge (u, v) that is not on the shortest path.
- The extra cost of the sidetrack, or 'sidetracked-ness', of such an edge is (w + d(v)) - d(u), where w is weight edge of edge u to v. All these detour costs are non-negative.

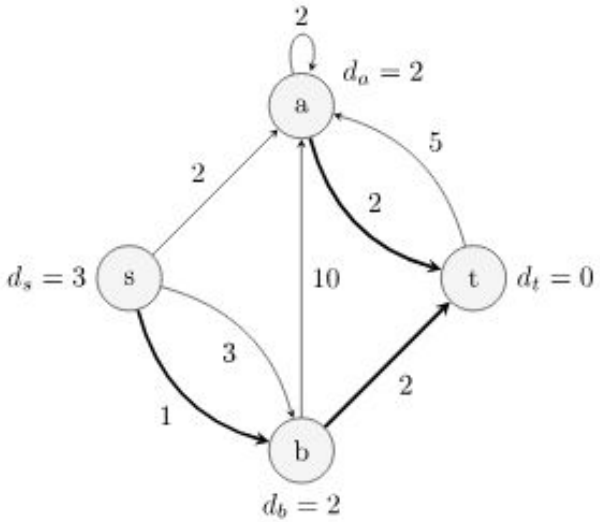
K-th Shortest Walk: Algorithm 2

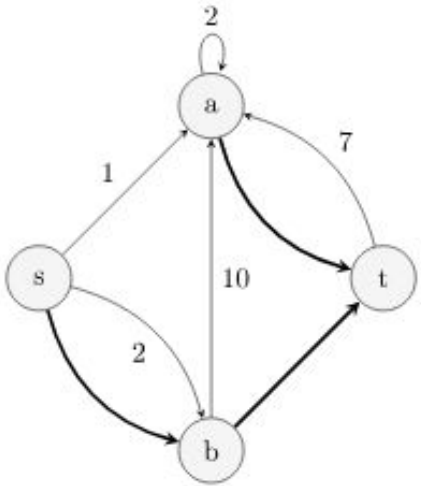
Consider this example.

The bold edges are the spanning tree generated by running Dijkstra on t.

For example, say we went through the path $s \to a$.

The sidetracked-ness is $w + d_a - d_s = 2 + 2 - 3 = 1$





K-th Shortest Walk: Algorithm 2

- ... We say the "sidetracked-ness" of walking along an edge (u, v) to be st(u, v) = (w + d_v) - d_u. What does this do?

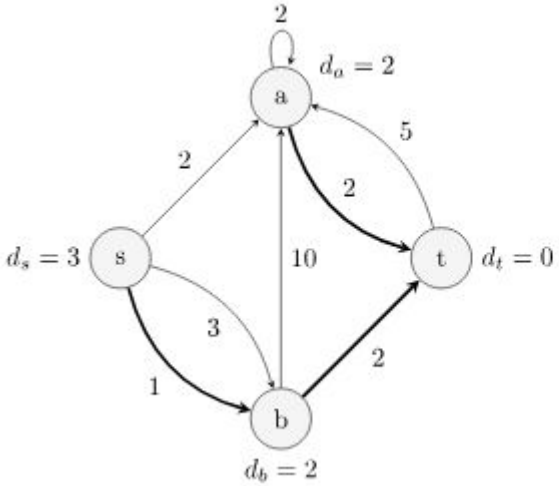
- Generate a graph G' as follows:
  - First, ignore all the edges in the spanning tree.
  - Then, for all sidetracked edges (u, v, st(u, v)), add a directed edge from all descendants of u to v with cost st(u, v).
- Then, **there exists a bijection between s-t walks in G and s walks in G' (with arbitrary ending node).**
  - In other words, every s-t walk in G corresponds to **exactly one** s walk in G'.

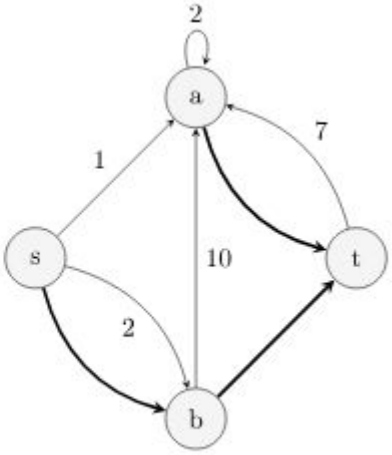
K-th Shortest Walk: Algorithm 2

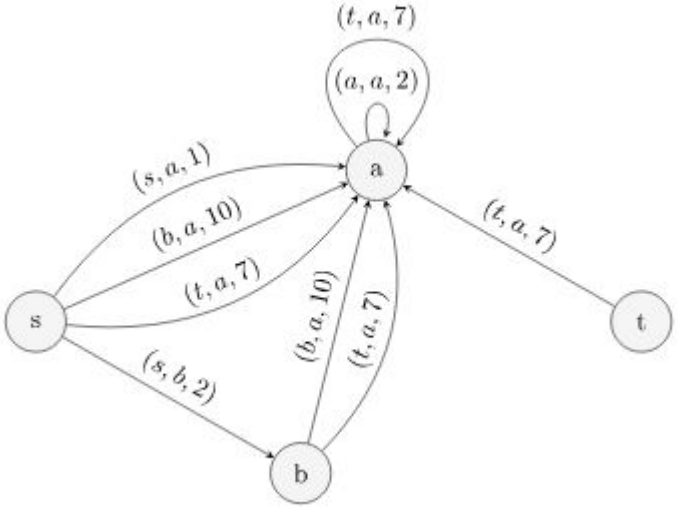
- Generate a graph G' as follows:
  - First, ignore all the edges in the spanning tree.
  - Then, for all sidetracked edges (u, v, st(u, v)), add a directed edge from all descendants of u to v with cost st(u, v).

- Bijection: suppose I want to visit sidetracked edges $\{(u_1, v_1), (u_2, v_2), \ldots\}$
  - The path on G would be $s \to \ldots$ (up the tree) $\to u_1 \to v_1 \to \ldots$ (up the tree) $\to u_2 \to v_2 \to \ldots$
  - Here, "up the tree" refers to going up the spanning tree generated after running Dijkstra.
  - We add all descendants of u to v in G' to save the "walking up the tree" process.
  - Therefore, in G', the path would look like $s \to v_1 \to v_2 \to \ldots$

K-th Shortest Walk: Algorithm 2







K-th Shortest Walk: Algorithm 2

Question

Why is this bijection true?

Challenge

Why is this “transformation” helpful? Provide some intuition on why this makes the task easier.

K-th Shortest Walk: Algorithm 2

Answer

Effectively, G' describes how long you follow the optimal path before taking a "sidetracked" edge.

Challenge

Now we know that the K shortest walks on G' from s bijects to the K shortest walks on G from s to t. We can run a Dijkstra-like algorithm on G'. Every time we pop something, it is sure to be a walk from s to t on G.

K-th Shortest Walk: Algorithm 2

We can perform a Dijkstra-like algorithm on G', popping from the heap K times:

```
q = empty min heap
push (d[s], s) on q
repeat k times:
    (l, u) = pop q
    found a path of length l
    for each outgoing edge (u, v, w) from u:
        push (l + w, v) on q
```

K-th Shortest Walk: Algorithm 2

A brief recap:

1. Run a reverse Dijkstra from t to get the distance between every node and u, as well as one of the edges to achieve so. This generates a spanning tree (“distance tree”, i.e. BFS tree but Dijkstra). We root this tree at t.
   - O(M + N log N) (or O(M log M)) because of Dijkstra

K-th Shortest Walk: Algorithm 2

A brief recap:

2. Construct G'. For all edges (u, v, w) that are not in the tree, add an edge from all descendants of u to v with weight st(u, v), where st(u, v) = (w + d_v) - d_u.
   - Actually, we only have to construct the adjacency list of G'.
   - Instead of storing an edge for every descendant of u, we only store the edge at u. Then, when querying, take all the stored edges from the root t to u.
   - O(M) time for construction only

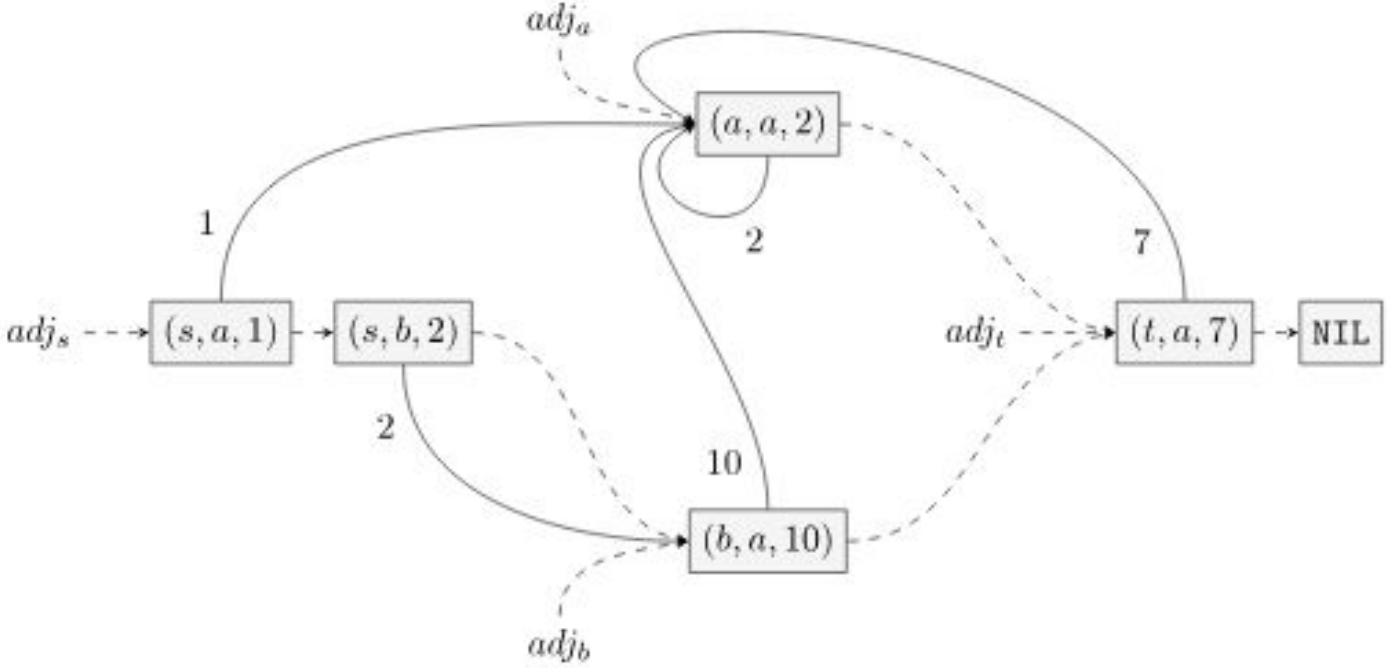
K-th Shortest Walk: Algorithm 2

A brief recap:

3. Run the Dijkstra-like algorithm. Each time, pop the walk with the shortest "sidetracked cost", then add all adjacent nodes to the priority queue.
   - Actually, a node can have O(M) edges (consider a line graph)...
   - O(MK log MK)

Total time complexity: O(M + N log N + M + MK log MK) =
O(N log N + MK log MK), dominated by step 3
Hmm... I wonder if we can do better here...

K-th Shortest Walk: Algorithm 2



K-th Shortest Walk: Algorithm 3

Use fracturing search!!! But on what?

Question
Where can we use fracturing search?

K-th Shortest Walk: Algorithm 3

Answer

On the adjacency lists!

In this context, it works similar to lazy evaluation.

K-th Shortest Walk: Algorithm 3

So, what do we want?

For each adj_u, we want:

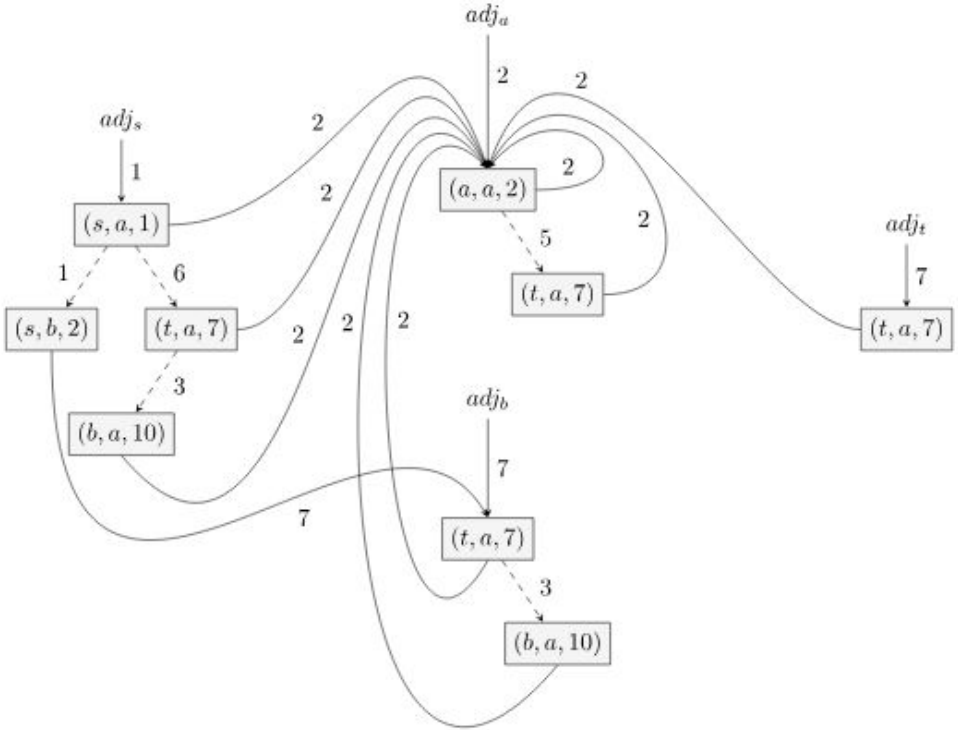
- A min-heap (on weight, for Dijkstra)
- Bounded degree (for fracturing search)
- Supports inserting items persistently / merging heaps efficiently (for merging ancestors)

K-th Shortest Walk: Algorithm 3

- There are multiple data structures that solves this in O(K) (or O(K log K)), hence optimising the solution to O(M + N log N + K).
- The data structures and details are rather complicated and is beyond the scope of discussion of this lecture. You can refer to this Codeforces blog for more information.

- Intuitively, think of it as doing Dijkstra “in parallel”, searching for “useful stuff only”. This is the crux of fracturing search.

K-th Shortest Walk: Algorithm 3



Q&A

Any questions?

HKOI Training Camp 2025

Further Brute Force

Group Problem Solving Session

Problem: K Smallest Sums I

Given K non-decreasing arrays $A^{(1)}, \ldots, A^{(K)}$, each of length K, form an **element-wise sum** by picking one entry from array i for $1 \leq i \leq K$.
Output the K smallest of them (duplicates count multiple times) in non-decreasing order.

Problem: K Smallest Sums II

Given K non-decreasing arrays $A^{(1)}, \ldots, A^{(K)}$, each of length K, form an **element-wise sum** by picking between L_i to R_i entries from array i for $1 \leq i \leq K$.
Output the K smallest of them (duplicates count multiple times) in non-decreasing order.

Problem: Queueing

There are N people in a physical queue (person 1 at the front, N at the back), each with a value A_i. Alice can choose any of the $N!$ possible orders (permutations) of the N people. For a chosen order, person i's anger $=$

$$P_i A_i + \sum_{j>i, P_j<P_i} A_j$$

where P_i means the order of the i-th person among the N people. The total anger of an order is the sum of all individuals' anger values. Among the $N!$ totals, sort them ascending and output the K-th smallest one.

Problem: Olympiad

Each of two neighbouring cities takes part in an annual competition comprising K distinct events. Every year each city chooses exactly K contestants out of N contestants they have, and *every* contestant competes in *all* K events.
For any fixed event, a team's score is the highest individual score achieved by one of its members in that event. The total score of a team is the sum of these K event scores:

$$\text{Total}(T) \;=\; \sum_{j=1}^{K} \big(\max_{i \in T} s_{i,j}\big),$$

where $s_{i,j}$ denotes the score earned by contestant i in event j.
As an illustration, suppose $K = 3$ and the three selected contestants record scores $(4, 5, 3)$, $(7, 3, 6)$ and $(3, 4, 5)$ in the three events. The event scores for the team are then $(\max\{4, 7, 3\},\ \max\{5, 3, 4\},\ \max\{3, 6, 5\}) = (7, 5, 6)$, so the total team score equals $7 + 5 + 6 = 18$.
Because a city can form many different K-member teams from its pool of eligible contestants, we rank all those teams in descending order of total score. Rank 1 is the best team, rank 2 the second best, and so forth. For any integer $C \geq 1$, the *C-th best team* is the team occupying rank C. Two teams are regarded as different if they differ in at least one contestant.
Considering every possible K-member team as equally likely, determine the total score of the city's C-th best team.
Constraints: $N \leq 500, K \leq 6, C \leq 2000$.