АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА МАКСИМАЛЕН ПЪТ

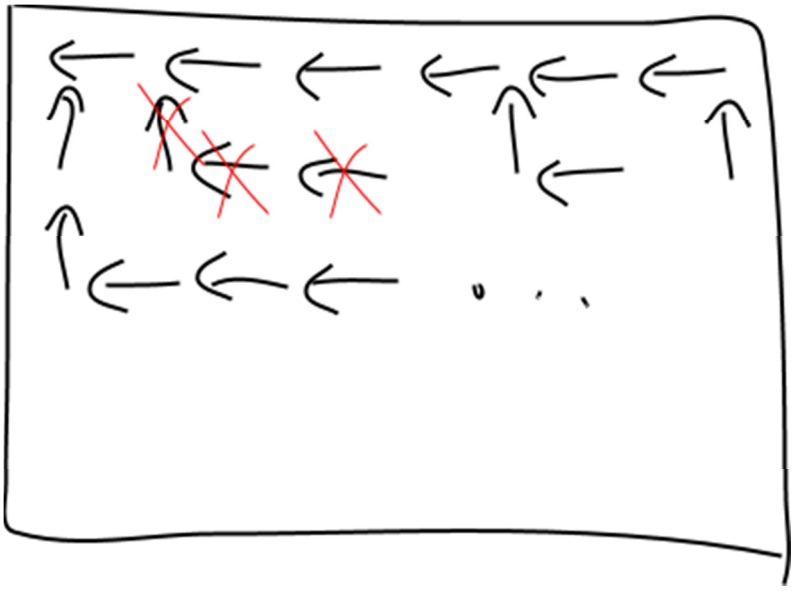
- **Общи коментари:**
  - Задачата изглежда прекалено позната. Очаква се повечето състезатели да са решавали подобна задача многократно, но да не са се замисляли, че съществуват и други по-ефективни нейни решения. Дадените ограничения по памет и време предполагат, че също толкова бързо решение трябва да заема много по-малко от O(N^2) памет.
  - Дадената функция, генерираща входната матрица, е доста „случайна“: стойностите във всяка клетка са досущ като взети от генератор за случайни числа. Това означава, че не е нужно да решаваме задачата за в най-лошия случай, а в средния.
  - Приложенята на задачата са напълно реални: в биоинформатиката, например, се налага решаването на тази задача за изключително големи матрици, които не се побират в RAM-паметта на един компютър.
- **стандартно динамично: O(N^2) време и памет, maxpath-dummy.cpp, ~10т.**
  - Нека dp(i,j) е максималната сума/дължина на път от (1,1) до (i,j). Тогава dp(i,j) = max{ dp(i-1,j), dp(i,j-1) } + F(i,j), при предположение, че dp(0,.) и dp(.,0) сме подставили формално равни на нула. Можем последователно от ляво надясно и от горе надолу да изчислявим всяка от стойностите dp(i,j), като в path(i,j) записваме по един бит информация – дали за максималния път до (i,j) сме направили последния ход от горе или от ляво. Максималният път възстановяваме като от (N,N) се върнем назад до (1,1) по стъпките, които path подсказва.
- **N стандартни динамични: O(N^3) време, O(N) памет: ~20т., maxpath-slow.cpp**
  - След изчисляване на dp(N,N) със запомняне на предишните два реда (като при предишното решение), можем лесно да разберем дали сме дошли от dp(N-1,N) или dp(N,N-1): от което е по-голямо. Свеждаме задачата до търсене на максимален път, включващ една клетка по-малко. Повтаряме процедурата 2N-1 пъти (докато не стигнем до първоначалния връх).
- **Оптимизация №1: Оптимизация за последните два реда на dp и с компресия на path: O(N^2) време и памет, maxpath-dummy.cpp, ~25т.**
  - Рекурентната формула dp(i,j) зависи само от стойносите на вляво и отгоре (които са в последните два реда), а значи можем за предишните редове да забравим. Т.е. ще изхарчим не O(N^2), а само O(N) за dp.
  - В масива path не можем да си позволим да забравим минали редове, понеже ще са ни нужни за възстановяване на пътя чак до (1,1), но можем да съкратим паметта 8 пъти: bool path[N][N] би заемал по един байт на клетка, а vector<bool> path[N] с дължина по N във всеки vector, ще заемат само по бит).
- **Оптимизация №2: Разглеждане само на клетки, близки до диагонала: O(N*) maxpath-diag.cpp:**
  - Да разгледаме първо следната проста задача, известна като Random walk. Пиян човек последователно прави стъпка вляво или дясно с вероятност 50%. Къде се очаква да се намира той след k стъпки. Отговор: при четно k – в началното положение, а при нечетно – в съседните на началното положение. Логиката е проста: всички 2^k възможни пътя са

еднаквовероятни, а броят на пътищата с точно r десни и n-r леви стъпки е равен на броят на комбинациите от N елемента r-ти клас: k!/(r!(k-r)!). А максимумът на тези биномни коефициенти се достига при r=k/2.

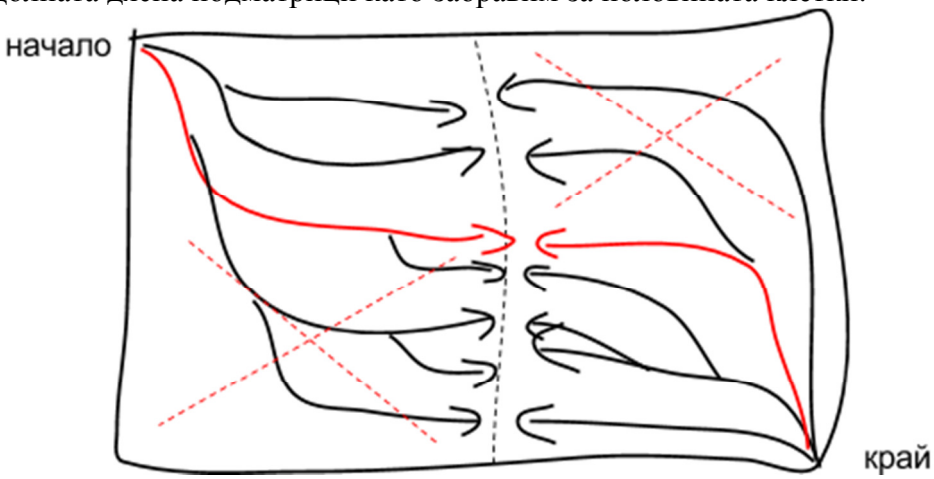
- При N клонящо към безкрайно (безкрайно голяма матрица) имаме абсолютно аналогична ситуация: ако максималният път е достигнал която и да е клетка, то с еднаква вероятност той би могъл да продължи както надолу, така и надясно. Това значи, че максималният път с висока вероятност не се отдалечава твърде от главния диагонал на матрицата. При краен размер на матрицата, „стремежът“ на максималния път към диагонала е още по-силен: от клетка под диагонала съществуват повече пътища с първи ход надясно (към диагонала), отколкото с първи ход наляво

- Теоретическите оценки не са прости за извеждане, но практика, всички от случайно генерираните 20 теста имат максимален път, който не се отдалечава на повече от d=800 клетки от диагонала (погледнете max_diag_offset в profiling.txt). Подобни изследвания биха били сами по себе си интересна задача.

- **Оптимизация №3: Забравяне на бесперспективни пътища, maxpath-cut.cpp:**
  - За основа имаме стандартното динамично програмиране, като за всяка клетка съхраняваме посоката на предишната клетка (нагоре или наляво) в максималния път. Но непрекъснато “трием” запомнените посоки максимални пътища до клетки, от които гарантирано не следва оптимален път до (N,N). Вместо двумерен масив можем да използваме sparse matrix или hash_map<pair<int,int>, bool> (от клетка към посока нагоре или наляво). В най-добрия случай такъв подход ще достига O(N) памет, а в най-лошия – O(N^2) (постройте примери). За равномерно разпрелени и независими стойности, както е в случая, не е ясно каква е сложността по памет и са необходими допълнителни изследвания.



- **Комбиниране на оптимизациите – maxpath-cut-diag.cpp, maxpath-cut-diag-compress.cpp: предположително до 100т.:**
  - Просто е съчетаването на оптимизациите №2: Разглеждане само на клетки, близки до диагонала и №3: Забравяне на пътища.
  - По-сложно е комбинирането на оптимизациите №1: Компресия на path и №3: Забравяне на пътища, понеже трябва да запомним разредена матрица от булеви стойности. Това може да бъде реализирано с използване на някакво хитро хеширане (напр. Bloom filter). Авторът ще се радва да получи различни предложения.
- **Изненада! Разделяй и владей + динамични: Задачата се решава за O(N^2) време и O(N) памет дори в най-лошия случай, maxpath-pesho.cpp и maxpath-iskren.cpp, 100т.**
  - Накратко: Разделяме матрицата на лява и дясна част (или, аналогично, на горна и долна част). Намираме стойносите на най-дългите пътища от началото до всяка средна клетка, както и от края до всяка средна клетка. Средната клетка с най-голяма сума на двойката максимални пътища до нея принадлежи на най-дългия път от началото до края. Значи можем да сведем задачата до търсене на най-дългите пътища в горната лява и долната дясна подматрици като забравим за половината клетки.



  - Надълго: Да разделим матрицата на две части между двете средни колони. С тривиалното dp за O(N*M) време и O(N) памет намираме dp1[i] – максималните дължини/суми на пътищата (но самите пътища не можем

да възстановим) от горната лява клетка до всяка клетка i в средната колона. Аналогично (но „наобратно“) намираме и максималните суми dp2[i] от долната дясна клетка до всяка клетка i в средната колона. Така за всяка клетка i от средната колона знаем максималната сума, която се получава от път минаващ през тази клетка: dp1[i]+dp2[j]. Значи оптимален път ще минава през средната колона в стълба opt_i = maxarg_i(dp1[i]+dp2[j]). Запомняме opt_i като част от пътя.

- Остава аналогично да решим двете независими подзадачи в правоъгълника от горната лява клетка до (opt_i, M/2) и в правоъгълника от (opt_i, M/2) до долната дясна клетка. Всяко такова разбиване на две части води до изхвърляне на половината клетки, така че времето остава асимптотически същото: O(N*M) (геом. прогресия с коефициент 1/2: времето за обработка на k клетки се получава T(k) = O(k) + T(k/2) = O(k+k/2+k/4+...) = O(2k) = O(k)). Паметта също остава линейна, понеже за всяко ниво в дървото на разделяй-и-владей подзадачите имаме само изчисляване на максимални суми, но не и самите пътища.
- Реализацията на това решение може да бъде много кратко – около 50 реда, но е технически интересно. Погледнете кода за подробности.
- Възможно е фактът, че се изисква решаването на задачата в средния случай (само за произволна матрица) да скрие от състезателите съществуването на това решение. Е, извинявайте! :”)

Автор: Петър Иванов