Chapter 13
Shortets paths

Mencari jalur terpendek antara dua node pada sebuah graf adalah masalah penting yang memiliki banyak aplikasi praktis. Sebagai contoh, masalah yang terkait dengan jaringan jalan adalah untuk menghitung panjang rute terpendek antara dua kota, diberikan panjang jalan-jalannya.

Pada graf tak berbobot (*unweighted graph*), panjang sebuah jalur sama dengan jumlah sisi-sisinya, dan kita bisa langsung menggunakan *breadth-first search* untuk mencari jalur terpendek. Namun, pada bab ini, kita fokus pada graf berbobot (*weighted graphs*) di mana algoritma yang lebih canggih diperlukan untuk mencari jalur terpendek.

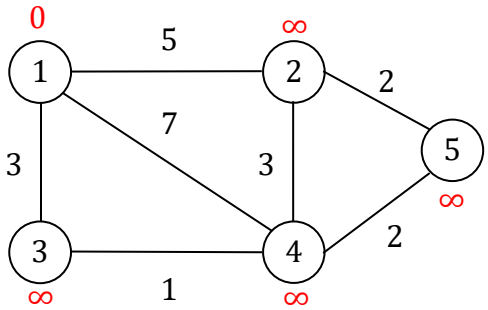
Algoritma Bellman–Ford

Algoritma Bellman–Ford menemukan jalur terpendek dari sebuah node awal ke semua node dalam graf. Algoritma ini dapat memproses segala jenis graf, asalkan graf tersebut tidak mengandung siklus atau cycle dengan panjang negatif. Jika graf mengandung siklus negatif, algoritma ini dapat mendeteksinya.

Algoritma ini melacak jarak dari node awal ke semua node dalam graf. Pada awalnya, jarak ke node awal adalah 0 dan jarak ke semua node lainnya adalah tak hingga. Algoritma ini mengurangi jarak dengan menemukan sisi-sisi yang memperpendek jalur hingga tidak ada lagi pengurangan jarak yang bisa dilakukan.

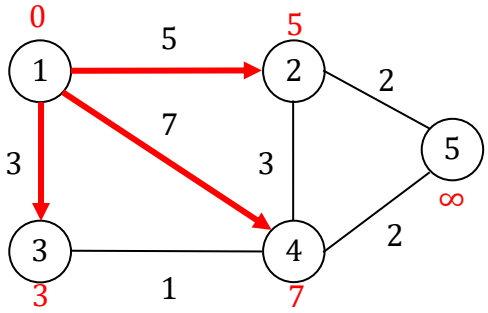
Contoh

Mari kita pertimbangkan bagaimana algoritma Bellman–Ford bekerja pada graf berikut:

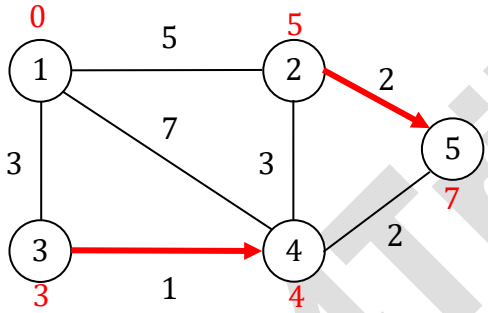


Setiap node pada graf diberikan jarak. Awalnya, jarak ke node awal adalah 0, dan jarak ke semua node lainnya adalah tak hingga.

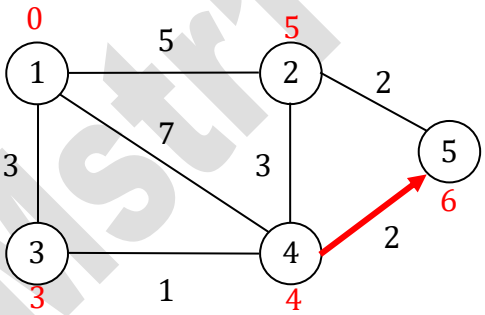
Algoritma mencari sisi yang mengurangi jarak. Pertama, semua sisi dari node 1 mengurangi jarak:



Setelah ini, sisi 2 → 5 dan 3 → 4 mengurangi jarak:

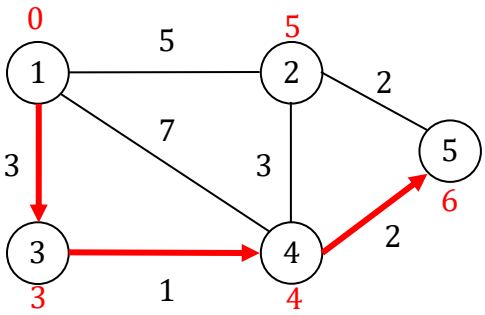


Akhirnya, ada satu perubahan lagi:



Setelah ini, tidak ada lagi tepi yang dapat mengurangi jarak. Ini berarti bahwa jarak-jarak tersebut sudah final, dan kita telah berhasil menghitung jarak terpendek dari node awal ke semua node dalam graf.

Misalnya, jarak terpendek 3 dari node 1 ke node 5 sesuai dengan jalur berikut:



Implementasi

Implementasi berikut dari algoritma Bellman–Ford menentukan jarak terpendek dari node x ke semua node dalam graf. Kode ini mengasumsikan bahwa graf disimpan sebagai daftar tepi `edges` yang terdiri dari tuple dengan format (a, b, w), yang berarti terdapat sebuah edge dari node a ke node b dengan bobot w.

Algoritma ini terdiri dari $n - 1$ putaran, dan pada setiap putaran, algoritma melewati semua tepi dalam graf dan mencoba untuk mengurangi jarak. Algoritma ini membangun sebuah array `distance` yang akan berisi jarak dari x ke semua node dalam graf. Konstanta `INF` menyatakan jarak tak terhingga.

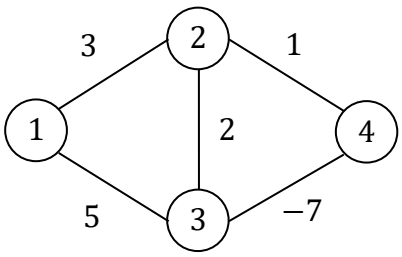
```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
for (int i = 1; i <= n-1; i++) {
    for (auto e : edges) {
        int a, b, w;
        tie(a, b, w) = e;
        distance[b] = min(distance[b], distance[a] + w);
    }
}
```

Kompleksitas waktu algoritma ini adalah $O(nm)$, karena algoritma ini terdiri dari $n - 1$ putaran dan mengiterasi semua m edge dalam setiap putaran. Jika tidak ada siklus negatif dalam graf, maka semua jarak akan final setelah $n - 1$ putaran, karena setiap jalur terpendek dapat memuat paling banyak $n - 1$ edge.

Dalam praktiknya, jarak akhir biasanya dapat ditemukan lebih cepat daripada dalam $n - 1$ putaran. Oleh karena itu, cara yang mungkin untuk membuat algoritma ini lebih efisien adalah dengan menghentikan algoritma jika tidak ada jarak yang dapat dikurangi dalam satu putaran.

Siklus Negatif

Algoritma Bellman-Ford juga dapat digunakan untuk memeriksa apakah graf mengandung siklus dengan panjang negatif. Sebagai contoh, graf berikut



mengandung siklus negatif $2 \to 3 \to 4 \to 2$ dengan panjang -4.

Jika graf mengandung siklus negatif, kita dapat memendekkan banyak jalur yang mengandung siklus tersebut dengan mengulang siklus itu berulang kali. Oleh karena itu, konsep jalur terpendek tidak bermakna dalam situasi ini.

Siklus negatif dapat dideteksi menggunakan algoritma Bellman-Ford dengan menjalankan algoritma selama n putaran. Jika putaran terakhir mengurangi jarak mana pun, graf mengandung siklus negatif. Perlu dicatat bahwa algoritma ini dapat digunakan untuk mencari siklus negatif di seluruh graf tanpa memandang node awal.

Algoritma SPFA

Algoritma SPFA ("Shortest Path Faster Algorithm") adalah varian dari algoritma Bellman-Ford, yang seringkali lebih efisien daripada algoritma asli. Algoritma SPFA tidak memeriksa semua edge pada setiap putaran, tetapi sebaliknya, memilih edge yang akan diperiksa dengan cara yang lebih cerdas.

Algoritma ini mempertahankan antrian node yang mungkin digunakan untuk mengurangi jarak. Pertama, algoritma menambahkan node awal x ke dalam queue. Kemudian, algoritma selalu memproses node pertama dalam queue, dan ketika sebuah edge $a \rightarrow b$ mengurangi jarak, node b ditambahkan ke dalam queue.

Efisiensi algoritma SPFA tergantung pada struktur graf: algoritma ini seringkali efisien, tetapi kompleksitas waktu kasus terburuknya tetap $O(nm)$, dan memungkinkan untuk membuat input yang dapat membuat algoritma ini seperlahan algoritma Bellman-Ford asli.

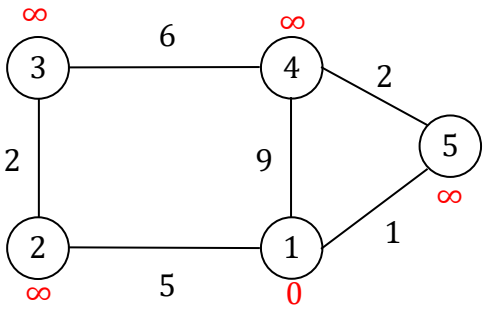
Algoritma Dijkstra

Algoritma Dijkstra menemukan jalur terpendek dari node awal ke semua node dalam graf, seperti algoritma Bellman-Ford. Keuntungan dari algoritma Dijkstra adalah bahwa algoritma ini lebih efisien dan dapat digunakan untuk memproses graf yang besar. Namun, algoritma ini memerlukan bahwa tidak ada edge dengan bobot negatif dalam graf.

Seperti algoritma Bellman-Ford, algoritma Dijkstra mempertahankan jarak ke node dan menguranginya selama pencarian. Algoritma Dijkstra efisien karena hanya memproses setiap edge dalam graf satu kali, dengan menggunakan fakta bahwa tidak ada edge negatif.

Contoh

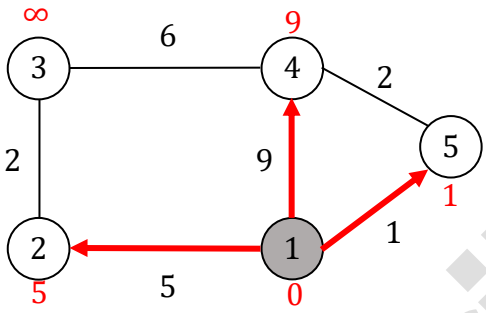
Mari kita pertimbangkan bagaimana algoritma Dijkstra bekerja pada graf berikut ketika node awal adalah node 1:



Seperti pada algoritma Bellman-Ford, pada awalnya jarak ke node awal adalah 0 dan jarak ke semua node lainnya adalah tak terhingga.

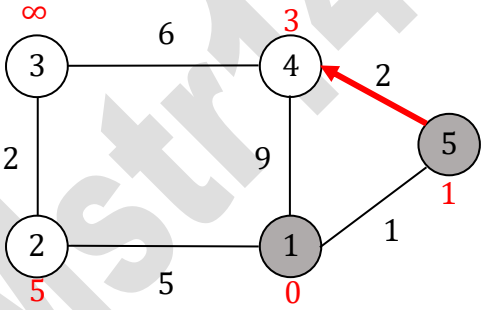
Pada setiap langkah, algoritma Dijkstra memilih node yang belum diproses dan memiliki jarak terkecil. Node pertama yang dipilih adalah node 1 dengan jarak 0.

Ketika sebuah node dipilih, algoritma memeriksa semua edge yang dimulai dari node tersebut dan mengurangi jarak menggunakan edge-edge tersebut:

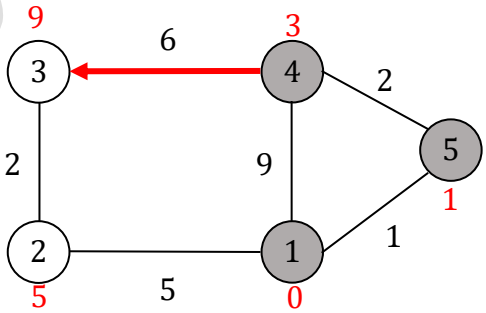


Dalam hal ini, edge-edge dari node 1 mengurangi jarak ke node 2, 4, dan 5, yang sekarang memiliki jarak 5, 9, dan 1.

Node berikutnya yang diproses adalah node 5 dengan jarak 1. Ini mengurangi jarak ke node 4 dari 9 menjadi 3:

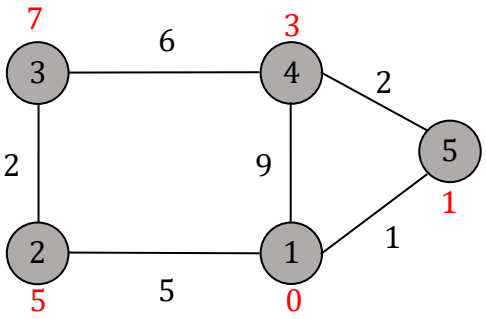


Setelah ini, node berikutnya adalah node 4, yang mengurangi jarak ke node 3 menjadi 9:



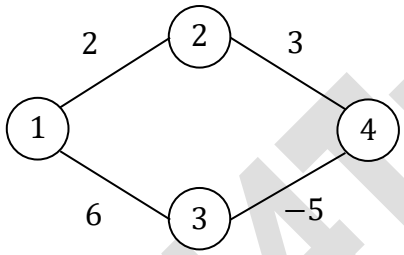
Sebuah sifat yang luar biasa dalam algoritma Dijkstra adalah bahwa setiap kali sebuah node dipilih, jaraknya adalah final. Sebagai contoh, pada titik ini dalam algoritma, jarak 0, 1, dan 3 adalah jarak final untuk node 1, 5, dan 4.

Setelah ini, algoritma memproses dua node yang tersisa, dan jarak finalnya adalah sebagai berikut:



Negative Edges

Efisiensi algoritma Dijkstra didasarkan pada kenyataan bahwa graf tersebut tidak mengandung edge negatif. Jika terdapat edge negatif, algoritma ini bisa memberikan hasil yang salah. Sebagai contoh, pertimbangkan graf berikut:



Jalur terpendek dari node 1 ke node 4 adalah $1 \rightarrow 3 \rightarrow 4$ dan panjangnya adalah 1. Namun, algoritma Dijkstra menemukan jalur $1 \rightarrow 2 \rightarrow 4$ dengan mengikuti tepi dengan bobot terkecil. Algoritma ini tidak mempertimbangkan bahwa pada jalur lainnya, bobot negatif -5 mengimbangi bobot besar sebelumnya yaitu 6.

Implementasi

Implementasi berikut dari algoritma Dijkstra menghitung jarak minimum dari sebuah node x ke node-node lain di graf. Graf disimpan sebagai adjacency lists sehingga `adj`$[a]$ berisi pasangan (b, w) setiap kali ada edge dari node a ke node b dengan bobot w.

Implementasi yang efisien dari algoritma Dijkstra memerlukan struktur data yang memungkinkan pencarian node dengan jarak minimum yang belum diproses secara efisien. Struktur data yang tepat untuk ini adalah *priority queue* yang berisi node yang diurutkan berdasarkan jaraknya. Dengan menggunakan *priority queue*, node yang akan diproses berikutnya dapat diambil dalam waktu logaritmik.

Pada kode berikut, *priority queue* `q` berisi pasangan dalam bentuk $(-d, x)$, yang berarti jarak saat ini ke node x adalah d. Array `distance` menyimpan jarak ke setiap node, dan array `processed` menunjukkan apakah sebuah node sudah diproses. Awalnya, jarak untuk node x adalah 0 dan untuk semua node lainnya adalah tak hingga (∞).

```
for (int i = 1; i <= n; i++) distance[i] = INF;
distance[x] = 0;
q.push({0, x});
while (!q.empty()) {
    int a = q.top().second; q.pop();
    if (processed[a]) continue;
    processed[a] = true;
    for (auto u : adj[a]) {
        int b = u.first, w = u.second;
        if (distance[a] + w < distance[b]) {
            distance[b] = distance[a] + w;
            q.push({-distance[b], b});
        }
    }
}
```

Perhatikan bahwa *priority queue* berisi jarak negatif ke node. Alasan untuk ini adalah bahwa versi default dari *priority queue* C++ menemukan elemen maksimum, sementara kita ingin menemukan elemen minimum. Dengan menggunakan jarak negatif, kita dapat langsung menggunakan *priority queue* default. Juga perhatikan bahwa mungkin ada beberapa instance dari node yang sama dalam *priority queue*; namun, hanya instance dengan jarak minimum yang akan diproses.

Kompleksitas waktu dari implementasi di atas adalah $O(n + m \log m)$, karena algoritma ini melewati semua node dalam graf dan menambahkan untuk setiap edge paling banyak satu jarak ke *priority queue*.

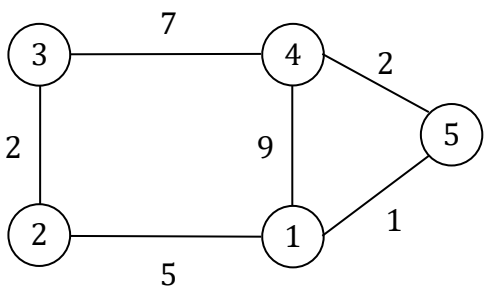
Algoritma Floyd-Warshall

Algoritma Floyd-Warshall memberikan pendekatan alternatif untuk masalah mencari *shortest path*. Berbeda dengan algoritma lain dalam bab ini, algoritma ini menemukan semua *shortest path* antar node dalam satu kali proses.

Algoritma ini mempertahankan array dua dimensi yang berisi jarak antar node. Pertama, jarak dihitung hanya menggunakan edge langsung antar node, dan setelah itu, algoritma ini mengurangi jarak dengan menggunakan node perantara dalam jalur.

Contoh

Mari kita pertimbangkan bagaimana algoritma Floyd-Warshall bekerja dalam graf berikut:



Awalnya, jarak dari setiap node ke dirinya sendiri adalah 0, dan jarak antara node a dan b adalah x jika ada edge antara node a dan b dengan bobot x. Semua jarak lainnya adalah tak terhingga.

Dalam graf ini, array awalnya adalah sebagai berikut:

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	∞	∞
3	∞	2	0	7	∞
4	9	∞	7	0	2
5	1	∞	∞	2	0

Algoritma ini terdiri dari putaran berturut-turut. Pada setiap putaran, algoritma memilih node baru yang dapat bertindak sebagai node perantara dalam jalur mulai sekarang, dan jarak-jarak diperpendek menggunakan node ini.

Pada putaran pertama, node 1 adalah node perantara baru. Ada jalur baru antara node 2 dan 4 dengan panjang 14, karena node 1 menghubungkan keduanya. Ada juga jalur baru antara node 2 dan 5 dengan panjang 6.

	1	2	3	4	5
1	0	5	∞	9	1
2	5	0	2	**14**	**6**
3	∞	2	0	7	∞
4	9	**14**	7	0	2
5	1	**6**	∞	2	0

Pada putaran kedua, node 2 adalah node perantara baru. Ini menciptakan jalur baru antara node 1 dan 3 serta antara node 3 dan 5.

	1	2	3	4	5
1	0	5	**7**	9	1
2	5	0	2	14	6
3	**7**	2	0	7	**8**
4	9	14	7	0	2
5	1	6	**8**	2	0

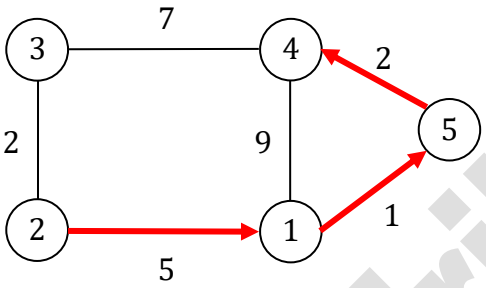
Pada putaran ketiga, node 3 adalah node perantara baru. Terdapat jalur baru antara node 2 dan 4.

	1	2	3	4	5
1	0	5	**7**	9	1
2	5	0	2	**9**	6
3	**7**	2	0	7	8
4	9	**9**	7	0	2
5	1	6	8	2	0

Algoritma ini berlanjut seperti ini, hingga semua node telah ditunjuk sebagai node perantara. Setelah algoritma selesai, array akan berisi jarak minimum antara setiap dua node.

	1	2	3	4	5
1	0	5	**7**	3	1
2	5	0	2	8	6
3	**7**	2	0	7	8
4	9	8	7	0	2
5	1	6	8	2	0

Misalnya, array memberi tahu kita bahwa jarak terpendek antara node 2 dan 4 adalah 8. Ini sesuai dengan jalur berikut:



Implementasi

Keuntungan dari algoritma Floyd-Warshall adalah mudahnya implementasi. Kode berikut membangun matriks jarak di mana `distance`$[a][b]$ adalah jarak terpendek antara node a dan b. Pertama, algoritma menginisialisasi `distance` menggunakan adjacency matrix `adj` dari graf:

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        if (i == j) distance[i][j] = 0;
        else if (adj[i][j]) distance[i][j] = adj[i][j];
        else distance[i][j] = INF;
    }
}
```

Setelah ini, jarak terpendek dapat ditemukan sebagai berikut:

```
for (int k = 1; k <= n; k++) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j]);
        }
    }
}
```

Kefektifan waktu dari algoritma ini adalah $O(n^3)$, karena mengandung tiga loop bertingkat yang melalui node-node dalam graf.

Karena implementasi algoritma Floyd-Warshall yang sederhana, algoritma ini bisa menjadi pilihan yang baik meskipun hanya dibutuhkan untuk menemukan satu jalur terpendek dalam graf. Namun, algoritma ini hanya dapat digunakan ketika graf cukup kecil sehingga kompleksitas waktu kubik masih cukup cepat.