Начални наблюдения

Задачата, ако не беше на Стекланг, би била доста лесна за решаване линейно. За по-незапознатите може да изглежда сякаш е нужно да използваме алгоритъм на Дийкстра, защото ребрата имат тегла, но всъщност, когато теглата са само 0 или 1, може да се използва 0-1 BFS. При този алгоритъм поддържаме дек (двустранна опашка). На всяка итерация попваме връх от предната страна на дека и добавяме всички негови съседи в дека. Тези с нулево ребро добавяме отпред, а тези с единично ребро добавяме отзад. Естествено, както при повечето подобни алгоритми в графи, трябва да не посещаваме един връх над веднъж или поне, ако го посетим втори път, да не добавяме съседите му пак, също така трябва да записваме разстоянието до всеки връх и така нататък.

Сега да видим как да се оправим с първия привиден проблем тук: трудно ще запазваме и проверяваме кои върхове се посетени използвайки само стекове. Това обаче не е проблем, защото езикът донякъде го прави вместо нас. Инструкцията `getEdges` не може да ни върне ребрата от някой връх втори път дори и да искаме, т.е. освен ако много не се постараем е трудно алгоритъмът да разглежда съседите на даден връх по повече от веднъж. В крайна сметка това привидно ограничение всъщност ни помага, нещо което не може да бъде казано за еквивалентно ограничение на `getEnd`, както ще видим по-нататък.

Подзадача 1: Всички ребра са с тегло 0

Лесно се вижда, че в тази подзадачи, до всеки връх или има път с тегло 0 или няма път. Т.е. само трябва да проверим дали има път от $Start$ до End. Можем да използваме просто DFS, което е доста лесно в този език, защото стек е базовия вид памет (а именно той е нужен за DFS). В най-лесната имплементация ни трябва 3 стека: `front`, `end` и `edges`. Във `front` ще държим текущите върхове, като в началото пушваме началният връх там, в `end` – крайният връх, който пушваме там в началото, а в `edges` – ребрата излизащи от върха, който текущо разглеждаме. Естествено, ако достигнем `end`, ще върнем 0, а ако `front` е празен ще върнем -1. Така можем да имплементираме DFS на този език без особени затруднения.

Подзадача 2: От всеки връх има не повече от един път до всеки друг

Първо, нека отбележим, че за тази подзадача, не ни е нужно да се възползваме от това, че `getEdges` връща ребрата от даден връх само по веднъж. Това обаче не е от значение за решението, защото то си се случва винаги. Важното е, че тук единственият път от $Start$ до End, ако съществува, ще бъде открит от DFS, т.е. можем отново това да правим, но да поддържаме разстоянието до всеки връх във `front`. Най-лесната имплементация е да използваме 4 стека: предишните 3 и `dist`. Планът е във `front` да се редуват номера на върхове с разстоянията до тях. След това, след като пушнем ребрата от даден връх в `edges` и го попнем от `front`, ще запазим неговото разстояние в `dist` и и него ще попнем от `front`. Така всеки път като пушваме нов връх във `front`, първо ще пушваме разстоянието до него, което е равно на `dist` плюс теглото на реброто. Другата промяна, която трябва да правим е да връщаме текущото разстояние до върха (вместо 0), когато достигнем `end`.

Забележете, че макар и по ограничения тази подзадача да не включва подзадача 1, всяко (разумно) решение за нея ще решава и предната.

Подзадача 3: Всички ребра са с тегло 1

Сега най-накрая се налага да правим BFS, но не 0-1 BFS, а нормално такова. В най-лесната му имплементация са ни нужни 5 стека: предните и `back`. За разлика от в предната подзадача обаче, няма да държим разстояния във `front`, копирайки текущото в `dist`. Вместо ще поддържаме текущото разстояние само в `dist` и ще го увеличаваме с 1, всеки път, когато BFS-то ни влезе едно ниво „по-навътре“. В началото, ще пушнем 0 в `dist`. След това на всяка стъпка разглеждаме всички върхове във `front`, пушваме всички техни съседи в `back` (естествено ще връщаме `dist`, ако сме достигнали до end), след това увеличаваме `dist` с 1 и накрая прехвърляме всички върхове от `back` във `front`. Това може да се имплементира, като най-външния цикъл е буквално по-дълбочина в BFS-то и в него има цикъл по `front`. Алтернативно (както е в авторовите решения) може направо външният цикъл да си е по `front` и, когато той е празен, ако и `back` е празен връщаме -1, а ако не е, увеличаваме `dist` с 1 и прехвърляме всичко от `back` във `front`.

Забележете, че макар и по ограничения тази подзадача да не включва подзадача 1, всяко решение за нея може лесно да се модифицира да я решава, като връщането на `dist` се замени с връщане на 0. Тъй като подзадачите се оценяват отделно от системата, това значи, че всеки написал решение за подзадача 3 трябва веднага след това (в нов събмит) да може да вземе и подзадача 1.

Подзадача 4: Пълната задача

Тук най-накрая се налага да напишем пълното 0-1 BFS. Наивната му имплементация е ефективно същата като тази на нормалното BFS от подзадача 3. Единствената основна разлика е, че всеки път, когато пушваме нов връх в опашката, първо ще проверим теглото на реброто му – ако е 1, ще го пушнем в `back` (както преди), но ако е 0, ще го пушнем във `front`.

Това всъщност е стандартен начин за имплементиране на дек/опашка чрез два стека (релевантно например във функционални езици като Haskell), но тук двата стека си имат допълнително значение, а именно `front` са всички върхове на разстояние `dist` от Start, а `back` са върхове на разстояние `dist + 1`. Именно това ни позволява да държим разстоянието извън тези два стека, а не за всеки връх да го пазим поотделно за всеки връх както в решението на подзадача 2. Това е важно, както за по-удобно писане на програмата, така и за оптимизациите, които сега ще разгледаме.

Първо махане на стек

Първото махане на стек, което ще разгледаме, може да се приложи към решенията ни от всичките подзадачи, а именно да слеем стековете `end` и `edges` в стек `endEdges`. В него нормално ще държим само End, но когато обработваме ребрата на текущия връх, първо ще пушнем разделител -1, който ще третираме като дъното на стека, и после нормално ще обработим ребрата, като най-накрая ще попнем разделителя.

Алтернативно, можем да слеем стековете `dist` и `end` (във всички решения освен това за първата подзадача), тъй като и двете са просто единични променливи. Това е малко по-досадно, защото всеки път, когато искаме достъп до долната от двете стойности, трябва да местим горната върху някой трети стек. Също така, както сега ще видим, има по добро място за `dist` (във решенията на подзадачи 3 и 4).

Второ махане на стек

Тук ще разгледаме как да махнем още един от стековете в решенията на подзадачи 3 и 4 преди да стигнем до най-сложната такава оптимизация, която е нужна за финалното махане (включително и за подзадача 2).

Очевидното нещо за оптимизиране е стека `dist`, защото както end той държи само една стойност. Както беше споменато и в края на предната част, можем да го слеем със `endEdges` стека, и това също може да доведе до решение с 2 стека, но то би било малко по-досадно за писане и малко по-неефикасно от към брой итерации.

Вместо това, можем да забележим, че увеличаваме `dist` само, когато `front` е празен (когато и прехвърляме съдържанието на `back` към `front`). Това ни навежда на мисълта, че можем да държим текущото разстояние в дъното на `front` (или на `back`), като над него слагаме разделител -1, който третираме за дъно на стека. Така, когато той се „изпразни“ махаме разделителя, увеличаваме разстоянието с 1 и връщаме разделителя. Остава да видим само как да върнем отговорът накрая, защото той вече не е така лесно достъпен. Това обаче си е доста лесно – просто попваме всичко от `front`, докато не стигнем разделителя, попваме и него и така накрая връщаме разстоянието.

Забележете, че нито една от тези две идеи не важи за решението на подзадача 2, защото там `dist` е нещо изцяло различно, което се използва на всяко пушване на нов връх, т.е. не може да стои някъде скрито.

Финално махане на стек

Финалната оптимизация, свеждането на решенията ни до използване на само 2 стека, е доста по-сложна от досегашните ни оптимизационни идеи. Основният проблем е, че със само 2 стека, нямаме допълнителен трети стек, който да използваме за всякакви местения на неща между стековете. Т.е. дори и задачата да разменим горните два елемента на стековете `A` и `B` изглежда едва ли не невъзможна. Всъщност всички други (решими) логистически проблеми със 2 стека (например вмъкване на горния елемент на `A` под горния елемент на `B` или размяна на горните 2 елемента на `A`) се свеждат до тази задача. Тъй като говорим за най-горните елементи на двата стека, можем направо да ги третираме като числа, а не като стекове. Т.е. чудим се можем ли да разменим две числа, без да използваме трето за временно съхраняване. Всъщност операциите събиране и вадене са ни достатъчни: `A := A + B; B := A - B; A := A - B`. (Може да си разпишете стойностите на `A` и `B` спрямо началните им, за да се убедите, че това е вярно.) Първата и третата стъпка могат директно да бъдат имплементирани с по една инструкция на Стекланг. Втората стъпка е малко по-проблемна, но въпреки това може да се разбие на 4 инструкции: `push A A; sub A B; pop B; push B A; pop A`.

След като вече знаем как да разменяме стойностите на върховете на двата стека, т.е. да правим суапове, решаването на задачата със само 2 стека далеч не изглежда толкова непосилно. Има много варианти за точните логистики, но като груба насока: първо да си представим как бихме решили задачата със само 2 стека плюс няколко регистър за само едно число `end` (потенциално и един за `dist`). След това решаваме, че ще държим крайния връх върху някой от двата стека и всеки път, когато иначе бихме пушнали върху него (или от него) първо използваме суапове, за да нагласим нещата така че всичко релевантно да е на върховете на стековете.

Другият основен проблем е какво да правим с ребрата. За пълната задача, трябва например да пушнем ребрата на текущия връх върху `back`, после да прехвърлим всички

към `front`, разделяйки единичните от нулевите с разделител -1, но премахвайки самите тегла, накрая трябва да прехвърлим единичните обратно към `back` и да махнем разделителя след това. Това води до решение, което използва около 1.8 милиона итерации, но би трябвало всякакви разумни, относително ефикасни имплементации да използват под 3 милиона итерации и да получат 100 точки.

Алтернативна идея, предложена от Luke Miles използва само около 870 хиляди итерации. Това се постига с няколко ключови оптимизации. Първо, вместо да прехвърляме всички ребра по 1-2 пъти, в стековете `front` и `back` държим и нули, които не значат нищо, над върховете. Така можем да пушнем текущите ребра върху `front` и да преместим само единичните ребра към `back`, заменяйки единиците с нули. Второ, при изпразване на `front`, вместо да прехвърляме всичко от `back` към `front`, можем просто да „разменим“ двата стека, като имаме две версии на програмата – една с разменени стекове и една с неразменени стекове, т.е. те вече са `even` и `odd`. Това изисква държане на разстоянието в дъната и на двата стека (като едното е с едно повече от другото) и това то да се увеличава с по 2 на изпразване. Финално, вместо суап, който всъщност после използваме за вмъкване, можем директно да използваме вмъкване, спестявайки по 3 итерации на „суап“ (което е доста тежка операция). Разбира се, вероятно могат да се правят и още много микро-оптимизации по кода, но това са ключовите идеи.

Автор: Емил Инджев