RADIX HEAP: ACELERANDO O DIJKSTRA COM PESOS INTEIROS PEQUENOS

Lucas Pavanelli, FGV EMAp

Resumo

O algoritmo de Dijkstra é onipresente em programação competitiva, e geralmente o implementamos com um binary heap (std::priority_queue do C++). Quando os pesos das arestas são inteiros pequenos, porém, é possível fazer melhor. Este tutorial apresenta o radix heap: uma fila de prioridade que tira proveito do fato de que o Dijkstra extrai as distâncias em ordem não-decrescente. Explicamos como ele funciona, como integrá-lo ao Dijkstra e quando ele compensa.

1. O gargalo do Dijkstra

O problema do caminho mínimo de fonte única é frequente em programação competitiva, e o algoritmo de Dijkstra é a ferramenta padrão quando os pesos são não-negativos. A implementação usual emprega um *std::priority_queue* (um *binary heap*), em que cada inserção e cada remoção do mínimo custam $\mathcal{O}(\log n)$, levando ao total conhecido $\mathcal{O}((m + n) \log n)$.

O Dijkstra, porém, tem uma propriedade que esse *heap* não aproveita. Como o algoritmo extrai os vértices em ordem não-decrescente de distância, as chaves removidas da fila nunca diminuem, e uma fila com essa característica é chamada de monótona. Quando os pesos são inteiros e o maior deles é C, o *radix heap* [1] explora essa monotonicidade para trocar o fator $\log n$ por um fator que depende de $\log C$. Para C pequeno, essa troca compensa.

2. Como funciona o radix heap

A ideia é distribuir as chaves em *buckets* conforme a distância delas até o mínimo corrente. Mantemos o mínimo u (a última chave removida, inicialmente 0) e uma sequência de *buckets*. Uma chave $k \geq u$ vai para o *bucket* de índice $b + 1$, onde b é a posição do bit mais significativo em que k difere de u; o *bucket* 0 fica reservado às chaves iguais a u, daí o deslocamento de um. Por exemplo, com $u = 0$, a chave $13 = 1101_2$ difere no bit de posição 3 e cai no bucket 4 (faixa 8–15), ao lado das chaves 8 e 10. Essas são as distâncias dos vértices a, b e c alcançados a partir da fonte s: a Figura 1 mostra o *radix heap* guardando os nós nesse *bucket*, indexados por suas distâncias.

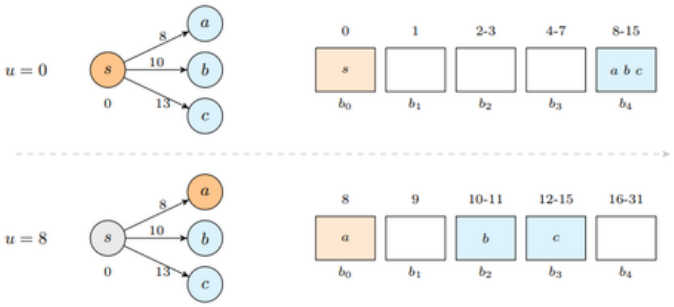

FIGURA 1: DO GRAFO AOS BUCKETS, QUE GUARDAM OS NÓS INDEXADOS PELA DISTÂNCIA. COM $u = 0$ (ACIMA), A FONTE s OCUPA O BUCKET 0 E SEUS VIZINHOS a, b E c (DISTÂNCIAS 8, 10 E 13) CAEM NO BUCKET 4, DE FAIXA 8–15. AO EXTRAIR a COMO NOVO MÍNIMO (ABAIXO, $u = 8$), AS FAIXAS PASSAM A VALER A PARTIR DE 8 E OS NÓS b E c MIGRAM PARA BUCKETS MAIS ESTREITOS.

O ponto central é que os buckets têm larguras diferentes, com o *bucket* i cobrindo uma faixa de 2^{i-1} valores acima de u. Os de índice baixo são estreitos; os de índice alto, largos (linha de cima da Figura 1). Como as distâncias nunca passam de nC, bastam $\mathcal{O}(\log(nC))$ *buckets*: 65 cobrem qualquer valor de 64 bits.

O índice resulta de duas operações: um XOR entre a chave e u, seguido de um *count-leading-zeros* (Código 1). O XOR é o truque essencial: ele isola o bit em que k se afasta de u, fazendo a mesma regra valer para qualquer valor de u.

```
std::size_t bucket_index(Weight key) const {
    Weight diff = key - last_deleted_key;
    if (diff == 0) return 0;
    return most_significant_bit(diff) + 1;
}

// most_significant_bit(v) = 63 - __builtin_clzll(v)
```

CÓDIGO 1: ÍNDICE DE BUCKET NO RADIX HEAP.

Inserir uma chave é $\mathcal{O}(1)$, pois basta calcular seu *bucket* e adicioná-la a ele. A extração do mínimo é onde o *bucket* 0 atua. Enquanto ele tem elementos, todos valem u e são removidos diretamente. Quando esvazia, percorremos os buckets em ordem até o primeiro não-vazio, sua menor chave passa a ser o novo mínimo u', e os elementos desse *bucket* são redistribuídos segundo u'. Como esse *bucket* é largo e u' está dentro de sua faixa, as chaves se espalham por *buckets* mais estreitos, enquanto os seguintes ficam intactos. A linha de baixo da Figura 1 mostra esse passo para $u' = 8$.

A eficiência vem de uma invariante simples: uma chave nunca sobe de *bucket*. Cada chave só desce, e o faz no máximo $\mathcal{O}(\log C)$ vezes ao longo de sua vida. Somando sobre as n chaves, o Dijkstra inteiro custa $\mathcal{O}(m + n \log C)$. O $\log n$ do *binary heap* deu lugar a $\log C$.

3. Refinando: dois níveis

Podemos reduzir ainda mais o número de níveis lendo a chave como dígitos numa base $K = 2^b$ (usamos $b = 4$, $K = 16$) em vez de bit a bit. O primeiro nível indica a posição d do dígito que difere do mínimo; o segundo, entre K *sub-buckets*, indica o valor s desse dígito. Retomando os nós a, b e c da Figura 1: com $u = 0$, suas distâncias 8, 10 e 13 são menores que 16, logo têm $d = 0$, e o nível 2 as separa pelos valores $s = 8$, 10 e 13 (Figura 2); uma distância maior, como $300 = 12C_{16}$, ficaria na posição $d = 2$. Assim cada elemento percorre $\mathcal{O}(\log C / \log\log C)$ posições em vez de $\log C$, e o custo cai para $\mathcal{O}(m + n \log C / \log\log C)$. Nos experimentos essa variante supera a de um nível em quase todos os cenários, ao custo de uma implementação mais complexa.

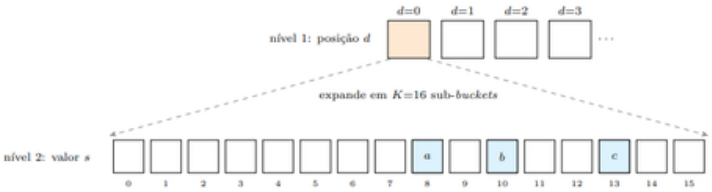

FIGURA 2: RADIX HEAP DE DOIS NÍVEIS (BASE $K = 16$) PARA OS MESMOS NÓS DA FIGURA 1. O NÍVEL 1 AGRUPA AS CHAVES PELA POSIÇÃO d DO DÍGITO QUE DIFERE DE u: COM $u = 0$, AS DISTÂNCIAS 8, 10 E 13 (MENORES QUE 16) TÊM $d = 0$. O NÍVEL 2 ABRE ESSE GRUPO EM K SUB-BUCKETS E SEPARA OS NÓS PELO VALOR s DO DÍGITO, QUE AQUI COINCIDE COM A PRÓPRIA DISTÂNCIA.

4. Como usar

A única exigência do *radix heap* é que as chaves extraídas não decresçam, o que o Dijkstra satisfaz naturalmente, pois a distância d do vértice removido só aumenta. Para adotar o radix heap, basta substituir a *std::priority_queue* por ele e abrir mão do *decrease-key*: em vez de atualizar uma entrada, inserimos uma nova com a chave menor e, ao extrair, descartamos qualquer entrada cuja chave não corresponda mais à distância corrente (*lazy deletion*). O *loop* fica como no Código 2.

```
RadixHeapQueue pq;
dist[s] = 0;
pq.push(0, s);

while (!pq.empty()) {
    auto [d, u] = pq.pop_min();
    if (d != dist[u]) continue;  // entrada obsoleta

    for (auto [v, w] : adj[u]) {
        if (d + w < dist[v]) {
            dist[v] = d + w;
            pq.push(dist[v], v);  // sem decrease-key
        }
    }
}
```

CÓDIGO 2: DIJKSTRA COM RADIX HEAP E LAZY DELETION.

Dois cuidados: os pesos precisam ser inteiros não-negativos, e as distâncias (até nC) devem caber no tipo das chaves (64 bits bastam na prática).

5. Quando vale a pena

O *radix heap* nem sempre é melhor. Medimos o Dijkstra com *binary heap*, *radix* de um nível e *radix* de dois níveis em quatro famílias de grafos, variando C: *grid* (malha 2D esparsa), *sparse* (aleatório com $m \approx 3n$), *dense* (Erdős–Rényi, $m \approx n^2$) e *layered* (DAG em camadas). A Tabela 1 traz o tempo mediano em ms nas colunas *binary*, *radix-1* e *radix-2* (*binary heap* e *radix heaps* de um e dois níveis). Com $C = 10$, um radix vence em todas as famílias, sendo até ~1.7× mais rápido nos grafos esparsos, chegando a ~1.8× com $C = 10^3$. O cruzamento ocorre entre $C = 10^3$ e $C = 10^6$; para pesos grandes, o *std::priority_queue* volta a vencer.

Família	C	binary	radix-1	radix-2
dense (n=2k)	10	1.03	**0.93**	0.96
	1 000	1.30	1.24	**1.08**
	1 000 000	**1.35**	1.98	1.73
grid (n=360k)	10	17.52	14.73	**13.28**
	1 000	21.31	41.48	**21.22**
	1 000 000	**21.43**	59.59	51.41
layered (n=120k)	10	4.41	2.99	**2.82**
	1 000	6.02	8.26	**4.74**
	1 000 000	**5.66**	13.67	12.54
sparse (n=250k)	10	19.47	**11.47**	11.76
	1 000	25.35	15.19	**13.89**
	1 000 000	**26.03**	44.13	31.29

TABELA 1: TEMPO MEDIANO DE EXECUÇÃO (MS) POR FILA DE PRIORIDADE. MELHOR VALOR POR LINHA EM NEGRITO.

Como regra prática, vale trocar pelo *radix* de dois níveis quando os pesos são pequenos ($C \lesssim 10^3$) e o grafo é grande e esparso. Para pesos grandes ($C \geq 10^6$) ou grafos pequenos e densos, prefira o *binary heap*, mais simples e mais rápido.

6. Considerações finais

O *radix heap* é uma substituição direta da *std::priority_queue* que pode acelerar consideravelmente o Dijkstra quando os pesos são inteiros pequenos, cenário comum na competição. A ideia de organizar chaves em *buckets* remonta ao algoritmo de Dial [3], que o *radix heap* refina [1]; o levantamento de Costa et al. [2] cobre estas e outras filas monótonas. A implementação completa em C++, com os geradores de grafos e os experimentos, está em https://github.com/pavalucas/monotone-dijkstra.

Referências

[1] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. Journal of the ACM, 37(2):213–223, 1990.

[2] V. Costa, M. Castro, and R. de Freitas. Exploring monotone priority queues for dijkstra optimization. RAIRO Operations Research, 59, 2025.

[3] Robert B. Dial. Algorithm 360: Shortest-path forest with topological ordering. Communications of the ACM, 12(11):632–633, 1969.