Рекуррентные соотношения

Скачки

Для поиска кратчайшего пути представим трассу в виде графа, где вершина хранит позицию [x,y] (координаты клетки в матрице), скорость, с которой в эту клетку можно перейти, время движения с данной скоростью на данном этапе и номер хода.

```
struct Vertex {
	int y;
	int x;
	int v = 0;
	int k = 0;
	int step = 0;
};
```

При заполнении двумерного массива значениями трассы запоминаем координаты стартовой клетки, с которой впоследствии начнем обход в ширину (если бы можно было делать сразу несколько ходов, то использовали бы алгоритм Дейкстры).

```
for (int i = 0; i < m; i++) {
		for (int j = 0; j < n; j++) {
			int a;
			fin >> a;
			trace[i][j] = a;
			if (a == 4) {
				xStart = j;
				yStart = i;
			}
		}
	}
	fin >> maxV >> maxK;
```

В четырехмерном векторе хранится информация о каждой вершине:
0-вершина посещена, 500 - не посещена.

```
dp.resize(m+1, vector<vector<vector<int>>>(n+1, vector<vector<int>>(maxV+1, vector<int>(maxK+1, 500))));
```

Обход в ширину, как уже говорилось ранее, начинаем с вершины s с параметрами (y=yStart, x=xStart, v=0, k=0, step=0).

Затем по обычному принципу bfs, пока очередь не пуста, выполняем операции поиска вершин, в которые можно перейти с учетом параметров. Алгоритм перехода такой:

- если скорость меньше максимальной, то мы можем либо увеличить скорость, либо уменьшить, либо не менять;
- если скорость максимально возможная, то в зависимости от времени ее поддержания выполняем те же операции, что и в предыдущем пункте (только увеличить скорость нельзя).

```
q.push(s);
      while (!q.empty()) {
            int x = q.front().x;
            int y = q.front().y;
            int v = q.front().v;
            int k = q.front().k;
            int step = q.front().step;
            q.pop();

                  if (trace[y][x] == 3) {
                        vec.push_back(step);
                        continue;
                  }
                  if (v < maxV) {
                        if (v != 0) {
                              ways(y, x, v - 1, 0, step);
                        }
                        ways(y, x, v, 0, step);
                        if (v + 1 == maxV) {
                              ways(y, x, v + 1, 1, step);
                        } else {
                              ways(y, x, v + 1, 0, step);
                        }
                        continue;

                  }
                  if (v == maxV) {
                        if (k < maxK) {
                              ways(y, x, v, k + 1, step);
                              ways(y, x, v - 1, 0, step);
                        }
                        if (k == maxK) {
                              ways(y, x, v - 1, 0, step);
                        }
                  }
      }
```

Сама функция поиска вершин представляет собой проход по значениям матрицы, удаленным от текущей клетки на расстояние v1=|x1-x|+|y1-y|. Таким образом, если клетка равна 0, 3, 4 (в клетку старта также можно вернуться), а вершина еще не была посещена, то добавляем ее в очередь и обнуляем соответствующее ей значение четырехмерного массива.

```
//right up
for (int i = 0; i <= v1; i++) {
	y1 = y - v1 + i;
	x1 = x + i;
	if (isMatrix(y1, x1)) {
		if (trace[y1][x1] == 0 || trace[y1][x1] == 3
||trace[y1][x1]==4) {
			if (dp[y1][x1][v1][k] != 0) {
				Vertex vertex = { y1, x1 , v1, k, step + 1 };
				dp[y1][x1][v1][k] = 0;
				q.push(vertex);
			}
		}
	}

}
```

Обязательным условием является принадлежность клетки матрице. Эту проверку осуществляет отдельная функция.

```
bool isMatrix(int y, int x) {
	return (x < n && y < m && x >= 0 && y >= 0);
}
```

Таким образом, мы ищем длины путей до финишных клеток, а затем среди них выбираем минимальную.
Поиск в ширину работает за “количество вершин + количество ребер”. Количество вершин - количество состояний ДП, то есть количество четверок (x, y, v, k), значит, равно $n \cdot m \cdot maxV \cdot maxK$. Количество ребер равно количеству переходов из каждого состояния. Так как мы перебираем 3 варианта для скорости (увеличить/уменьшить/не менять) и для каждого варианта 4 направления (right up/left up/left down/right down), то в общей сложности количество переходов может быть порядка $12 \cdot maxV$.
В худшем случае алгоритм работает примерно за $O(n \cdot m \cdot maxV^2 \cdot maxK)$.

Алгоритмы на графах

Проезд с посещением города С

В условии задачи сказано, что можно проехать из города i в город j туда и обратно, следовательно, граф неориентированный. Однако чтобы проверить, получится ли проехать из A в B через C, нужно построить по графу сеть и вычислить максимальный поток.

Отступление. Мой предыдущий алгоритм основывался на поиске в глубину путей от C до A и B, однако на ряде тестов он давал сбой, так как отметка о посещении вершины зависела от порядка занесения ребра в граф (одна из причин неправильной работы).

Для поиска потока используем алгоритм Диница.
На первой фазе алгоритма построим остаточную сеть: вместо одного ребра добавим две дуги, одной назначим пропускную способность 1, другой - 0.
Вершины раздвоим, введем новую вершину s, соединим ее с городами A и B, а в качестве t возьмем C.

> Остаточное ребро можно интуитивно понимать как меру того, насколько ещё можно увеличить поток вдоль какого-то ребра. В самом деле, если по ребру (u, v) с пропускной способностью c_{uv} протекает поток f_{uv}, то потенциально по нему можно пропустить ещё $c_{uv} - f_{uv}$ единиц потока, а в обратную сторону можно пропустить до f_{uv} единиц потока, что будет означать отмену потока в первоначальном направлении.

Хранить граф будем в виде списка смежности. Для ребер создадим специальную структуру, в которой будет храниться из какой вершины в какую отправлена дуга, пропускная способность,текущее значение потока и индекс.

```
struct Edge {
      int from;
      int where;
      int capacity;
      int flow;
      int inverseIndex;
      Edge(int _from, int _where, int _capacity, int _flow, int
_inverseIndex) :
            from(_from), where(_where), capacity(_capacity), flow(_flow),
```

```
inverseIndex(_inverseIndex) {};
};

void addEdge(int u, int v, int capacity = 1) {
      edge[u].emplace_back(u, v, capacity, 0, edge[v].size());
      edge[v].emplace_back(v, u, 0, 0, edge[u].size() - 1);
}
```

На следующем этапе построим “слоистую сеть” с помощью bfs: сначала определяем длины кратчайших путей из s до всех остальных вершин и заносим значения в массив, а затем вносим в очередь все ребра, которые не принадлежат целиком одному уровню, а также те, что ведут назад, к предыдущему уровню.

```
int bfs(int s, int t) {
      for (int i = 0; i < level.size(); ++i) {
            level[i] = MAX;
      }
      level[s] = 0;
      queue<int> q;
      q.push(s);
      while (!q.empty()) {
            int v = q.front();
            q.pop();
            for (auto& to : edge[v]) {
                  if (level[to.where] == MAX && to.flow < to.capacity) {
                        level[to.where] = 1 + level[v];
                        q.push(to.where);
                  }
            }
      }
      return level[t] != MAX;
}
```

Следующая фаза - поиск блокирующего потока.

Блокирующим потоком в данной сети называется такой поток, что любой путь из истока s в сток t содержит насыщенное этим потоком ребро. Иными словами, в данной сети не найдётся такого пути из истока в сток, вдоль которого можно беспрепятственно увеличить поток.

Блокирующий поток не обязательно максимален. Теорема Форда-Фалкерсона говорит о том, что поток будет максимальным тогда и только тогда, когда в остаточной сети не найдётся $s-t$ пути; в блокирующем же потоке ничего не утверждается о существовании пути по рёбрам, появляющимся в остаточной сети.

Поиск организуем за счет dfs. Пока находятся пути s-t, продолжаем поиск. Насыщаем ребра по пути.

```
int dfs(int v, int t, int currFlow) {
	if (v == t) {
		return currFlow;
	}
	if (currFlow == 0) {
		return currFlow;
	}

	for (; isVisited[v] < edge[v].size(); ++isVisited[v]) {
		auto& currEdge = edge[v][isVisited[v]];
		int to = currEdge.where;
		if (level[to] == 1 + level[v] && currEdge.flow <
currEdge.capacity) {
			int mainFlow = min(currFlow, currEdge.capacity -
currEdge.flow);
			int returnedFlow = dfs(to, t, mainFlow);
			if (returnedFlow) {
				currEdge.flow += returnedFlow;
				edge[to][currEdge.inverseIndex].flow -=
returnedFlow;
				return returnedFlow;
			}
		}
	}
	return 0;
}
```

Последние две фазы алгоритма Диница, в которой проверяем, достижима ли t из s.

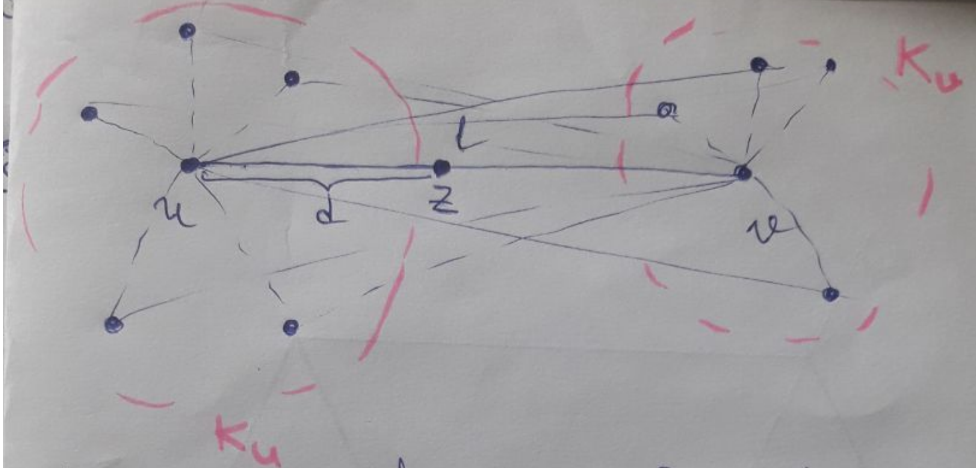
```
while (bfs(source, trash)) {
		for (int i = 0; i < isVisited.size(); ++i) {
			isVisited[i] = 0;
		}

		while (true) {
			int newFlow = dfs(source, trash, MAX);
			flow += newFlow;
			if (!newFlow) {
				break;
			}
		}
}
```

Алгоритм Диница выполняется за $O(Fm)$, где F – величина потока. Так как поток равен 2, то асимптотика сводится к $O(m)$.

Встреча

Из условия ясно, что на входе мы получаем взвешенный граф. Необходимо доказать, что точка встречи во всех случаях (кроме того, когда вершины только две) может находится в домике.



K_u – множество домиков, которые ближе к u.
K_v – множество домиков, кот. ближе к v.
z – точка встречи (предполагаемая).
l – расстояние между u и v.
d – расстояние от u до z.

(I) $K_v > K_u$

$\rho(z, d(k))$ – расстояние от всех домиков до z.

Пусть p – расстояние, на кот перенесём z по напр. к v, $p < (l-d)$. $< \rho(z, d(k))$

$\rho(z, d(k)+p) = \rho(z, d(k)) + K_u p - K_v p = \rho(z, d(k)) + (K_u - K_v)p$

(II) Сдвигаем z к вершине u.

$\rho(z, d(k)-p) = \rho(z, d(k) - (K_v p - K_u p)$ – аналогично.

(III) $\rho(z, d(k)) = \rho(z, d(k)+p)$, если $K_u = K_v$.

Таким образом, задача сводится к поиску кратчайших расстояний между всеми вершинами графа. Для этого реализуем алгоритм Флойда-Уоршелла.

Хранить граф будем в матрице смежности, где ребру i-j в позиции [i][j] соответствует его вес. Диагональные элементы матрицы должны быть равны нулю. Все остальные клетки, в которые не были занесены веса, заполняются максимальным значением.

```
for (int i = 0; i < m; i++) {
            int u, v, w;
            fin >> u >> v >> w;
            g[u - 1][v - 1] = w;
            g[v - 1][u - 1] = w;

            if (m == 1) {
                  fout << u << " " << v << " " << w / 2;
                  return 0;
            }
}
```

Далее непосредственно сам алгоритм Флойда-Уоршелла

Существует два варианта значения d_{ij}^k, $k \in (1, \ldots, n)$:

1. Кратчайший путь между i, j не проходит через вершину k, тогда $d_{ij}^k = d_{ij}^{k-1}$
2. Существует более короткий путь между i, j, проходящий через k, тогда он сначала идёт от i до k, а потом от k до j. В этом случае, очевидно, $d_{ij}^k = d_{ik}^{k-1} + d_{kj}^{k-1}$

Таким образом, для нахождения значения функции достаточно выбрать минимум из двух обозначенных значений.

Тогда рекуррентная формула для d_{ij}^k имеет вид:

d_{ij}^0 — длина ребра (i, j);

$d_{ij}^k = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$.

```
for (int k = 0; k < n; ++k) {
            for (int i = 0; i < n; ++i) {
                  for (int j = 0; j < n; ++j) {
                        g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
                  }
            }
}
```

Таким образом, чтобы найти домик, в котором состоится встреча, нужно пройтись по матрице еще раз, суммируя для каждой вершины расстояния от нее до всех остальных. Минимальное будет ответом.

```
int house = 0;
      int min = MAX;
      for (int i = 0; i < n; i++) {
            distance[i] = accumulate(g[i].begin(), g[i].end(), 0);
            if (distance[i] < min) {
                  min = distance[i];
                  house = i + 1;
            }
      }
```

Алгоритм работает за $O(n^3)$, так как реализуется с помощью трех вложенных циклов.

Приближенные алгоритмы

Всемирная выставка

Очевидно, что данная задача о минимальном вершинном покрытии графа. Для решения этой задачи можно использовать жадный алгоритм или же простой.

Жадный алгоритм.

1. $S := \{\emptyset\}$
2. Выбираем вершину с максимальной степенью
3. Добавляем в решение S эту вершину
4. Удаляем из графа все ребра, инцидентные выбранной вершине
5. Если остались ребра, возвращаемся к шагу 2

Однако к данному алгоритму можно привести контрпример с двудольным графом.

Рассматриваем двудольный граф на долях U и V

$|U| = n!$

$V = V_1 \cup V_2 \cup \dots \cup V_n$, $|V_i| = \frac{n!}{i}$

Каждая вершина $u \in U$ имеет ровно одного соседа в группе V_i

$\forall v \in V_i \;\; deg(v) = i$

В таком случае жадный алгоритм может последовательно выбрать сначала все вершины из V_n, затем из V_{n-1} и так далее. Следовательно, полученное покрытие будет равно:

$$n!(\tfrac{1}{n} + \tfrac{1}{n-1} + \ldots + 1)$$

Для жадного алгоритма можно вывести следующую оценку.

Для любой константы C найдется граф такой, что алгоритм выдаст покрытие, которое по размеру будет в C раз хуже оптимального.

Рассмотрим простой алгоритм.

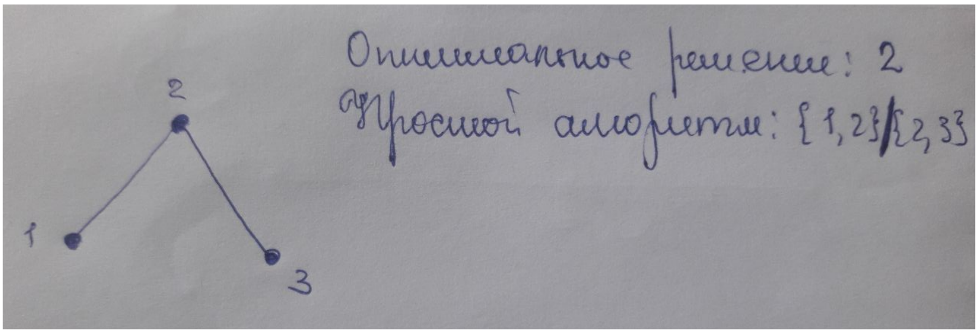
1. $S := \{\emptyset\}$
2. Выбираем случайное ребро графа
3. Добавляем в решение S оба конца ребра
4. Удаляем из графа все ребра, инцидентные концам ребра
5. Если остались ребра, возвращаемся к шагу 2

Так как требуется выбрать случайное ребро, то нам не обязательно хранить граф, можно генерировать ответ при считывании значений из файла. А вместо удаления инцидентных ребер поставим флаг, что вершина уже есть в ответе.

```
set<int>answer;
vector<bool>used(n, false);
for (int i = 0; i < m; i++) {
	int u, v;
	fin >> u >> v;
	if (!used[u-1] && !used[v-1]) {
		answer.insert(u);
		answer.insert(v);
		used[u-1] = true;
		used[v-1] = true;
	}
}
fout << answer.size() << endl;
for (auto it = answer.begin(); it != answer.end(); it++) {
	auto i = it;
	i++;
	if (i ==answer.end()) {
		fout << *it;
	}
	else {
		fout << *it << " ";
	}
}
```

Так как мы заносим в ответ неповторяющиеся вершины, то есть каждые две вершины соответствуют одному ребру, каждое из которых не имеет общих вершин с любым другим, то можно сделать вывод, что решение задачи с помощью простого алгоритма не превзойдет оптимальное более чем в 2 раза.

Нагляднее можно убедиться на следующем тривиальном примере.



Так как считываем m ребер и в ответе может быть максимум n вершин, а в set вершины добавляются за $O(\log n)$, то в худшем случае решение выдаст асимптотику $O(n \cdot \log n + m)$.