

Implementasi *Dynamic Programming* Untuk Menentukan *Rendezvous Point* Terdekat

Ihsan Muhammad Asnadi / 135-16-028

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia

13516028@std.stei.itb.ac.id

Abstract—Sebagai makhluk sosial, manusia pasti akan berinteraksi dengan manusia lain, tidak terkecuali mahasiswa. Mahasiswa akan berkumpul dengan teman-temannya entah untuk mengerjakan tugas, belajar untuk mempersiapkan lomba ataupun sekedar *hangout* untuk melepaskan penat. Banyak tempat yang dapat dipilih untuk berkumpul: *mall*, *café*, *restaurant*, kampus, maupun indekos seseorang. Salah satu pertimbangan yang digunakan untuk memilih tempat berkumpul adalah dengan memilih tempat untuk berkumpul yang terdekat dari kediaman masing-masing orang. Pada makalah ini, penulis akan membahas implementasi *Dynamic Programming* untuk menentukan tempat berkumpul yang paling dekat dari kediaman masing-masing.

Keywords—*Overlapping Subproblem*, *Dynamic Programming*, *Floyd Warshall*.

I. PENDAHULUAN

Bidang *Computer Science* ada untuk memecahkan masalah-masalah yang terjadi di masyarakat dan mempermudah hidup manusia. Pada bidang *Computer Science*, terdapat beberapa teknik pemecahan masalah seperti *Brute Force*, *Greedy*, *Backtracking*, *Divide and Conquer*, *Decrease and Conquer*, *Dynamic Programming*, *Recursion*, dan *Branch and Bound*. Setiap teknik memiliki kelebihan dan kekurangan masing-masing, tergantung dari jenis permasalahan yang akan dihadapi. Beberapa teknik mungkin tidak dapat diaplikasikan pada suatu masalah, tetapi teknik tersebut mungkin akan menghasilkan solusi yang optimal pada permasalahan lain.

Di dunia industri, pemilihan teknik untuk menyelesaikan masalah tidak hanya dilihat dari keberhasilan teknik dalam menyelesaikan masalah, tetapi juga dari *performance* teknik tersebut. *Software* yang menyelesaikan *Travelling Salesman Problem* dengan teknik *Brute Force* mungkin tidak akan laku di pasaran karena membutuhkan waktu yang lama untuk menghitung solusi yang diinginkan.

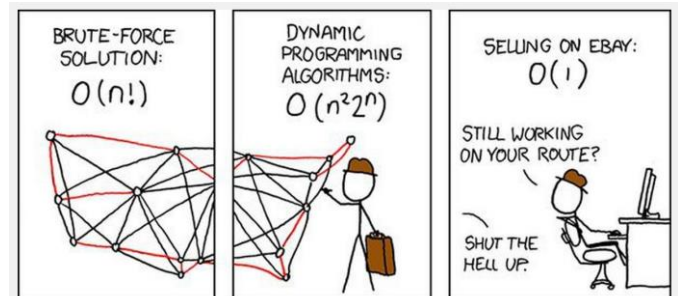


Figure I-1

Sumber: <http://codepreneur.github.io/2014/11/09/Aptitude-Attitude-and-Little-Core-Knowledge/>

II. LANDASAN TEORI

A. Recursion

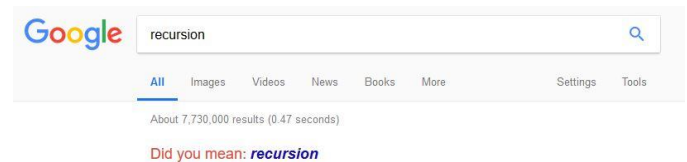


Figure II-1

Recursion adalah sebuah *function call* yang memanggil kembali dirinya sendiri. Beberapa masalah dapat diselesaikan dengan mudah jika menggunakan *recursion*, contohnya adalah menghitung Deret Fibonacci.

Recursion terdiri dari dua bagian, yaitu *base case* yang biasanya telah diketahui, dan solusi dari *problem* yang diekspresikan dalam *subproblem*.

Contoh kode perhitungan Deret Fibonacci:

```
def computeFibonacci(n):  
    if n == 0:  
        return 0  
    elif n == 1 or n == 2:  
        return 1  
    else:  
        return computeFibonacci(n-1) +  
        computeFibonacci(n-2)
```

Perhatikan bahwa fungsi computeFibonacci memanggil dirinya sendiri.

B. Overlapping subproblem

Sebuah *problem* disebut memiliki *overlapping subproblem* jika *problem* tersebut terdiri dari menyelesaikan *subproblem* yang sama secara berulang-ulang.

Fibonacci adalah contoh dari *overlapping subproblem* karena fungsi computeFibonacci(n-2) akan beririsan dengan nilai yang dihitung oleh fungsi computeFibonacci(n-1).

Contoh:

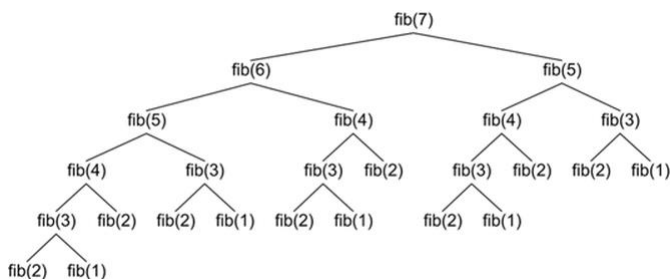


Figure II-2

Sumber: <http://www.rubyguides.com/2015/08/ruby-Recursion-and-memoization/>

Perhatikan bahwa fib(5) yang dipanggil oleh fib(7) adalah *subtree* dari fib(6). Fib(4) yang dipanggil oleh fib(5) juga merupakan *subtree* dari fib(6). Perhitungan nilai *subproblem* yang sama inilah yang dimaksud dengan *overlapping subproblem*. Perhitungan ini masih “dibenarkan” jika nilai N adalah nilai yang kecil, tapi untuk menghitung nilai N yang sangat besar akan dibutuhkan waktu yang lama. Bahkan jika waktu bukanlah *constraint*, maka akan terjadi *maximum depth recursion exceeded*, sebuah *state* ketika batas rekursi maksimum sudah tercapai, artinya program tidak dapat menghitung lagi karena *depth* dari *recursion tree* sudah sangat besar.

C. Dynamic Programming

Dynamic Programming adalah sebuah teknik yang membagi sebuah masalah menjadi beberapa *subproblem*. Setiap *subproblem* ini dapat diselesaikan secara independen yang pada akhirnya akan menghasilkan solusi dari masalah yang utuh.

Masalah yang dapat diselesaikan dengan *Dynamic Programming* biasanya memiliki karakteristik sebagai berikut:

- Masalah memiliki basis dan rekurens.
- Masalah dapat dibagi menjadi beberapa *subproblem*.
- Masalah merupakan persoalan optimasi yang mempertimbangkan banyak kemungkinan.
- *Subproblem* dapat diselesaikan secara independen.
- Memiliki *overlapping subproblem*.

Dynamic Programming adalah metode yang cocok untuk menyelesaikan *overlapping subproblem* karena metode ini menawarkan “penyimpanan” untuk solusi dari *subproblem* yang telah dihitung sehingga tidak akan ada perhitungan ulang untuk *subproblem* yang telah *solved*.

Dynamic Programming dibagi menjadi dua:

- *Memoization* (Top Down)

Memoization adalah metode yang mirip dengan *recursion*, bedanya, *memoization* akan melihat terlebih dahulu apakah *subproblem* tersebut sudah pernah diselesaikan dan disimpan pada tabel, jika belum, akan dimulai perhitungan baru.

Contoh:

```
arr = [-1]*100  
def fibTD(n):  
    if arr[n] == -1:  
        if n == 0:  
            arr[n] = 0  
        elif n == 1 or n == 2:  
            arr[n] = 1  
        else:  
            arr[n] = fibTD(n-1) +  
            fibTD(n-2)  
    return arr[n]
```

- *Tabulation* (Bottom Up)

Tabulation adalah metode yang menghitung solusi mulai dari nilai basis, menaik ke *subproblem* yang lebih besar. Solusi *problemnya* adalah nilai tabel di indeks terakhir.

Contoh:

```
arr = [-1]*100
def fibBU(n):
    arr[0] = 0
    arr[1] = 1
    for i in range(2, n+1):
        arr[i] = arr[i-1] + arr[i-2]
    return arr[n]
```

D. Graph

Graph adalah himpunan titik-titik yang dihubungkan oleh garis. Secara formal, definisi *graph* adalah *graph* terdiri dari pasangan himpunan *vertices* V yang tidak kosong, dan himpunan E yang anggotanya adalah pasangan dua elemen sebagai subset dari V . Anggota E disebut juga *edges*. *Graph* dituliskan sebagai $G = (V, E)$.

Graph banyak digunakan dalam dunia computer science karena *graph* bisa memodelkan hubungan antara objek. Banyak masalah yang dapat direpresentasikan dengan *graph*, contoh yang terkenal adalah *Seven Bridges of Königsberg* dan *Travelling Salesman Problem*.

Dalam *graph theory*, terdapat beberapa istilah yaitu:

- **Vertex**
Vertex adalah sebuah *node* (titik) yang terdapat di dalam *graph*.
- **Edge**
Edge adalah sebuah garis yang menghubungkan dua buah *vertices*.
Pada directed *graph*, *edge* disebut juga sebagai *arc*.
- **Adjacent**
Dua buah *vertices* pada *graph* dikatakan *adjacent* bila terdapat suatu *edge* yang menghubungkan keduanya.
- **Incident**
Untuk sembarang *edge* e pada himpunan E dalam *graph*. *Edge* e *incident* dengan simpul yang terhubung pada kedua ujungnya.
- **Path**
Path adalah himpunan *edges* $\{e_1, e_2, e_3, \dots, e_N\}$ yang menghubungkan *vertex* awal v_0 dan *vertex* akhir v_N sedemikian sehingga $e_1 = (v_0, v_1)$, $e_2 = (v_1, v_2)$, ..., dan $e_N = (v_{N-1}, v_N)$.
- **Cycle**
Path yang berawal dan berakhir pada *vertex* yang sama disebut *cycle*.

- **Connected**

Sebuah pasangan *vertices* dikatakan *connected* jika terdapat *path* yang menghubungkan keduanya.

Sedangkan sebuah *graph* dikatakan *connected* jika untuk setiap pasang *vertices* dalam himpunan V , terdapat *path* yang menghubungkan keduanya.

Jika ada pasangan *vertices* yang tidak terhubung oleh suatu *path*, maka *graph* tersebut dikatakan *disconnected*.

- **Unweighted Graph**

Unweighted graph adalah *graph* yang tiap *edge*-nya tidak memiliki nilai, jadi *graph* jenis ini hanya menunjukkan keterhubungan. Contoh: A berteman dengan B direpresentasikan dengan *vertex* A terhubung dengan *vertex* B pada suatu *edge*.

- **Weighted Graph**

Weighted graph adalah *graph* yang tiap *edge*-nya memiliki nilai, nilai ini dapat merepresentasikan kuantitas keterhubungan objek. Contoh: Jarak kota A ke kota B sejauh seratus kilometer dapat direpresentasikan dengan *vertex* A terhubung dengan *vertex* B pada suatu *edge*, lalu terdapat angka "100" di dekat *edge* tersebut.

E. Floyd Warshall Algorithm

Algoritma Floyd Warshall adalah algoritma yang digunakan untuk menghitung *All Pairs Shortest Path problem*, yaitu masalah yang meminta solusi untuk menemukan jarak terpendek dari setiap pasangan *vertices* di dalam *weighted graph*. Algoritma ini menggunakan *Dynamic Programming* karena setiap jarak minimum dari *vertex* A ke B akan dicatat pada *Adjacency Matrix*.

Ide utama dari algoritma Floyd Warshall adalah dengan memilih nilai minimum antara jarak A ke B tanpa melewati C dan jarak A ke B dengan melewati C. Implementasinya menggunakan *Adjacency Matrix* awal yang berisi *weight* dari *edge* (A, B). *Adjacency Matrix* ini akan terus di-update selama perhitungan jarak.

Contoh kode Floyd Warshall:

```
for k in range(lenOfArr):
    for i in range(lenOfArr):
        for j in range(lenOfArr):
            adjMatrix[i][j] =
min(adjMatrix[i][j], adjMatrix[i][k]
+ adjMatrix[k][j])
```

III. IMPLEMENTASI

Pada bagian ini, penulis akan membahas implementasi *Dynamic Programming* untuk mencari *Rendezvous Point* terdekat dari kediaman setiap orang.

Pencarian sebuah titik yang memiliki total jarak terdekat dari semua titik lain dapat diselesaikan dengan algoritma Floyd Warshall.

Berikut langkah-langkah yang dapat dilakukan untuk menghitung *Rendezvous Point* terdekat:

1. Representasikan posisi sebagai kelas "Point".

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def print(self):
        print(self.x, self.y)

    def computeDistance(self, p):
        return
        math.sqrt(math.pow(self.x-p.x, 2)
        + math.pow(self.y-p.y, 2))
```

2. Menyimpan posisi kediaman setiap orang dan *landmark* lain yang mungkin dijadikan sebagai *Rendezvous Point* ke dalam *list*.

```
if __name__ == "__main__":
    os.system("CLS")
    a = Point(0, 0)
    b = Point(5, 0)
    c = Point(5, 12)
    arr = [a, b, c]
```

3. Pass *list* ke dalam fungsi *FloydWarshall*.

```
FloydWarshall(arr)
```

4. Deklarasi variabel

```
def FloydWarshall(arr):
    INF = "INF"
    lenOfArr = len(arr)
```

5. Representasikan *Point* sebagai *Adjacency matrix*.

```
adjMatrix = []
for _ in range(lenOfArr):
    ctemp = []
    for _ in range(lenOfArr):
        ctemp.append(INF)
    adjMatrix.append(ctemp)
```

6. Hitung jarak antar *Point* sebagai *Weight* dari *Graph*.

```
for i in range(lenOfArr):
    for j in range(lenOfArr):
        ans = -1
        if i == j:
            ans = 0
        else:
            ans =
            arr[i].computeDistance(arr[j])
            adjMatrix[i][j] =
            adjMatrix[j][i] = ans
```

7. Gunakan *Dynamic Programming* untuk menghitung jarak terdekat tiap semua pasangan titik.

```
for k in range(lenOfArr):
    for i in range(lenOfArr):
        for j in range(lenOfArr):
            adjMatrix[i][j] =
            min(adjMatrix[i][j],
            adjMatrix[i][k] +
            adjMatrix[k][j])
```

8. Mencari posisi yang memiliki jarak terdekat.

```
temp = int(1e9)
ans = -1
for i in range(lenOfArr):
    totalDistance =
    sum(adjMatrix[i])
    if temp > totalDistance:
        ans = i
        temp = totalDistance
```

9. Mengeluarkan *Adjacency Matrix* dan tempat yang memiliki total jarak terdekat dari tempat lain.

```
for x in adjMatrix:
    print(x)
print(ans+1, temp)
```

Kelebihan algoritma Floyd Warshall dibandingkan *Minimum Spanning Tree* adalah algoritma Floyd Warshall dapat menghitung jarak terdekat yang membentuk sirkuit.

Contoh:

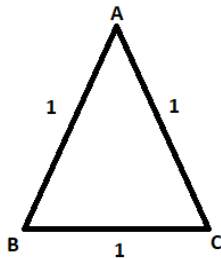


Figure III-1

Pada gambar tersebut, *Minimum Spanning Tree* akan menghasilkan sebuah *path* berbentuk “V”, dengan asumsi pivot di titik “A”, sehingga jarak A-B = 1, A-C = 1, dan B-C = 2. Sedangkan jika menggunakan Floyd Warshall, maka akan terbentuk *path* berbentuk segitiga ABC, sehingga jarak A-B = 1, A-C = 1, dan B-C = 1.

IV. HASIL EKSPERIMEN

Pada bagian ini, penulis akan mencantumkan hasil pengujian terhadap program yang telah ditulis.

Untuk tiga titik A = (0,0), B = (5,0), dan C = (5,12), didapatkan hasil sebagai berikut:

```
C:\Window...
[0, 5.0, 13.0]
[5.0, 0, 12.0]
[13.0, 12.0, 0]
2 17.0
```

Figure IV-1

Solusi yang diberikan adalah menggunakan titik B sebagai *Rendezvous Point* karena jarak yang perlu ditempuh hanyalah A-B = 5, dan B-C = 12, sehingga total jarak = 17. Pemilihan titik lain sebagai *Rendezvous Point* tidak memberikan hasil yang optimal.

Untuk empat titik A = (0,0), B = (5,0), C = (5,12), dan D = (0,24) didapatkan hasil sebagai berikut:

```
C:\Windows\System32\cmd.exe
[0, 5.0, 13.0, 24.0]
[5.0, 0, 12.0, 24.515301344262525]
[13.0, 12.0, 0, 13.0]
[24.0, 24.515301344262525, 13.0, 0]
3 38.0
```

Figure IV-2

Solusi yang diberikan adalah menggunakan titik C sebagai *Rendezvous Point* karena jarak yang perlu ditempuh hanyalah A-C = 13, B-C = 12, dan C-D = 13 sehingga total jarak = 38. Seperti pada *test* sebelumnya, pemilihan titik lain sebagai *Rendezvous Point* tidak memberikan hasil yang optimal.

V. KESIMPULAN

Dynamic Programming adalah sebuah metode yang cukup efisien walaupun harus memperhitungkan banyak kemungkinan. Alasannya, *Dynamic Programming* menyimpan hasil perhitungan *subproblem* sehingga perhitungan *subproblem* hanya dilakukan sekali. Hasil perhitungan *Dynamic Programming* juga dipastikan dapat memberikan hasil yang optimal.

UCAPAN TERIMA KASIH

Penulis mengucapkan syukur kepada Tuhan Yang Maha Esa karena atas berkat dan rahmat-Nya penulis dapat menyelesaikan makalah ini tanpa halangan yang berarti. Penulis juga mengucapkan terima kasih kepada orang tua penulis yang telah memberikan bantuan materiil dan moril kepada penulis.

REFERENSI

- [1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. London: MIT Press.
- [2] Gozali, W., & Aji, A. F. (2018). *Pemrograman Kompetitif Dasar*. unpublished.
- [3] Halim, S. (2013). *Competitive Programming 3: The New Lower Bound of Programming Contests*. Singapore: School of Computing, National University of Singapore.
- [4] Laaksonen, A. (n.d.). *Competitive Programmer's Handbook*. unpublished.
- [5] Prihatmaja, P. A. (n.d.). *Perbandingan Performa Algoritma Greedy dan Dynamic Programming*.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 14 Mei 2018

A handwritten signature in black ink, consisting of a stylized 'S' shape with a horizontal line crossing it.

Ihsan Muhammad Asnadi

135-16-028