Discrete Optimization

Instructor: Yuan Zhou

Notes Taker: Zejin Lin

TSINGHUA UNIVERSITY.

linzj23@mails.tsinghua.edu.cn

lzjmaths.github.io

March 18, 2025

Contents

1 Introduction

Discrete (combinatorial) optimization is a subfield of mathematical optimization that consists of finding an optimal object from a finite set of objects, where the set of feasible solution is discrete or can be reduced to a discrete set.

However, usually this feasible solution set is very large (due to combinatorial explosion) and it is computationally infeasible to go through all feasible solutions and find the one with opimal objective function value.

Example 1.1 (Task Assignment)**.** There are n tasks and n workers. Each task has an importance score a_i and each worker has a skill level b_i. We need to assign each task to a worker such that the sum of $\sum\limits_{i=1}^{n} a_i b_{\sigma(i)}$ is maximized.

1.1 Models of Computation: Turing Machines

Definition 1.2 (A Deterministic Turing Machine(DTM))**.** It consists of an infinitely-long tape (memory) and a deterministic finite automata that controls the head to move along the tape and read/write symbols from/to the tape cells.

Definition 1.3 (Complexity measure)**.** Running time is the number of steps of Turing machine.

Memory is the number of tape cells used.

Definition 1.4 (Caveat)**.** No random access of memory

- Single-tape DTM requires $\geqslant n^2$ steps to detect n bit palindromes.
- EAsy to detect palindromes within c_n steps on a real computer.

1.2 Models of Computation: word RAM

Definition 1.5. Each memory location and input/output cell stores a w-bit integer (assume $w \geqslant \log_2 \omega$).

Primitive Operations:

1.3 Polynomial Running Time

Definition 1.6. We say that an algorithm is **efficient** if its running time is polynomial of input size n.

Example 1.7 (Task machine)**.** Polynomial-time algorithm: selection sort/inserting sort/quick sort/merge sort.

Non-polynomial-time algorithm: try all possible matching and output the one with the highest score.

- Definition is relatively insensitive to model of computation.
- The poly-times algorithm that people develop have both small constants and small exponents
- Breaking through the exponential barrier is a major challenge.

1.4 Notation

Definition 1.8. $f(n)$ is $O(g(n))$ if there exist constants $c > 0$ and $n_0 \geqslant 1$ such that $0 \leqslant f(n) \leqslant c \cdot g(n)$ for all $n \geqslant n_0$.

$f(n)$ is $\Omega(g(n))$ is $g(n) \in O(f(n))$.

$f(n)$ is $\Theta(g(n))$ is both $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$.

1.5 Tentative Syllabus

We will introduce three exact discrete optimization algorithms(6 weeks):

- Greedy algorithms
- Dynamic programming
- Network flows

And some approximation algorithms for intractable discrete optimization problems(9 weeks)

- Definition of approximation algorithms
  - Algorithm techniques: greedy, linear programming relaxation, semidefinite programming relaxation.
- Hardness of approximation
  - Techniques: hardness reductions, Fourier analysis of Boolean functions.
- Problems studied: Set-Cover, facility location, K-center, Multi-Cut, Max-Cut, $\cdots$

2 Greedy Algorithms

2.1 Interval Scheduling

Example 2.1 (Interval Scheduling)**.** Input: n jobs, $\{(s_i, f_i)\}_{i=1}^n$. Goal: How to choose jobs with maximized number such that each pair of intervals do not intersect.

Greedy Framework Consider jobs in order $\pi(1), \pi(2), \cdots, \pi(n)$. For each $\pi(i)$, $i = 1, 2, \cdots, n$, if $\pi(i)$ compatible with all selected jobs, then select $\pi(i)$.

The choice of π: Earliest-start-time-first, Earliest-finish-time-first, Longest-job-first, Shortest-job-first, etc.

Theorem 2.2. *Earliest-finish-time-first greedy returns an optimal solution.*

Proof. Suppose algorithm selects $i_1, i_2, \cdots, i_k$, opt selects $k' > k$ jobs.

Choose an optimal solution agrees with algorithm in first r jobs so that r maxmized, $j_1, j_2, \cdots, j_{k'}$.

Obviously, $r < k$. Then $f_{i_{r+1}} < f_{j_{r+1}}$. Therefore, we can replace i_{r+1} with j_{r+1} to get another optimal solution, which contradicts to the fact that r maxmized. □

2.2 Interval Partitioning

Example 2.3 (Interval Partitioning)**.** Input: n lectures, $\{(s_i, f_i)\}_{i=1}^n$.

Goal: Position lectures into minimum number of classrooms so that in each classroom lectures are compatible.

Greedy Framework Lectures in order $\pi(1), \cdots, \pi(n)$, the number of opening classrooms is zero in the beginning. For each $\pi(i)$,

If $\exists$ opening classroom j *s.t.* lecture $\pi(i)$ compatible with lectures in j, then $\pi(i) \to$ classroom j.

Else, open a new classroom for $\pi(i)$.

Proof. Introduce a concept **Depth**: $d(t) =$ Number of lectures active at time t, and $d = \max_t \{d(t)\}$.

Claim. OPT $\geqslant d$.

Lemma 2.4. *Alg* $\leqslant d$

Proof. Assume for contradiction.

At some point, Alg opens $d+1$ classroom.

Denote the lecture being considered by i. Then it is not compatible with other d lectures. Hence, there should be a time when $d+1$ lectures are active, which causes contradiction. □

□

2.3 Single-Source Shortest Path

Example 2.5 (Single-Source Shortest Path(SSSP))**.** Input: Graph $G = (V, E, w)$, V is the set of point and E is the set of edge with direction and $\omega : E \to \mathbb{R}_{\geqslant 0}$.

We want to find a path from s to t with minimum total cost.

Dijkstra's Algorithm Choose s as a source. $d[s] = 0, d[u] = \begin{cases} \omega(s,u) & \text{if } (s,u) \in E \\ +\infty & \text{otherwise} \end{cases}$, $S = \{s\}$ first. To record the path, we can use $\mathrm{Pred}[u] \leftarrow s$.

Algorithm 1 Dijkstra's Algorithm

```
1:  while $S \neq V$ do
2:      Choose $u \in \arg\min_{\notin S}\{d[x]\}$.
3:      Update $S \leftarrow S \cup \{u\}$.
4:      for each $x \in V - S$,$(u,x) \in E$ do
5:          $d[x] \leftarrow \min\{d[x], d[u] + \omega(u,x)\}$.
6:          if $d[u] + \omega(u,x) < d[x]$ then
7:              $d[x] \leftarrow d[u] + \omega(u,x)$
8:              $\mathrm{Pred}[x] \leftarrow u$
9:          end if
10:     end for
11: end while
```

Theorem 2.6 (Invariant)**.** *$\forall u \in S$, $d[u]$ is the shortest path distance $s \rightsquigarrow u$*

Proof. Induction on $|S|$.

For $|S| = 1$ true.

Induction Step: Every time executing 2 in Algorithm 1, we need to prove $d[u]$ is the shortest distance $s \rightsquigarrow u$.

If $v = \text{Pred}[u] \in S$, then $d[u] = d[v] + \omega(v, u)$.

For any path from s to u, there exists $(\alpha, \beta) \in E$ such that $\alpha \in S, \beta \notin S$. Then

$$\begin{aligned} \text{length}(P) &\geqslant \text{length}(P[s \to \beta]) \\ &= \text{length}(P[s \to \alpha]) + \omega(\alpha, \beta) \\ &\geqslant d[\alpha] + \omega(\alpha, \beta) \\ &\geqslant d[\beta] \geqslant d[u] \end{aligned}$$

□

Remark 2.7. The straightforward implementation of Dijkstra's Algorithm is of $O(|v|^2)$.

If we use priority queue: Q with priority $Q.\pi()$. It has some methods:

- ExtractMin: Return $\arg\min_{x \in Q}\{Q.\pi(x)\}$ and remove x from Q.
- DecreaseKey: Update $Q.\pi(v)$ with newkey.

The time complexity is $|V| \times$ ExtractMin + $|E| \times$ DecreaseKey

Runtime	ExtractMin	DecreaseKey	Dijkstra
Simple Array	$O(\|V\|)$	$O(1)$	$O(\|v\|^2)$
Binary Heap	$O(\log \|V\|)$	$O(\log \|V\|)$	$O(\|E\| \cdot \log \|V\|)$
Fibonacci Heap	$O(\log \|V\|)$	$O(1)$ (amorized)	$O(\|E\| + \|V\| \log \|V\|)$

2.4 Minimum Spanning Tree

Example 2.8 (Minimum Spanning Tree (MST))**.** Input: Connected, undirected graph $G = (V, E, \omega)$.

Definition 2.9 (Spanning Tree)**.** $T \subset E$ is a **spanning tree** if $|T| = |V| - 1$, $G' = (V, T)$ is connected.

Goal of MST Find spanning tree T so that $\omega(T) = \sum\limits_{e \in T} \omega(e)$ minimized.

Theorem 2.10 (Cayley Theorem)**.** *The number of spanning trees of n-vertex complete graph is n^{n-2}*

A **cut** $(S, V - S)$ has a **cutset** of $S = \{e = (u, v) : u \in S, v \notin S\}$.

Claim. Any cycle C and cutset D has intersection $|C \cap D|$ even.

Fundamental Cycle: Given G and spanning tree $T \subset E$, for each $e \in E \backslash T$, the unique cycle in $T \cup \{e\}$ is called **Fundamental cycle**.

Claim. For a fundamental cycle C related with e, $\forall f \in C \cap T$, $(T \cup \{e\}) \backslash \{f\}$ is also a spanning tree.

If T is MST, then $\omega(e) \geqslant \omega(f)$.

Fundamental Cut: Spanning tree $T \subset E$. For each $f \in T$, $T \backslash \{f\}$ has two connected components, whose cutset is called **fundamental cut**.

Claim. $\forall e \in D \backslash T$, $(T \cup \{e\}) \backslash \{f\}$ is a spanning tree.

If T is MST, then $\omega(e) \geqslant \omega(f)$.

MST Algorithm There are some rules. **Red rule**:Let C a cycle without red edges. Select an uncolored edge in C with max weight and color it red.

Blue rule: Let D be a cutset without blue edges. Select an un colored edge in D with min weight and color it blue.

Greedy Algorithm:Apply red or blue rules in any order iteratively until all edges colored.

Theorem 2.11. *Greedy algorithm terminates and blue edges from MST.*

Proof. Observed that during the algorithm, blue edges always from a forest. □

Invariant $\exists$ MST T^* *s.t.* T^* contains all blue edges and no red edges.

Proof. Proof by induction. If there is a MST T^* contains all blue edges no red edges now. If we apply blue rule, with cutset D and $f \in D$ but $f \notin T^*$, then for fundamental cycle C of f, $\forall e \in C \cap T, \omega(e) \geqslant \omega(f)$. Since C has even edges in the cutset by the claim, $\exists e \in C \cap T$ *s.t.* $e \in D$, which contradicts the fact that f is the edge in cutset D with min weight.

The case that we apply red rule is similar. □

Algorithm 2 Prim's Algorithm

```
1: Initialize $S \leftarrow \{s\}$.
2: while $n - 1$ times do
3:    Choose $e$ be the min weight edge in the cutset $(S, V \backslash S)$
4:    add $e$ to $T$, another endpoint of $e$ to $S$.
5: end while
```

Remark 2.12. It is compatible with the simple idea: Each time chooses the min weight edge. However, it is more powerful since we only need to do this process in the cutset.

It is similar to Dijkstra's Algorithm. So its time complexity is $O(|E| + |V| \log |V|)$

Remark 2.13. The first step need time complexity $O(|E| \log |E|)$.

The second step need time complexity $O(|E| \cdot \alpha(|V|))$ using **Union-Find** data structure.

Algorithm 3 Kruskal's Algorithm

```
1: Consider edges in weight increasing order.
2: Add each edge to $T$ if not introducing a cycle.
```

WLOG we can assume edge weights are distinct.

Algorithm 4 Boruvka's Algorithm

```
1: while $< (n-1)$ blue edges do
2:     Simultaneously apply blue rule to each blue compunent.
3: end while
```

Claim. WHILE loop iterates $\leqslant O(\log|V|)$.

So time complexity is $O(|E|\log|V|)$.

Remark 2.14. There is a "contraction View". For each step, we can view each component as a single point with edges to other components.

If the graph is **Planar Graph**, then $|E| \leqslant O(V)$. At the i-th WHILE iteration, $|V_i| \leqslant \frac{|V|}{2^{i-1}}$, $|E_i| \leqslant O(|V_i|)$.

So the time complexity is $\sum_i O(|E_i|) \leqslant \sum_i O\left(\frac{|V|}{2^{i-1}}\right) \leqslant O(|V|)$ which is linear!

Using the contraction view, we can get another algorithm:

Prim+Boruvka

- Run Boruvka for k iterates.
- Run Prim on the contracted graph.

Remark 2.15.

For step 1, time complexity is $k \cdot |E|$.

For step 2, time complexity is $|E| + \frac{|V|}{2^k} \cdot \log\frac{|V|}{2^k}$.

So the total time complexity is $k|E| + \frac{|V|}{2^k} \cdot \log\frac{|V|}{2^k}$.

Choose $k = \log_2\log_2|V|$, it comes to $(\log\log|V|) \cdot |E| + \frac{|V|}{\log_2|V|} \cdot \log_2|V| \leqslant O(|E|\log\log|V| + |V|)$.

2.5 Minimum Arborescence

Example 2.16 (Minimum Arborescence)**.** Input: Directed $G = (V, E)$, source $s \in V$ and weight $\omega : E \to \mathbb{R}$.

We want to find an **arborescence** $T = (V, E)$ with root r of minimum total weight.

Definition 2.17. Given directed $G = (V, E)$ and $r \in V$, $n = |V|, m = |E|$, $F \subset E$ is an **arborescence** if

- F is a spanning tree if ignoring directions
- $\forall v \in V$, $\exists$ unique path $r \to v$ in F.

Or equivalently, F has no directed cycles and every node $v \neg r$ has a unique incoming edge.

For this problem, WLOG we can assuem that the root r has no in-degree and assume $\omega \geqslant 0$.

For each $n \neq r$, let

$$\text{cheap}(v) = \text{argmin}_{e=(u,v)\in E}\{\omega(e)\}$$

Claim. Let $F = \{\text{cheap}(v) | v \neq r\}$. F is arborescense $\Rightarrow F$ is min-cost.

Define $\omega_r(u, v) = \omega(u, v) - \omega(\text{cheap}(v))$. Suffices to find the min-cost arborescence under ω_r.

If F is not an arborescense, then $\exists$ a directed cycle C with all edges of weight 0.

Using the contraction view, if we contract "0-cycle" and keep this process recursively. By taking degrees carefully we can easily confirm the legallity of the contraction view. Then suffices to prove it is indeed the min-cost arborescence when we expand after.

Theorem 2.18. *The min-cost arborescence $\widetilde{F}$ when we apply contraction to $0-$cycle is exactly the min-cost arborescence in the original graph after expanding.*

Lemma 2.19. *$\exists$ min-cost F^* s.t. only 1 edge in F^* entering C.*

Proof. Our goal is to prove $\omega_r(F) \leqslant \omega_r(F^*)$.

Let $F^*_C = F^* \cap (C \times C)$. Then $|F^*_C| = |C| - 1$.

Apply C-contraction to $F^* \backslash F^*_C$ we obtain an arborescence of $\widetilde{G}$. (Easy to check)

So

$$\sum_{e \in F^* \backslash F^*_C} \omega_r(e) \geqslant \sum_{e \in \widetilde{F}} \omega_r(e)$$

So $\omega_r(F^*) \geqslant \omega_r(F)$ □

Proof of Lemma. Choose any $v \in C$.

Let $(x, y) \in r \to v$ be the first edge entering C.

Delete the edge entering $C \backslash \{y\}$ and add the edge of circle except the edge entering y.

Then it is an arborescence of less cost. □

3 Dynamic Programming

3.1 Weighted Interval Scheduling

Example 3.1 (Weighted Interval Scheduling)**.** Input: n jobs, $\{(s_i, f_i), \omega_i\}_{i=1}^n$. Want to find $\sum \omega_{i_k}$ maximum.

To make the structure simpler, we WLOG assume $s_1 \leqslant s_2 \leqslant \cdots \leqslant s_n$. We may find that there is a lot of repetitive computation. We can record each $\mathrm{Search}(i)$

Algorithm 5 $\mathrm{Search}(i)$

```
1: $j \leftarrow \min \mathrm{id} > i, s_j \geqslant f_i$.
2: Return $\max\{\mathrm{Search}(j) + \omega_i, \mathrm{Search}(i+1)\}$.
```

Algorithm 6 $\mathrm{Search} - \mathrm{Memorization}(i)$

```
1: If $i > n$, RETURN 0
2: If $i \neq$ bottom, RETURN $F[i]$.
3: $j(i) \leftarrow \min\{j | s_j \geqslant f_i\}$.
4: $F[i] \leftarrow \max\{\mathrm{Search} - \mathrm{M}(j(i)) + \omega_i, \mathrm{Search} - \mathrm{M}(i+1)\}$
5: RETURN $F[i]$
```

It can be written as

$$\begin{cases} F[i] = \max\{F[j(i)] + \omega_i, F[i+1]\} \\ F[n+1] = 0 \end{cases}$$

Such an equation is called **Bellman Equation**. So Dynamic Programming is a method to solve the problem by finding the optimal solution of each subproblem. We sometimes need to record the optimal solution of each subproblem to avoid repetition.

3.2 Segmented Least Square

Example 3.2 (Least Square)**.** We have n points $\{(x_i, y_i)\}_{i=1}^n$. We want to find a line $y = ax + b$ to minimize

$$\mathrm{SSE} = \sum_{i=1}^{n} [y_i - (ax_i + b)]^2 \tag{3.1}$$

Actually,

$$\begin{cases} a = \dfrac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2} \\ \\ b = \dfrac{\sum_i y_i - a \sum_i x_i}{n} \end{cases}$$

Example 3.3 (Segmented Least Square)**.** Input: $\{(x_i, y_i)\}_{i=1}^n$, $c > 0$.

Goal: Minimize $l = E + cL$ for piecewise line, where c is the **hyperparameter**, L is the number of the segments.

WLOG, assume $x_1 < x_2 < \cdots < x_n$.

We can define its subproblem as

$$\mathrm{OPT}[i] : \text{min loss}$$

when in put is $(x_1, y_1), \cdots, (x_i, y_i)$.

Find solution $\mathrm{OPT}[n]$. The boundary condition is $\mathrm{OPT}[1] = \mathrm{OPT}[2] = c$ and the **Bellman Equation** is

$$\mathrm{OPT}[i] = \min_{1 \leqslant j \geqslant i} \{\mathrm{OPT}[j-1] + l_{ji} + c\}$$

3.3 Knapsack Problem

Example 3.4 (Knapsack Problem)**.** Input: n items, w_i, v_i for its weight and value. The capacity of knapsack is w.

If assume integral weight, then denote $\mathrm{OPT}[i, w]$ as the optimal total value when in put is first knapsack capacity is w.

The **Bellman Equation** is

$$\mathrm{OPT}[i, w] = \begin{cases} \mathrm{OPT}[i-1, w] & w < w_i \\ \max\{\mathrm{OPT}[i-1, w], v_i + \mathrm{OPT}[i-1, w - w_i]\}, w \geqslant w_i \end{cases}$$

It has time complexity $O(nw)$, which is not a polynomial algorithm.

We can find another Value-Based DP: (Also assume integral values)

$$\mathrm{OPT}[i, v]\text{: choose min weight items.}$$

from item $1, 2, \cdots, i$ so that total value $\geqslant v$.

The final solution for maxmial v *s.t.* $\mathrm{OPT}[n, v] \leqslant w$.

$$OPT[i, v] = \min \begin{cases} \mathrm{OPT}[i-1, v] \\ w_i + \mathrm{OPT}[i-1, (v - v_i)^+] \end{cases}$$

$\mathrm{OPT}[0, v] = \begin{cases} 0 & v = 0 \\ +\infty & v > 0 \end{cases}$ The time complexity is $O(n^2 v)$.

Now we consider a α**-approximation algorithm** that $\mathrm{ALG} \geqslant \alpha \cdot \mathrm{OPT}$ for $\alpha \in (0, 1]$.

Let $\varepsilon = 1 - \alpha$.

Algorithm 7 Knapsack Problem

```
1: Assume WLOG $w_i \leqslant W$ so that $V \geqslant \mathrm{OPT}$.
2: Set $K = \frac{\varepsilon V}{n}$. Let $v_i' = \left\lfloor \frac{v_i}{K} \right\rfloor$
3: Run value-based DP to find optimal solution $T$ for $I'$
4: Return $T$ as a solution to $I$.
```

It is a feasible solution and

$$\begin{aligned} \sum_{i \in T} v_i' &= \mathrm{OPT}(I') \\ &\geqslant v(S; I'),\ \forall \text{feasible} S \\ &\geqslant v(T^*, I') \\ &= \sum_{i \in T^*} v_i' \\ &= \sum_{i \in T^*} \left\lfloor \frac{v_i}{K} \right\rfloor \\ &\geqslant \sum_{i \in T^*} \left(\frac{v_i}{K} - 1 \right) \\ &\geqslant \frac{1}{k} \sum_{i \in T^*} \sum_{i \in T^*} v_i - n \end{aligned}$$

$$= \frac{1}{K}\text{OPT}(I) - n$$

So $\text{ALG} \geqslant \sum_{i\in T} K \cdot v_i' \geqslant \text{OPT}(I) - nK \geqslant (1-\varepsilon)\text{OPT}(I)$.

The time complextity is $O(n^2 V') = O(n^2 \frac{V}{K}) = O(n^3\varepsilon^{-1})$.

Remark 3.5. The time complextity depends on the accuracy ε instead of the maximum value V since the accuracy is based on scale.

In other words, ε^{-1} in time complexity represents not only accuracy but also the "size" of scale.

Fully Polynomial-Time Approximation Scheme(FPTAS) $\forall\varepsilon$, $\exists$ $(1-\varepsilon)$-approximation algorithm with time complexity $f(n,\varepsilon) = \text{poly}(n, \frac{1}{\varepsilon})$.

PTAS : $\forall\varepsilon$, $\exists$ $(1-\varepsilon)$-approximation in time $f_\varepsilon(n) = \text{poly}(n)$. For this algorithm, it is $(n \cdot 2^{\frac{1}{\varepsilon}}, n^{\frac{1}{\varepsilon}})$.

3.4 RNA Secondary Structure

Example 3.6 (RNA Secondary Structure)**.** RNA is a string $b_1 b_2 \cdots b_n$ where $b_i \in \{A, C, G, U\}$.

The secondary structure is what fold to form "base pairs" including:

$$U \cdots A, A \cdots U, C \cdots G, G \cdots C$$

Mathematically, second structure represented by set of base pairs $S = \{(i,j)\}$,

*) $\forall (i,x) \in S, (b_i, b_j) \in \{U \cdots A, A \cdots U, C \cdots G, G \cdots C\}$

*) no sharp turns: $\forall (i,j) \in S, i < j - 4$,

*) non-crossing: $\forall (i,j), (k,l) \in S$, cannot have $i < k < j < l$.

Goal: Maximize $|S|$.

A direct idea is to construct those subproblems:

$$\text{OPT}[i,j] = \max_{i \leqslant k < j-4} \begin{cases} \text{OPT}[i,j-1] & b_j \text{ not matched} \\ 1 + \text{OPT}[i,k-1] + \text{OPT}[k+1,j-1] & b_j \text{ matched with } b_k \end{cases}$$

$$\text{OPT}[i,j] = 0 \text{ when } i \leqslant j < i+4$$

3.5 Sequence Alignment(Edit Distance)

Example 3.7. For a wrong-spelled word, what cost do we need to make it right, using the gap and mismatch.

Or what is its edit distance to the correct word.

Mathematically, for string $(a_1 \cdots a_n), (b_1 \cdots b_m)$, a matching $M = \{(i,j)\}$ such that there is no $(i_1, j_1), (i_2, j_2) \in M \;\; s.t.\; i_1 < i_2$ but $j_2 < j_1$. Define its cost

$$\text{cost}(M) = \sum_{(i,j)\in M} \alpha_{a_i b_j} + \sum_{i\in[n], i \text{ not in } M} + \sum_{\substack{j\in[m] \\ j \text{ not in } M}} \delta$$

$\sum\limits_{(i,j)\in M} \alpha_{a_i b_j}$ is the mismatch cost and $\sum_{i\in[n], i \text{ not in } M} + \sum_{j\in[m], j \text{ not in } M} \delta$ is the gap cost

Define $\text{OPT}[i,j]$ is the edit distance between $a_1 a_2 \cdots a_i$ and $b_1 b_2 \cdots b_j$.

$$\text{OPT}[i,j] = \min_{1 \leqslant k \leqslant j} = \begin{cases} \delta + \text{OPT}[i-1,j] & a_i \text{ not matched} \\ \alpha_{a_i b_k} + \delta \cdot (j-k) + \text{OPT}[i-1,k-1] & a_i \text{matched with } b_k \end{cases}$$

However, for each case it can be divided into three cases:

$$\mathrm{OPT}[i,j]=\min\begin{cases}\mathrm{OPT}[i-1,j-1]+\alpha_{a_ib_j}\\ \mathrm{OPT}[i-1,j]+\delta\\ \mathrm{OPT}[i,j-1]+\delta\end{cases}$$

The question is, if we need to trace the matching process, the space complexity is $O(nm)$, too large.

Here we use binary search.

Algorithm 8 Binary Search

```
1: Compute $A[j]=d[(0,0)\to(\frac{n}{2},j)]$ and $B[j]=d[(\frac{n}{2},j)\to(n,m)]$,
2: find $j^*=\mathrm{argmin}_j\, A[j]+B[j]$.
3: Run the sub-process $(0,0)\to(\frac{n}{2},j^*)$ and $(\frac{n}{2},j^*)\to(n,m)$
```

The complexity is still $O(nm)+\frac{1}{2}O(nm)+\cdots+\frac{1}{2^k}O(nm)=O(nm)$.

3.6 Matrix Multiplication

Example 3.8 (Matrix Multiplication). Consider $M_1\cdot M_2\cdots M_k$ where M_i is a $n_{i-1}\times n_i$ matrix.

We want to find the optimal multiplicative order such that the time cost is minimal.

Denote $\mathrm{OPT}[i,j]$ is the min from M_i to M_j.

Using the binary tree, consider the last multiplication

$$\mathrm{OPT}[i,j]=\min_{i\leqslant l<j}\{\mathrm{OPT}[i,l]+\mathrm{OPT}[l+1,j]+n_{i-1}n_ln_j\}$$

4 Flow Network

4.1 Definition

Example 4.1. For directed graph $G = (V, E, s, t, c)$ where s is the source and t is the sink. $c : E \to \mathbb{R}_{\geqslant 0}$ is the capacity function.

The **st-flow** is $f : E \to \mathbb{R}_{\geqslant 0}$ *s.t.*

1) $\forall e \in E$, $f(e) \leqslant c(e)$.

2) $\forall v \in V \backslash \{s, t\}$, $\sum\limits_{(u,v)\in E} f(u, v) = \sum\limits_{(v,u)\in E} f(v, u)$, *i.e.* flow conservation.

$$\text{val}(f) = \sum_{(s,u)\in E} f(s, u) - \sum_{(u,s)\in E} f(u, s)$$

Our goal is to maximize $\text{val}(f)$

An **st-cut** is a partition (A, B) of V such that $s \in A, t \in B$, the capacity

$$c(A, B) = \sum_{\substack{(u,v)\in E \\ u\in A, v\in B}} c(u, v)$$

Claim. $\forall$ feasible flow f and st-cut (A, B),

$$\text{val}(f) \leqslant c(A, B)$$

Residual Network Given flow network G, feasible flow f, the residual network $G_f(v, E_f, s, t, c_f)$ is for each $e \in E$

$$c_f(e) = c(e) - f(e) + f(e^{\text{reverse}})$$

where $u \to v$ is on the flow.

Claim (Weak Duality). f' is a feasible flow in G_f if and only if $f \oplus f'$ is feasible in G, where

$$(f \oplus f')(e) = f(e) + f'(e) - f'(e^{\text{reverse}})$$

An **augmenting path** P is an unsaturated $s \to t$ path in G_f.

Algorithm 9 Augment (f, P)

```
1: Let $\delta = \min_{e \in P} c_f(e)$.
2: for $e = (u, v) \in P$ do
3:   if $e \in E$ then
4:     $f(e) \leftarrow f(e) + \delta$
5:   else
6:     $f(v, u) \leftarrow f(v, u) - \delta$
7:   end if
8: end for
```

Now we give the Ford-Fulkerson Algorithm.

Algorithm 10 Ford-Fulkerson Algorithm

```
1: $f \leftarrow 0$
2: while $\exists$ augmenting path $P$ in $G_f$ do
3:   Augment$(f, P)$
4: end while
5: return $f$
```

Theorem 4.2. *If F-F algorithm terminates, it finds a max flow.*

Claim. $\forall$ st-cut (A, B), st-flow f, we have

$$\text{val}(f) = \sum_{\substack{u \in A, v \in B \\ (u,v) \in E}} f(u, v) - \sum_{\substack{u \in E, v \in A \\ (u,v) \in E}} f(u, v)$$

It proves the previous claim weak duality.

Proof.

$$\begin{aligned}\text{val}(f) = &\sum_{(s,v)\in E} f(s,v) - \sum_{(u,s)\in E} f(u,s) \\ &+ \sum_{\omega\in A-\{s\}} \left(\sum_{(u,w)\in W} f(u,w) - \sum_{(w,v)\in E} f(w,v) \right)\end{aligned}$$

□

Proof of the Theorem 4.2. Consider the residue graph G.

Denote A to be the set of nodes reachable from s. $B = V\backslash A$. $t \in B$ since there is no path from s to t.

Then st-cut (A, B) has capacity $c_f(A, B) = 0$. So for $u \in B, v \in A$, since $f(u, v) \neq 0 \Rightarrow c_f(v, u) > 0$, we have $c_f(v, u) = 0 \Rightarrow f(u, v) = 0$.

$$\begin{aligned}\text{val}(f) &= \sum_{\substack{u\in A, v\in B \\ (u,v)\in E}} f(u,v) - \sum_{\substack{u\in B, v\in A \\ (u,v)\in E}} f(u,v) \\ &= \sum_{\substack{u\in A, v\in B \\ (u,v)\in E}} c(u,v) - 0 \\ &= c(A,B)\end{aligned}$$

□

Now suffices to proof that the algorithm terminates.

Lemma 4.3. *If capacities are integral and less than c, then F-F terminates in $O(nmC)$ time and returns an integral max flow.*

The lemma implies we should choose some proper path so that it will terminate fast.

Assume the integral capacities $\leqslant C$ and $G_f(\Delta)$ denoted as G_f with edges of capacites $\geqslant \Delta$.

Algorithm 11 Capacity-Scaling Algotihm

```
1: Initiate $f \equiv 0$, $\Delta \leftarrow$ largest $2^k \leqslant c$.
2: while $\Delta \geqslant 1$ do
3:     while $\exists$ augmenting path $P$ in $G_f(\Delta)$ do
4:         Augment$(f, P)$
5:     end while
6:     $\Delta \leftarrow \Delta/2$
7: end while
```

Theorem 4.4. *The C-S runs in time $O(m^2 \log c)$ since the step 2 runs for $O(m)$ iterations.*

Lemma 4.5. *Every time inner WHILE terminates, max-flow value is less than* $\mathrm{val}(f) + m\Delta$*.*

Corollary 4.6. *Each inner WHILE iterates $\leqslant 2m$. The times complexity is $O(m^2 \log C)$*

Proof of Lemma.

$$\begin{aligned}
\mathrm{val}(f) &= \sum_{e \in E \text{ from } A \text{ to } B} f(e) - \sum_{e \in E \text{ from } B \text{ to } A} f(e) \\
&> \sum_{e \in E \text{ from } A \text{ to } B} (c(e) - \Delta) - \sum_{e \in E \text{ from } B \text{ to } A} (\Delta) \\
&= c(A, B) - \sum_{e \in E \text{ between } A, B} \Delta \\
&\geqslant c(A, B) - m\Delta \\
&\geqslant \mathrm{MaxFlow} - m\Delta
\end{aligned}$$

□

Lemma 4.7. *Length of the shortest augmenting path never decreases.*

Algorithm 12 Shortest Augment Path

```
Initiate $f \leftarrow 0$.
while $\exists s \to t$ path in $G_f$ do
    Find $P : s \to t$ in $G_f$ using least number of edges.
    Augment $(f, P)$.
end while
```

Lemma 4.8. *After $\leqslant m$ iterations, length of the shortest augmenting path strictly increases. Time complexity is $O(nm^2)$*

Proof. Assume $f \xrightarrow{\text{augment(f,P)}} f'$ Denote $l(u), l'(u)$ as the length of the shortest $s \to u$ path in $G_f, G_{f'}$ respectively.

Our goal is to prove $l(u) \leqslant l'(u)$.

$l(u)$ determines "distance" to s.

Define the level graph as the set of all $(u, v) \in E(G_f)$ such that $l(u) + 1 = l(v)$.

Call edges not belong to level graph as *back edge*.

Observation Consider any $e \in E(G_{f'})\backslash E(G_f)$, e must be a back edge in G_f.

Choose u such that $l'(u) < l(u)$ and $l'(u)$ minimized.

If (v, u) is the edge in the shortest path of $G_{f'}$

$$l(v) \leqslant l'(v) = l'(u) - 1 < l(u) - 1 \leqslant l(u) - 2$$

so (u, v) is not a back edge in G_f, hence $(v, u) \notin E(G_{f'})$, which causes contradiction. □

Lemma 4.9. *After $\leqslant m$ augmentation, $\exists u$, $l(u)$ strictly increases. It goes on no more n^2 times, so the time complexity is $O(n^2m^2)$.*

Proof. This lemma is much easier than the previous lemma.

Noticed that each augmentation adds back edges and removes at least one edges in level graph. □

Proof of Lemma 4.8.

Claim. During the period when $l(t)$ doesn't increase, the added edges in residual graph does not appear in shortest augmenting path.

Suppose for contradiction: $\exists j < i$, $l_j(t) = l_i(t)$, $\exists\ (v, u)$ appears in the shortest augmenting path P in G_{f_i} and $l_j(v) = l_j(u) + 1$.

Choose the edge (v, u) with smallest i and then with largest $l_i(u)$.

Then $l_i(u) \geqslant l_j(u) + 2$. So

$$\begin{aligned} l_j(t) &\leqslant l_j(u) + |P[u \to t]| \\ &\leqslant l_i(u) + |P[u \to t]| - 2 \\ &\leqslant l_i(t) - 2 \end{aligned}$$

□

Recent work [Chen et at al. '2022] we can do in $O(m^{1+o(1)})$.

4.2 Appllication

4.2.1 Bipartite Matching

Example 4.10 (Bipartite Matching)**.** For Bipartite graph $G = (U, V, E)$, a matching $M \subset E$, we want to find M to maximize $|M|$.

We can construct two virtual nodes s, t such that $s \to$ all nodes in U and $t \to$ all nodes in V, with capacity 1.

Then the maximum capacity of flow in the augmented graph is what we need.

So the meaning of capacity can be generalized as the number of one node who can accommonded

Now consider so-called "perfect matching", *i.e.* $|M| = |U| = |V|$.

Note that $\exists$ perfect matching *s.t.* $\forall S \subset U$, $|\Gamma(S)| \geqslant |S|$.

Theorem 4.11 (Hall's Theorem). *The inverse still holds. i.e. If $\forall S \subset U$, $|T(S)| \geqslant |S|$, then $\exists M$ is perfect matching if $|M| = n$.*

Proof. It suffices to prove max-flow$= n$. Or equivalently, to prove min-cut$\geqslant n$.

i.e. $\forall$ s-t cut (A, B), $c(A, B) \geqslant n$.

$c(A, B) < +\infty \overset{1}{\Rightarrow}$ if $u \in A$ then $\Gamma(u) \in A \Rightarrow \Gamma(A \cap U) \subset A \cap V$.

$$\begin{aligned} c(A, B) &\geqslant |B \cap U| + |A \cap V| \\ &\geqslant n - |A \cap U| + |\Gamma(A \cap U)| \\ &\geqslant n \end{aligned}$$

□

Remark 4.12. Here we mark the weight between U and V to be ∞ such that the fact 1 holds.

The duality of max-flow and min-cut is very useful in this problem.

4.2.2 Network Connectivity

Example 4.13 (Network Connectivity). Directed $G = (V, E)$, source s, sink t. Then Max-flow = the maximum numbered of edge-disjoint $s \to t$ path. Two paths are called *edge-disjoint* if they have no edge in common.

Connectivity of the graph is defined as the $\min\limits_{E' \subset E} |E'|$ such that $s \to t$ disconnected in $(V, E \backslash E')$

Theorem 4.14 (Menger's Theorem). *Connectivity=min-cut=max-flow=maximum number of edge-disjoint $s \to t$ path.*

4.2.3 Circulation

Example 4.15. Directed graph $G = (V, E)$ with capacity $c : E \to \mathbb{R}_{\geqslant 0}$ and node demand $d : V \to \mathbb{R}$. ($d(u) < 0$ means the supply node)

We have the flow conservation

$$\sum_{e \text{ into } u} f(e)0 - \sum_{e \text{ out of } u} f(e) = d(u), \forall u$$

Our task is to decide whether there exists a feasible flow f satisfies the flow conservation.

Indeed, we can consrtuct two vitual nodes s, t such that s to all nodes with demand $d < 0$, equipped with capacity d and t has edges from all nodes with demand $d > 0$, equipped with capacity d.

Then the task is equivalent to check whether the max-flow saturates all edges out of s and in of t.

Moreover, we can use the cut to discuss.

We have:

$$\nexists \text{ feasible circulation} \Leftrightarrow \exists \text{ cut } (A, B) \quad s.t.\ c(A, B) < \sum_{v \in B} \mathrm{d}(v).$$

Remark 4.16. This crieterion, similar as Hall's Theorem 4.11, is so-called "*polynomial proof*", under the meaning that for a specific case, we can give a proof in polynomial time to check.

Flow Lower Bounds If we have a capacity constraint such that

$$l(e) \leqslant f(e) \leqslant c(e), \forall e \in E$$

Then it suffices to add the lower flow at first. For example, for two nodes with demand 0 and edge with amount in $[4,6]$, we can replace it with

$$4 \xrightarrow{[0,6]} -4$$

4.2.4 Survey Design

Example 4.17 (Survey Design)**.** We ask n_1 customers about n_2 products. Ask customer i the number between $[c_i, c_i']$ products and ask the number between $[p_j, p_j']$ customers questions about product j.

We want to find if there is a feasible survey design.

It is equivalent to give each edge a weight interval, where $s \to i$ with $[c_i, c_i']$, $i \to j$ with $[0,1]$ and $j \to t$ with $[p_j, p_j']$.

4.2.5 Airline Scheduling

Example 4.18 (Airline Scheduling)**.** Flight i from the origin o_i at time s_i to the destination d_i at time f_i.

We want to know what is the minimum number of crews in flights that can be scheduled. A feasible schedule for one crew is a set of flights $\{i_1, i_2, \cdots\}$ such that $f_{i_k} \leqslant s_{i_{k+1}}$, $d_k = o_{k+1}$.

We can construct a graph with nodes $o_i \to d_i$. The edges $o_i \to d_i$ with weight $[1,1]$. If the schedule from flight i to flight j is feasible *i.e.* $f_i \leqslant s_j$, we let edge $i \to j$ with weight interval $[0,1]$.

Then a feasible flow gives a feasible schedule. To limit the total amount, we can determine the minimum number.

Remark 4.19. The weight interval have a broader meaning in this problem. With different view of nodes and edges, we can transform it into different limitations.

4.2.6 Image Segmentation

Example 4.20 (Image Segmentation)**.** For an image, $p_{ij} \geqslant 0$ is the separation penalty if neighbors i, j belongs to different partitions.

$a_i \geqslant 0$ is the likelihood that $i \in A$ (foreground) $b_i \geqslant 0$ is the likelihood that $i \in B$ (background)

Our goal is to partition pixels into A, B, to maximize

$$\sum_{i\in A} a_i + \sum_{j\in B} - \sum_{\substack{i,j \text{ neighbors} \\ |\{i,j\}\cap A|=1}}$$

It is equivalent to

$$\begin{aligned} &\text{minimize } -\sum_{i\in A} a_i - \sum_{j\in B} b_j + \sum_{\substack{i,j \text{ neighbors} \\ |\{i,j\}\cap A|=1}} p_{ij} \\ \Leftrightarrow\ &\text{minimize } \sum_{i\in B} a_i \sum_{j\in A} b_j + \sum_{\substack{i,j \text{ neighbors} \\ |\{i,j\}\cap A|=1}} p_{ij} \end{aligned}$$

Then we can construct a visual source with edges to all pixels with weight a_i and a visual sink with edges from all pixels with weight b_i. All neighbors of pixel have edges of weight p_{ij} from each other

Remark 4.21. This example focuses on the optimal sum of net flow. To construct visual source, sink and proper edges, we can optimize some sum of structure with related constraints.

4.2.7 Project Selection

Example 4.22 (Project Selection). $v \to w$, v depends on w. Our goal is to find a feasible set S of projects(if $v \in S$, then all prequisites of $v \in S$) to maximize

$$\sum_{v \in S} p(v)$$

Index

List of Theorems