Universal Optimality of Dijkstra Algorithm

Using Fibonacci-Like Priority Queue with Working Sets

Soham Chatterjee

July 24, 2025

Oral Qualifier, STCS

Introduction

- Dijkstra algorithm is a foundation algorithm solving Single Source Shortest Path problem (SSSP) both for directed and undirected graphs.
- Using Fibonacci Heaps we have the worst-case time complexity $O(m + n \log n)$.

Introduction

- Dijkstra algorithm is a foundation algorithm solving Single Source Shortest Path problem (SSSP) both for directed and undirected graphs.

- Using Fibonacci Heaps we have the worst-case time complexity $O(m + n \log n)$.

- Recently Duan, Mao, Shu and Yin in 2023 solved SSSP for undirected graphs with expected time $O(m\sqrt{\log n \log \log n})$

Introduction

- Dijkstra algorithm is a foundation algorithm solving Single Source Shortest Path problem (SSSP) both for directed and undirected graphs.
- Using Fibonacci Heaps we have the worst-case time complexity $O(m + n \log n)$.
- Recently Duan, Mao, Shu and Yin in 2023 solved SSSP for undirected graphs with expected time $O(m\sqrt{\log n \log\log n})$
- This year in STOC Duan, Mao, Mao, Shu, Yin solved SSSP for directed graphs in $O(m \log^{\frac{2}{3}} n)$ time.

Assumptions

- Input graph is always connected.
- All trees are rooted at s.
- For any vertex v, $T(v)$ denote the subtree of T rooted at v.
- The weights of the graph are positive real numbers.
- We allow the ∞ in the weights.

Dijkstra Algorithm

Algorithm: $\textsc{Dijkstra}(G, s, w)$

$F \longleftarrow \emptyset$, $\textsc{Insert}(F, s)$, $dist(s) \longleftarrow 0$

while $F \neq \emptyset$ **do**

  $u \longleftarrow \textsc{ExtractMin}(F)$

  for $e = (u, v) \in E$ **do**

    If v is unseen, $\textsc{Insert}(F, v)$

    $\textsc{DecreaseKey}(F, v, \min\{dist(v), dist(u) + w(u, v)\})$

Dijkstra Algorithm

Algorithm: $\textsc{Dijkstra}(G, s, w)$

$F \longleftarrow \emptyset$, $\textsc{Insert}(F, s)$, $dist(s) \longleftarrow 0$
while $F \neq \emptyset$ **do**
    $u \longleftarrow \textsc{ExtractMin}(F)$
    for $e = (u, v) \in E$ **do**
        If v is unseen, $\textsc{Insert}(F, v)$
        $\textsc{DecreaseKey}(F, v, \min\{dist(v), dist(u) + w(u, v)\})$

Dijkstra solves three problems:

- Computes Shortest Distances

Dijkstra Algorithm

Algorithm: $\textsc{Dijkstra}(G, s, w)$

$F \longleftarrow \emptyset$, $\textsc{Insert}(F, s)$, $dist(s) \longleftarrow 0$
while $F \neq \emptyset$ **do**
  $u \longleftarrow \textsc{ExtractMin}(F)$
  for $e = (u, v) \in E$ **do**
    If v is unseen, $\textsc{Insert}(F, v)$
    $\textsc{DecreaseKey}(F, v, \min\{dist(v), dist(u) + w(u, v)\})$

Dijkstra solves three problems:

- Computes Shortest Distances
- Build Shortest Path Tree

Dijkstra Algorithm

Algorithm: $\textsc{Dijkstra}(G, s, w)$

$F \longleftarrow \emptyset$, $\textsc{Insert}(F, s)$, $dist(s) \longleftarrow 0$
while $F \neq \emptyset$ **do**
  $u \longleftarrow \textsc{ExtractMin}(F)$
  for $e = (u, v) \in E$ **do**
    If v is unseen, $\textsc{Insert}(F, v)$
    $\textsc{DecreaseKey}(F, v, \min\{dist(v), dist(u) + w(u, v)\})$

Dijkstra solves three problems:

- Computes Shortest Distances
- Build Shortest Path Tree
- Sorts vertices by Shortest Distance (DO)

Comparison-Addition Model

Notice the Dijkstra algorithm does the following operations:

- Adds two values
- Compares two values.
- Stores Values.

So we will work on a model where all possible operations are addition, compare and storage.

Comparison-Addition Model

Notice the Dijkstra algorithm does the following operations:

- Adds two values
- Compares two values.
- Stores Values.

So we will work on a model where all possible operations are addition, compare and storage.

For a given graph:

- $OPT_Q(G)$ is the number of comparison queries of an optimal algorithm for this graph.
- $OPT(G)$ be the number of total steps taken by an optimal correct algorithm for the graph.

Universal Optimality

- Let $\mathcal{A}$ is the set of all correct algorithms.
- $\mathcal{G}_{n,m}$ is the set of all graphs with n vertices and m edges.
- $\mathcal{W}_G$ is the set of all possible weights for a graph $G \in \mathcal{G}_{n,m}$.

Universal Optimality

- Let $\mathcal{A}$ is the set of all correct algorithms.
- $\mathcal{G}_{n,m}$ is the set of all graphs with n vertices and m edges.
- $\mathcal{W}_G$ is the set of all possible weights for a graph $G \in \mathcal{G}_{n,m}$.

A correct algorithm A^* is *existentially optimal* if

$$\forall\, n, m : \sup_{\substack{G \in \mathcal{G}_{n,m} \\ w \in \mathcal{W}_G}} A^*(G, w) \leq \alpha \inf_{A \in \mathcal{A}} \sup_{\substack{G \in \mathcal{G}_{n,m} \\ w \in \mathcal{W}_G}} A(G, w)$$

where $\alpha = \tilde{O}(1)$. This corresponds to being optimal wrt worst-case complexity.

Universal Optimality

- Let $\mathcal{A}$ is the set of all correct algorithms.
- $\mathcal{G}_{n,m}$ is the set of all graphs with n vertices and m edges.
- $\mathcal{W}_G$ is the set of all possible weights for a graph $G \in \mathcal{G}_{n,m}$.

A correct algorithm A^* is *existentially optimal* if

$$\forall\, n, m : \sup_{\substack{G \in \mathcal{G}_{n,m} \\ w \in \mathcal{W}_G}} A^*(G, w) \leq \alpha \inf_{A \in \mathcal{A}} \sup_{\substack{G \in \mathcal{G}_{n,m} \\ w \in \mathcal{W}_G}} A(G, w)$$

where $\alpha = \tilde{O}(1)$. This corresponds to being optimal wrt worst-case complexity.

But this is not good. It is just saying A^* may take as much time as it takes in a star-graph or more complicated one.

Universal Optimality

We want a notion of optimality which says your algorithm is optimal compared to any other algorithm if you **fix the graph**.

Universal Optimality

We want a notion of optimality which says your algorithm is optimal compared to any other algorithm if you **fix the graph**.

A correct algorithm A^* is *universally optimal* if

$$\forall\ n, m,\ \forall\ G \in \mathcal{G}_{n,m} : \sup_{w \in \mathcal{W}_G} A^*(G, w) \leq \alpha \inf_{A \in \mathcal{A}} \sup_{w \in \mathcal{W}_G} A(G, w)$$

where $\alpha = \tilde{O}(1)$.

Universal Optimality

We want a notion of optimality which says your algorithm is optimal compared to any other algorithm if you **fix the graph**.

A correct algorithm A^* is *universally optimal* if

$$\forall\, n, m,\ \forall\, G \in \mathcal{G}_{n,m} : \sup_{w \in \mathcal{W}_G} A^*(G, w) \leq \alpha \inf_{A \in \mathcal{A}} \sup_{w \in \mathcal{W}_G} A(G, w)$$

where $\alpha = \tilde{O}(1)$.

In this discussion we will focus solely on $\alpha = O(1)$.

Exploration Tree and DO

Consider a run of Dijkstra. Whenever a vertex is extracted add the unexplored neighbors of that vertex as children of that vertex. The tree built this way is called the exploration tree.

Exploration Tree and DO

Consider a run of Dijkstra. Whenever a vertex is extracted add the unexplored neighbors of that vertex as children of that vertex. The tree built this way is called the exploration tree.

- Let T be the exploration tree. Let $\prec$ be the final distance ordering of the vertices.

Exploration Tree and DO

Consider a run of Dijkstra. Whenever a vertex is extracted add the unexplored neighbors of that vertex as children of that vertex. The tree built this way is called the exploration tree.

- Let T be the exploration tree. Let $\prec$ be the final distance ordering of the vertices.

- Then for every edge $(u, v) \in T$, $u \prec v$.

Order of Vertices by a Tree

Definition (Order of T)

Let T be any tree in G. An order of T is a total order of $V(T)$ such that for every edge $(u, v) \in E(T)$ we have $u \prec v$ in the order.

Order of Vertices by a Tree

Definition (Order of T)

Let T be any tree in G. An order of T is a total order of $V(T)$ such that for every edge $(u, v) \in E(T)$ we have $u \prec v$ in the order.

The DO after Dijkstra is an order of exploration tree.

- L is an order of G if there exists a spanning tree T of G such that L is an order of T.

Order of Vertices by a Tree

Definition (Order of T)

Let T be any tree in G. An order of T is a total order of $V(T)$ such that for every edge $(u, v) \in E(T)$ we have $u \prec v$ in the order.

The DO after Dijkstra is an order of exploration tree.

- L is an order of G if there exists a spanning tree T of G such that L is an order of T.
- $\mathsf{Order}(G)$ is the number of all possible orders of G.

Order of Vertices by a Tree

Definition (Order of T)

Let T be any tree in G. An order of T is a total order of $V(T)$ such that for every edge $(u, v) \in E(T)$ we have $u \prec v$ in the order.

The DO after Dijkstra is an order of exploration tree.

- L is an order of G if there exists a spanning tree T of G such that L is an order of T.
- $\mathsf{Order}(G)$ is the number of all possible orders of G.

Lemma

For any graph G, L is an order of G iff there exists non-negative weights w such that

1. *For every two nodes $u \neq v$, $d_w(s, u) \neq d_w(s, v)$.*
2. *$u \prec_L v$ if and only if $d_w(s, u) < d_w(s, v)$.*

Dijkstra Induced Interval Set

For any vertex $v \in V(G)$

- l_v: When v was first discovered and added to the heap.
- r_v: When v was removed from heap.
- $[l_v, r_v]$: Interval set of v

Dijkstra Induced Interval Set

For any vertex $v \in V(G)$

- l_v: When v was first discovered and added to the heap.
- r_v: When v was removed from heap.
- $[l_v, r_v]$: Interval set of v

A run of Dijkstra induces intervals for each vertex $v \in V$ with the operations $\textsc{Insert}$ and $\textsc{ExtractMin}$.

Dijkstra Induced Interval Set

For any vertex $v \in V(G)$

- l_v: When v was first discovered and added to the heap.
- r_v: When v was removed from heap.
- $[l_v, r_v]$: Interval set of v

A run of Dijkstra induces intervals for each vertex $v \in V$ with the operations Insert and ExtractMin.

An interval set $\mathcal{I}$ is collection of intervals for each vertex. It is called Dijkstra Induced when all the intervals for each vertex in $\mathcal{I}$ is induced by a run of Dijkstra on some (G, w).

Working set of an Interval Set

Let $\mathcal{I}$ any interval set.

Working set of an Interval Set

Let $\mathcal{I}$ any interval set.

- For any vertex $v \in V(G)$ at any time $t \in I(v)$ the working set $W_{v,t}$ is the set of vertices inserted after v and still present at time t. So

$$W_{v,t} = \{[l_u, r_u] \in \mathcal{I} : l_v \leq l_u \leq t \leq r_u\}$$

Working set of an Interval Set

Let $\mathcal{I}$ any interval set.

- For any vertex $v \in V(G)$ at any time $t \in I(v)$ the working set $W_{v,t}$ is the set of vertices inserted after v and still present at time t. So

$$W_{v,t} = \{[l_u, r_u] \in \mathcal{I} : l_v \leq l_u \leq t \leq r_u\}$$

- Working set of v, $W_v = W_{v,t^*}$ such that $t^* = \arg\max_t |W_{v,t}|$.

Working set of an Interval Set

Let $\mathcal{I}$ any interval set.

- For any vertex $v \in V(G)$ at any time $t \in I(v)$ the working set $W_{v,t}$ is the set of vertices inserted after v and still present at time t. So

$$W_{v,t} = \{[l_u, r_u] \in \mathcal{I} : l_v \leq l_u \leq t \leq r_u\}$$

- Working set of v, $W_v = W_{v,t^*}$ such that $t^* = \arg\max_t |W_{v,t}|$.

- The cost of a vertex $v \in V(G)$ is $Cost(v) = \log|W_v|$. And so $Cost(\mathcal{I}) = \sum_{v \in V(G)} \log|W_v|$.

Fibonacci-Like Priority Queue with Working Set Property

FPQWSP is a type of Fibonacci Heap which satisfies the amortized time complexity for any sequence of operations as follows:

Fibonacci-Like Priority Queue with Working Set Property

FPQWSP is a type of Fibonacci Heap which satisfies the amortized time complexity for any sequence of operations as follows:

	FPQWSP	Fibonacci Heap
Insert	$O(1)$	$O(1)$
DecreaseKey	$O(1)$	$O(1)$
ExtractMin	$O(1 + \log \lvert W_x \rvert)$	$O(\log n)$

Fibonacci-Like Priority Queue with Working Set Property

FPQWSP is a type of Fibonacci Heap which satisfies the amortized time complexity for any sequence of operations as follows:

	FPQWSP	Fibonacci Heap
Insert	$O(1)$	$O(1)$
DecreaseKey	$O(1)$	$O(1)$
ExtractMin	$O(1 + \log \lvert W_x \rvert)$	$O(\log n)$

Fact

There is a FPQWSP for Dijkstra. We will use this data structure in every argument from now on by default.

Time Complexity of Dijkstra

In Dijkstra Algorithm it runs n times ExtractMin calls for each vertex and m times DecreaseKey calls.

Time Complexity of Dijkstra

In Dijkstra Algorithm it runs n times ExtractMin calls for each vertex and m times DecreaseKey calls.

- Hence total time taken by all DecreaseKey calls is $O(m)$.

Time Complexity of Dijkstra

In Dijkstra Algorithm it runs n times ExtractMin calls for each vertex and m times DecreaseKey calls.

- Hence total time taken by all DecreaseKey calls is $O(m)$.
- Total time taken by all ExtractMin calls is

$$\sum_{v \in V(G)} O(1 + \log |W_v|) = O\left(n + \sum_{v \in V(G)} \log |W_v|\right) = O(n + Cost(\mathcal{I}))$$

- Total time taken by Dijkstra is $O(m + n + Cost(\mathcal{I}))$

Main Theorem

Theorem

Dijkstra implemented by FPQWSP in Comparison-Addition model has time complexity $O(OPT_Q(G) + m + n)$.

Goal: We'll show $OPT_Q(G) = \Omega(Cost(\mathcal{I}))$.

- $OPT_Q(G) \leq OPT(G)$

Main Theorem

Theorem

Dijkstra implemented by FPQWSP in Comparison-Addition model has time complexity $O(OPT_Q(G) + m + n)$.

Goal: We'll show $OPT_Q(G) = \Omega(Cost(\mathcal{I}))$.

- $OPT_Q(G) \leq OPT(G)$
- $OPT(G) = \Omega(n)$

Main Theorem

Theorem

Dijkstra implemented by FPQWSP in Comparison-Addition model has time complexity $O(OPT_Q(G) + m + n)$.

Goal: We'll show $OPT_Q(G) = \Omega(Cost(\mathcal{I}))$.

- $OPT_Q(G) \leq OPT(G)$
- $OPT(G) = \Omega(n)$
- $OPT(G) = \Omega(m)$

So $OPT_Q(G) + n + m = O(OPT(G))$.

Proof Flow

Fact

$OPT_Q(G) = \Omega(\log(\mathrm{Order}(G)))$

Proof Flow

Fact

$OPT_Q(G) = \Omega(\log(\mathsf{Order}(G)))$

- Partition the exploration tree into non-comparable sets $(B_1, \ldots, B_k)$ with $i < j$ then no node of B_j is ancestor of any node of B_i.

Proof Flow

Fact

$OPT_Q(G) = \Omega(\log(\mathsf{Order}(G)))$

- Partition the exploration tree into non-comparable sets $(B_1, \ldots, B_k)$ with $i < j$ then no node of B_j is ancestor of any node of B_i.
- For any such partition $\log(\mathsf{Order}(G)) = \Omega\left(\sum_{i=1}^{k} |B_i| \log |B_i|\right)$

Proof Flow

Fact

$OPT_Q(G) = \Omega(\log(\mathrm{Order}(G)))$

- Partition the exploration tree into non-comparable sets $(B_1, \ldots, B_k)$ with $i < j$ then no node of B_j is ancestor of any node of B_i.
- For any such partition $\log(\mathrm{Order}(G)) = \Omega\left(\sum_{i=1}^{k} |B_i| \log |B_i|\right)$
- There is a partition such that $2\sum_{i=1}^{k} |B_i| \log |B_i| \geq Cost(\mathcal{I})$

Barrier Sequence

Definition (Barrier)

Let T be any tree. A *Barrier*, $B \subseteq V(T)$ is a set of nodes where for any two vertices $u, v \in B$, u is not ancestor of v in T.

Barrier Sequence

Definition (Barrier)

Let T be any tree. A *Barrier*, $B \subseteq V(T)$ is a set of nodes where for any two vertices $u, v \in B$, u is not ancestor of v in T.

- For two disjoint barriers, $B_1 \prec B_2$ if no node of B_2 is predecessor of a node in B_1.

Barrier Sequence

Definition (Barrier)

Let T be any tree. A *Barrier*, $B \subseteq V(T)$ is a set of nodes where for any two vertices $u, v \in B$, u is not ancestor of v in T.

- For two disjoint barriers, $B_1 \prec B_2$ if no node of B_2 is predecessor of a node in B_1.
- $(B_1, \ldots, B_k)$ is a *barrier sequence* if $i < j \implies B_i \prec B_j$.

Barrier Sequence

Definition (Barrier)

Let T be any tree. A *Barrier*, $B \subseteq V(T)$ is a set of nodes where for any two vertices $u, v \in B$, u is not ancestor of v in T.

- For two disjoint barriers, $B_1 \prec B_2$ if no node of B_2 is predecessor of a node in B_1.
- $(B_1, \ldots, B_k)$ is a *barrier sequence* if $i < j \implies B_i \prec B_j$.

Lemma

A sequence $(B_1, \ldots, B_k)$ of pairwise disjoint vertex sets is barrier sequence if and only if for all $1 \leq i \leq j \leq k$, $v \in B_j$ is not ancestor of any $u \in B_i$ in T.

Barriers Give Lower Bounds

Lemma

Let T be any spanning tree and $(B_1, \ldots, B_k)$ be a barrier sequence of T.

Then $\log(\mathrm{Order}(G)) = \Omega\left(\sum_{i=1}^{k} |B_i| \log |B_i|\right)$

Barriers Give Lower Bounds

Lemma

Let T be any spanning tree and $(B_1, \ldots, B_k)$ be a barrier sequence of T. Then $\log(\mathrm{Order}(G)) = \Omega\left(\sum_{i=1}^{k} |B_i| \log |B_i|\right)$

- We have $\mathrm{Order}(G)) \geq \mathrm{Order}(T)$. We'll show $\mathrm{Order}(T) \geq |B_1|!|B_2|! \cdots |B_k|!$.

Barriers Give Lower Bounds

Lemma

Let T be any spanning tree and $(B_1, \dots, B_k)$ be a barrier sequence of T. Then $\log(\mathrm{Order}(G)) = \Omega\left(\sum_{i=1}^{k} |B_i| \log |B_i|\right)$

- We have $\mathrm{Order}(G)) \geq \mathrm{Order}(T)$. We'll show $\mathrm{Order}(T) \geq |B_1|!|B_2|! \cdots |B_k|!$.
- Delete vertices of B_k to get T'. By induction for the barrier sequence $(B_1, \dots, B_{k-1})$ for T', $\mathrm{Order}(T') \geq |B_1|!|B_2|! \cdots |B_{k-1}|!$.

Barriers Give Lower Bounds

- We can order vertices of B_k in any order we want. There are $|B_k|!$ many orders.

Barriers Give Lower Bounds

- We can order vertices of B_k in any order we want. There are $|B_k|!$ many orders.
- For each order of B_k and any order of $\mathsf{Order}(T')$ we can just concatenate them to get an order of T.

Barriers Give Lower Bounds

- We can order vertices of B_k in any order we want. There are $|B_k|!$ many orders.
- For each order of B_k and any order of $\mathsf{Order}(T')$ we can just concatenate them to get an order of T.

So finally we got the result:

Result

If T is a spanning tree of G and $(B_1, \ldots, B_k)$ is a barrier sequence for T then

$$OPT_Q(G) = \Omega\left(\sum_{i=1}^{k} |B_i| \log |B_i|\right)$$

Barriers in the Heap

Consider running Dijkstra algorithm until some time. Let S is the set of nodes that are in the priority queue.

Barriers in the Heap

Consider running Dijkstra algorithm until some time. Let S is the set of nodes that are in the priority queue.

- Notice that S is the leaves of the partial exploration tree built so far which is a subgraph of final exploration tree.

Barriers in the Heap

Consider running Dijkstra algorithm until some time. Let S is the set of nodes that are in the priority queue.

- Notice that S is the leaves of the partial exploration tree built so far which is a subgraph of final exploration tree.
- Therefore, S is an incomparable set of the final exploration tree.
- S forms a barrier.

Barriers in the Heap

Consider running Dijkstra algorithm until some time. Let S is the set of nodes that are in the priority queue.

- Notice that S is the leaves of the partial exploration tree built so far which is a subgraph of final exploration tree.
- Therefore, S is an incomparable set of the final exploration tree.
- S forms a barrier.

Result

At any time of the algorithm the set of elements in the priority queue forms a barrier

Intersecting Coloring

A barrier sequence is basically coloring vertices in a certain way where vertices in a barrier have same color.

Intersecting Coloring

A barrier sequence is basically coloring vertices in a certain way where vertices in a barrier have same color.

Definition (Intersecting Coloring)

An intersecting coloring of $\mathcal{I}$ with k colors is a function $C : \mathcal{I} \to [k]$ that assigns a color to every interval and additionally for every color $i \in [k]$,

$$\bigcap_{I \in \mathcal{I}, C(I) = i} I \neq \emptyset.$$

Intersecting Coloring

A barrier sequence is basically coloring vertices in a certain way where vertices in a barrier have same color.

Definition (Intersecting Coloring)

An intersecting coloring of $\mathcal{I}$ with k colors is a function $C : \mathcal{I} \to [k]$ that assigns a color to every interval and additionally for every color $i \in [k]$, $\bigcap_{I \in \mathcal{I}, C(I)=i} I \neq \emptyset$.

Every intersecting coloring induces a barrier sequence in the exploration tree in following way: For any color c,

Intersecting Coloring

A barrier sequence is basically coloring vertices in a certain way where vertices in a barrier have same color.

Definition (Intersecting Coloring)

An intersecting coloring of $\mathcal{I}$ with k colors is a function $C : \mathcal{I} \to [k]$ that assigns a color to every interval and additionally for every color $i \in [k]$, $\bigcap_{I \in \mathcal{I}, C(I) = i} I \neq \emptyset$.

Every intersecting coloring induces a barrier sequence in the exploration tree in following way: For any color c,

- $B_c = \{v \in V(G) \mid C(I(v)) = c\}$

Intersecting Coloring

A barrier sequence is basically coloring vertices in a certain way where vertices in a barrier have same color.

Definition (Intersecting Coloring)

An intersecting coloring of $\mathcal{I}$ with k colors is a function $C : \mathcal{I} \to [k]$ that assigns a color to every interval and additionally for every color $i \in [k]$, $\bigcap_{I \in \mathcal{I}, C(I)=i} I \neq \emptyset$.

Every intersecting coloring induces a barrier sequence in the exploration tree in following way: For any color c,

- $B_c = \{v \in V(G) \mid C(I(v)) = c\}$
- $t_c = \min\{t \mid \forall\, v \in B_c, t \in I(v)\}$

Intersecting Coloring

A barrier sequence is basically coloring vertices in a certain way where vertices in a barrier have same color.

Definition (Intersecting Coloring)

An intersecting coloring of $\mathcal{I}$ with k colors is a function $C : \mathcal{I} \to [k]$ that assigns a color to every interval and additionally for every color $i \in [k]$, $\bigcap_{I \in \mathcal{I}, C(I)=i} I \neq \emptyset$.

Every intersecting coloring induces a barrier sequence in the exploration tree in following way: For any color c,

- $B_c = \{v \in V(G) \mid C(I(v)) = c\}$
- $t_c = \min\{t \mid \forall\, v \in B_c, t \in I(v)\}$
- Order $\{B_c\}$ by increasing order of $\{t_c\}$. WLOG $t_1 < \cdots < t_k$.

Intersecting Coloring

A barrier sequence is basically coloring vertices in a certain way where vertices in a barrier have same color.

Definition (Intersecting Coloring)

An intersecting coloring of $\mathcal{I}$ with k colors is a function $C : \mathcal{I} \to [k]$ that assigns a color to every interval and additionally for every color $i \in [k]$, $\bigcap_{I \in \mathcal{I}, C(I)=i} I \neq \emptyset$.

Every intersecting coloring induces a barrier sequence in the exploration tree in following way: For any color c,

- $B_c = \{v \in V(G) \mid C(I(v)) = c\}$
- $t_c = \min\{t \mid \forall\ v \in B_c, t \in I(v)\}$
- Order $\{B_c\}$ by increasing order of $\{t_c\}$. WLOG $t_1 < \cdots < t_k$.
- $(B_1, \ldots, B_k)$ is a barrier sequence for exploration tree.

Intersecting Coloring Gives Lower Bounds

Let C be an intersecting coloring of $\mathcal{I}$ with k colors. Let $(B_1, \ldots, B_k)$ is the barrier sequence induced by C. Then let the energy of C is defined to be

$$E(C) = 2\sum_{i=1}^{k} |B_i| \log |B_i|$$

Intersecting Coloring Gives Lower Bounds

Let C be an intersecting coloring of $\mathcal{I}$ with k colors. Let $(B_1, \ldots, B_k)$ is the barrier sequence induced by C. Then let the energy of C is defined to be

$$E(C) = 2\sum_{i=1}^{k} |B_i| \log |B_i|$$

Result

If $\mathcal{I}$ is the interval set induced by Dijkstra and C be any arbitrary intersecting coloring of $\mathcal{I}$ then

$$OPT_Q(G) = \Omega(E(C))$$

Good Intersecting Coloring gives Optimality

Goal: Find an intersecting coloring of $\mathcal{I}$, C such that $E(C) \geq Cost(\mathcal{I})$

Good Intersecting Coloring gives Optimality

Goal: Find an intersecting coloring of $\mathcal{I}$, C such that $E(C) \geq Cost(\mathcal{I})$

- Then time complexity of all ExtractMin operations is $O(n + Cost(\mathcal{I})) = O(n + E(C))$.

Good Intersecting Coloring gives Optimality

Goal: Find an intersecting coloring of $\mathcal{I}$, C such that $E(C) \geq Cost(\mathcal{I})$

- Then time complexity of all ExtractMin operations is $O(n + Cost(\mathcal{I})) = O(n + E(C))$.
- We have $OPT_Q(G) = \Omega(E(C))$.

Good Intersecting Coloring gives Optimality

Goal: Find an intersecting coloring of $\mathcal{I}$, C such that $E(C) \geq Cost(\mathcal{I})$

- Then time complexity of all ExtractMin operations is $O(n + Cost(\mathcal{I})) = O(n + E(C))$.
- We have $OPT_Q(G) = \Omega(E(C))$.
- So overall Cost of ExtractMin in Dijkstra is upper bounded by $O(n + OPT_Q(G))$.

Good Intersecting Coloring gives Optimality

Goal: Find an intersecting coloring of $\mathcal{I}$, C such that $E(C) \geq Cost(\mathcal{I})$

- Then time complexity of all ExtractMin operations is $O(n + Cost(\mathcal{I})) = O(n + E(C))$.
- We have $OPT_Q(G) = \Omega(E(C))$.
- So overall Cost of ExtractMin in Dijkstra is upper bounded by $O(n + OPT_Q(G))$.
- Dijkstra achieves universal optimality for time complexity.

Good Intersecting Coloring gives Optimality

Goal: Find an intersecting coloring of $\mathcal{I}$, C such that $E(C) \geq Cost(\mathcal{I})$

- Then time complexity of all ExtractMin operations is $O(n + Cost(\mathcal{I})) = O(n + E(C))$.
- We have $OPT_Q(G) = \Omega(E(C))$.
- So overall Cost of ExtractMin in Dijkstra is upper bounded by $O(n + OPT_Q(G))$.
- Dijkstra achieves universal optimality for time complexity.

We will find such a good intersecting coloring recursively.

Finding Good Intersecting Coloring

- We will construct C by induction on $|\mathcal{I}|$.

Finding Good Intersecting Coloring

- We will construct C by induction on $|\mathcal{I}|$.
- Find the interval $x \in \mathcal{I}$ with the largest W_x. Use induction on $\mathcal{I}' = \mathcal{I} \setminus W_x$

Finding Good Intersecting Coloring

- We will construct C by induction on $|\mathcal{I}|$.
- Find the interval $x \in \mathcal{I}$ with the largest W_x. Use induction on $\mathcal{I}' = \mathcal{I} \setminus W_x$
- Let C' is the coloring for $\mathcal{I}'$ such that $E(C') \geq Cost(\mathcal{I}')$. Add a new color for all the elements in W_x to get new coloring C.

Finding Good Intersecting Coloring

- We will construct C by induction on $|\mathcal{I}|$.
- Find the interval $x \in \mathcal{I}$ with the largest W_x. Use induction on $\mathcal{I}' = \mathcal{I} \setminus W_x$
- Let C' is the coloring for $\mathcal{I}'$ such that $E(C') \geq Cost(\mathcal{I}')$. Add a new color for all the elements in W_x to get new coloring C.
- $E(C) = E(C') + 2|W_x| \log |W_x|$ by definition.

Finding Good Intersecting Coloring

- We will construct C by induction on $|\mathcal{I}|$.
- Find the interval $x \in \mathcal{I}$ with the largest W_x. Use induction on $\mathcal{I}' = \mathcal{I} \setminus W_x$
- Let C' is the coloring for $\mathcal{I}'$ such that $E(C') \geq Cost(\mathcal{I}')$. Add a new color for all the elements in W_x to get new coloring C.
- $E(C) = E(C') + 2|W_x| \log |W_x|$ by definition.

Fact

For working set W_x with the largest size

$$Cost(\mathcal{I}) \leq Cost(\mathcal{I} \setminus W_x) + 2|W_x| \log |W_x|$$

Finding Good Intersecting Coloring

- We will construct C by induction on $|\mathcal{I}|$.
- Find the interval $x \in \mathcal{I}$ with the largest W_x. Use induction on $\mathcal{I}' = \mathcal{I} \setminus W_x$
- Let C' is the coloring for $\mathcal{I}'$ such that $E(C') \geq Cost(\mathcal{I}')$. Add a new color for all the elements in W_x to get new coloring C.
- $E(C) = E(C') + 2|W_x| \log |W_x|$ by definition.

Fact

For working set W_x with the largest size

$$Cost(\mathcal{I}) \leq Cost(\mathcal{I} \setminus W_x) + 2|W_x| \log |W_x|$$

- $Cost(\mathcal{I}) \leq Cost(\mathcal{I}') + 2|W_x| \log |W_x|$. Hence, $E(C) \geq Cost(\mathcal{I})$.

Thank You

$OPT_Q(G) = \Omega(\log(\mathrm{Order}(G)))$

Lemma

For any directed or undirected graph G, any algorithm for the DO problem needs $\Omega(\log(\mathrm{Order}(G)))$ comparison queries in expectation.

- Let A is any correct algorithm and $L \in \mathrm{Order}(G)$.
- Given L we have a weight assignment w_L such that L is unique order obtained from w_L upon running Dijkstra. For each L fix w_L. Let $\mathcal{W}$ be the collection of all such w_L.
- Let $C_L \in \{-1, 0, 1\}^*$ be the sequence of answers of comparisons made by A on (G, w_L). Then $C : \mathcal{W} \to \{-1, 0, 1\}^*$, $C(w_L) = C_L$ is a ternary prefix free code.
- By Shannon's source coding lemma for symbol codes any such code has expected length $\Omega(\log(|\mathcal{W}|)) = \Omega(\log(\mathrm{Order}(G)))$

Deleting Intervals from $\mathcal{I}$

Lemma

Let $\mathcal{I}$ *an interval set and* $x \in \mathcal{I}$. $k = \max_t |\{I \in \mathcal{I} \mid t \in I\}|$. *Then*

$$Cost(\mathcal{I}) \leq Cost(\mathcal{I} \setminus \{x\}) + \log |W_x| + \log k$$

- Let $I_1, \ldots, I_l \in \mathcal{I}$ are the only intervals which had nonempty intersection with x. So $l \leq k - 1$.
- Let t_i is starting point of I_i. WLOG assume $t_l > \cdots > t_1$.
- Let W_i, W_i' are working sets of I_i before and after removing x.

Deleting Intervals from $\mathcal{I}$

- Let t is starting point of x. Then $W_{i,t}$ contains $x, I_1, \ldots, I_i$. So $|W_i| \geq i + 1$.
- $|W_i| \in \{|W_i'|, |W_i'| + 1\}$ for all $i \in [l]$.

$$
\begin{aligned}
&Cost(\mathcal{I}) - Cost(\mathcal{I} \setminus \{x\}) - \log |W_x| \\
&= \sum_{i=1}^{l} \log |W_i| - \log |W_i'| \\
&\leq \sum_{i=1}^{l} \log(i+1) - \log i = \log(l+1) \leq \log k
\end{aligned}
$$

Fact

For any working set $|W_x| = k$ we have

$$Cost(\mathcal{I}) \leq Cost(\mathcal{I} \setminus W_x) + 2|W_x| \log |W_x|$$