TUGAS AKHIR

PERANCANGAN & ANALISIS ALGORITMA

Disusun Oleh :

Nama : Muhammad Razief

NIM : 21346015

Prodi : Informatika (NK)

Dosen Pengampu : Widya Darwin, S.Pd., M.Pd.T

PROGRAM STUDI INFORMATIKA

JURUSAN TEKNIK ELEKTRONIKA

FAKUTAS TEKNIK

UNIVERSITAS NEGERI PADANG

2023

KATA PENGANTAR

Rasa syukur kita hanya milik Allah SWT atas segala semua rahmatnya, sehingga saya dapat menyelesaikan tugas akhir yang saya susun ini. Meskipun banyak rintangan dan hambatan yang saya alami dalam proses pekerjaan tetapi saya berhasil mengerjakan dengan baik dan tetap pada waktunya.

Dan harapan saya di sini semoga atas tugas akhir yang saya buat ini bisa menambah pengetahuan dan pengalaman bagi para pembaca, dan untuk kedepan nya kiat juga sama-sama memperbaiki bentuk atau menambah isi dari makalah agar semua akan lebih baik dengan sebelumnya.

Saya mengucapkan terima kasih kepada Ibu Widya Darwin, S.Pd., M.Pd.T sebagai Dosen Pengampu pada mata kuliah Perancangan & Analisis Algoritma yang telah membimbing saya dalam menyelesaikan tugas akhir yang saya buat ini.

Karena dari semua keterbatasan dari pengetahuan atau pun pengalaman saya, saya yakin masih banyak sekali dari kekurangan yang terdapat pada makalah ini. Oleh karena itu saya sangat berharap untuk saran dan kritik yang bisa membangun dari pembaca demi semua tugas akhir ini akan terselesaikan dengan benar.

Painan, Juni 2023

Muhammad Razief

DAFTAR ISI

BAB I
ANALISIS ALGORITMA

A. Pengertian

Analisis algoritma adalah proses untuk mempelajari dan menganalisis kinerja suatu algoritma. Algoritma adalah serangkaian langkah atau aturan yang digunakan untuk menyelesaikan suatu masalah atau mencapai suatu tujuan. Dalam analisis algoritma, kita melihat berbagai aspek seperti waktu eksekusi (runtime) dan kebutuhan ruang (space complexity) yang diperlukan oleh algoritma tersebut.

Tujuan utama dari analisis algoritma adalah untuk memahami seberapa efisien atau efektif sebuah algoritma dalam menyelesaikan masalah. Hal ini penting karena terdapat banyak cara berbeda untuk menyelesaikan masalah yang sama, dan analisis algoritma membantu kita dalam memilih algoritma terbaik yang paling cocok untuk suatu tugas.

B. Langkah – Langkah dalam Melakukan Analisis Algoritma

Berikut adalah langkah-langkah umum yang dapat diikuti dalam melakukan analisis algoritma:

1. Pahami masalah: Identifikasi dengan jelas masalah yang ingin diselesaikan dengan menggunakan algoritma. Pahami persyaratan, tujuan, dan batasan masalah tersebut.

2. Tentukan input dan output: Ketahui jenis data yang akan digunakan sebagai input dan output dari algoritma. Pahami format data yang diperlukan dan hasil yang diharapkan.

3. Rancang langkah-langkah: Gunakan pengetahuan dan kreativitas untuk merancang langkah-langkah yang logis untuk menyelesaikan masalah. Berpikir secara sistematis dan pecah masalah menjadi langkah-langkah yang lebih kecil.

4. Analisis kompleksitas: Evaluasi kompleksitas algoritma yang dirancang untuk memahami seberapa efisien algoritma tersebut. Tinjau faktor-faktor seperti waktu eksekusi, penggunaan memori, dan sumber daya lainnya yang diperlukan.

5. Implementasikan algoritma: Terjemahkan langkah-langkah yang telah dirancang ke dalam bahasa pemrograman yang relevan. Buat kode yang sesuai dengan desain algoritma.

6. Uji dan evaluasi: Uji algoritma dengan memberikan berbagai input yang berbeda untuk memastikan bahwa algoritma berfungsi dengan benar dan menghasilkan

output yang diharapkan. Identifikasi masalah atau kekurangan dalam algoritma dan lakukan perbaikan jika diperlukan.

7. Analisis kinerja: Evaluasi kinerja algoritma setelah diimplementasikan dan diuji. Periksa waktu eksekusi, penggunaan memori, dan efisiensi algoritma secara keseluruhan. Bandingkan dengan algoritma lain yang mungkin ada untuk melihat mana yang lebih baik dalam menyelesaikan masalah tersebut.

8. Optimalisasi: Jika diperlukan, coba cari cara untuk meningkatkan kinerja algoritma dengan melakukan perubahan atau optimasi. Tinjau kembali langkah-langkah, struktur data, atau strategi yang digunakan untuk mencari cara yang lebih efisien untuk menyelesaikan masalah.

9. Dokumentasikan: Pastikan untuk mendokumentasikan algoritma, langkah-langkah, dan hasil analisis secara rinci. Ini akan membantu orang lain memahami algoritma dan mempermudah pemeliharaan atau pengembangan lebih lanjut di masa depan.

Langkah-langkah di atas memberikan kerangka dasar untuk melakukan analisis algoritma, tetapi perlu diingat bahwa pendekatan dapat bervariasi tergantung pada masalah yang dihadapi.

C. Dasar – dasar Algortima

Berikut adalah beberapa dasar-dasar algoritma yang perlu dipahami:

1. Definisi algoritma: Algoritma adalah serangkaian instruksi langkah-demi-langkah yang dirancang untuk menyelesaikan masalah atau mencapai tujuan tertentu. Algoritma harus memiliki langkah-langkah yang jelas, terstruktur, dan dapat dijalankan secara sistematis.

2. Struktur kontrol: Algoritma menggunakan struktur kontrol untuk mengatur alur eksekusi langkah-langkah. Struktur kontrol umum termasuk pengulangan (looping), pengambilan keputusan (if-else), dan pengulangan terkondisi (while, for).

3. Variabel: Variabel digunakan untuk menyimpan data dalam algoritma. Variabel dapat menampung berbagai jenis data seperti angka, teks, boolean, dan lainnya. Variabel memungkinkan manipulasi dan penggunaan data dalam algoritma.

4. Operasi aritmatika dan logika: Algoritma dapat melibatkan operasi aritmatika seperti penjumlahan, pengurangan, perkalian, dan pembagian untuk memanipulasi angka. Operasi logika seperti AND, OR, dan NOT digunakan untuk menggabungkan dan memanipulasi kondisi boolean.

5. Struktur data: Struktur data adalah cara untuk mengorganisir dan menyimpan data dalam algoritma. Beberapa struktur data umum meliputi array, daftar (list), tumpukan (stack), antrian (queue), dan pohon (tree). Pemilihan struktur data yang tepat dapat meningkatkan efisiensi dan kinerja algoritma.

6. Rekursi: Rekursi adalah konsep di mana sebuah fungsi atau algoritma dapat memanggil dirinya sendiri. Rekursi digunakan untuk menyelesaikan masalah yang dapat dibagi menjadi submasalah yang lebih kecil. Namun, penggunaan rekursi

harus dilakukan dengan hati-hati untuk menghindari rekursi tak terbatas dan memastikan kondisi dasar (base case) tercapai.

7. Kompleksitas algoritma: Kompleksitas algoritma adalah ukuran kinerja algoritma dalam hal waktu dan ruang yang diperlukan untuk menjalankan algoritma. Hal ini dipengaruhi oleh faktor-faktor seperti ukuran input, jumlah operasi yang dilakukan, dan penggunaan memori. Analisis kompleksitas algoritma membantu dalam membandingkan dan memilih algoritma yang paling efisien untuk menyelesaikan masalah.

Pemahaman dasar-dasar ini akan membantu dalam merancang, menerapkan, dan menganalisis algoritma dengan lebih baik.

D. Problem Solving

Problem solving refers to the process of finding solutions to problems or overcoming challenges. It involves using critical thinking, logical reasoning, and creativity to analyze the problem, develop a strategy, and implement a solution. Here are some key aspects of problem solving:

1. Identify the problem: Clearly define the problem or challenge that needs to be addressed. Understand the desired outcome and the constraints or limitations involved.

2. Gather information: Collect relevant information and data related to the problem. This may involve conducting research, gathering facts, and seeking input from others who may have knowledge or experience in the area.

3. Analyze the problem: Break down the problem into smaller components and examine the relationships and dependencies between them. Identify patterns, trends, or underlying causes that contribute to the problem.

4. Generate potential solutions: Use creative thinking and brainstorming techniques to generate a variety of possible solutions. Encourage diverse perspectives and explore different approaches to tackle the problem.

5. Evaluate alternatives: Assess each potential solution based on its feasibility, effectiveness, and potential impact. Consider the advantages and disadvantages of each option and select the most promising ones for further consideration.

6. Implement the solution: Develop a plan of action to implement the chosen solution. Break down the solution into manageable steps and allocate necessary resources. Execute the plan and monitor the progress.

7. Review and learn: Evaluate the effectiveness of the solution by analyzing the results and comparing them to the desired outcome. Identify any lessons learned and areas for improvement. Use this knowledge to refine future problem-solving approaches.

8. Adaptability and resilience: Problem solving often requires adaptability and the ability to adjust strategies or solutions as new information or challenges arise. Remain open to feedback, be willing to make adjustments, and persist in finding

viable solutions.

9. Collaboration and communication: In many cases, problem solving is a collaborative process. Engage with others, seek their input and expertise, and foster effective communication to gain diverse perspectives and increase the likelihood of successful outcomes.

Developing problem-solving skills is essential in various domains, including academics, professional settings, and everyday life. By approaching problems systematically and employing analytical thinking, individuals can enhance their problem-solving abilities and find effective solutions to a wide range of challenges.

BAB II

ANALISIS EFISIENSI ALGORITMA

A. Measuring an Input's Size

Dalam analisis algoritma, pengukuran ukuran input mengacu pada menentukan ukuran atau skala dari data input yang digunakan oleh algoritma. Ukuran input dapat bervariasi tergantung pada masalah yang diselesaikan dan representasi khusus dari input tersebut.

Ukuran input biasanya diukur dalam hal jumlah elemen atau total jumlah data yang perlu diproses oleh algoritma. Contohnya:

- Untuk algoritma yang mengurutkan daftar angka, ukuran input dapat diukur sebagai jumlah elemen dalam daftar.
- Untuk algoritma yang mencari item tertentu dalam sebuah database, ukuran input dapat diukur sebagai total jumlah rekaman atau entri dalam database.
- Untuk algoritma yang memecahkan masalah graf, ukuran input dapat diukur sebagai jumlah simpul (vertices) dan sisi (edges) dalam graf.

Ukuran input merupakan faktor penting dalam analisis algoritma karena mempengaruhi kinerja dan efisiensi algoritma. Secara umum, ketika ukuran input meningkat, algoritma mungkin membutuhkan lebih banyak waktu dan sumber daya untuk memproses data. Dengan menganalisis bagaimana waktu eksekusi dan kebutuhan ruang algoritma berkembang seiring dengan ukuran input, kita dapat memahami efisiensinya dan membuat keputusan yang terinformasi tentang pemilihan dan optimasi algoritma.

B. Unit for Measuring Running Time

Satuan yang biasa digunakan untuk mengukur waktu berjalan atau waktu eksekusi suatu algoritme adalah "kompleksitas waktu". Kompleksitas waktu mengkuantifikasi jumlah waktu yang diperlukan algoritme untuk dijalankan sebagai fungsi dari ukuran input. Ini membantu kami menganalisis dan membandingkan efisiensi berbagai algoritme.

Kompleksitas waktu biasanya diungkapkan menggunakan notasi Big O, yang memberikan batas atas pada laju pertumbuhan waktu eksekusi. Misalnya, jika suatu algoritma memiliki kompleksitas waktu O(n), itu berarti waktu eksekusi meningkat secara linier dengan ukuran input. Jika ukuran input menjadi dua kali lipat, waktu eksekusi juga sekitar dua kali lipat.

Kompleksitas waktu lain yang umum meliputi:

- Waktu konstan: O(1) - waktu eksekusi tetap konstan terlepas dari ukuran input.

- Waktu logaritmik: O(log n) - waktu eksekusi meningkat secara logaritmik dengan ukuran input. Contohnya termasuk algoritma pencarian biner.

- Waktu kuadratik: O(n^2) - waktu eksekusi tumbuh secara kuadratik dengan ukuran input. Contohnya termasuk perulangan bersarang.

- Waktu eksponensial: O(2^n) - waktu eksekusi tumbuh secara eksponensial dengan ukuran input. Algoritma dengan kompleksitas waktu eksponensial umumnya dianggap tidak efisien untuk input yang besar.

Perlu dicatat bahwa kompleksitas waktu memberikan analisis asimptotik, berfokus pada laju pertumbuhan saat ukuran input menjadi sangat besar. Ini tidak memberikan pengukuran yang tepat tentang waktu eksekusi sebenarnya dalam detik atau milidetik. Waktu eksekusi sebenarnya dapat dipengaruhi oleh faktor seperti perangkat keras, bahasa pemrograman, dan detail implementasi spesifik.

C. Order of Growth

Order of growth (tingkat pertumbuhan) mengacu pada bagaimana waktu eksekusi atau kompleksitas sebuah algoritma berkembang seiring dengan peningkatan ukuran input. Dalam analisis algoritma, order of growth dinyatakan menggunakan notasi Big O.

Order of growth menunjukkan bagaimana waktu eksekusi algoritma berubah relatif terhadap ukuran input. Beberapa contoh order of growth yang umum meliputi:

- O(1) - Konstanta: Waktu eksekusi konstan, tidak tergantung pada ukuran input. Contohnya adalah akses langsung ke elemen dalam array.

- O(log n) - Logaritmik: Waktu eksekusi meningkat secara logaritmik dengan ukuran input. Contohnya adalah pencarian biner di array terurut.

- O(n) - Linier: Waktu eksekusi meningkat secara linier dengan ukuran input. Contohnya adalah iterasi melalui array untuk mencari elemen tertentu.

- O(n log n) - N log N: Waktu eksekusi meningkat dengan perkalian antara ukuran input dan logaritma dari ukuran input. Contohnya adalah algoritma pengurutan seperti quicksort atau mergesort.

- O(n^2) - Kuadratik: Waktu eksekusi meningkat secara kuadratik dengan ukuran input. Contohnya adalah nested loop yang mengiterasi melalui matriks dua dimensi.

- O(2^n) - Eksponensial: Waktu eksekusi meningkat eksponensial dengan

ukuran input. Contohnya adalah algoritma brute-force untuk subset sum problem.

Dalam analisis algoritma, kita mencari order of growth terburuk (worst-case) yang memberikan batas atas pada waktu eksekusi algoritma. Dengan mengetahui order of growth sebuah algoritma, kita dapat memprediksi bagaimana waktu eksekusi algoritma akan berubah saat ukuran input meningkat, dan membuat keputusan yang lebih baik dalam memilih algoritma yang efisien untuk penyelesaian masalah yang diberikan.

D. Worst-Case, Best-Case, and Average-Case Efficiency

Efisiensi dalam analisis algoritma dapat diukur dalam tiga skenario yang berbeda: worst-case (kasus terburuk), best-case (kasus terbaik), dan average-case (kasus rata-rata). Setiap skenario ini memberikan gambaran tentang kinerja algoritma dalam situasi yang berbeda.

- Worst-Case Efficiency (Efisiensi Kasus Terburuk):

  Worst-case efficiency mengukur kinerja terburuk yang dapat dicapai oleh algoritma pada input terburuk yang mungkin. Ini memberikan batas atas pada waktu eksekusi algoritma dalam situasi paling tidak menguntungkan. Analisis worst-case efficiency sangat penting karena menjamin bahwa algoritma akan menyelesaikan tugas dalam waktu yang dapat diterima dalam semua situasi.

- Best-Case Efficiency (Efisiensi Kasus Terbaik):

  Best-case efficiency mengukur kinerja terbaik yang dapat dicapai oleh algoritma pada input terbaik yang mungkin. Ini memberikan batas bawah pada waktu eksekusi algoritma dan menggambarkan skenario optimal di mana algoritma berjalan dengan cepat. Namun, best-case efficiency tidak memberikan gambaran lengkap tentang kinerja sebenarnya, karena input terbaik mungkin jarang terjadi dalam praktiknya.

- Average-Case Efficiency (Efisiensi Kasus Rata-rata):

  Average-case efficiency mengukur kinerja rata-rata algoritma pada input yang dianggap terjadi secara acak. Ini mencerminkan kinerja yang diharapkan dalam keadaan umum saat algoritma digunakan dalam praktiknya. Analisis efisiensi kasus rata-rata melibatkan penggunaan statistik dan probabilitas untuk menghitung waktu eksekusi rata-rata algoritma.

Dalam analisis algoritma, fokus utama sering kali pada worst-caseefficiency, karena memberikan jaminan bahwa algoritma akan menyelesaikan tugas dalam waktu yang dapat diterima dalam semua situasi. Namun, dalam beberapa kasus, analisis kasus terbaik atau kasus rata-rata juga dapat penting, tergantung pada aplikasi dan distribusi input yang diharapkan.

BAB III

BRUTE-FORCE EXHAUSTIVE SEARCH

A. Selection Sort and Bubble Sort

Selection Sort dan Bubble Sort adalah dua algoritma pengurutan sederhana yang digunakan untuk mengurutkan elemen dalam suatu array. Berikut adalah penjelasan singkat tentang kedua algoritma tersebut:

1. Selection Sort (Pengurutan Pilihan):
Selection Sort bekerja dengan membagi array menjadi dua bagian: bagian yang sudah terurut dan bagian yang belum terurut. Pada setiap iterasi, algoritma ini mencari elemen terkecil dari bagian yang belum terurut dan menukar posisinya dengan elemen terdepan dari bagian terurut. Dengan demikian, elemen-elemen terkecil secara bertahap dipindahkan ke bagian terurut hingga seluruh array terurut.

Langkah-langkah dalam Selection Sort:
- Cari elemen terkecil dalam bagian yang belum terurut.
- Tukar elemen terkecil dengan elemen terdepan dalam bagian terurut.
- Perluas bagian terurut dengan memindahkan batas satu langkah ke depan.
- Ulangi langkah-langkah di atas hingga seluruh array terurut.

Selection Sort memiliki kompleksitas waktu O(n^2), di mana n adalah jumlah elemen dalam array. Algoritma ini sederhana dan mudah dipahami, tetapi tidak efisien untuk array dengan jumlah elemen yang besar.

2. Bubble Sort (Pengurutan Gelembung):
Bubble Sort bekerja dengan membandingkan pasangan elemen bersebelahan dalam array dan menukar posisi jika urutannya salah. Pada setiap iterasi, elemen dengan nilai yang lebih besar akan "naik ke atas" seperti gelembung, sehingga elemen terbesar secara bertahap dipindahkan ke posisi yang benar.

Langkah-langkah dalam Bubble Sort:
- Bandingkan pasangan elemen bersebelahan dalam array.
- Jika urutan pasangan tersebut salah, tukar posisi mereka.
- Perulangan langkah-langkah di atas untuk setiap pasangan elemen dalam array.
- Ulangi langkah-langkah di atas hingga seluruh array terurut.

Bubble Sort juga memiliki kompleksitas waktu O(n^2), di mana n adalah jumlah elemen dalam array. Algoritma ini juga sederhana namun tidak efisien untuk array dengan jumlah elemen yang besar. Selain itu, Bubble Sort dapat menghasilkan banyak pertukaran jika elemen terbesar berada di posisi yang jauh dari akhir array, sehingga membuatnya kurang efisien.

Kedua algoritma ini merupakan contoh pengurutan sederhana yang cocok untuk digunakan pada array kecil atau dalam kasus-kasus di mana kompleksitas waktu bukanlah faktor yang kritis. Untuk array dengan jumlah elemen yang besar, algoritma pengurutan yang lebih efisien seperti Quick Sort atau Merge Sort lebih disarankan.

B. Sequential Search and Brute –Force String Maching

Sequential Search (Pencarian Sekuensial):
Sequential Search adalah algoritma pencarian sederhana yang digunakan untuk mencari elemen tertentu dalam urutan linier, seperti array atau daftar. Algoritma ini bekerja dengan memeriksa setiap elemen secara berurutan hingga menemukan elemen yang dicari atau mencapai akhir urutan.

Langkah-langkah dalam Sequential Search:
- Mulai pencarian dari elemen pertama dalam urutan.
- Bandingkan elemen tersebut dengan elemen yang dicari.
- Jika elemen tersebut cocok, pencarian dihentikan dan elemen ditemukan.
- Jika elemen tidak cocok, lanjutkan pencarian ke elemen berikutnya.
- Ulangi langkah-langkah di atas hingga menemukan elemen yang dicari atau mencapai akhir urutan.

Sequential Search memiliki kompleksitas waktu terburuk O(n), di mana n adalah jumlah elemen dalam urutan. Algoritma ini sederhana dan mudah diimplementasikan, tetapi tidak efisien untuk urutan dengan jumlah elemen yang besar.

Brute-Force String Matching (Pencocokan String secara Brute-Force):
Brute-Force String Matching adalah algoritma pencocokan string yang sederhana dan langsung. Algoritma ini digunakan untuk mencari kemunculan pola atau substring tertentu dalam sebuah string. Brute-Force String Matching bekerja dengan memeriksa setiap kemungkinan posisi pemulaan dalam string utama dan membandingkannya dengan pola yang dicari. Jika ada kesamaan lengkap antara pola dan substring dalam string utama, maka pola ditemukan.

Langkah-langkah dalam Brute-Force String Matching:
- Mulai pemindaian dari posisi awal dalam string utama.
- Bandingkan setiap karakter dari pola dengan karakter yang sesuai dalam substring pada posisi saat ini.
- Jika semua karakter cocok, pola ditemukan pada posisi saat ini.
- Jika ada ketidakcocokan, geser posisi pemindaian ke kanan dan ulangi langkah-langkah di atas.
- Ulangi langkah-langkah di atas hingga menemukan semua kemunculan pola atau mencapai akhir string utama.

Brute-Force String Matching memiliki kompleksitas waktu terburuk O(m * n), di mana m adalah panjang pola dan n adalah panjang string utama. Algoritma ini sederhana dan mudah diimplementasikan, tetapi juga bisa menjadi tidak efisien untuk kasus dengan pola dan string utama yang sangat panjang.

Keduanya, Sequential Search dan Brute-Force String Matching, adalah algoritma pencarian sederhana yang sesuai untuk digunakan dalam kasus dengan ukuran input yang kecil atau ketika kompleksitas waktu bukanlah faktor yang kritis. Untuk masalah pencarian dengan ukuran input yang besar, algoritma yang lebih efisien seperti Binary Search atau KMP (Knuth-Morris-Pratt) Algorithm untuk pencarian string, direkomendasikan.

C. Closest -Pair and Convex -Hull Problems

Closest Pair Problem (Masalah Pasangan Terdekat):
Masalah Pasangan Terdekat adalah masalah dalam komputasi geometri yang melibatkan mencari dua titik terdekat dalam himpunan titik dalam ruang Euclidean. Tujuan dari masalah ini adalah untuk menemukan pasangan titik yang memiliki jarak terkecil di antara semua pasangan titik yang mungkin.

Untuk menyelesaikan masalah Pasangan Terdekat, dapat digunakan pendekatan berbasis pemilahan (sorting) seperti Algoritma Divide and Conquer. Pendekatan umum untuk masalah ini adalah sebagai berikut:
1. Urutkan semua titik berdasarkan koordinat x.
2. Pisahkan himpunan titik menjadi dua bagian hingga mencapai titik tengah.
3. Cari pasangan titik terdekat di setiap bagian secara rekursif.
4. Temukan pasangan titik terdekat antara dua bagian dan cari pasangan titik terdekat di sekitar garis batas dengan lebar 2 kali jarak terdekat yang sudah ditemukan.
5. Kembalikan pasangan titik terdekat.

Kompleksitas waktu algoritma ini adalah O(n log n), di mana n adalah jumlah titik dalam himpunan.

Convex Hull Problem (Masalah Cangkang Cembung):
Masalah Cangkang Cembung melibatkan mencari cangkang cembung terkecil yang meliputi semua titik dalam himpunan titik dalam ruang Euclidean. Cangkang cembung adalah poligon dengan sisi-sisi lurus yang meliputi semua titik dalam himpunan, dengan sisi poligon tidak ada yang menyimpang ke dalam.

Pendekatan umum untuk masalah Cangkang Cembung adalah dengan menggunakan Algoritma Graham Scan atau Algoritma Jarvis. Pendekatan umum menggunakan Algoritma Graham Scan sebagai berikut:
1. Pilih titik paling bawah sebagai titik awal.
2. Urutkan semua titik berdasarkan sudut polar terhadap titik awal.
3. Mulai dengan titik pertama dalam urutan, dan lakukan iterasi untuk setiap titik berikutnya.
4. Jika tiga titik terakhir membentuk tikungan berlawanan arah (searah jarum jam atau berlawanan arah jarum jam), hapus titik terakhir dari cangkang cembung.
5. Tambahkan titik saat ini ke cangkang cembung.
6. Ulangi langkah-langkah 4-5 hingga kembali ke titik awal.

Kompleksitas waktu algoritma ini adalah O(n log n), di mana n adalah jumlah titik dalam himpunan.

Kedua masalah tersebut, Pasangan Terdekat dan Cangkang Cembung, adalah masalah yang lebih kompleks dalam komputasi geometri. Untuk menyelesaikan masalah ini secara efisien, diperlukan algoritma yang sesuai dan dapat menghasilkan solusi yang akurat dengan kompleksitas waktu yang masuk akal.

D. Exhaustive Search

Exhaustive Search (Pencarian Eksaustif), juga dikenal sebagai Brute-Force Search, adalah pendekatan dalam pemecahan masalah yang mencoba semua kemungkinan solusi secara sistematis untuk menemukan solusi optimal. Pendekatan ini melibatkan mencoba semua kombinasi atau permutasi solusi yang mungkin, dan memeriksa satu per satu apakah setiap solusi memenuhi kriteria yang diinginkan.

Langkah-langkah dalam Exhaustive Search:
1. Tentukan ruang solusi yang mungkin, yaitu kumpulan semua solusi yang mungkin.
2. Buat metode atau fungsi yang akan memeriksa apakah suatu solusi memenuhi kriteria yang diinginkan.
3. Gunakan nested loop, rekursi, atau kombinasi dari keduanya untuk mencoba semua kemungkinan solusi.
4. Saat mencoba setiap solusi, periksa apakah solusi memenuhi kriteria yang diinginkan.
5. Jika ditemukan solusi yang memenuhi kriteria, gunakan solusi tersebut sebagai jawaban.
6. Ulangi langkah-langkah 4-5 hingga semua kemungkinan solusi telah dicoba.
7. Jika tidak ada solusi yang memenuhi kriteria, berikan jawaban yang sesuai, misalnya "Tidak ada solusi yang ditemukan".

Pendekatan Pencarian Eksaustif ini dapat digunakan untuk memecahkan berbagai masalah, terutama ketika ukuran ruang solusi tidak terlalu besar dan tidak ada struktur khusus yang dapat dimanfaatkan. Namun, pendekatan ini memiliki kompleksitas waktu yang tinggi karena mencoba semua kemungkinan solusi. Oleh karena itu, dalam kasus ruang solusi yang besar, pendekatan ini mungkin tidak efisien dan memerlukan waktu yang lama untuk menemukan solusi optimal.

Penting untuk mencatat bahwa Pencarian Eksaustif dapat menjadi solusi yang baik dalam beberapa kasus, terutama ketika tidak ada pendekatan yang lebih efisien yang tersedia atau ketika ukuran ruang solusi masih dapat ditangani secara wajar. Namun, jika terdapat pendekatan lain yang lebih efisien, seperti algoritma khusus atau teknik optimisasi, disarankan untuk digunakan untuk mempercepat proses pemecahan masalah.

E. Depth-First Search and Breath -First Search

Depth-First Search (Pencarian Terdalam):Depth-First Search (DFS) adalah algoritma pencarian yang digunakan untuk melintasi atau mencari informasi dalam struktur data berupa graf. Algoritma ini menjelajahi setiap cabang secara terdalam sebelum melanjutkan ke cabang lainnya. DFS menggunakan pendekatan "mendalam" untuk menjelajahi graf dan mencapai titik terminasi tertentu.

Langkah-langkah dalam Depth-First Search:

1. Mulai dari simpul awal atau simpul sumber.

2. Periksa simpul saat ini, tandai sebagai dikunjungi.

3. Jika simpul saat ini memiliki simpul tetangga yang belum dikunjungi, pilih satu simpul tetangga yang belum dikunjungi dan pergi ke simpul tersebut.

4. Ulangi langkah 2-3 untuk simpul tetangga yang baru dikunjungi.

5. Jika simpul saat ini tidak memiliki simpul tetangga yang belum dikunjungi, kembali ke simpul sebelumnya dan periksa simpul tetangga lainnya yang belum dikunjungi.

6. Ulangi langkah 2-5 hingga mencapai titik terminasi yang diinginkan atau telah mengunjungi semua simpul yang mungkin.

Depth-First Search dapat dilakukan secara rekursif atau menggunakan tumpukan (stack) untuk melacak simpul yang akan dikunjungi berikutnya. Algoritma ini berguna untuk mencari jalan melalui graf, menemukan siklus, atau mencari solusi dalam permasalahan pencarian.

Breath-First Search (Pencarian Terlebar): Breath First Search (BFS) adalah algoritma pencarian yang digunakan untuk melintasi atau mencari informasi dalam struktur data berupa graf. Algoritma ini menjelajahi setiap tingkat graf secara bertahap sebelum melanjutkan ke tingkat berikutnya. BFS menggunakan pendekatan "terlebar" untuk menjelajahi graf dan mencapai titik terminasi tertentu.

Langkah-langkah dalam Breath-First Search:

1. Mulai dari simpul awal atau simpul sumber.

2. Tandai simpul saat ini sebagai dikunjungi dan tambahkan ke antrian.

3. Selama antrian tidak kosong, ambil simpul pertama dari antrian.

4. Periksa simpul tersebut dan tambahkan semua simpul tetangga yang belum dikunjungi ke antrian.

5. Tandai simpul tetangga yang baru ditambahkan sebagai dikunjungi.

6. Ulangi langkah 3-5 hingga mencapai titik terminasi yang diinginkan atau telah mengunjungi semua simpul yang mungkin.

Breath-First Search menggunakan struktur data antrian untuk melacak simpul yang akan dikunjungi berikutnya. Algoritma ini berguna untuk mencari jarak terpendek antara dua simpul dalam graf, menemukan jalur optimal, atau melakukan pemrosesan berbasis tingkat pada graf.

Baik Depth-First Search maupun Breath-First Search memiliki berbagai aplikasi dalam pemecahan masalah, terutama dalam analisis graf, pemodelan jaringan, pengenalan pola, dan kecerdasan buatan. Pilihan antara DFS dan BFS tergantung pada tujuan pencarian dan struktur data yang digunakan.

BAB IV

DECREASE & CONQUER

A. Three Major Varian of Decrease-andConquer

Terdapat tiga varian utama dari paradigma desain algoritma decrease-and-conquer dalam bahasa Indonesia. Varian-varian ini melibatkan pengurangan ukuran masalah dengan cara yang berbeda untuk akhirnya menyelesaikan masalah tersebut.

- Pengurangan dengan Faktor Konstan:
  Pada varian ini, masalah dikurangi dengan faktor konstan pada setiap langkah rekursif. Algoritma menyelesaikan submasalah yang lebih kecil dengan ukuran n/c, di mana c adalah konstanta yang lebih besar dari 1. Pengurangan ini dilakukan hingga mencapai kasus dasar, di mana ukuran masalah menjadi cukup kecil untuk langsung diselesaikan. Contoh algoritma yang mengikuti varian ini termasuk pencarian biner dan pengurutan gabung (merge sort).

- Pengurangan dengan Faktor Variabel:
  Pada varian ini, ukuran masalah dikurangi dengan faktor variabel pada setiap langkah rekursif. Algoritma menyelesaikan submasalah dengan ukuran n/k, di mana k adalah variabel yang bergantung pada instansi masalah atau input. Nilai k dapat bervariasi berdasarkan karakteristik dari masalah yang sedang diselesaikan. Varian ini sering digunakan dalam algoritma seperti quicksort, di mana pemilihan pivot menentukan ukuran submasalah.

- Pengurangan dengan Jumlah Konstan:
  Pada varian ini, ukuran masalah dikurangi dengan jumlah konstan pada setiap langkah rekursif. Algoritma menyelesaikan submasalah yang lebih kecil dengan ukuran n-d, di mana d adalah konstanta. Pengurangan ini terus dilakukan hingga mencapai kasus dasar, di mana ukuran masalah menjadi cukup kecil untuk langsung diselesaikan. Contoh algoritma yang mengikuti varian ini termasuk selection sort dan insertion sort.

  Varian-varian algoritma decrease-and-conquer ini ditandai dengan cara spesifik mereka dalam mengurangi ukuran masalah dan menyelesaikan submasalah. Dengan secara iteratif menyelesaikan instansi masalah yang lebih kecil, algoritma tersebut akhirnya mencapai solusi dari masalah asli.

B. Sort

Pengurutan (sort) adalah proses mengatur elemen-elemen data dalam urutan tertentu, seperti secara menaik (ascending) atau menurun (descending), berdasarkan suatu kunci atau kriteria tertentu. Pengurutan merupakan salah satu operasi dasar dalam pemrograman dan sering digunakan dalam berbagai aplikasi.

Terdapat berbagai algoritma pengurutan yang dapat digunakan, masing-masing memiliki kelebihan dan kelemahan dalam hal waktu eksekusi, kebutuhan memori, dan kompleksitasnya. Berikut ini beberapa algoritma pengurutan yang umum digunakan:

- Selection Sort (Pengurutan Pilihan):Algoritma ini secara berulang mencari elemen terkecil dari sisa array dan menukar posisinya dengan elemen pertama yang belum terurut. Proses ini terus berlanjut hingga seluruh array terurut. Selection sort

memiliki kompleksitas waktu O(n^2) dan cocok untuk digunakan pada array dengan ukuran kecil atau ketika kebutuhan memori rendah.

- Insertion Sort (Pengurutan Sisip):Algoritma ini secara berulang membandingkan setiap elemen dengan elemen-elemen sebelumnya dalam array terurut dan menyisipkannya ke posisi yang tepat. Insertion sort juga memiliki kompleksitas waktu O(n^2), tetapi lebih efisien daripada selection sort dalam kasus terbaik ketika array sudah hampir terurut.

- Bubble Sort (Pengurutan Gelembung):Algoritma ini berulang kali membandingkan pasangan elemen berturut-turut dan menukar posisi mereka jika urutannya salah. Proses ini terus berlanjut hingga seluruh array terurut. Bubble sort juga memiliki kompleksitas waktu O(n^2) dan umumnya digunakan pada array dengan ukuran kecil atau pada kasus yang hampir terurut.

- Merge Sort (Pengurutan Gabung):Algoritma divide-and-conquer yang membagi array menjadi subarray yang lebih kecil, mengurutkan masing-masing subarray secara rekursif, dan menggabungkannya kembali untuk mendapatkan array terurut. Merge sort memiliki kompleksitas waktu O(n log n), yang membuatnya efisien untuk array dengan ukuran besar.

- Quick Sort (Pengurutan Cepat):Algoritma divide-and-conquer yang menggunakan pendekatan pemilihan elemen pivot dan membagi array menjadi dua subarray berdasarkan pivot tersebut. Subarray kemudian diurutkan secara rekursif. Quick sort memiliki kompleksitas waktu rata-rata O(n log n), tetapi dapat mencapai O(n^2) pada kasus terburuk. Namun, quick sort sering kali lebih cepat dalam praktiknya dibandingkan dengan algoritma pengurutan lainnya.

Selain algoritma-algoritma di atas, ada juga algoritma pengurutan lainnya seperti heap sort, radix sort, dan tim sort. Pemilihan algoritma pengurutan yang tepat tergantung pada sifat data, ukuran masalah, dan kebutuhan performa yang diinginkan.

C. Topological Sorting

Topological Sorting (Pengurutan Topologis) adalah proses mengurutkan simpul-simpul dalam suatu graf berarah (directed graph) sedemikian rupa sehingga setiap busur (edge) mengarah dari simpul sebelumnya ke simpul sesudahnya. Topological Sorting umumnya digunakan pada graf yang merepresentasikan ketergantungan antara tugas atau kegiatan.

Langkah-langkah dalam melakukan Topological Sorting adalah sebagai berikut:

1. Tentukan graf berarah yang akan diurutkan. Graf ini terdiri dari simpul-simpul (nodes) yang mewakili tugas atau kegiatan, dan busur-busur (edges) yang menghubungkan antara simpul-simpul tersebut.

2. Cari simpul yang tidak memiliki busur masuk (in-degree) yaitu simpul yang tidak memiliki ketergantungan terhadap simpul lain. Simpul ini akan menjadi simpul awal dalam urutan topologis.

3. Letakkan simpul awal ke dalam hasil urutan topologis dan hapus simpul tersebut beserta semua busur yang keluar darinya dari graf.

4. Ulangi langkah 2 dan 3 untuk semua simpul yang tersisa, tetapi kali ini mencari simpul berikutnya yang tidak memiliki ketergantungan dengan simpul yang telah diproses sebelumnya.

5. Terus ulangi langkah 2 dan 3 hingga semua simpul telah dimasukkan ke dalam hasil urutan topologis.

Hasil akhir dari Topological Sorting adalah urutan topologis, yaitu urutan linier dari simpul-simpul graf yang memenuhi semua ketergantungan yang ada dalam graf tersebut.

Topological Sorting memiliki berbagai aplikasi, terutama dalam perencanaan proyek, analisis jaringan, perutean (routing), pemecahan masalah yang melibatkan urutan tugas atau kegiatan, serta dalam pemrosesan bahasa alami dan analisis sintaksis

BAB V

DEVIED & CONQUER

A. Mergesort

Mergesort adalah salah satu algoritma pengurutan yang menggunakan pendekatan divide-and-conquer. Algoritma ini membagi array yang akan diurutkan menjadi dua bagian yang lebih kecil, mengurutkan masing-masing bagian secara rekursif, dan kemudian menggabungkan kedua bagian tersebut untuk mendapatkan array terurut.

Berikut adalah langkah-langkah dalam algoritma Mergesort:

- Divide: Bagi array menjadi dua bagian yang sama besar. Ini dilakukan dengan mencari titik tengah array.

- Conquer: Urutkan kedua bagian array secara rekursif menggunakan mergesort.

- Combine: Gabungkan kedua bagian yang sudah diurutkan menjadi satu array terurut. Proses ini melibatkan membandingkan elemen-elemen dari kedua bagian dan menempatkannya ke dalam array hasil.

- Ulangi langkah-langkah 1-3 sampai seluruh array terurut.

  Mergesort memiliki kompleksitas waktu yang stabil dan seimbang, yaitu O(n log n), di mana n adalah ukuran array yang akan diurutkan. Hal ini membuat mergesort efisien dalam mengurutkan array dengan ukuran besar. Namun, mergesort juga membutuhkan penggunaan memori tambahan untuk menyimpan array sementara saat proses penggabungan.

B. Quicksort

Quicksort (Pengurutan Cepat) adalah algoritma pengurutan yang menggunakan pendekatan divide-and-conquer. Algoritma ini memilih sebuah elemen pivot dari array dan membagi array menjadi dua bagian berdasarkan nilai pivot tersebut. Bagian kiri dari pivot berisi elemen-elemen yang lebih kecil daripada pivot, sedangkan bagian kanan berisi elemen-elemen yang lebih besar. Setelah itu, kedua bagian tersebut diurutkan secara rekursif menggunakan algoritma quicksort.

Berikut adalah langkah-langkah dalam algoritma Quicksort:

- Pilih Pivot: Pilih sebuah elemen pivot dari array yang akan diurutkan. Elemen pivot dapat dipilih secara acak atau dengan menggunakan strategi tertentu, seperti memilih elemen di tengah array atau elemen pertama/terakhir.

- Partitioning: Bagi array menjadi dua bagian, yaitu bagian yang memiliki elemen yang lebih kecil daripada pivot (bagian kiri) dan bagian yang memiliki elemen yang lebih besar daripada pivot (bagian kanan). Biasanya,

langkah ini melibatkan penggunaan dua pointer, yaitu pointer i dan pointer j. Pointer i akan bergerak dari kiri ke kanan untuk mencari elemen yang lebih besar dari pivot, sedangkan pointer j akan bergerak dari kanan ke kiri untuk mencari elemen yang lebih kecil dari pivot. Jika ditemukan pasangan elemen yang tidak sesuai, yaitu elemen yang lebih besar di bagian kiri dan elemen yang lebih kecil di bagian kanan, kedua elemen tersebut akan ditukar posisinya.

- Rekursif: Terapkan langkah-langkah 1 dan 2 secara rekursif pada kedua bagian array yang dihasilkan. Panggil algoritma quicksort untuk bagian kiri (elemen yang lebih kecil) dan bagian kanan (elemen yang lebih besar).

- Combine: Setelah kedua bagian array terurut, gabungkan hasilnya menjadi satu array terurut. Hal ini dapat dilakukan dengan menggabungkan bagian kiri, elemen pivot, dan bagian kanan.

  Quicksort memiliki kompleksitas waktu rata-rata O(n log n), di mana n adalah ukuran array yang akan diurutkan. Namun, dalam kasus terburuk (misalnya jika pivot yang dipilih selalu merupakan elemen terbesar atau terkecil), kompleksitas waktu dapat mencapai O(n^2). Untuk menghindari kasus terburuk tersebut, dapat digunakan beberapa strategi dalam pemilihan elemen pivot, seperti memilih pivot secara acak atau menggunakan teknik median-of-three pivot. Quicksort juga memiliki keunggulan dalam penggunaan memori yang efisien.

C. Binary Tree Traversals and Related Properties

Binary Tree Traversal adalah proses mengunjungi (melihat) setiap simpul dalam pohon biner tepat satu kali dengan urutan tertentu. Ada tiga metode utama untuk melakukan binary tree traversal:

1. Inorder Traversal:

- Langkah 1: Traversing (mengunjungi) simpul kiri secara rekursif.

- Langkah 2: Mengunjungi simpul saat ini.

- Langkah 3: Traversing (mengunjungi) simpul kanan secara rekursif.

Urutan kunjungan dalam inorder traversal adalah kiri-akar-kanan. Dalam pohon biner, ini menghasilkan urutan data yang terurut secara menaik.

2. Preorder Traversal:

- Langkah 1: Mengunjungi simpul saat ini.

- Langkah 2: Traversing (mengunjungi) simpul kiri secara rekursif.

- Langkah 3: Traversing (mengunjungi) simpul kanan secara rekursif.

Urutan kunjungan dalam preorder traversal adalah akar-kiri-kanan. Dalam pohon biner, ini menghasilkan urutan data yang serupa dengan struktur pohon.

3. Postorder Traversal:

- Langkah 1: Traversing (mengunjungi) simpul kiri secara rekursif.

- Langkah 2: Traversing (mengunjungi) simpul kanan secara rekursif.

- Langkah 3: Mengunjungi simpul saat ini.

Urutan kunjungan dalam postorder traversal adalah kiri-kanan-akar. Dalam pohon biner, ini menghasilkan urutan data yang mirip dengan urutan penempatan simpul dalam pohon.

Selain traversal, ada beberapa sifat terkait dalam binary tree:

1. Tinggi Pohon (Height of Tree): Tinggi pohon biner adalah jumlah maksimum simpul yang harus dilewati dari akar ke salah satu daun terjauh. Tinggi pohon dapat dihitung dengan menggunakan rekursi, di mana tinggi setiap subpohon kiri dan kanan dihitung dan tinggi maksimumnya ditambahkan dengan 1.

2. Kedalaman Simpul (Depth of Node): Kedalaman simpul adalah jumlah simpul yang harus dilewati dari akar hingga simpul tersebut. Kedalaman simpul dapat dihitung dengan menggunakan rekursi, di mana kedalaman simpul induk ditambahkan dengan 1.

3. Jumlah Simpul (Number of Nodes): Jumlah simpul dalam pohon biner adalah total jumlah simpul di seluruh pohon, termasuk akar, simpul internal, dan daun.

4. Jumlah Daun (Number of Leaves): Jumlah daun dalam pohon biner adalah total jumlah simpul daun (simpul yang tidak memiliki anak).

Traversal dan sifat-sifat ini digunakan secara luas dalam pemrosesan dan analisis pohon biner dalam berbagai aplikasi seperti struktur data, algoritma pencarian, komputasi grafis, kecerdasan buatan, dan banyak lagi.

BAB VI

TRANSFORM & CONQUER

A. Instance Simplification

Instance Simplification (Pemudahan Instansi) adalah teknik yang digunakan dalam analisis algoritma untuk mengurangi kompleksitas masalah dengan mengubah atau menyederhanakan instansi masalah yang diberikan menjadi instansi yang lebih mudah dipecahkan.

Tujuan dari instance simplification adalah memperoleh pemahaman yang lebih baik tentang sifat masalah dan mencari solusi yang lebih efisien. Dengan menyederhanakan instansi masalah, kita dapat menghilangkan beberapa aspek yang tidak relevan atau memperkecil ukuran masalah sehingga memungkinkan penggunaan algoritma yang lebih efisien.

Proses instance simplification dapat melibatkan beberapa strategi, antara lain:

- Menghilangkan Informasi yang Tidak Relevan: Kadang-kadang, instansi masalah mengandung informasi yang tidak diperlukan dalam pencarian solusi. Dalam hal ini, langkah pertama adalah mengidentifikasi informasi yang tidak relevan dan menghapusnya untuk mengurangi kompleksitas masalah.

- Menyederhanakan Kasus Khusus: Beberapa masalah memiliki kasus khusus yang dapat disederhanakan secara terpisah. Dengan mengidentifikasi kasus khusus dan menyederhanakannya, kita dapat mengurangi kompleksitas keseluruhan masalah.

- Mengganti Representasi Masalah: Dalam beberapa kasus, mengubah representasi masalah menjadi bentuk yang lebih sederhana atau lebih efisien dapat membantu dalam analisis dan pemecahan masalah. Representasi masalah yang baru mungkin lebih mudah dipahami atau lebih mudah dipecahkan menggunakan algoritma tertentu.

- Memperkecil Ukuran Instansi Masalah: Jika masalah terdiri dari banyak entitas atau objek, mungkin memungkinkan untuk memperkecil ukuran masalah dengan mempertimbangkan hanya sebagian dari entitas tersebut atau dengan menggabungkan beberapa entitas yang setara menjadi satu entitas.

Penerapan instance simplification dapat membantu meningkatkan efisiensi dalam analisis algoritma dan pemecahan masalah. Dengan menyederhanakan instansi masalah, kita dapat mengurangi kompleksitas waktu atau ruang yang diperlukan untuk mencari solusi yang optimal atau memahami karakteristik masalah dengan lebih baik.

B. Representation Change

Representation Change (Perubahan Representasi) adalah teknik yang digunakan dalam analisis algoritma untuk mengubah cara data atau masalah direpresentasikan. Tujuan dari perubahan representasi adalah untuk memperoleh pemahaman yang lebih baik tentang masalah dan mencari solusi yang lebih efisien.

Dalam beberapa kasus, representasi asli dari data atau masalah mungkin tidak cocok untuk penggunaan algoritma tertentu atau mungkin menghasilkan kompleksitas yang tinggi. Dalam hal ini, perubahan representasi dapat membantu memperkecil kompleksitas masalah atau membuatnya lebih mudah untuk dipecahkan.

Beberapa strategi yang dapat digunakan dalam perubahan representasi meliputi:

- Struktur Data yang Berbeda: Mengubah struktur data yang digunakan untuk merepresentasikan masalah dapat memiliki dampak signifikan pada efisiensi algoritma. Misalnya, mengubah array menjadi struktur data seperti linked list, stack, atau queue dapat memungkinkan operasi yang lebih efisien.

- Encoding atau Komprimasi Data: Jika masalah melibatkan representasi data yang berulang atau memiliki pola tertentu, encoding atau kompresi data dapat digunakan untuk mengurangi ukuran data yang diperlukan. Ini dapat mengurangi kompleksitas ruang yang diperlukan dan mempercepat operasi pada data yang dikodekan atau dikompresi.

- Representasi Matematis: Dalam beberapa kasus, masalah dapat diubah menjadi bentuk matematis yang lebih sederhana atau lebih mudah untuk dianalisis. Misalnya, memodelkan masalah sebagai graf atau matriks dapat membantu menerapkan algoritma graf atau operasi matriks yang efisien.

- Representasi Berbasis Aturan: Beberapa masalah kompleks dapat dipecahkan dengan merumuskan aturan atau konstrain yang lebih sederhana. Dengan memperumum masalah dan mengidentifikasi aturan yang membatasi solusi yang mungkin, kompleksitas masalah dapat berkurang.

Perubahan representasi dapat membantu dalam analisis algoritma, pemecahan masalah, dan optimisasi performa. Dengan memilih representasi yang lebih sesuai dan efisien, kita dapat mengurangi kompleksitas waktu dan ruang yang diperlukan untuk mencari solusi dan meningkatkan efisiensi algoritma yang digunakan.

C. Problem Reduction

Problem Reduction (Reduksi Masalah) adalah teknik yang digunakan dalam pemecahan masalah untuk mengurangi kompleksitas atau kesulitan suatu masalah dengan mengubahnya menjadi masalah yang lebih sederhana atau sudah diketahui solusinya. Tujuan utama dari reduksi masalah adalah untuk

mengidentifikasi hubungan antara masalah yang sulit dan masalah yang lebih mudah, sehingga solusi yang ada untuk masalah yang lebih mudah dapat diterapkan pada masalah yang sulit.

Proses reduksi masalah melibatkan dua langkah utama:

1. Reduksi dari Masalah Target ke Masalah Referensi: Pertama, masalah yang sulit atau kompleks (masalah target) diubah menjadi masalah yang lebih sederhana atau sudah diketahui solusinya (masalah referensi). Ini dilakukan dengan mengidentifikasi kemiripan atau kesamaan antara masalah target dan masalah referensi. Dalam beberapa kasus, masalah referensi dapat menjadi varian khusus dari masalah target.

2. Penerapan Solusi Masalah Referensi ke Masalah Target: Setelah masalah target direduksi menjadi masalah referensi, solusi yang ada atau algoritma yang diketahui untuk masalah referensi dapat diterapkan pada masalah target. Ini memanfaatkan kesamaan atau korelasi antara dua masalah untuk menghasilkan solusi untuk masalah yang sulit.

Reduksi masalah adalah teknik yang penting dalam pemecahan masalah kompleks, dan digunakan dalam berbagai bidang seperti teori kompleksitas, kecerdasan buatan, dan optimisasi kombinatorial. Ini memungkinkan pemecahan masalah yang lebih efisien dan memperluas ruang solusi dengan memanfaatkan pengetahuan dan solusi yang sudah ada. Dalam beberapa kasus, reduksi masalah juga dapat digunakan untuk membuktikan sifat-sifat tertentu tentang kekerasan atau kesulitan suatu masalah.

BAB VII

SPACE & TIME TRADE -OFFS

A. Sorting by Counting

Sorting by Counting (Pengurutan dengan Menghitung) adalah algoritma pengurutan yang digunakan untuk mengurutkan sebuah rangkaian data dengan rentang nilai terbatas. Algoritma ini bekerja dengan menghitung frekuensi kemunculan setiap elemen dalam rangkaian data, kemudian menggunakan informasi tersebut untuk membangun rangkaian data yang terurut.

Berikut adalah langkah-langkah umum dalam algoritma Sorting by Counting:

- Menentukan rentang nilai: Pertama, kita perlu menentukan rentang nilai yang terdapat dalam rangkaian data yang akan diurutkan. Rentang nilai ini digunakan untuk menentukan ukuran array bantu yang akan digunakan dalam algoritma.

- Menghitung frekuensi kemunculan: Selanjutnya, kita menghitung frekuensi kemunculan setiap elemen dalam rangkaian data. Untuk setiap elemen, kita meningkatkan nilai frekuensinya di array bantu yang sesuai dengan indeks yang mencerminkan nilai elemen tersebut.

- Mengakumulasikan frekuensi: Setelah menghitung frekuensi kemunculan, kita mengakumulasikan frekuensi tersebut. Ini dilakukan dengan menjumlahkan setiap elemen array bantu dengan elemen sebelumnya. Tujuan dari langkah ini adalah untuk menentukan posisi awal setiap elemen dalam rangkaian data terurut.

- Membangun rangkaian data terurut: Setelah mengakumulasikan frekuensi, kita menggunakan informasi ini untuk membangun rangkaian data yang terurut. Kita mulai dari elemen terakhir dalam rangkaian data asli dan menempatkannya pada posisi yang sesuai dalam rangkaian data terurut. Setelah itu, kita mengurangi frekuensi elemen tersebut di array bantu. Proses ini diulang untuk setiap elemen dalam rangkaian data, sehingga menghasilkan rangkaian data terurut.

Algoritma Sorting by Counting memiliki kompleksitas waktu yang bergantung pada rentang nilai dalam rangkaian data. Jika rentang nilai terbatas, algoritma ini dapat menjadi sangat efisien dengan kompleksitas waktu linier O(n), di mana n adalah jumlah elemen dalam rangkaian data. Namun, algoritma ini membutuhkan penggunaan array bantu dengan ukuran sesuai dengan rentang nilai, sehingga membutuhkan ruang tambahan.

Algoritma Sorting by Counting sangat berguna ketika kita memiliki rangkaian data dengan rentang nilai terbatas, seperti ketika mengurutkan data non-negatif atau data dengan nilai-nilai diskrit.

B. Input Enhancement in String Matching

Input Enhancement (Peningkatan Input) dalam String Matching adalah teknik yang digunakan untuk meningkatkan efisiensi atau kinerja algoritma pencocokan string dengan memanipulasi atau memodifikasi input yang akan dicocokkan. Tujuan dari peningkatan input adalah untuk mengurangi jumlah operasi pencocokan yang perlu dilakukan oleh algoritma dan mempercepat proses pencocokan.

Beberapa teknik umum yang digunakan dalam peningkatan input dalam string matching adalah sebagai berikut:

- Preprocessing: Teknik ini melibatkan pengolahan atau persiapan awal terhadap input yang akan dicocokkan sebelum proses pencocokan dimulai. Contohnya, dapat dilakukan pengindeksan atau pembangunan struktur data seperti tabel hash atau tabel sufiks untuk mempercepat pencocokan. Preprocessing ini dilakukan satu kali sebelum pencocokan dilakukan.

- Normalisasi Input: Normalisasi input melibatkan mengubah input ke dalam bentuk yang lebih terstruktur atau standar. Ini dapat mencakup penghapusan karakter non-alfanumerik, penggantian huruf besar ke huruf kecil, atau penghilangan spasi yang tidak relevan. Dengan normalisasi input, kemungkinan kecocokan yang relevan dapat ditingkatkan dan mengurangi kompleksitas pencocokan.

- Penggunaan Indeks atau Struktur Data Khusus: Menggunakan indeks atau struktur data khusus, seperti trie (pohon pencarian), dapat meningkatkan efisiensi pencocokan string. Indeks atau struktur data ini memungkinkan pencocokan berbasis indeks yang lebih cepat dan efisien daripada pencocokan karakter per karakter.

- Praproses Pencocokan: Praproses pencocokan melibatkan melakukan beberapa operasi pencocokan sebelum mencocokkan input sebenarnya. Contohnya, dapat dilakukan pencocokan kasus khusus terlebih dahulu untuk menghindari pencocokan karakter yang lebih kompleks atau memanfaatkan pola yang sudah diketahui sebelumnya.

- Pembatasan Pencarian: Dalam beberapa kasus, pembatasan pencarian dapat digunakan untuk membatasi ruang pencarian dan mengurangi jumlah operasi pencocokan yang perlu dilakukan. Misalnya, dengan membatasi pencarian hanya pada subset input yang relevan atau menggunakan teknik seperti pencocokan aproksimasi atau pencarian dengan jarak Levenshtein yang terbatas.

Peningkatan input dalam string matching dapat membantu meningkatkan efisiensi dan kinerja algoritma pencocokan string. Dengan mengoptimalkan input sebelum pencocokan atau dengan menggunakan struktur data dan teknik yang sesuai, jumlah operasi pencocokan dapat dikurangi, sehingga meningkatkan kecepatan dan efisiensi pencocokan.

C. Hasing

Hashing (Penghasingan) adalah teknik yang digunakan untuk mengonversi data input menjadi nilai hash yang unik dan dapat digunakan untuk identifikasi atau pencarian data dengan efisien. Proses hashing melibatkan penggunaan fungsi hash yang mengambil data input sebagai masukan dan menghasilkan nilai hash yang berukuran tetap.

Fungsi hash bertujuan untuk menghasilkan nilai hash yang unik untuk setiap data input yang berbeda. Idealnya, fungsi hash harus menghasilkan nilai hash yang berbeda untuk setiap data input yang berbeda, dan sebaiknya tidak menghasilkan nilai hash yang sama untuk data input yang berbeda. Namun, terkadang terjadi situasi di mana dua data input berbeda menghasilkan nilai hash yang sama, yang disebut sebagai "tabrakan hash" atau "collision". Dalam desain fungsi hash yang baik, tabrakan hash harus dihindari sebisa mungkin.

Hashing digunakan dalam berbagai aplikasi, termasuk:

1. Penyimpanan dan Pemulihan Data: Dalam struktur data seperti tabel hash atau pohon hash, hashing digunakan untuk menyimpan data dan memulihkannya dengan cepat. Nilai hash digunakan sebagai kunci untuk memetakan data ke posisi penyimpanan yang sesuai.

2. Identifikasi dan Pencocokan: Hashing digunakan untuk mengidentifikasi atau mencocokkan data dengan cepat. Nilai hash digunakan sebagai identitas unik atau tanda tangan data, yang memungkinkan pencocokan atau pencarian dengan waktu yang konstan.

3. Keamanan dan Enkripsi: Hashing digunakan dalam algoritma keamanan dan enkripsi untuk mengamankan data. Nilai hash yang unik digunakan untuk memverifikasi integritas data atau untuk mengenkripsi dan memverifikasi kata sandi.

Beberapa algoritma hashing yang umum digunakan adalah MD5 (Message Digest Algorithm 5), SHA-1 (Secure Hash Algorithm 1), SHA-256, dan CRC32 (Cyclic Redundancy Check). Algoritma hashing yang dipilih tergantung pada kebutuhan spesifik aplikasi dan tingkat keamanan yang diinginkan.

Hashing adalah teknik penting dalam pengolahan data dan keamanan informasi. Dengan menggunakan fungsi hash yang baik dan struktur data yang tepat, hashing dapat memberikan kinerja yang cepat dan efisien dalam identifikasi, pencocokan, penyimpanan, dan keamanan data.

BAB VIII

DYNAMIC PROGRAMMING

A. Three Basic

Tiga teknik dasar dalam algoritma komputer adalah:

1. Sequential Search (Pencarian Berurutan): Teknik ini melibatkan pencarian secara berurutan satu per satu dari awal hingga akhir elemen dalam rangkaian data. Pada setiap langkah, elemen yang sedang dicari dibandingkan dengan elemen saat ini dalam urutan. Jika ada kesamaan, elemen ditemukan. Jika tidak, pencarian berlanjut hingga seluruh rangkaian data diperiksa. Sequential search sederhana namun memiliki kompleksitas waktu linear O(n), di mana n adalah jumlah elemen dalam rangkaian data.

2. Binary Search (Pencarian Biner): Teknik ini digunakan pada rangkaian data yang telah diurutkan. Pencarian dimulai dengan membandingkan elemen tengah dengan elemen yang sedang dicari. Jika elemen tengah adalah elemen yang dicari, pencarian selesai. Jika elemen tengah lebih besar dari elemen yang dicari, pencarian dilanjutkan pada setengah kiri dari rangkaian data. Jika elemen tengah lebih kecil, pencarian dilanjutkan pada setengah kanan. Binary search memiliki kompleksitas waktu logaritmik O(log n), di mana n adalah jumlah elemen dalam rangkaian data.

3. Hashing: Teknik ini melibatkan penggunaan fungsi hash untuk mengonversi data menjadi nilai hash, yang kemudian digunakan sebagai kunci untuk penyimpanan, pencocokan, atau identifikasi data. Hashing memungkinkan akses langsung ke data dengan kompleksitas waktu konstan O(1) dalam kasus terbaiknya. Namun, ada kemungkinan tabrakan hash jika dua data menghasilkan nilai hash yang sama. Untuk mengatasi tabrakan hash, teknik seperti chaining atau open addressing dapat digunakan.

Ketiga teknik dasar ini memiliki kelebihan dan kekurangan masing-masing. Pemilihan teknik yang tepat tergantung pada karakteristik data dan kebutuhan spesifik dalam pemrosesan data.

B. The Knapsack Problem and Memory Fungctions

Masalah Knapsack (Knapsack Problem) adalah sebuah masalah optimasi yang sering dihadapi dalam ilmu komputer dan matematika. Dalam masalah ini, terdapat sebuah knapsack (tas) dengan kapasitas terbatas dan sejumlah objek yang memiliki nilai dan bobot yang berbeda. Tujuan adalah memilih kombinasi objek yang akan dimasukkan ke dalam knapsack sedemikian rupa sehingga nilai total objek yang dipilih maksimum, tetapi total bobotnya tidak melebihi kapasitas knapsack.

Dalam konteks masalah knapsack, memory functions (fungsi memori) merujuk pada penggunaan penyimpanan sementara untuk menyimpan dan mengingat solusi-solusi submasalah yang telah dihitung. Dalam pendekatan pemrograman dinamis untuk memecahkan masalah knapsack, memory functions digunakan untuk menghindari perhitungan berulang pada submasalah yang sama.

Memory functions dapat berupa struktur data seperti tabel atau matriks yang digunakan untuk menyimpan hasil perhitungan submasalah. Setiap entri dalam struktur data tersebut mewakili solusi terbaik yang ditemukan untuk submasalah tertentu. Dengan menggunakan memory functions, kita dapat mengakses solusi submasalah yang telah dihitung sebelumnya secara efisien, tanpa perlu menghitung ulang.

C. **Warshall's and Floyd's AlgorithmsWarshall's Algorithm (Algoritma Warshall)** dan Floyd's Algorithm (Algoritma Floyd) adalah dua algoritma yang digunakan dalam teori graf untuk menyelesaikan masalah jalur terpendek antara semua pasangan simpul (all-pairs shortest path).

1. Warshall's Algorithm:

Algoritma Warshall digunakan untuk menemukan jalur terpendek antara semua pasangan simpul dalam graf berbobot positif atau negatif (tetapi tanpa siklus negatif). Algoritma ini mengoperasikan matriks kedekatan (adjacency matrix) yang merepresentasikan graf. Langkah-langkah utama dalam algoritma Warshall adalah sebagai berikut:

- Inisialisasi matriks kedekatan dengan bobot langsung antara simpul-simpul yang terhubung secara langsung, dan mengisi nilai tak hingga (Infinity) untuk pasangan simpul yang tidak terhubung secara langsung.

- Lakukan iterasi untuk semua simpul sebagai simpul tengah dan perbarui nilai bobot antara simpul-simpul dengan mempertimbangkan simpul tengah. Jika bobot jalur baru lebih kecil dari bobot jalur sebelumnya, maka nilai bobot diperbarui.

- Setelah semua iterasi selesai, matriks kedekatan akan berisi bobot jalur terpendek antara semua pasangan simpul.

2. Floyd's Algorithm:

Algoritma Floyd, juga dikenal sebagai Algoritma Floyd-Warshall, digunakan untuk menemukan jalur terpendek antara semua pasangan simpul dalam graf berbobot positif atau negatif (tetapi tanpa siklus negatif). Algoritma ini juga mengoperasikan matriks kedekatan yang merepresentasikan graf. Langkah-langkah utama dalam algoritma Floyd adalah sebagai berikut:

- Inisialisasi matriks kedekatan dengan bobot langsung antara simpul-simpul yang terhubung secara langsung, dan mengisi nilai tak hingga (Infinity) untuk pasangan simpul yang tidak terhubung secara langsung.

- Lakukan iterasi untuk semua simpul sebagai simpul tengah dan perbarui nilai bobot antara simpul-simpul dengan mempertimbangkan simpul tengah. Jika bobot jalur baru lebih kecil dari bobot jalur sebelumnya, maka nilai bobot diperbarui.

- Setelah semua iterasi selesai, matriks kedekatan akan berisi bobot jalur terpendek antara semua pasangan simpul.

Perbedaan utama antara Algoritma Warshall dan Algoritma Floyd terletak pada urutan iterasinya. Pada Algoritma Warshall, iterasi dilakukan pada semua pasangan simpul sebagai simpul tengah secara berurutan, sedangkan pada Algoritma Floyd, iterasi dilakukan pada semua simpul sebagai simpul tengah secara berurutan. Algoritma Warshall memiliki kompleksitas waktu O(n^3) dan Algoritma Floyd juga memiliki kompleksitas waktu O(n^3), di mana n adalah jumlah simpul dalam graf.

Kedua algoritma ini sangat berguna dalam menemukan jalur terpendek antara semua pasangan simpul dalam graf berbobot.

BAB IX

GREEDY TECHNIQUE

A. Prims's Algorithm

algoritma yang digunakan untuk mencari Minimum Spanning Tree (MST) dari sebuah graf berbobot. MST adalah subgraf terhubung yang menghubungkan semua simpul dalam graf dengan bobot total yang minimum.

Berikut adalah langkah-langkah utama dalam algoritma Prim:

Pilih simpul awal secara acak sebagai simpul awal MST.
Inisialisasi set MST kosong dan set T berisi semua simpul yang belum termasuk dalam MST.
Selama set T tidak kosong, lakukan langkah-langkah berikut:
a. Pilih sisi minimum yang menghubungkan simpul dalam MST dengan simpul di luar MST.
b. Tambahkan simpul tersebut ke MST.
c. Hapus simpul tersebut dari set T.
Ulangi langkah 3 sampai set T menjadi kosong.

Algoritma Prim bekerja dengan memperluas MST satu simpul pada satu waktu. Pada setiap iterasi, simpul baru ditambahkan ke MST berdasarkan sisi minimum yang menghubungkan simpul dalam MST dengan simpul di luar MST. Dengan demikian, algoritma membangun MST secara bertahap dengan memilih sisi-sisi minimum yang membentuk MST dengan bobot minimum.

B. Kruskal's Algorithm

Algoritma Kruskal (Kruskal's Algorithm) adalah Algoritma Kruskal (Kruskal's Algorithm) adalah algoritma yang digunakan untuk mencari Minimum Spanning Tree (MST) dari sebuah graf berbobot. MST adalah subgraf terhubung yang menghubungkan semua simpul dalam graf dengan bobot total yang minimum.

Berikut adalah langkah-langkah utama dalam algoritma Kruskal:

1. Urutkan semua sisi dalam graf berdasarkan bobotnya dari yang terkecil hingga yang terbesar.
2. Inisialisasi set MST kosong.
3. Untuk setiap sisi dalam urutan terurut, lakukan langkah-langkah berikut:
   a. Jika menambahkan sisi tersebut ke MST tidak membentuk siklus, tambahkan sisi tersebut ke MST.
   b. Jika menambahkan sisi tersebut ke MST membentuk siklus, lewati sisi tersebut dan lanjutkan ke sisi berikutnya.
4. Ulangi langkah 3 sampai semua simpul terhubung dalam MST atau jumlah sisi dalam MST sama dengan V - 1, di mana V adalah jumlah simpul dalam graf.

Algoritma Kruskal bekerja dengan memilih sisi-sisi dengan bobot terkecil dan

menambahkannya ke MST jika sisi tersebut tidak membentuk siklus dengan sisi-sisi sebelumnya yang sudah ada dalam MST. Algoritma ini membangun MST secara bertahap dengan menambahkan sisi-sisi yang memiliki bobot terkecil dan memastikan tidak terbentuknya siklus.

Keuntungan dari algoritma Kruskal adalah kompleksitas waktu yang efisien. Dalam implementasinya dengan menggunakan struktur data seperti disjoint-set atau union-find, kompleksitas waktu algoritma Kruskal adalah O(E log E), di mana E adalah jumlah sisi dalam graf. Oleh karena itu, algoritma ini efisien untuk graf dengan jumlah sisi yang besar.

Algoritma Kruskal sering digunakan dalam aplikasi yang melibatkan jaringan komunikasi, optimasi jaringan, pemetaan jaringan, dan masalah lain yang melibatkan penentuan jalur terpendek atau jaringan terhubung dengan biaya minimum.

C. Djikstra's Algorithm

Algoritma Dijkstra (Dijkstra's Algorithm) adalah algoritma yang digunakan untuk mencari jalur terpendek antara dua simpul dalam graf berbobot yang terhubung. Algoritma ini dapat diterapkan pada graf berarah maupun tidak berarah, tetapi bobot sisi-sisi harus non-negatif.

Berikut adalah langkah-langkah utama dalam algoritma Dijkstra:

→Inisialisasi jarak awal untuk semua simpul dalam graf dengan nilai tak hingga, kecuali simpul awal yang memiliki jarak awal 0.
→Buat sebuah himpunan kosong S yang akan berisi simpul-simpul yang sudah memiliki jalur terpendek yang telah diketahui.
→Selama himpunan S belum mencakup semua simpul dalam graf, lakukan langkah-langkah berikut:
a. Pilih simpul u yang belum termasuk dalam S dengan jarak terpendek dari simpul awal.
b. Tambahkan simpul u ke dalam S.
c. Untuk setiap simpul tetangga v dari u yang belum termasuk dalam S, perbarui jarak terpendek dari simpul awal ke v jika melalui simpul u memberikan jarak yang lebih pendek daripada jarak terdahulu yang diketahui.
→Setelah himpunan S mencakup semua simpul dalam graf, jalur terpendek dari simpul awal ke simpul tujuan dapat dikonstruksi dengan mengikuti jalur terpendek yang disimpan dalam tabel atau struktur data yang mempertahankan informasi tentang simpul sebelumnya dalam jalur terpendek.

Algoritma Dijkstra menggunakan pendekatan Greedy (Serakah) dengan memilih simpul dengan jarak terpendek pada setiap langkah. Dalam setiap iterasi, algoritma mengambil simpul dengan jarak terpendek yang belum termasuk dalam himpunan S, dan memperbarui jarak terpendek ke semua tetangga simpul tersebut. Proses ini dilakukan hingga jalur terpendek ke semua simpul telah diketahui.

Keuntungan dari algoritma Dijkstra adalah kemampuannya untuk menemukan jalur terpendek dalam graf berbobot dengan efisien. Dalam implementasinya

dengan menggunakan struktur data seperti heap binomial atau heap Fibonacci, kompleksitas waktu algoritma Dijkstra adalah O((V + E) log V), di mana V adalah jumlah simpul dan E adalah jumlah sisi dalam graf. Namun, jika menggunakan representasi graf menggunakan matriks ketetanggaan, kompleksitas waktu dapat mencapai O(V^2).

D. Huffman Tress and Codes

Pohon Huffman (Huffman Tree) dan kode Huffman (Huffman Codes) adalah teknik kompresi data yang digunakan untuk mengurangi ukuran file dengan memanfaatkan frekuensi kemunculan simbol-simbol dalam data yang akan dikompresi.

Pohon Huffman adalah pohon biner khusus yang digunakan untuk menghasilkan kode Huffman. Pohon ini dibangun berdasarkan frekuensi kemunculan simbol-simbol dalam data. Simbol-simbol dengan frekuensi kemunculan yang lebih tinggi akan ditempatkan lebih dekat ke akar pohon, sementara simbol-simbol dengan frekuensi yang lebih rendah akan ditempatkan lebih jauh dari akar. Setiap simpul dalam pohon Huffman merepresentasikan sebuah simbol atau kombinasi simbol-simbol.Langkah-langkah untuk membangun pohon Huffman:

Proses pembentukan pohon Huffman melibatkan beberapa langkah berikut:

- Menghitung frekuensi kemunculan setiap simbol dalam data yang akan dikompresi.
- Membangun pohon Huffman menggunakan pendekatan serakah:

  a. Buat simpul untuk setiap simbol dengan frekuensi kemunculan dan masukkan ke dalam sebuah priority queue.

  b. Ambil dua simpul dengan frekuensi terendah dari priority queue dan gabungkan menjadi satu simpul baru. Simpul baru ini akan memiliki frekuensi yang merupakan jumlah frekuensi simpul-simpul yang digabungkan.

  c. Masukkan simpul baru ke dalam priority queue.

  d. Ulangi langkah b dan c hingga hanya tersisa satu simpul di dalam priority queue, yang merupakan akar pohon Huffman.

- Buat tabel yang mengaitkan setiap simbol dengan kode Huffman yang sesuai berdasarkan pohon Huffman yang telah dibangun.

Setelah pohon Huffman terbentuk, data dapat dikompresi dengan menggantikan setiap simbol dalam data dengan kode Huffman yang sesuai. Proses ini menghasilkan file yang lebih kecil dalam ukuran, karena representasi biner digunakan untuk menggantikan representasi asli simbol-simbol.

BAB X

ITERATIVE IMPROVEMENT

A. The Simplex Method

Metode Simpleks (Simplex Method) adalah sebuah algoritma yang digunakan dalam pemrograman linier untuk menemukan solusi optimal dari sebuah masalah pemrograman linier. Metode ini dikembangkan oleh George Dantzig pada tahun 1947 dan menjadi salah satu metode yang paling umum digunakan dalam pemecahan masalah pemrograman linier.

Langkah-langkah dalam Metode Simpleks adalah sebagai berikut:

1. Bentuk masalah pemrograman linier dalam bentuk standar:
   - Tentukan fungsi objektif yang ingin dioptimalkan.
   - Tentukan batasan-batasan yang membatasi nilai variabel dalam masalah.

2. Ubah masalah ke dalam bentuk tableau atau tabel:
   - Tambahkan variabel slack untuk setiap batasan untuk mengubah batasan ulang menjadi kesetaraan.
   - Tambahkan variabel surplus untuk setiap batasan ketidaksamaan yang tidak diselesaikan.
   - Tambahkan variabel bukaan untuk setiap variabel basis.

3. Tentukan aturan pivot:
   - Pilih variabel masukan (entri masuk) dengan koefisien negatif terbesar dalam baris fungsi objektif.
   - Pilih variabel keluar (entri keluar) dengan aturan rasio terkecil antara solusi optimal dan koefisien kolom pada baris yang dipilih.

4. Lakukan operasi pivot:
   - Ubah baris pivot menjadi 1 dengan membaginya dengan entri pivot.
   - Gunakan eliminasi Gauss-Jordan untuk membuat semua entri pada kolom pivot menjadi 0, kecuali entri pivot itu sendiri.

5. Ulangi langkah 3 dan 4 hingga tidak ada variabel masukan yang memiliki koefisien negatif dalam baris fungsi objektif. Ini menunjukkan bahwa solusi optimal telah ditemukan.

6. Interpretasikan solusi:
   - Nilai variabel basis memberikan solusi optimal untuk variabel yang diwakili oleh kolom basis.

Metode Simpleks memberikan solusi optimal untuk masalah pemrograman linier yang memenuhi persyaratan dan batasan tertentu. Namun, kompleksitas waktu metode ini tergantung pada jumlah variabel dan batasan dalam masalah, sehingga pada kasus tertentu, metode ini mungkin tidak efisien.

B. The maximum-Flow Problem

Masalah aliran maksimum (maximum-flow problem) adalah sebuah masalah dalam teori jaringan yang bertujuan untuk menemukan aliran maksimum yang dapat mengalir melalui jaringan dari satu simpul sumber ke satu simpul tujuan. Dalam konteks ini, jaringan direpresentasikan sebagai graf terarah yang terdiri dari simpul-simpul (node) dan tepi-tepi (edge) dengan kapasitas tertentu.

Langkah-langkah untuk memecahkan masalah aliran maksimum adalah sebagai berikut:

1. Tentukan simpul sumber (source) dan simpul tujuan (sink) dalam jaringan.

2. Atur aliran awal di setiap tepi dalam jaringan menjadi 0.

3. Selama terdapat jalur sumber-tujuan (path) yang dapat diambil dalam jaringan, lakukan langkah-langkah berikut:
   - Temukan jalur sumber-tujuan menggunakan algoritma pencarian seperti Breadth-First Search (BFS) atau Depth-First Search (DFS).
   - Tentukan kapasitas tersisa (residual capacity) pada jalur tersebut, yaitu kapasitas maksimum dikurangi dengan aliran yang sudah ada.
   - Tingkatkan aliran pada jalur tersebut dengan jumlah yang tidak melebihi kapasitas tersisa.
   - Kurangi aliran pada jalur balik (reverse path) dengan jumlah yang sama.

4. Ulangi langkah 3 hingga tidak ada jalur sumber-tujuan lagi dalam jaringan.

Setelah algoritma selesai, aliran maksimum yang ditemukan akan memiliki sifat bahwa tidak ada jalur sumber-tujuan lagi dalam jaringan yang dapat mengalirkan lebih banyak aliran. Algoritma ini mengoptimalkan aliran melalui jaringan dengan memaksimalkan jumlah aliran yang dapat dikirim dari simpul sumber ke simpul tujuan.

Masalah aliran maksimum memiliki berbagai aplikasi dalam dunia nyata, seperti dalam perencanaan transportasi, routing jaringan komunikasi, manajemen aliran air, dan optimasi operasi sistem. Algoritma yang umum digunakan untuk memecahkan masalah ini adalah Algoritma Edmonds-Karp dan Algoritma Ford-Fulkerson, yang berbasis pada ide-ide seperti pemotongan minimum (minimum cut) dan jaringan residual.

C. Maximum Matching in Bipartite Graphs

Matching maksimum dalam graf bipartit (maximum matching in bipartite graphs) adalah masalah yang bertujuan untuk menemukan jumlah maksimum dari pasangan yang dapat dipasangkan dalam sebuah graf bipartit. Graf bipartit terdiri dari dua himpunan simpul, di mana setiap simpul dalam himpunan pertama hanya terhubung dengan simpul dalam himpunan kedua dan sebaliknya.

Langkah-langkah untuk mencari matching maksimum dalam graf bipartit adalah sebagai berikut:

1. Representasikan graf bipartit sebagai sebuah graf dengan dua himpunan simpul (misalnya A dan B).

2. Inisialisasi matching awal dengan himpunan kosong.

3. Ulangi langkah-langkah berikut hingga tidak ada lagi peningkatan yang dapat dilakukan pada matching:
   - Temukan jalur peningkatan (augmenting path) dalam graf menggunakan algoritma seperti Breadth-First Search (BFS) atau Depth-First Search (DFS).
   - Jika jalur peningkatan ditemukan, lakukan peningkatan pada matching dengan menukar pasangan yang dipasangkan dan pasangan yang tidak dipasangkan pada jalur tersebut.
   - Jika tidak ada jalur peningkatan yang ditemukan, berarti matching sudah mencapai jumlah maksimum.

4. Setelah tidak ada peningkatan lagi yang mungkin dilakukan pada matching, jumlah pasangan yang terbentuk dalam matching adalah matching maksimum dalam graf bipartit.

Algoritma untuk mencari matching maksimum dalam graf bipartit memiliki kompleksitas waktu yang bergantung pada implementasinya. Dalam kasus graf bipartit dengan V simpul, algoritma menggunakan Breadth-First Search (BFS) memiliki kompleksitas waktu O(V^2 E), di mana E adalah jumlah tepi dalam graf. Terdapat juga algoritma yang lebih efisien, seperti Algoritma Hopcroft-Karp, yang memiliki kompleksitas waktu O(E√V).

D. The Stable Marriage Problem

Masalah Pernikahan Stabil (Stable Marriage Problem) adalah sebuah masalah yang mencari pencocokan optimal antara dua kelompok orang, di mana setiap individu dari kelompok pertama memiliki preferensi terhadap individu dari kelompok kedua, dan sebaliknya. Tujuan dari masalah ini adalah untuk mencari pencocokan yang stabil, di mana tidak ada pasangan individu yang lebih memilih satu sama lain daripada pasangan mereka yang ada.

Tujuan dari masalah pernikahan stabil adalah untuk menemukan pencocokan yang stabil, di mana tidak ada pasangan yang lebih memilih satu sama lain daripada pasangan mereka saat ini. Dalam pencocokan stabil, tidak ada pasangan yang saling berkeinginan untuk meninggalkan pasangan mereka dan membentuk pasangan baru.

Berikut adalah langkah-langkah dalam penyelesaian Masalah Pernikahan Stabil:

- Setiap pria mengajukan proposal kepada wanita yang berada di urutan teratas dalam daftar preferensinya yang belum menolaknya.
- Setiap wanita menerima proposal dari pria yang dia preferensikan paling tinggi, atau menolak dan mempertimbangkan proposal dari pria berikutnya dalam daftar preferensinya.
- Jika seorang wanita menerima proposal dari seorang pria, tetapi dia telah menerima proposal dari pria lain sebelumnya, dia membandingkan dua proposal tersebut dan memilih pria yang lebih tinggi dalam preferensinya. Pria yang ditolak akan mengajukan proposal kepada wanita berikutnya dalam daftar preferensinya yang belum menolaknya.
- Langkah 2 dan 3 diulang sampai tidak ada lagi proposal yang diajukan dan semua individu telah dipasangkan.

Hasil dari penyelesaian Masalah Pernikahan Stabil adalah pencocokan yang stabil, di mana tidak ada pasangan yang lebih memilih satu sama lain daripada pasangan mereka yang ada. Artinya, tidak ada pasangan yang cenderung berpisah untuk membentuk pasangan yang baru yang lebih disukai. Solusi ini memberikan keadilan dan kestabilan dalam konteks pernikahan.

BAB XI

LIMITATIONS OF ALGORITHM POWER

A. Lower-Bounf Arguments

Argumen batas bawah (lower-bound arguments) adalah pendekatan dalam analisis algoritma yang digunakan untuk membuktikan bahwa tidak ada algoritma yang lebih efisien atau lebih cepat dari batas bawah tertentu dalam memecahkan suatu masalah. Dengan kata lain, argumen batas bawah berfokus pada menentukan batasan terendah dari kinerja algoritma yang mungkin dapat dicapai dalam memecahkan masalah tersebut.

Argumen batas bawah digunakan untuk memberikan bukti bahwa suatu masalah memiliki kompleksitas waktu tertentu, sehingga algoritma yang lebih cepat tidak mungkin ada. Dalam konteks ini, kompleksitas waktu didefinisikan sebagai fungsi yang menggambarkan hubungan antara ukuran input masalah dan waktu yang dibutuhkan oleh algoritma untuk memecahkan masalah tersebut.

Untuk membuktikan batas bawah dalam analisis algoritma, beberapa teknik yang umum digunakan adalah sebagai berikut:

1. Reduksi: Mengurangi suatu masalah yang diketahui memiliki batas bawah tertentu ke masalah yang sedang dianalisis. Dengan mengasumsikan bahwa algoritma yang lebih cepat ada untuk masalah yang sedang dianalisis, maka akan ditemukan algoritma yang lebih cepat untuk masalah yang dikurangi, yang akan bertentangan dengan asumsi tersebut.

2. Pembuktian informasi yang hilang: Mengidentifikasi informasi yang tidak ada dalam input masalah yang mempengaruhi hasil akhir algoritma. Dengan membuktikan bahwa informasi tersebut diperlukan untuk memecahkan masalah dengan tepat, dapat dijelaskan mengapa algoritma dengan kompleksitas yang lebih rendah tidak mungkin ada.

3. Pembuktian pengurangan waktu: Mengasumsikan bahwa algoritma yang lebih cepat ada, dan kemudian menunjukkan bahwa hal itu akan mengakibatkan hasil yang tidak mungkin atau melanggar sifat-sifat tertentu yang diketahui tentang masalah.

Argumen batas bawah merupakan alat penting dalam analisis algoritma untuk menentukan batas terendah dari kompleksitas waktu yang mungkin dapat dicapai dalam memecahkan suatu masalah. Dengan memahami batas bawah, kita dapat mengetahui kinerja algoritma yang ada dan mengidentifikasi kapan suatu masalah memerlukan solusi yang lebih efisien atau lebih canggih.

B. Decision Tree

Decision Tree adalah struktur pohon seperti diagram alir di mana setiap simpul internal menunjukkan fitur, cabang menunjukkan aturan dan simpul daun menunjukkan hasil dari algoritma. Ini adalah algoritma pembelajaran mesin

terawasi serbaguna, yang digunakan untuk masalah klasifikasi dan regresi. Ini adalah salah satu algoritma yang sangat kuat. Dan itu juga digunakan di Hutan Acak untuk melatih subset data pelatihan yang berbeda, yang menjadikan hutan acak salah satu algoritme paling kuat dalam pembelajaran mesin.

Terminologi Pohon Keputusan(Decision Tree)
Beberapa Terminologi umum yang digunakan dalam Pohon Keputusan adalah sebagai berikut:
Root Node: Ini adalah simpul paling atas di pohon, yang mewakili kumpulan data lengkap. Ini adalah titik awal dari proses pengambilan keputusan.
Decision/Internal Node: Node yang melambangkan pilihan mengenai fitur input. Bercabang dari node internal menghubungkan mereka ke node daun atau node internal lainnya.
Leaf/Terminal Node: Node tanpa node anak yang menunjukkan label kelas atau nilai numerik.
Splitting: Proses pemisahan node menjadi dua atau lebih sub-node menggunakan kriteria pemisahan dan fitur yang dipilih.
Branch/Sub-Tree: Sebuah subbagian dari pohon keputusan dimulai dari simpul internal dan berakhir di simpul daun.
Parent Node: Node yang terbagi menjadi satu atau lebih node anak.
Child Node: Node yang muncul ketika parent node dipisah.
Impurity: Pengukuran homogenitas variabel target dalam subset data. Ini mengacu pada tingkat keacakan atau ketidakpastian dalam serangkaian contoh. Indeks Gini dan entropi adalah dua pengukuran ketidakmurnian yang umum digunakan dalam pohon keputusan untuk tugas klasifikasi
Variance: Varians mengukur seberapa banyak prediksi dan variabel target bervariasi dalam sampel dataset yang berbeda. Ini digunakan untuk masalah regresi di pohon keputusan. Mean squared error, Mean Absolute Error, friedman_mse, atau penyimpangan Half Poisson digunakan untuk mengukur varian tugas regresi dalam pohon keputusan.
Information Gain: Penguatan informasi adalah ukuran pengurangan pengotor yang dicapai dengan memisahkan kumpulan data pada fitur tertentu di pohon keputusan. Kriteria pemisahan ditentukan oleh fitur yang menawarkan perolehan informasi terbesar, digunakan untuk menentukan fitur yang paling informatif untuk dibagi pada setiap simpul pohon, dengan tujuan menciptakan himpunan bagian murni
Pruning: Proses menghilangkan cabang dari pohon yang tidak memberikan informasi tambahan atau menyebabkan overfitting..

C. P, NP and-Complete Problems

P, NP, dan NP-complete adalah istilah yang digunakan dalam teori kompleksitas komputasi untuk menggolongkan masalah berdasarkan tingkat kesulitan mereka dalam dipecahkan oleh algoritma.

P:
Kelas P (Polynomial Time) adalah kelas masalah yang dapat dipecahkan oleh algoritma dengan kompleksitas waktu yang dapat dibatasi oleh fungsi polinomial dari ukuran input. Dalam kelas P, solusi optimal dapat ditemukan dalam waktu yang efisien. Contoh masalah dalam kelas P termasuk penjumlahan matriks, pengurangan bilangan bulat, dan sorting.

NP:
Kelas NP (Nondeterministic Polynomial Time) adalah kelas masalah di mana solusi dapat diverifikasi dalam waktu yang efisien. Dalam kelas NP, jika diberikan solusi yang diajukan, kebenarannya dapat diverifikasi dalam waktu polinomial. Namun, tidak ada jaminan bahwa solusi optimal dapat ditemukan dengan cepat. Contoh masalah dalam kelas NP termasuk problem penggantian NP-complete, seperti Travelling Salesman Problem (TSP) dan Knapsack Problem.

NP-complete:
Masalah NP-complete adalah subset dari masalah dalam kelas NP yang memiliki sifat tertentu. Sebuah masalah NP-complete adalah masalah yang secara teoritis sulit untuk dipecahkan dengan cepat. Jika ada algoritma efisien yang dapat menyelesaikan salah satu masalah NP-complete, maka algoritma tersebut dapat digunakan untuk menyelesaikan semua masalah NP-complete dengan efisien. Contoh masalah NP-complete termasuk Boolean Satisfiability Problem (SAT), Traveling Salesman Problem (TSP), dan Knapsack Problem.

D. Challenge of Numerical Algorithms

Tantangan dalam algoritma numerik melibatkan pemecahan masalah matematis dengan menggunakan metode komputasi. Algoritma numerik berfokus pada pengembangan teknik dan metode untuk menyelesaikan masalah yang melibatkan perhitungan angka, seperti perhitungan integral, sistem persamaan linear, atau optimisasi numerik.

Berikut ini adalah beberapa tantangan utama dalam algoritma numerik:

•Ketelitian Numerik: Pada saat melakukan perhitungan dengan angka, terjadi pembulatan dan kesalahan numerik. Tantangan ini melibatkan bagaimana mengendalikan dan meminimalkan kesalahan yang terjadi selama perhitungan. Hal ini penting dalam memastikan keakuratan hasil numerik.

•Kestabilan Numerik: Beberapa masalah numerik dapat sangat sensitif terhadap perubahan kecil dalam data masukan. Ketika masalah ini terjadi, algoritma harus dirancang agar tetap memberikan hasil yang stabil dan konsisten, bahkan dengan variasi kecil pada data masukan.

•Skala Masalah: Beberapa masalah numerik melibatkan jumlah data yang sangat besar atau skala yang ekstrem. Tantangan ini meliputi efisiensi perhitungan dan

pengelolaan memori untuk menangani data dalam skala besar dengan cepat dan efisien.

•Konvergensi: Beberapa algoritma numerik membutuhkan iterasi berulang untuk mencapai solusi yang akurat. Tantangan ini adalah memastikan bahwa algoritma konvergen ke solusi yang benar dan mencapai akurasi yang diinginkan dalam jumlah iterasi yang wajar.

•Ketersediaan Sumber Daya: Algoritma numerik seringkali membutuhkan sumber daya komputasi yang besar, seperti kecepatan pemrosesan yang tinggi, memori yang cukup, atau kemampuan komputasi paralel. Tantangan ini adalah memastikan ketersediaan sumber daya yang memadai untuk menjalankan algoritma dengan efisiensi.

Untuk mengatasi tantangan ini, pengembang algoritma numerik perlu memperhatikan desain algoritma yang cermat, analisis kesalahan numerik, pemilihan metode yang tepat, dan optimisasi implementasi algoritma. Selain itu, perlu juga melakukan pengujian yang cermat untuk memastikan keakuratan dan keandalan algoritma numerik yang dikembangkan.

BAB XII

COPING WITH THE LIMITATIONS OF ALGORITHM POWER

A. Backtracking

Backtracking adalah sebuah metode atau pendekatan dalam pemrograman yang digunakan untuk mencari solusi dari masalah yang melibatkan pemilihan langkah-langkah yang tepat secara iteratif. Metode ini sering digunakan untuk masalah optimasi, kombinatorik, dan permasalahan yang melibatkan pemilihan dari sejumlah besar kemungkinan.

Ide dasar dari backtracking adalah mencoba setiap kemungkinan langkah yang mungkin untuk mencapai solusi yang benar. Jika langkah yang sedang dipertimbangkan tidak mengarah ke solusi yang diinginkan, maka kita mundur ke langkah sebelumnya (backtrack) dan mencoba langkah alternatif lainnya.

Proses backtracking melibatkan beberapa langkah penting, yaitu:

1. Langkah pertama: Memilih langkah awal dan memulai pencarian solusi.

2. Evaluasi langkah: Memeriksa apakah langkah yang sedang dipertimbangkan memenuhi kriteria yang ditetapkan untuk solusi. Jika iya, maka kita melanjutkan ke langkah berikutnya. Jika tidak, kita melakukan backtrack ke langkah sebelumnya.

3. Langkah berikutnya: Memilih langkah berikutnya dari kumpulan kemungkinan langkah yang tersedia. Langkah ini akan menjadi langkah yang sedang dipertimbangkan pada langkah evaluasi selanjutnya.

4. Evaluasi kondisi berhenti: Setiap langkah dievaluasi untuk menentukan apakah solusi telah ditemukan atau belum. Jika solusi telah ditemukan, pencarian dihentikan. Jika solusi belum ditemukan, proses backtracking dilanjutkan dengan memilih langkah alternatif atau melakukan backtrack ke langkah sebelumnya.

Proses ini terus berlanjut hingga semua kemungkinan langkah telah dijelajahi atau solusi telah ditemukan.

Backtracking adalah metode yang efektif untuk menyelesaikan masalah dengan ruang pencarian yang besar. Namun, kompleksitas waktu dari algoritma backtracking dapat menjadi sangat tinggi tergantung pada ruang pencarian dan jumlah langkah yang harus dieksplorasi. Dalam beberapa kasus, optimisasi dan teknik pruning dapat digunakan untuk membatasi jumlah langkah yang dievaluasi, meningkatkan efisiensi algoritma backtracking.

B. Branch and Bound

Branch and Bound (BnB) adalah salah satu algoritma yang digunakan dalam pemecahan masalah optimasi kombinatorial, terutama dalam kasus di mana pencarian solusi eksponensial dapat dihindari dengan memanfaatkan batasan dan pemotongan yang cerdas.

Algoritma Branch and Bound bekerja dengan cara yang mirip dengan algoritma cabang dan batas (Branch and Bound) yang digunakan dalam pemecahan masalah pemrograman linier. Langkah-langkah utama algoritma ini adalah sebagai berikut:

1. Inisialisasi: Tentukan batas atas (upper bound) awal sebagai suatu nilai tak terbatas dan inisialisasi himpunan solusi terbaik dengan nilai kosong.
2. Pembagian (Branching): Pilih satu variabel atau subset variabel dalam masalah yang belum ditentukan nilainya. Bagi masalah menjadi dua atau lebih submasalah baru dengan memilih nilai yang mungkin untuk variabel tersebut. Setiap submasalah memiliki batasan dan fungsi objektif yang diperbarui berdasarkan nilai-nilai yang dipilih.
3. Penyelesaian Submasalah: Selesaikan setiap submasalah secara rekursif dengan menerapkan langkah-langkah 2 dan 3 pada setiap submasalah hingga mencapai kasus dasar yang dapat dipecahkan langsung.
4. Pemotongan (Bounding): Evaluasi batasan bawah (lower bound) pada setiap submasalah. Jika batasan bawah suatu submasalah melebihi batas atas terbaik yang telah ditemukan, maka submasalah tersebut dapat dipotong (pruned) karena tidak akan menghasilkan solusi yang lebih baik daripada solusi terbaik yang telah ada.
5. Pembaruan Solusi Terbaik: Jika suatu submasalah menghasilkan solusi yang lebih baik daripada solusi terbaik yang ada sejauh ini, perbarui solusi terbaik dengan solusi dari submasalah tersebut.
6. Kembali ke Langkah 2: Ulangi langkah-langkah 2 hingga 5 untuk setiap submasalah yang belum terselesaikan.
7. Terminasi: Algoritma berakhir ketika tidak ada lagi submasalah yang perlu diselesaikan.
8. Output: Solusi terbaik yang ditemukan selama proses algoritma adalah solusi optimal untuk masalah optimasi kombinatorial.

Algoritma Branch and Bound memanfaatkan batasan bawah dan batasan atas untuk mempersempit ruang pencarian solusi dan menghindari pencarian yang tidak perlu. Dengan cara ini, algoritma dapat mengurangi kompleksitas waktu yang dibutuhkan untuk menemukan solusi optimal dalam masalah yang kompleks.

Algoritma Branch and Bound digunakan dalam berbagai masalah optimasi kombinatorial, seperti masalah penugasan (assignment problem), masalah rute terpendek (shortest path problem), dan masalah penjadwalan (scheduling problem).

C. Algorithms for Solving Nonlinear Problems

Algoritma-algoritma untuk memecahkan masalah nonlinier dapat bervariasi tergantung pada jenis persamaan atau fungsi nonlinier yang ingin diselesaikan. Berikut adalah beberapa algoritma yang umum digunakan dalam konteks ini:

1. Metode Newton-Raphson: Metode ini digunakan untuk mencari akar dari suatu persamaan nonlinier. Algoritma ini menggunakan pendekatan iteratif dengan memperbaiki perkiraan akar pada setiap iterasi berdasarkan turunan fungsi. Metode Newton-Raphson sangat efisien untuk mengatasi persamaan nonlinier dengan kecepatan konvergensi cepat, tetapi membutuhkan turunan fungsi yang dapat dihitung dengan mudah.

2. Metode Bisection: Metode ini digunakan untuk mencari akar dari suatu persamaan nonlinier dengan pendekatan interval. Algoritma ini membagi interval awal menjadi interval yang lebih kecil secara iteratif hingga akar ditemukan. Metode Bisection bekerja dengan asumsi bahwa fungsi nonlinier bersifat kontinu dan memiliki tanda yang berbeda di ujung interval awal.

3. Metode Iterasi Sederhana: Metode ini menggunakan pendekatan iteratif sederhana untuk mencari akar persamaan nonlinier. Algoritma ini memilih titik awal yang sesuai dan mengulangi proses iterasi berdasarkan suatu persamaan iterasi yang ditentukan. Metode ini mungkin membutuhkan banyak iterasi dan sensitif terhadap pemilihan titik awal, tetapi dapat digunakan dalam beberapa kasus sederhana.

4. Metode Regula Falsi: Metode ini juga digunakan untuk mencari akar persamaan nonlinier dengan pendekatan interval. Algoritma ini menggabungkan metode bisection dengan interpolasi linier untuk mengestimasi posisi akar yang lebih baik pada setiap iterasi. Metode Regula Falsi memiliki konvergensi yang lambat dibandingkan metode lainnya, tetapi tetap berguna dalam beberapa kasus.

5. Metode Levenberg-Marquardt: Metode ini digunakan untuk memecahkan masalah pemodelan dan penyesuaian kurva nonlinier. Algoritma ini menggabungkan pendekatan iteratif Gauss-Newton dan metode gradien teredam (damped least squares) untuk mencari solusi yang paling cocok dalam penyesuaian kurva nonlinier.

6. Metode Optimisasi Nonlinier: Algoritma-algoritma optimisasi nonlinier, seperti Metode Gradien, Metode Newton, atau Metode Quasi-Newton, dapat digunakan untuk mencari ekstremum (maksimum atau minimum) suatu fungsi nonlinier. Algoritma-algoritma ini memanfaatkan turunan dan informasi gradient fungsi untuk menemukan titik stasioner yang mungkin merupakan ekstremum.

Pilihan algoritma untuk memecahkan masalah nonlinier tergantung pada sifat persamaan atau fungsi yang ingin diselesaikan, ketersediaan informasi turunan, dan efisiensi yang diinginkan. Beberapa masalah nonlinier mungkin membutuhkan kombinasi metode-metode ini atau algoritma-algoritma khusus yang disesuaikan untuk kasus tertentu.

DAFTAR PUSTAKA

- Sedgewick, R. (2001). Algorithms in C++. Addison-Wesley.
- Goodrich, M. T., & Tamassia, R. (2002). Algorithm Design: Foundations, Analysis, and Internet Examples. John Wiley & Sons.
- Brassard, G., & Bratley, P. (1996). Fundamentals of Algorithmics. Prentice Hall
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms. MIT Press.
- Skiena, S. S. (2008). The Algorithm Design Manual. Springer.
- Dasgupta, S., Papadimitriou, C. H., & Vazirani, U. V. (2008). Algorithms. McGraw-Hill.
- Kleinberg, J., & Tardos, É. (2005). Algorithm Design. Addison-Wesley.
- Goodrich, M. T., & Tamassia, R. (2014). Algorithm Design and Applications. John Wiley & Sons.
- Levitin, A. (2011). Introduction to the Design and Analysis of Algorithms. Pearson.
- Sedgewick, R., & Wayne, K. (2011). Algorithms, Part I (Online Course). Princeton University (available on Coursera).