# АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА СЪОБЩЕНИЕ

**Въведение**

В този анализ за Ваша полза ще са разгледани доста различни идеи, а Вие може да измислите/сте измислили и някои съвсем различни. Основното авторово решение обаче беше измислено и имплементирано (заедно с някои помощни материали като грейдър) в рамките на ограничено време с цел то (или нещо с подобна сложност) да е реалистично за реализиране на състезание. Затова и накрая ще намерите възможности за подобрение над него, които умишлено не са включени.

**Начални наблюдения**

Едно от първите неща, които можем да забележим е, че имаме два канала за въвеждане на грешки в реконструкцията ни. Първо, самите битове, които са изтрити. И второ (и по-важно) при изтриване на бит останалите се разместват и не знаем кой от къде е дошъл. Нашето решение може да се фокусира по-малко или повече върху двата различни вида грешки.

**Дубликация (duplication_random_fill) ~24 т.**

Като видим, че ще се трият неща, най-очевидното е може би да направим няколко копия. Можем да копираме цялото съобщение два пъти, но това макар и лесно осъществимо за първият събтаск изисква малко играчка за по-нататъшните. Можем също и да дубликираме съобщението два пъти, като втория е завъртян на обратно, но изключвайки последния събтаск за сега, можем да игнорираме този вариант, защото е еквивалентен на това всеки бит на място да дубликираме два пъти. Т.е. низът 0110 ще стане 00111100. Тук имаме и много лесна функция за декодиране – просто вървим по низът и, ако някъде има нечетна бройка еднакви битове, добавяме още един. Това ще изкарва оценки от около 0.5 на първия събтаск, тъй като винаги ще възстановява целият низ, което е и теоретичният максимум за това решение, защото изпраща съобщение с дължина $2N$ т.е. имаме $N/(2N - D)$.

Проблемът е, когато започнем да трием повече от един бит. Ако двата изтрити бита са в отделни групи (група ще наричаме поредни еднакви битове), тогава нямаме проблем, но при около $D = \sqrt{N}$ започваме да имаме по около ½ очаквани допиране на два съседни бита ($D^2/2$), но като включим и опцията да не са съседни но отново в същата група, виждаме, че често имаме проблеми. След като сме изгубили бит всичко е грешно. Едно решение е остатъкът да запълним с произволен шум.

**Дубликация с умно запълване (duplication_shift_fill) ~34 т.**

Има обаче по-хитра от тази идея. Очевидно нашето решение няма предпочитана посока – т.е. можем да го изпълним както наляво, така и надясно. Също така, очевидно, колкото по-близо сме до началото на изпълнението на решението, толкова по-вероятно е да нямаме грешка в реконструкцията. Така може да ни хрумне идеята първата половина да реконструираме от ляво надясно, а втората от дясно наляво. Например: имаме реконструиран низ 011001, но търсим низ с дължина 8, а не 6. Ще наложим копие на текущата реконструкция, както подравнено от ляво, така и от дясно:

011001__
__011001

И така ще възстановим очакваната вярна реконструкция: 01101001. Това ни гарантира, че при до едно изпускане на бит ще имаме поне сигурни ½ познати битове, а в средният случай и ¾ (казваме сигурни, за да не броим тези, които са познати на шанс.)

Това решение (както и горното до по-малка степен) имат не лоши точки на първите два събтаска, и доста лоши на следващите (с изключение на някои тестове на последния, на което ще бъде обърнато внимание по-долу).

**Само умно запълване (mono_shift_fill) ~34 т.**

След като измислихме тази хитра техника за запълване, можем всъщност да приложим само нея. Ще изпратим низа без модификации и после само ще приложим тази техника върху получения низ – така в средния случай ще имаме около ¾ от точките за първия събтаск. Това би се представяло доста зле на останалите обаче.

Тук ще разгледаме какво се случва и с последният събтаск, донякъде неочаквано това решение хваща малко над половината точки за него. Това се дължи на факта, че там често имаме малко на брой големи групи на триене, което води до това че макар и да се губят много битове, позициите на misalignment са редки, което е добра ситуация за решението/корекцията с шифтването.

**Много копия (multiplication_shift_fill) ~42 т.**

Съдейки по горните два резултата можем да се сетим, че е добра идея параметърът за колко копия искаме на всеки бит да варира спрямо D, оказва се, че няма смисъл те да са над 4, защото просто не си заслужават бройката копия (макар че и да го позволим не смъква значително резултата ни). С малко игра с параметри (които, обърнете внимание, можем да правим локално с Lgrader-а) за няколко минути можем да открием добри граници за различните стойности.

Това има добри точки за първите три събтаска, но доста лоши такива за последният.

**Много копия с лек чийт (multiplication_shift_fill_cheat) ~46 т.**

В горното решение не се възползвахме от факта, че последният събтаск е по-тежък от предните и съответно, ако успеем да изкараме доброто представяне на решението само с умното запълване, ще си подобрим точките значително. Това за жалост ще стане на цената на третият събтаск, тъй като нямаме добър начин да разберем в кой се намираме. (С още чийтене обаче вероятно бихте могли да се справите и изкарате още малко точки с тази идея.) Но с малко интуиция или пробване можем да се сетим да пускаме по-простото решение само на най големите тестове, където е най-важна смяната.

**Групиране на битовете (probabilistic_batches) ~78 т.**

Във всички до сега разгледани идеи, основният подход е бит по бит да кодираме и декодираме съобщението, като вземаме мерки за отделните битове да са максимално точни (с тази допълнителна корекция накрая). Основният ни проблем тук е, че колкото и да сме перфектни трудно ще надминем 50 точки, защото почти винаги правим поне по две копия и от там делим точките си поне на две. Една по-добра идея е да имаме до-някъде подобен процес, но вместо първо да групираме битовете на пакети и после да кодираме и декодираме пакет по пакет. Тук ще намерите обяснение на едно такова решение измислено и имплементирано от Енчо Мишинев:

Това решение се фокусира главно върху случая K=1, т.е. първите три подзадачи. Нека разделим низа на групи(пакети) от по $10$ последователни бита получавайки $N/10$ пакета, които искаме да изпратим. Понеже $D \leq N/10$, можем да очакваме, че ако просто изпратим всички пакети без модификации, бихме имали средно по 1 липсващ бит на пакет. Проблемът е, че ако получим пакет с един липсващ бит не е възможно еднозначно да определим оригиналния пакет.

Нека асоцииираме всяка поредица от $10$ бита с друга поредица от $10 + P$ бита. Най-простия начин да направим това е към всяка редица от $10$ бита да добавим неин детерминистичен хеш от $P$ бита. Така вместо оригиналните пакети от по $10$ бита можем да изпратим техните асоциирани пакети от по $10 + P$ бита. Тъй като голяма част от възможните редици от по $10 + P$ бита не са асоциирани с никоя редица от по 10 бита, то е възможно губейки няколко бита от пакет все пак еднозначно да възстановим оригиналният. Очевидно по-голяма стойност на $P$ увеличава размера на съобщението, което изпращаме, но подобрява вероятността за правилно декодиране на съобщението.

Ключова стъпка в този подход е реконструирането на оригиналния низ от полученото съобщение. Искаме да генерираме възможно най-вероятния списък от пакети по дадено съобщение. За да постигнем това можем да използваме следното динамично оптимиране:

$F[i][j]$ $=$ вероятността първите $i$ символа от полученото съобщение да отговарят на $j$ пакета.

Изчисляваме стойностите като итерираме по потенциалния размер $s$ на последния пакет (след потенциални загуби на битове от него):

$$F[i][j] = \max_{0 \leq s \leq 10+P} (F[i-s][j-1] \times \text{loseChance}(10+P-s) \times \text{decodable}(i-s+1, i))$$

Функциятя $\text{loseChance}(x)$ отговаря на шансът в произволен пакет от $10 + P$ бита от изпратеното съобщение да бъдат изтрити точно $x$ бита. Тези вероятности могат лесно да бъдат изчислени с биномна дистрибуция. Дори да се използват лоши приближения за тази функция, решението пак се справя доста добре.

Функцията $\text{decodable}(L, R)$ е равна на $1$ ако низът образуван от битовете на позиции от $L$ до $R$ в полученото съобщение е възможно да отговаря на пакет, и е равна на $0$ в противен случай. Това всъщност е ключовата функция за декодиране, тъй като при добър избор на $P$, част от низовете в полученото съобщение не е възможно да са получени от частично изтриване на валиден пакет. Това насочва декодирането към правилно откриване на позициите на пакетите. За изчислението на тази функция решението използва преизчисление базирано на пълно изчерпване със сложност $O\big((10+P) \times 2^{10+P}\big)$.

Използвайки това динамично оптимиране можем да разделим полученото съобщение на най-вероятната поредица от $N/10$ пакета от по $10 + P$ бита, някои от които имат липсващи битове. След това за всеки от тези пакети трябва да видим с кой пакет от $10$ бита е асоцииран. Често заради загуби това няма да е еднозначно, но при добра стойност за $P$, голяма част от пакетите ще бъдат правилно реконструирани.

Тъй като вероятностите при пресмятането са ужасно малки числа, то се прилага стандартния трик да се използват логаритмите от всички вероятности.

Очевиден проблем на това решение е, че при $K > 1$ е доста възможно да бъде изтрит цял пакет или няколко съседни пакета, което би довело до сериозно объркване на решението. Един опит за подобрение на това е да променим функцията loseChance за четвъртата подзадача, но това би било за сметка на третата подзадача, тъй като не е възможно да разберем в коя от двете сме.

Стойността на $P$ е най-добре да се определи емпирично използвайки или примери локално или събмити към системата. Тъй като дължината на изпратеното съобщението е точно $M = (N/10) \times (10 + P)$, то целта би била да минимизираме $P$. При твърде малко $P$, обаче, точността на решението рязко спада, тъй като голяма част от пакетите не могат да се реконструират еднозначно. Оказва се, че за първите две подзадачи $P = 1$ е достатъчно добър избор, а за третата (и съответно четвъртата) се налага $P = 5$.

Очевиден подход към подобрение на това решение е асоциирането на низове от по $10$ бита с низове от по $10 + P$ бита да се направи по по-умен начин. Ако низовете от по $10 + P$ бита се изберат така че да са възможно най-различни, то шансовете да имаме еднозначно реконструиране при частично изтрит пакет стават много по-добри. За целта на задачата, обаче, това не е нужна оптимизация, тъй като дори най-простото асоцииране споменато по-горе взима почти максимален брой точки за първите три подзадачи. Единствената причина решението да не е близо до 100т е невъзможността му да се справи с четвъртата подзадача.

**Скелет, произволен (skeleton_random_inefficient) ~96 т.**

Всички тези идеи страдат от един основен проблем – когато изгубят нишката на това кой бит къде принадлежи нямат начин да се възстановят. Т.е. ако се върнем към началните си наблюдения – те се фокусират върху първият вид загуба на информация много повече от колкото трябва.

Ако желаем да изкараме повече точки, трябва да измислим по-добър подход. Тук всъщност и започва същинското авторово решение. Като знаем какъв е проблемът, който искаме да решим, можем относително лесно да се сетим, че едно възможно решение е да поставяме разни маркери из съобщението. След това ще ги засичаме при получателя и така ще можем да следим къде се намираме в съобщението. Естествено изглежда сякаш би имало много усложнения – можем да изтрием битове от маркера, може да изтрием цял маркер, може той да се среща другаде в съобщението и вероятно други. Нека обаче за момент игнорираме тези и помислим малко по абстрактно.

Тъй като искаме и изпращача, и получателя трябва да се съгласуват за това какви и къде са маркерите можем да кажем, че двете на база $N$ и $D$ трябва да генерират еднакъв скелет, който се състои от разни фиксирани независещи от съобщението битове и празни места, където ще попълним данните си. След това искаме да решим два проблема: първо как ще оперира получателя и второ какви са добри скелети.

За щастие тъй като вече не мислим за отделни маркери ами за цялостен скелет, можем да използваме един много общ подход за мачване със скелета. Реално имаме две думи и знаем че едната е получена от другата (полученото съобщение от изпратеното такова) чрез $D$ изтривания. Тук можем да се сетим да ползваме алгоритъмът за minimum edit distance, който е едно просто динамично и с малка модификация може да реконструираме пътят от промени във думите. Всъщност, тъй като в случая има само един вид промяна – триене от единия стринг (без добавяне или смяна на символ) няма нужда да попълваме самото dp с edit distance-и да вземаме минимуми, ами само dp-то с инструкции за реконструкция. Проблемът е, че нямаме оригиналното съобщение, с което да мачваме, така че да открием позициите на триене. Това обаче може лесно да се реши, ако просто мачваме директно с непопълнения скелет като празните му полета винаги третираме за мач. Така реконструираме в този ред: местата на триене, оригиналното изпратено съобщение, оригиналните данни.

Сега да се върнем на вторият въпрос – какъв скелет да използваме. Ами едно важно наблюдение е, че искаме в него да има вариации с всякакви честоти. Т.е. не е добра идея да има нисък период на повтаряне, защото тогава едно триене на един период или повече ще ни прецака. Не е добре обаче да имаме и само ниско-честотни вариации, защото тогава няма да можем да реконструираме позициите на триене с добра точност. (Всъщност това е въпросът на accuracy срещу precision.) Интересното наблюдение обаче е, че произволният шум има точно свойствата, които търсим, т.е. можем да си фиксираме размер на скелета и да попълваме произволни битове в него, докато не останат точно $N$ празни места.

Остава само да решим колко дълъг искаме да е скелета. Това може да се установи с малко интуиция и малко експерименти (и тук отново на локална среда) и се оказва, че броя допълнителни битове е горе-долу пропорционален на $\sqrt{D}$, а оптималната константа можем да установим експериментално, но и близки стойности ще донесат близки точки, така че решението не е твърде чувствително към тази формула (няколко точки вариация).

**Оптимизация (skeleton_random_optimized) ~96 т.**

Една оптимизация по време (която всъщност не е нужна), която можем да направим, е да видим как, макар и алгоритъмът ни за реконструкция започна като версия на minimum edit distance, всъщност не ни трябва никакво динамично тъй като в никой момент не ни интересуват предните стойности при попълване на таблицата, ами можем просто да се движим назад по двата низа и когато няма съвпадение, да го броим за изгубен бит. Този процес всъщност можем да го правим и от ляво надясно (но в решението е имплементиран от дясно наляво, за да е по подобен на версията с динамично) и така реално можем да реконструираме низът онлайн, ако това беше нужно (например в реална ситуация, ако получаваме данни дълго време, а не само един пакет). Така всъщност алгоритъмът ни има сложност $O(N)$, но това не е фокусът на задачата и решения до около $O(N^2)$ трябва да влизат в тайм лимита.

**Подобрен скелет (skeleton_regular) 100 т.**

Можем всъщност да подобрим представянето на решението си с няколко процента, ако вместо изцяло произволен скелет използваме такъв с малко повече и по-равно разпределени високо-честотни вариации.

Основният проблем на изцяло произволният е, че е неравно разпределен из съобщението на локално ниво. Т.е. докато гледаме глобално, е разпределен добре, но като погледнем един малък подниз, може да има няколко фиксирани бита един до друг (от които не печелим толкова) и после дълго разстояние без никакви такива битове (това обяснение е малко опростено но идеята е подобна). Тогава като имаме загуба на бит в това празно място, нямаме никакъв начин да видим как точно ще подравним останалите битове вътре.

Затова едно (в общия случай) подобрение е да слагаме фиксирани битове на равни интервали (или горе-долу равни например x__x___x__x___x__x…), а самите битове да са произволни, за да няма някакъв малък период скелета ни. Това е подобрение при повечето видове тестове освен в екстремните случаи когато трием много битове наведнъж (много голямо $K$), защото тогава локалното разпределение няма никакво значение, а само ниско-честотните вариации, които по този начин все пак леко намаляваме.

**Следващи подобрения**

Една посока на развитие на решението е да се избира по-умен скелет. Тук разгледахме два подхода: произволен и конструиран според някакъв зададен шаблон, който е лесно да се програмира. Може да се окаже обаче, че има скелети или класове скелети, които се разпознават по-лесно или по друг начин подобряват оценката ни. Един подход е да използваме генетични алгорими или други оптмизиационни техники, за да открием такива скелети.

Друг аспект за развитие се базира на следното: при финалното авторово решение не губим повече от 1000-1500 (в зависимост от $D$) бита и дори това е в най-лошите случаи. Тук обаче загуба на бит, ефективно значи, че той просто е променен, т.е. този ни протокол за комуникация превръща триения към просто промени на разни битове. Над него можем да насложим друг протокол, който да се оправя със самите промени. Няколко обещаващи идеи са:

- Добавяне на parity bit на всеки $T$ бита (т.е. добавяме 0 или 1 накрая на всяка група от $T$ бита, така че броят единици да е четен).
- Добавяне на хеш или нещо друго накрая на разните групи или преди фиксираните от скелета битове.
- Решението **probabilistic_batches**, потенциално с намалени параметри и други модификации, тъй като има по-малко/по-прости грешки за коригиране.
- Други методи, които се представят добре при малък брой флипвания на битове.

*Автор: Емил Инджев*