Лекция по алгоритмам #12
Тема: BFS, Dijkstra

25 ноября

Собрано 27 декабря 2014 г. в 22:42

---

1. Обозначения

n – количество вершин в графе
m – количество ребер в графе

2. Поиск в ширину

Задача найти кратчайший путь на графе с целыми весами из $\{0, 1\}$.

Решение #1. Индукция. База. Запускаем из стартовой вершины dfs/bfs по нулевым ребрам, получаем множество вершин A_0, до которых расстояние 0. Переход. Делаем из всех $v \in A_d$ переход по единичным ребрам. Получаем B_{d+1} – ранее не посещенные вершины, до них расстояние $d+1$. Из B_{d+1} запускаем bfs/dfs по нулевым ребрам, получаем A_{d+1} – все вершины, до которых расстояние $d+1$.

Корректность: очевидно. Сложность: $\mathcal{O}(n+m)$ времени, $\mathcal{O}(n)$ памяти.

Решение #2. Тоже самое, но реализация проще. Дек. Берем вершину v из начала, делаем из нее переход по всем ребрам, если вес ребра 0, добавляем в начало, иначе в конец.

```
void bfs (int s){
  //s --- стартовая вершина
  d.assign (n, INF); // вектор расстояний
  p.assign (n, -1);  // вектор предков

  d[s] = 0;
  deque<int> q;
  q.push_front (s);
  while (!q.empty ()) {
    int v, u, len;
    v = q.front ();
    q.pop_front ();
    for (int i = 0; i < g[v].size (); ++i) {
      //g[v] --- список смежности вершины v
      u = g[v][i].first; // вершина в которую ведет ребро
      len = g[v][i].second; // 0 или 1, в зависимости от веса ребра
      if (d[u] == INF) {
        d[u] = d[v] + (len);
```

```
        p[u] = v;
        if ((len) == 0)
          q.push_front (u);
        else
          q.push_back (u);
      }
    }
  }
}
```

Корректность: замечаем, что каждую вершину положим в дек не более двух раз – сперва один раз в конец, затем один раз в начало. Сложность: $\mathcal{O}(n+m)$ времени, $\mathcal{O}(n)$ памяти.

Задача найти кратчайший путь на графе с целыми весами из $\{1, 2, 3, \dots, k\}$.

Решение #1. Скажем, что ребро длины l это на самом деле l ребер длины 1. Получили новый граф из $\mathcal{O}(n+km)$ вершин и $\mathcal{O}(km)$ ребер. Сложность bfs-а на новом графе – $\mathcal{O}(n+km)$ времени и $\mathcal{O}(n+km)$ памяти.

Решение #2. Предложено Dial'1969. Заводим $(n-1)k+1$ очередь. q_i – вершины, до которых текущее оптимальное расстояние i. $(n-1)k$ – максимальное расстояние от начальной вершины до любой. Перебираем i в порядке возрастания, обрабатываем все вершины из q_i, когда делаем переход по ребру веса w, пытаемся улучшить расстояние до вершины и добавить ее в очередь q_{i+w}. Из той очереди, где вершина лежала до этого её по-хорошему нужно было бы удалить. Но мы не будем, сделаем иначе: когда достаем вершину из очереди, проверяем что расстояние такое, как мы думаем. Если нет, вершину нужно было удалить, поэтому мы её пропускаем (если все-таки очень хочется удалять, то можно хранить двусвзяные списки и удалять каждый раз, это уменьшит количество используемой памяти). Заметим, что после того, как все вершины из $q_0, q_1, \dots, q_i$ обработаны, непустыми будут очереди только $q_{i+1}, q_{i+2}, \dots, q_{i+k}$. Поэтому для экономии памяти можно хранить не все $\Theta(nk)$, а только k очередей.

Сложность: каждая вершина обрабатывается ровно один раз, каждое ребро порождает не более чем одно добавление в очередь, количество просмотренных очередей $\mathcal{O}(nk)$. Итого время работы алгоритм $\mathcal{O}(m+nk)$. Если хранить только k очередей и честно удалять вершину из старой очереди при перемещении в новую очередь, используемая дополнительная память будет $\mathcal{O}(n+k)$. Реализация алгоритма:

```
vector<pair<int,int>> g[n]; // наш граф, пары (vertex,weight)
int maxD = (n-1)k + 1, d[n];
queue<int> q[maxD];
q[0].push(s);
fill(d, d + n, INT_MAX);
for (int D = 0; D < maxD; D++)
  while (q[D].size()) {
    int v = q[D].top(); q[first].pop();
    if (d[v] != D) continue; // уже должны были удалить
    for (auto p : c[v])
      if (d[p.first] > D + p.second) {
```

```
        d[p.first] = D + p.second;
        q[D + p.second].push(p.first);
      }
  }
```

Оптимизация алгоритма

Заметим, что время работы алгоритма линейно, если мы умеем быстро выбирать ближайшую непустую очередь. Можно, например, номера всех непустых очередей хранить в куче. Тогда время работы будет $\mathcal{O}((n+m)\log k)$.

Также можно взять $s = \lfloor\sqrt{k}\rfloor$ и для каждого i поддерживать количество непустых очередей на отрезке $[is, (i+1)s)$. Тогда, если при перемещении в новую очередь вершины удаляются, и поэтому операцию "поиск непустой очереди" нужно сделать n раз, суммарное время на обработку всех n поисков непустых очередей будет $\mathcal{O}(n\sqrt{k})$, а суммарное время работы алгоритма $\mathcal{O}(m + n\sqrt{k})$.

Задача найти кратчайший путь на графе с целыми весами из $\{0, 1, 2, 3, \ldots, k\}$.

Объединяем две последние идеи в одну, получаем алгоритм за $\mathcal{O}(m + nk)$ времени.

Задача найти кратчайший путь на графе с вещественными весами из $[1, k]$.

Модифицируем алгоритм для целых весов. Теперь в очереди q_i будет храниться вершины, до которых текущее оптимальное расстояние лежит в $[i, i+1)$. Оценки на время и память алгоритма и всех его модификаций останутся прежними. Очевидно, что такой алгоритм не будет работать для $[0, k]$ графа.

3. Radix Heap

Автор идеи Radix Heap – Ahuja'1990. Близкий аналог с худшей асимптотикой был предложен Johnson'1977. В этом разделе мы реализуем Дейкстру на графе с целочисленными весами от 1 до C за время $\mathcal{O}(m + n\log C)$ (в предыдущем разделе были другие обозначения, но все же заметим, что мы уже умеем решать поставленную задачу за время $\mathcal{O}(m + n\sqrt{C})$). Для этого нам понадобится Radix Heap, который реализует интерфейс Monotone Priority Queue. Monotone Priority Queue: (a) `Insert`, (b) `DecreaseKey`, (c) `ExtractMin` с дополнительным ограничением "минимум, вытаскиваемый из очереди, должен возрастать". В алгоритме Дейкстры как раз нужна структура данных, реализующая Monotone Priority Queue. Время работы операций в Radix Heap:

1. Insert: $\mathcal{O}(\log C)$
2. ExtractMin: амортизированное $\mathcal{O}(\log C)$
3. DecreaseKey: амортизированное $\mathcal{O}(1)$

Radix Heap хранит целочисленные ключи. Здесь C – ограничение сверху на разницу максимального и минимального ключей в куче.

Вернемся к Дейкстре. В каждый момент времени есть вершины трех типов. A – уже обработанные, расстояние до них известно. B – до вершины известно какое-то расстояние, возможно, не оптимальное. Z – до вершины расстояние бесконечно. В нашей структуре данных мы храним вершины из B. Пусть минимальное расстояние до вершин из B равно d, тогда все вершины в B имеют текущее расстояние от d до $d + C$.

Radix Heap хранит вершины в бакетах (кучках, корзинах) $u_0, u_1, ..., u_{\lceil \log C \rceil}$. У каждого бакета u_i есть границы $l_i \le r_i$. Длина отрезка $r_i - l_i \le 2^{\max(i-1,0)} = size_i = |u_i|$. Количество вершин в u_i произвольно. Расстояния до вершин в u_i лежат в $[l_i, r_i)$, $r_i = l_{i+1}$. Изначально все $l_0 = 0, r_i = l_i + size_i$, последнее r равно $+\infty$, в бакете u_0 лежит стартовая вершина s, от которой мы ищем расстояния. В каждом бакете вершины хранятся в списке, чтобы можно было за $\mathcal{O}(1)$ добавлять и удалять.

• Insert(v): найти нужный bucket, положить туда вершину v.

• DecreaseKey(v, x): для вершины v помним номер бакета i_v, в котором она лежит и указатель на элемент списка. Пока $l_{i_v} > d_v$ уменьшаем на один i_v, перекладываем вершину v в бакет с индексом i_v. Для каждой вершины v суммарное время работы всех DecreaseKey(v) – $\mathcal{O}(\log C)$. Суммарное время работы m операций DecreaseKey равно $\mathcal{O}(m + n \log C)$.

• ExtractMin(): ищем непустой бакет u_i с минимальным индексом i. Минимум содержится в данном бакете. Чтобы его найти, нужно перебрать все вершины этого бакета. Если $i \le 1$, это работает за $\mathcal{O}(1)$. Иначе пусть мы нашли вершину v с минимальным расстоянием d_v и удалили её из u_i. $|r_i - l_i| \le 2^i = 1 + 1 + 2 + 4 + ... + 2^{i-1}$. $\forall j < i \colon u_j$ пуст, поэтому скажем, что $l_0 = d_v, l_{j+1} = r_j = d_v + size_j$ и положим каждую из вершин $|u_i|$ в соответствующий бакет. Заметим, что время работы $\mathcal{O}(|u_i| + \log C)$, но для каждой вершины $t \in u_i$ уменьшился i_t (номер бакета, в котором она лежит).

Для того, чтобы получить строгие временные оценки, рассмотрим потенциальную функцию $\Phi = \sum i_t$ (для вершин из Z прибавим $\lceil \log C \rceil$). Начальное значение Φ_0 равно $n\lceil \log C \rceil$, $\Phi \searrow$. Амортизированное время DecreaseKey равно $\mathcal{O}(1)$. Амортизированное время ExtractMin равно $\mathcal{O}(\log C)$. Здесь амортизированное время $a = t + \Delta\Phi$, t – реальное время работы, $\Delta\Phi = \Phi' - \Phi$.

КОНЕЦ

4. Multilevel Radix Heap

Если останется время, можно обсудить реализации Дейкстры за $\mathcal{O}(m + n\frac{\log C}{\log\log C})$ и $\mathcal{O}(m + \sqrt{\log C})$. В любом случае про существование таких оценок стоит упомянуть.

Зафиксируем K, i-й бакет теперь имеет диапазон K^i и разбит слева направо на K частей. Чтобы найти в бакете минимум, нужно сперва за $\mathcal{O}(K)$ выбрать непустую часть с минимальным индексом, а затем перебрать вершины в этой части и перераспределить их на меньших бакеты. Время работы решения $\mathcal{O}(m + n(\log_K C + K))$. Оптимально выбрать $K = \frac{\log C}{\log\log C}$.

У нас получилась полноценная двухуровневая структура. Внутри мы выбираем минимум пока что за линейное время. Чтобы получить оценку $\sqrt{\log C}$, нужно выбрать $K = 2^{\sqrt{\log C}}$ и внутри уровня выбирать непустую из K частей с минимальным индексом за $\mathcal{O}(\log K)$. Для этого можно использовать модификацию кучи Фибоначчи.