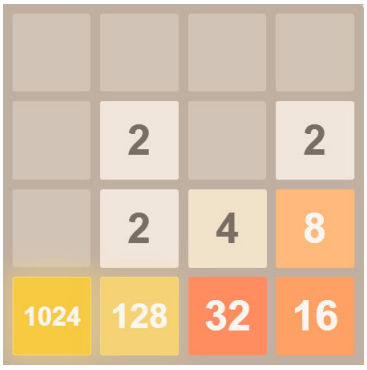
А Л Г О Р И Т М И В Ъ Р Х У Г Р А Ф И
ДОМАШНО № 2 ПО "ДИЗАЙН И АНАЛИЗ НА АЛГОРИТМИ"
ЗА СПЕЦИАЛНОСТ "КОМПЮТЪРНИ НАУКИ", 1. ПОТОК
(СУ, ФМИ, ЛЕТЕН СЕМЕСТЪР НА 2019 / 2020 УЧ. Г.)

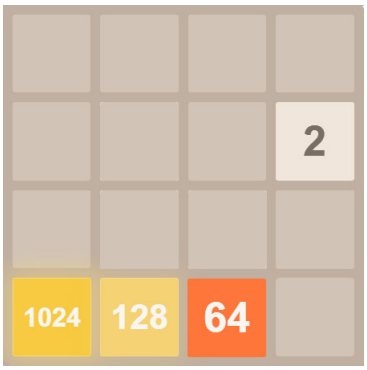
Задача 1 е вдъхновена от компютърната игра "2048". Тук използваме различни правила с цел опростяване на задачата.



Плочки с числа (степени на двойката) са разположени върху клетките на квадратно игрално табло. Някои клетки са заети с плочки, други клетки са свободни. Две клетки ще наричаме съседни, ако имат обща страна; всяка клетка, която не е по края на таблото, има четири съседни клетки. Играчът не може да мести старите плочки — тези, които вече са заели своите места. На всеки ход играчът получава по една нова плочка, която той има право да постави на произволно избрана от него свободна клетка от таблото.

Пример: на таблото, показано вдясно, играчът току-що е получил плочка с числото 2 и е избрал да я сложи върху главния диагонал.

После автоматично протича следната верижна реакция: ако в клетка, съседна на текущата (избраната от играча), има плочка със същото число, текущата клетка се опразва, като плочката от нея се придвижва към съседната клетка, където плочките се сливат (числата се събират), а клетката с промененото число става текуща и на свой ред се слива с някой от съседите си. Процесът продължава, докато това е възможно. Ако на някоя стъпка текущата клетка има две или повече подходящи съседни клетки, играчът избира с коя съседна клетка да се слее текущата.



В примера, показан по-горе, двете съседни клетки 2 се сливат, като горната (новата) плочка слиза. Получават се две съседни четворки. Те на свой ред се сливат, при което се получават две съседни осмици и тъй нататък. Накрая се стига до положението, показано на долната диаграма: числото 64 няма равен съсед и верижната реакция спира.

Ако играчът беше избрал да сложи новата плочка в горния десен ъгъл на таблото, тя щеше да се слее само с плочката под нея и верижната реакция щеше да спре до числото 4.

Целта на играча е да получи възможно най-голямо число в края на верижната реакция (и да опразни възможно най-много клетки на таблото). Помогнете на играча, като съставите алгоритъм, който определя в коя клетка да се постави новата плочка и накъде да продължи верижната реакция, ако на някоя стъпка има две или повече възможности. За целта обяснете как входните данни могат да се представят чрез граф и изберете някой изучен алгоритъм върху графи. Достатъчно е словесно описание на модела и алгоритъма, няма нужда от код. Алгоритъмът трябва да работи върху произволно голямо табло и да бъде възможно най-бърз.

(2 точки)

Демонстрирайте модела, който сте избрали, като начертаете графа, който съответства на по-горната диаграма. Изпълнете избрания алгоритъм върху този граф, за да покажете как алгоритъмът намира положението, показано върху по-долната диаграма.

(1 точка)

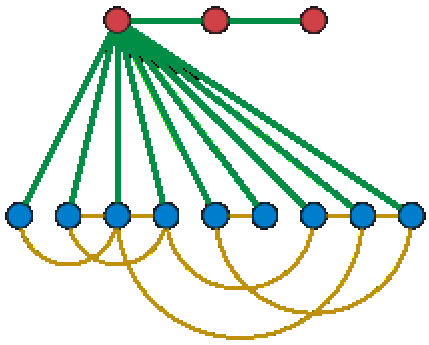
Задача 2. Дадени са тегловен граф G и едно негово минимално покриващо дърво T. (Дървото T е вече построено.) Към графа G се добавя ново ребро, чиито краища са измежду съществуващите върхове на G, тоест не се добавят върхове. Съставете възможно най-бърз алгоритъм, който да построява минимално покриващо дърво на новия граф. Такъв алгоритъм не е бил изучаван на лекции; очаква се да го съставите "от нулата" — без да разчитате на готови алгоритми.

а) Опишете алгоритъма с думи. **(1 точка)**

б) Демонстрирайте алгоритъма с пример. Демонстрацията трябва да бъде пълна — от входните данни до отговора на алгоритъма. **(1 точка)**

в) Докажете коректността на предложения алгоритъм. **(2 точки)**

Задача 3. Граф с $n \geq 5$ върха се нарича скорпион, ако притежава връх от първа степен (жилото), който е свързан с връх от втора степен (опашката), който е свързан с връх от степен $n-2$ (тялото), който е свързан с останалите $n-3$ върха (краката). Възможно е някои крака да са свързани помежду си.



Дадена е симетрична двоична матрица $n \times n$ със стойности *false* върху главния диагонал, която тълкуваме като матрица на съседствата на неориентиран граф с n върха (без примки).

Предложете алгоритъм, който за време $O(n)$ в най-лошия случай разпознава дали графът, зададен от матрицата, е скорпион. Обърнете внимание, че дължината на входа е $\Theta(n^2)$, тоест времевата сложност $O(n)$ е сублинейна и алгоритъмът не разполага с достатъчно време за прочитането на всички входни данни. Следователно алгоритъмът не може да провери дали дадената матрица е симетрична: тази проверка изисква време $\Omega(n^2)$. С други думи, алгоритъмът трябва да приеме наготово (без проверка), че дадената матрица е симетрична. Алгоритъмът може да приеме наготово и това, че матрицата съдържа само стойности *false* върху главния диагонал. Накратко, алгоритъмът приема без проверка, че дадената матрица е коректно представяне на неориентиран граф без примки. Единственото, което алгоритъмът трябва да провери, е дали графът представлява скорпион.

Опишете алгоритъма на програмния език Си във вид на функция със следния формат: `bool IsScorpion(Graph G)`. Типът `Graph` да бъде деклариран в програмата така: `typedef bool Graph[n][n]`, а идентификаторът `n` да бъде име на константа.

Ако в използваната версия на езика Си липсва тип `bool`, вместо него може да се ползва типът `int`, вместо константата *false* — числото 0, а вместо *true* — числото 1.

Програмата може да използва само типовете `int`, `bool` и масив. Да не се използват никакви готови функции (библиотеки).

(3 точки)

РЕШЕНИЯ

Задача 1. Разглеждаме ориентиран граф, чиито върхове съответстват на старите плочки. Графът съдържа ребро от върха x към върха y точно когато плочките x и y са съседни и $y = 2x$ (с имената на плочките означаваме и числата върху тях). Графът съдържа и един допълнителен връх s, съответстващ на новата плочка. От върха s излизат ребра към онези стари плочки, чиито числа са равни на числото на новата плочка и които имат свободна съседна клетка. Във върха s не влизат ребра.

Числото, което се получава в резултат на верижна реакция, е толкова по-голямо, колкото по-дълга е реакцията. Най-дълга верижна реакция съответства на ***най-дълъг път*** с начало s в построения граф. В общия случай (за произволен граф) задачата за намиране на най-дълъг път е NP-пълна. Обаче разглежданият вариант на играта "2048" поражда ***ориентиран ацикличен граф,*** защото при движение по ребрата на графа числата растат (с изключение на първата стъпка).

Най-дълъг път от даден връх в ориентиран ацикличен граф се търси бързо с помощта на ***динамично оптимиране.*** Времевата сложност на това решение при най-лоши входни данни е линейна спрямо размерите на графа, тоест тя е оптимална по порядък.

Но можем да намалим константния множител. Динамичното оптимиране обхожда графа два пъти — топологично сортиране и същинско търсене на най-дълъг път. Първото обхождане отпада, ако построим графа сортиран: слой № k съдържа старите плочки, надписани с 2^k, а новата плочка (s) е сама в слой (№ 0). Това решение е около два пъти по-бързо и носи 4 т.

Ненужно е и сравняването на дължините: всички пътища от s до произволен връх x имат равни дължини: $\log_2 x - \log_2 s + 1$; това води до решение ***без динамично оптимиране:*** обхождаме графа от върха s (няма значение как — в ширина или в дълбочина) и търсим плочката с най-голямо число, достижима от върха s. Тя е краят на най-дълъг път с начало s, който намираме от нея към s по указателите към родителите в дървото на обхождането. Това решение носи 6 т., но обосновката е задължителна: по принцип обхождането в ширина намира най-къс, а не най-дълъг път, а пък обхождането в дълбочина не прави никое от двете.

Задача 2. Алгоритъм за актуализиране на минимално покриващо дърво T след добавяне на ребро e към първоначалния граф G:

1) Добавяме реброто e към дървото T.
2) Тъй като дървото е покриващо, то добавеното ребро затваря цикъл в T. Намираме цикъла, като проследяваме пътищата от двата края на e до корена на дървото. (Предполагаме, че дървото T е зададено, като всеки връх без корена пази указател към своя родител.) Пътищата намираме от листата към корена, а ги сравняваме от корена към листата. При сравнението махаме еднаквите върхове без последния. Другите образуват цикъла.
3) Обхождаме цикъла и намираме най-тежкото ребро f в него. (То може да съвпада с e.)
4) Премахваме реброто f от дървото T. Полученото дърво $T_2 = T + e - f$ е минимално покриващо дърво за новия граф $G_2 = G + e$.

Доказателство за коректност на алгоритъма: От избора на реброто f е ясно, че $w(f) \geq w(e)$, където w означава тегло. За простота да разгледаме само случая, когато теглата на ребрата в графа G_2 са две по две различни. Тогава G_2 има единствено минимално покриващо дърво. Същото важи и за G, т.е. T е дървото, получено от алгоритъма на Крускал, пуснат върху G.

Сега да пуснем алгоритъма на Крускал върху графа G_2. Тъй като теглата на ребрата са две по две различни, то $w(f) > w(e)$, ако $f \neq e$. Останалите ребра се разделят на три вида:
— леки: с тегла, по-малки от $w(e)$;
— средни: с тегла между $w(e)$ и $w(f)$;
— тежки: с тегла, по-големи от $w(f)$.

Алгоритъмът на Крускал разглежда ребрата по нарастващ ред на теглата им. Тоест алгоритъмът разглежда най-напред ребрата от първия вид. Те са едни и същи в G и G_2 (двата графа се различават само по реброто e). Затова в минималното покриващо дърво на G_2 алгоритъмът ще вземе онези леки ребра, които е взел в дървото T. С други думи, до момента, когато алгоритъмът на Крускал достигне до реброто e, неговите действия върху графа G_2 са същите като действията му върху G: построената до този миг част от новото минимално покриващо дърво се състои само от ребра от дървото T.

След това алгоритъмът на Крускал разглежда реброто e. Възможни са два случая.

Първи случай: Реброто e с помощта на леките ребра, взети от алгоритъма на Крускал, затваря някакъв цикъл C. Тъй като взетите леки ребра са част от дървото T, то същият цикъл ще бъде затворен, ако добавим реброто e към дървото T. При добавяне на ребро към дърво се затваря най-много един цикъл. (Ако допуснем, че се затварят два или повече цикъла, то частите им, несъдържащи добавеното ребро, са образували цикъл в дървото още преди добавянето на реброто, а това е невъзможно: по определение дървото не съдържа цикли.) Ето защо цикълът, за който се говори в стъпка № 2 от нашия алгоритъм, съвпада с цикъла C. Тъй като цикълът C се състои само от леки ребра плюс реброто e, то реброто e е най-тежкото ребро в цикъла, тоест то съвпада с реброто f от стъпка № 3 от нашия алгоритъм. Следователно $T_2 = T + e - f = T$, тоест дървото T_2, построено от нашия алгоритъм, съвпада с дървото T (минималното покриващо дърво на G). За да бъде коректен нашият алгоритъм, остава да докажем, че T е минимално покриващо дърво и на графа G_2.

Действително, щом реброто e заедно с взетите по-леки ребра затваря цикъл, то следва, че алгоритъмът на Крускал отхвърля реброто e. Изпълнението на алгоритъма продължава с тежките ребра (средни ребра няма, тъй като f съвпада с e). Понеже реброто e е отхвърлено, изпълнението на алгоритъма продължава така, сякаш това ребро не е било част от графа. Тоест алгоритъмът на Крускал се изпълнява върху G_2 по същия начин, както се изпълнява върху G. Затова и крайният резултат е същият: получава се същото дърво T. С други думи, T е минимално покриващо дърво на G_2, което трябваше да се докаже.

Втори случай: Реброто e, добавено към леките ребра, взети от алгоритъма на Крускал, не затваря цикъл. Затова алгоритъмът на Крускал взима реброто e и продължава работа върху средните ребра. В този случай цикълът от стъпка № 2 от нашия алгоритъм съдържа поне едно средно или тежко ребро, следователно $f \neq e$ и $w(f) > w(e)$.

Твърдим, че от средните ребра алгоритъмът на Крускал, пуснат върху G_2, ще вземе същите, които е взел, когато е бил пуснат върху G. Действително, теглата са същите, така че разлика може да се получи само от затварянето на някакъв цикъл. Щом $G_2 = G + e$, няма как при изпълнението на алгоритъма на Крускал върху G да се затвори цикъл, който да не бъде затворен при изпълнението на алгоритъма на Крускал върху G_2. Възможно е обратното: при изпълнението на алгоритъма на Крускал върху G_2 да се появи ребро, затварящо цикъл, който не е бил затворен от това ребро при изпълнението на алгоритъма на Крускал върху G. Следователно този цикъл съдържа новото ребро e. Той се състои само от ребра, по-леки от f, тоест не е цикълът от стъпка № 2 на нашия алгоритъм. От друга страна, всичките му ребра (без e) са били взети (добавени към дървото T) от алгоритъма на Крускал, когато той е бил изпълняван върху G. Ето защо добавянето на реброто e към дървото T затваря два цикъла — цикъла, забелязан от алгоритъма на Крускал при изпълнението му върху графа G_2, и цикъла от стъпка № 2 на нашия алгоритъм. Това противоречи на доказаното по-горе: че добавянето на ребро към дърво затваря най-много един цикъл. Полученото противоречие доказва твърдението от началото на абзаца — че от средните ребра алгоритъмът на Крускал, пуснат върху G_2, ще вземе същите, които е взел, когато е бил пуснат върху G.

Сега идва ред на реброто f. От неговия избор (стъпка № 3 от нашия алгоритъм) следва, че има цикъл, образуван от e, f и ребра на T, по-леки от f (т.е. леки и средни ребра). Значи, реброто f затваря цикъл, затова ще бъде отхвърлено от алгоритъма на Крускал.

Досега алгоритъмът на Крускал, пуснат върху G_2, е построил покриваща гора, която се различава от покриващата гора на G, построена до същия етап при изпълнението на алгоритъма на Крускал върху G, само по това, че е взето реброто e вместо реброто f. А щом добавянето на f затваря цикъл, съдържащ e, то двата края на f са от едно и съща компонента на гората на G_2 (както са от една и съща компонента на гората на G). Накратко, двете изпълнения на алгоритъма на Крускал (върху G и върху G_2) до този миг са породили едно и също разпределение на върховете по компоненти на свързаност.

По-нататък алгоритъмът на Крускал обработва тежките ребра. Едно ребро се приема от алгоритъма на Крускал само ако краищата му са от различни компоненти на гората. Тъй като разпределението на върховете на графа по компоненти е едно и също в G_2 и в G, то алгоритъмът на Крускал при работа върху G_2 ще избере същите тежки ребра, които е избрал при обработката на G. Следователно минималното покриващо дърво на графа G_2, което алгоритъмът на Крускал ще върне, ще е същото като минималното покриващо дърво, върнато от алгоритъма на Крускал при изпълнението му върху графа G, само че ще съдържа реброто e вместо f. С други думи, при изпълнението си върху G_2 алгоритъмът на Крускал ще върне дървото $T + e - f = T_2$.

Тъй като алгоритъмът на Крускал е коректен, то T_2 е минимално покриващо дърво на G_2.

Дотук предполагахме, че между теглата на ребрата няма равни (затова минималното покриващо дърво беше единствено). Сега ще се освободим от това ограничение.

Ако между теглата на ребрата има равни, то графът може да има повече от едно минимално покриващо дърво. Въпреки това можем да смятаме, че дървото T е получено от алгоритъма на Крускал при работа върху G, защото това е свойство на този алгоритъм: всяко минимално покриващо дърво може да се получи от алгоритъма на Крускал, стига да зададем подходящи приоритети (ред на разглеждане) на ребрата с равни тегла.

За доказателството е важно сходството в обработката на графите G_2 и G. Запазваме това сходство, като зададем приоритетите в G и G_2 по еднакъв начин.

За леки ще смятаме ребрата, чиито тегла не надхвърлят $w(e)$, вкл. ребрата с тегло $w(e)$. Важно е реброто e да може да се смята за най-тежко в цикъла C и да се обработва от алгоритъма на Крускал след леките ребра. За целта даваме на e подходящ приоритет.

Задача 3. Следният алгоритъм разпознава за време $O(n)$ дали е скорпион граф с n върха, описан чрез матрица на съседствата. Времевата сложност се получава така: всички цикли са по брояч от 0 до $n - 1$ и няма вложени цикли — нито пряко, нито косвено (рекурсия).

```
void swap(int * x, int * y) {
  int z = * x;
  * x = * y;
  * y = z;
}

#define n 8
// n must be at least 5

typedef bool Graph[n][n]; // symmetric adjacency matrix;
// values 'false' assumed on the main diagonal

bool can_be_body[n];
bool can_be_sting[n];

void InitCanBeArrays() {
   for (int k = 0; k < n; k++) {
      can_be_body[k] = true;
      can_be_sting[k] = true;
   }
}
```

```
int Degree(Graph G, int v, int * v11, int * v12, int * v0) {
   int deg = 0;
   for (int k = 0; k < n; k++) {
      if (G[k][v]) {
         deg++;
         can_be_sting[k] = false; // correct <=> v is a foot
         switch (deg) {
            case 1: * v11 = k; break;
            case 2: * v12 = k; break;
         }
      }
      else if (k != v) {
         can_be_body[k] = false; // correct <=> v is a foot
         * v0 = k;
      }
   }
   return deg;
}

#define TheOtherVertex(v, v1, v2)  (((v) == (v2)) ? (v1) : (v2))

int FindNextCandidate(bool * arr, int index) {
   if (index < -1)
      index = -1;
   do
      index++;
   while (index < n && !arr[index]);
   return (index < n) ? index : -1;
}

int FindSting(Graph G) {
  int b = FindNextCandidate(can_be_body, -1);
  int s = FindNextCandidate(can_be_sting, -1);
  if (b == -1 || s == -1)
    return -1; // no sting or no body
  do {
     if (G[b][s]) {
        s = FindNextCandidate(can_be_sting, s); // sting <> s
        if (s == -1)
          return -1;
     }
     else
        b = FindNextCandidate(can_be_body, b);
        // body <> b unless sting = s
  } while (b != -1);
  // the body is omitted, but the sting is not
  return s; // sting = s unless the graph is not a scorpion
}
```

```
bool IsScorpion(Graph G) {
   int v, v11, v12, v0, body, sting;
   InitCanBeArrays();
   v = 1;
   int d = Degree(G, v, &v11, &v12, &v0);
   if (d == n-2) {
      body = v; sting = v0;
      if (Degree(G, sting, &v11, &v12, &v0) != 1) return false;
      return (Degree(G, sting, &v11, &v12, &v0) == 2);
   }
   if (d == 1) {
      sting = v;
      v = v11;
      d = Degree(G, v, &v11, &v12, &v0);
      if (d == n-2) {
         body = v; sting = v0;
         if (Degree(G, sting, &v11, &v12, &v0) != 1) return false;
         return (Degree(G, sting, &v11, &v12, &v0) == 2);
      }
      if (d == 2) {
         body = TheOtherVertex(sting, v11, v12);
         return (Degree(G, body, &v11, &v12, &v0) == n-2);
      }
      return false;
   }
   if (d == 2) {
      body = v11; sting = v12; // not certain yet
      d = Degree(G, sting, &v11, &v12, &v0);
      if (d == 1) {
         // the sting and tail are now certain
         return (Degree(G, body, &v11, &v12, &v0) == n-2);
      }
      if (d == n-2) {
         swap(&sting, &body);
         // the body and tail are now certain
         return (Degree(G, sting, &v11, &v12, &v0) == 1);
      }
   }
   // v is a foot => can_be_... arrays contain correct information
   sting = FindSting(G);
   // can_be_... arrays are not necessary any more
   if (sting == -1) return false;
   d = Degree(G, sting, &v11, &v12, &v0);
   if (d != 1) return false;
   v = v11; // v = tail
   d = Degree(G, v, &v11, &v12, &v0);
   if (d != 2) return false;
   body = TheOtherVertex(sting, v11, v12);
   d = Degree(G, body, &v11, &v12, &v0);
   return (d == n-2);
}
```