PERANCANGAN & ANALISIS ALGORITMA

TUGAS AKHIR

Disusun Oleh :

Nama : Fajri Hidayatul Ihsan

NIM : 21346007

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.

Sago, Juni 2023

Fajri Hidayatul Ihsan

DAFTAR ISI

BAB I
ANALISIS ALGORITMA

A. Pengertian

Analisis algoritma adalah proses mempelajari kinerja suatu algoritma dengan memperhatikan faktor-faktor seperti kecepatan (waktu eksekusi), penggunaan memori, dan keakuratan (ketepatan) dalam menghasilkan output yang diinginkan.

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

Measuring the size of an input is a crucial aspect of algorithm analysis and complexity estimation. The size of an input typically refers to the amount of data or the number of elements involved in the problem being solved. However, the way to measure the size may vary depending on the problem domain and the specific context. Here are a few common ways to measure the size of an input:

1. Counting elements: If the problem involves a collection of elements, such as an array or a list, the size can be determined by counting the number of elements in the collection. For example, the size of an array could be the number of elements in the array.

2. Measuring length: In problems involving strings or sequences, the size of the input can be measured by the length of the string or the number of elements in the sequence. For instance, in a problem that requires sorting a string, the size of the input could be the length of the string.

3. Quantifying parameters: In some cases, the input size may be determined by multiple parameters that are not directly related to the number of elements. For example, in a graph problem, the size of the input could be defined by the number of vertices and edges in the graph.

4. Bit representation: In certain scenarios, the size of the input can be measured in terms of the number of bits required to represent the input. This approach is commonly used in problems involving integers, where the size is proportional to the number of bits needed to represent the integer.

It's important to note that measuring the size of an input is closely tied to the problem being solved and the specific context. The size of the input is a crucial factor in analyzing the complexity of an algorithm and determining its efficiency in terms of time and space requirements. By accurately measuring the input's size, one can better estimate and compare the performance of different algorithms when solving the same problem.

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 dinyatakan dengan notasi Big O. Kelas kompleksitas waktu yang paling umum meliputi:

1. Waktu konstan (O(1)): Waktu berjalan tetap konstan terlepas dari ukuran input. Ini menunjukkan bahwa algoritme membutuhkan waktu yang konstan untuk dieksekusi, terlepas dari inputnya.

2. Waktu linier (O(n)): Waktu berjalan bertambah secara linier dengan ukuran masukan. Ini menunjukkan bahwa waktu eksekusi algoritma berbanding lurus dengan ukuran

input.

3. Waktu logaritmik (O(log n)): Waktu berjalan tumbuh secara logaritmik dengan ukuran input. Ini menunjukkan bahwa waktu eksekusi algoritme meningkat pada tingkat yang lebih lambat seiring bertambahnya ukuran input.

4. Waktu kuadrat (O(n^2)): Waktu berjalan tumbuh secara kuadrat dengan ukuran input. Ini menunjukkan bahwa waktu eksekusi meningkat dengan cepat seiring dengan bertambahnya ukuran input, sebanding dengan kuadrat dari ukuran input.

5. Waktu eksponensial (O(2^n)): Waktu berjalan tumbuh secara eksponensial dengan ukuran input. Ini menunjukkan bahwa waktu eksekusi meningkat secara signifikan seiring bertambahnya ukuran input, berlipat ganda dengan setiap elemen input tambahan.

Ini hanya beberapa contoh, dan ada banyak kelas kompleksitas waktu lainnya, masing-masing mewakili tingkat pertumbuhan waktu berjalan yang berbeda sehubungan dengan ukuran masukan.

Penting untuk diperhatikan bahwa kompleksitas waktu memberikan batas atas pada waktu berjalan dan membantu menganalisis skalabilitas algoritme. Itu tidak mewakili waktu berjalan aktual dalam detik atau unit spesifik lainnya, karena waktu eksekusi aktual dapat dipengaruhi oleh berbagai faktor, seperti perangkat keras, bahasa pemrograman, dan pengoptimalan khusus yang diterapkan.

C. Order of Growth

Analisis Urutan Pertumbuhan Efisiensi Algoritma (Order of Growth Analysis) adalah proses untuk mempelajari bagaimana kinerja suatu algoritma berubah seiring dengan pertambahan ukuran input. Dalam analisis ini, kita tertarik untuk mengetahui bagaimana waktu eksekusi atau penggunaan sumber daya (misalnya memori) berubah seiring dengan pertambahan ukuran input.

Dalam analisis urutan pertumbuhan, fokus utama adalah pada bagaimana waktu eksekusi atau penggunaan sumber daya berubah secara keseluruhan, bukan pada nilai absolut waktu eksekusi. Oleh karena itu, kita menggunakan notasi Big O untuk menggambarkan kompleksitas algoritma.

Notasi Big O digunakan untuk memperkirakan tingkat pertumbuhan terburuk algoritma seiring dengan ukuran input yang sangat besar. Misalnya, jika algoritma memiliki kompleksitas waktu O(n^2), itu berarti waktu eksekusi algoritma cenderung meningkat secara kuadrat seiring dengan pertambahan ukuran input (n adalah ukuran input).

Dalam analisis urutan pertumbuhan, beberapa kelas kompleksitas yang umum digunakan antara lain:

- O(1): Konstan, waktu eksekusi tetap konstan terlepas dari ukuran input.
- O(log n): Logaritmik, waktu eksekusi meningkat secara logaritmik seiring dengan pertambahan ukuran input.
- O(n): Linier, waktu eksekusi meningkat secara linier seiring dengan pertambahan ukuran input.
- O(n^2): Kuadratik, waktu eksekusi meningkat secara kuadratik seiring dengan pertambahan ukuran input.
- O(2^n): Eksponensial, waktu eksekusi meningkat secara eksponensial seiring dengan pertambahan ukuran input.

Dalam analisis urutan pertumbuhan, tujuannya adalah untuk mengidentifikasi

kompleksitas algoritma dan memahami bagaimana waktu eksekusi atau penggunaan sumber daya akan berubah seiring dengan ukuran input yang semakin besar. Dengan pemahaman ini, kita dapat memilih algoritma yang paling efisien untuk menyelesaikan suatu masalah.

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

Efisiensi dalam kasus terburuk, kasus terbaik, dan kasus rata-rata adalah konsep yang digunakan dalam analisis algoritma untuk mempelajari kinerja suatu algoritma dalam berbagai skenario. Berikut adalah penjelasan singkat tentang ketiga konsep tersebut:

1. Efisiensi dalam Kasus Terburuk (Worst-Case Efficiency):

Efisiensi dalam kasus terburuk mengacu pada kinerja terburuk yang dapat terjadi pada suatu algoritma ketika dihadapkan dengan input yang paling tidak menguntungkan. Dalam analisis ini, kita fokus pada waktu eksekusi atau penggunaan sumber daya yang paling tinggi dalam skenario terburuk. Efisiensi dalam kasus terburuk memberikan batas atas terhadap kinerja algoritma, yaitu seberapa buruk kinerja algoritma dapat menjadi dalam situasi terburuk.

2. Efisiensi dalam Kasus Terbaik (Best-Case Efficiency):

Efisiensi dalam kasus terbaik mengacu pada kinerja terbaik yang dapat dicapai oleh suatu algoritma ketika dihadapkan dengan input yang paling menguntungkan. Dalam analisis ini, kita fokus pada waktu eksekusi atau penggunaan sumber daya yang paling rendah dalam skenario terbaik. Efisiensi dalam kasus terbaik memberikan batas bawah terhadap kinerja algoritma, yaitu seberapa baik kinerja algoritma dapat menjadi dalam situasi terbaik.

3. Efisiensi dalam Kasus Rata-rata (Average-Case Efficiency):

Efisiensi dalam kasus rata-rata mengacu pada kinerja yang diharapkan dari suatu algoritma ketika dihadapkan dengan input acak yang diambil dari distribusi yang telah ditentukan. Dalam analisis ini, kita mempertimbangkan sebagian besar kemungkinan input dan menghitung waktu eksekusi atau penggunaan sumber daya yang diharapkan dalam kasus tersebut. Efisiensi dalam kasus rata-rata memberikan gambaran yang lebih realistis tentang kinerja algoritma dalam penggunaan praktis.

Penting untuk diingat bahwa efisiensi dalam kasus terburuk, kasus terbaik, dan kasus rata-rata memberikan pemahaman yang berbeda tentang kinerja algoritma dalam situasi yang berbeda. Biasanya, analisis algoritma fokus pada efisiensi dalam kasus terburuk, karena memberikan batas atas dan menjamin kinerja algoritma dalam situasi terburuk yang mungkin terjadi. Namun, pemahaman tentang efisiensi dalam kasus terbaik dan kasus rata-rata juga penting untuk mendapatkan gambaran yang lebih lengkap tentang kinerja algoritma dalam berbagai skenario.

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

Ada tiga varian utama dari paradigma algoritma decrease-and-conquer. Varian-varian tersebut adalah:

1. Pengurangan dengan Jumlah Konstan:
Pada varian ini, ukuran masalah dikurangi dengan jumlah konstan setiap iterasi hingga mencapai kasus dasar. Jumlah konstan dapat berupa angka tetap atau proporsi tetap dari ukuran masalah. Algoritma ini menyelesaikan submasalah yang lebih kecil pada setiap iterasi dan mengurangi ukuran masalah dengan jumlah konstan hingga dapat langsung diselesaikan.

2. Pengurangan dengan Jumlah Variabel:
Pada varian ini, ukuran masalah dikurangi dengan jumlah variabel setiap iterasi hingga mencapai kasus dasar. Jumlah dengan mana ukuran masalah dikurangi bergantung pada karakteristik masalah dan data input. Algoritma ini menyesuaikan pengurangan ukuran masalah secara dinamis berdasarkan properti dari input.

3. Pengurangan Secara Geometri:
Pada varian ini, ukuran masalah dikurangi secara geometri setiap iterasi hingga mencapai kasus dasar. Ukuran masalah dikurangi dengan proporsi tetap (seperti membagi ukuran masalah menjadi dua) pada setiap iterasi. Algoritma ini menyelesaikan submasalah yang lebih kecil pada setiap iterasi dengan mengurangi ukuran masalah secara eksponensial hingga dapat langsung diselesaikan.

Paradigma decrease-and-conquer adalah pendekatan umum untuk memecahkan masalah dengan membaginya menjadi submasalah yang lebih kecil dan menyelesainya secara iteratif. Setiap varian decrease-and-conquer memiliki karakteristiknya sendiri dan cocok untuk jenis masalah yang berbeda. Pemilihan varian tergantung pada struktur masalah, karakteristik data input, dan kompleksitas waktu yang diinginkan.

B. Sort

Sort (Pengurutan) adalah proses mengatur elemen-elemen dalam suatu himpunan data dalam urutan tertentu. Pengurutan digunakan untuk menyusun data sehingga mudah dicari, dibaca, dan diolah. Ada berbagai algoritma pengurutan yang tersedia, dan pemilihan algoritma yang tepat tergantung pada jumlah data, sifat data, dan persyaratan efisiensi.

Beberapa algoritma pengurutan yang umum digunakan meliputi:

1. Bubble Sort (Pengurutan Gelembung): Algoritma ini membandingkan pasangan elemen bersebelahan dan menukar posisinya jika urutannya tidak sesuai. Iterasi dilakukan berulang-ulang hingga seluruh himpunan data terurut dengan benar.

2. Insertion Sort (Pengurutan Sisip): Algoritma ini membagi himpunan data menjadi bagian terurut dan bagian tidak terurut. Setiap elemen tidak terurut dipindahkan ke posisi yang tepat dalam bagian terurut dengan cara membandingkannya dengan elemen-elemen sebelumnya.

3. Selection Sort (Pengurutan Seleksi): Algoritma ini secara berulang memilih elemen terkecil dari himpunan data yang belum terurut dan menukarnya dengan elemen pertama.

Langkah ini diulangi dengan himpunan data yang lebih kecil hingga seluruh himpunan terurut.

4. Merge Sort (Pengurutan Penggabungan): Algoritma ini menggunakan pendekatan divide and conquer (pembagian dan penaklukan) untuk mengurutkan himpunan data. Himpunan data dibagi menjadi bagian-bagian yang lebih kecil, kemudian bagian-bagian tersebut diurutkan secara terpisah dan digabungkan kembali dengan cara menggabungkan elemen-elemen secara berurutan.

5. Quick Sort (Pengurutan Cepat): Algoritma ini juga menggunakan pendekatan divide and conquer. Ia memilih elemen tertentu sebagai pivot, kemudian membagi himpunan data menjadi dua bagian: elemen-elemen yang lebih kecil dari pivot dan elemen-elemen yang lebih besar dari pivot. Bagian-bagian tersebut kemudian diurutkan secara terpisah dan digabungkan kembali.

6. Heap Sort (Pengurutan Tumpukan): Algoritma ini menggunakan struktur data tumpukan (heap) untuk mengurutkan himpunan data. Data dimasukkan ke dalam tumpukan dan kemudian diambil kembali secara berurutan dengan mengambil elemen teratas dari tumpukan.

Setiap algoritma pengurutan memiliki kompleksitas waktu yang berbeda-beda. Pemilihan algoritma yang tepat tergantung pada faktor-faktor seperti ukuran data, sifat data (apakah sudah hampir terurut atau tidak), serta persyaratan efisiensi waktu dan ruang.

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 algoritma pengurutan yang menggunakan pendekatan "divide and conquer" (pembagian dan penaklukan) untuk mengurutkan himpunan data. Algoritma ini membagi himpunan data menjadi dua bagian yang lebih kecil secara rekursif, kemudian menggabungkan kembali bagian-bagian tersebut dalam urutan yang benar.

Langkah-langkah dalam Mergesort adalah sebagai berikut:

1. Bagi himpunan data menjadi dua bagian yang lebih kecil, secara ideal sama ukurannya. Ini dapat dilakukan dengan mencari titik tengah himpunan data.

2. Selanjutnya, rekursif terapkan Mergesort pada kedua bagian tersebut hingga mencapai kasus dasar, yaitu ketika himpunan data hanya terdiri dari satu elemen atau kosong.

3. Setelah kedua bagian terurut, lakukan langkah penggabungan (merge) untuk menggabungkan kembali kedua bagian tersebut menjadi satu himpunan data terurut. Dalam langkah ini, bandingkan elemen-elemen dari kedua bagian dan letakkan elemen yang lebih kecil terlebih dahulu dalam urutan. Ulangi langkah ini hingga semua elemen tergabung menjadi himpunan data terurut.

4. Kembalikan himpunan data yang telah terurut.

Mergesort memiliki kompleksitas waktu O(n log n), di mana n adalah jumlah elemen dalam himpunan data. Algoritma ini cenderung efisien untuk pengurutan data dalam skala besar dan stabil dalam menjaga urutan relatif dari elemen-elemen dengan nilai yang sama.

Keunggulan Mergesort adalah kemampuannya untuk mengurutkan himpunan data dengan efisien dan konsisten. Namun, Mergesort juga membutuhkan penggunaan memori tambahan untuk menyimpan sementara bagian-bagian terurut dalam proses penggabungan.

Mergesort digunakan secara luas dalam pemrosesan data, pengolahan citra, analisis algoritma, dan berbagai aplikasi lain yang membutuhkan pengurutan efisien.

B. Quicksort

Quicksort (Pengurutan Cepat) adalah algoritma pengurutan yang menggunakan pendekatan "divide and conquer" (pembagian dan penaklukan) untuk mengurutkan himpunan data. Algoritma ini memilih suatu elemen sebagai pivot dan membagi himpunan data menjadi dua bagian: bagian elemen yang lebih kecil dari pivot dan bagian elemen yang lebih besar dari pivot. Setelah itu, algoritma tersebut secara rekursif menerapkan Quicksort pada kedua bagian tersebut.

Langkah-langkah dalam Quicksort adalah sebagai berikut:

1. Pilih salah satu elemen dalam himpunan data sebagai pivot. Biasanya, elemen pertama, terakhir, atau tengah dipilih sebagai pivot.

2. Bagi himpunan data menjadi dua bagian: bagian elemen yang lebih kecil dari pivot

dan bagian elemen yang lebih besar dari pivot. Untuk melakukan ini, iterasi melalui himpunan data dan pindahkan elemen-elemen yang lebih kecil ke satu sisi pivot dan elemen-elemen yang lebih besar ke sisi lainnya. Hal ini dapat dilakukan dengan menggunakan dua penunjuk (pointer) yang bergerak melalui himpunan data.

3. Setelah pembagian, pivot akan berada pada posisinya yang benar dalam himpunan data yang terurut. Elemen-elemen di kedua bagian dapat diurutkan secara rekursif dengan menerapkan Quicksort pada masing-masing bagian.

4. Ulangi langkah-langkah di atas untuk kedua bagian secara rekursif hingga setiap bagian hanya terdiri dari satu elemen atau kosong.

5. Setelah semua rekursi selesai, himpunan data akan terurut secara keseluruhan.

Pemilihan pivot yang baik adalah kunci untuk kinerja yang efisien dalam Quicksort. Pemilihan yang buruk dapat menghasilkan kinerja yang buruk dalam kasus terburuk, dengan kompleksitas waktu O(n^2). Namun, dengan pemilihan pivot yang baik, Quicksort memiliki kompleksitas waktu rata-rata O(n log n), di mana n adalah jumlah elemen dalam himpunan data.

Quicksort merupakan algoritma pengurutan yang cepat dan efisien dalam praktiknya. Algoritma ini digunakan secara luas dalam pemrosesan data, pengolahan citra, basis data, dan berbagai aplikasi lain yang membutuhkan pengurutan 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 proses mengurangi kompleksitas atau ukuran suatu instansi masalah ke dalam bentuk yang lebih sederhana atau lebih mudah dikelola, tetapi tetap mempertahankan informasi yang penting. Tujuan dari pemudahan instansi adalah untuk mempermudah analisis dan pemecahan masalah dengan mengurangi kompleksitas yang tidak perlu atau mengidentifikasi pola umum yang dapat digunakan dalam masalah yang lebih besar.

Beberapa teknik yang umum digunakan dalam pemudahan instansi adalah sebagai berikut:

1. Reduksi Dimensi: Jika instansi masalah memiliki banyak variabel atau fitur yang tidak relevan atau saling berkorelasi, maka dapat dilakukan reduksi dimensi untuk menghilangkan variabel yang tidak memberikan informasi penting. Misalnya, dengan menggunakan analisis faktor atau seleksi fitur, dimensi instansi masalah dapat dikurangi menjadi subset variabel yang lebih penting atau representatif.

2. Pengabstrakan: Dalam beberapa kasus, pemudahan instansi dapat dicapai dengan melakukan pengabstrakan pada data atau masalah yang kompleks. Misalnya, menggantikan detail atau elemen individual dengan representasi yang lebih umum atau tergeneralisasi. Ini membantu dalam mengurangi kompleksitas dan memungkinkan fokus pada aspek yang lebih penting atau esensial.

3. Generalisasi: Dalam pemudahan instansi, seringkali penting untuk mengidentifikasi pola umum atau kesamaan di antara instansi masalah yang berbeda. Dengan mengidentifikasi pola ini, dapat dibuat aturan atau solusi yang lebih umum yang dapat diterapkan pada berbagai instansi masalah. Hal ini membantu dalam mengurangi kerumitan dalam pemecahan masalah dan mempercepat proses analisis.

4. Penyederhanaan Representasi: Pada beberapa kasus, representasi instansi masalah dapat disederhanakan tanpa mengorbankan informasi penting. Misalnya, mengganti representasi grafis atau spasial dengan representasi teks atau simbolik yang lebih ringkas dan mudah diproses.

Pemudahan instansi adalah langkah penting dalam analisis dan pemecahan masalah. Dengan mengurangi kompleksitas dan menyederhanakan instansi masalah, dapat mempercepat waktu komputasi, meningkatkan efisiensi, dan memungkinkan fokus pada aspek yang lebih penting atau relevan.

B. Representation Change

Representation Change (Perubahan Representasi) adalah proses mengubah representasi atau format data dari satu bentuk ke bentuk lain yang lebih sesuai atau lebih mudah untuk dikelola atau diproses. Tujuan utama perubahan representasi adalah untuk meningkatkan efisiensi, memudahkan pemrosesan data, atau memfasilitasi analisis lebih lanjut.

Perubahan representasi dapat melibatkan transformasi data dari satu jenis representasi ke jenis representasi lainnya, seperti:

1. Perubahan Format Data: Ini melibatkan mengubah format data dari satu bentuk ke bentuk lain yang lebih sesuai untuk pemrosesan atau analisis. Misalnya, mengubah data dari format teks menjadi format numerik, atau mengubah data dari format tabel

menjadi format grafis.

2. Konversi Struktur Data: Ini melibatkan mengubah struktur data dari satu bentuk ke bentuk lain yang lebih cocok untuk kebutuhan tertentu. Misalnya, mengkonversi data dari array menjadi linked list, atau mengubah data dari bentuk hierarkis menjadi bentuk grafik.

3. Transformasi Representasi: Ini melibatkan transformasi data dari satu representasi ke representasi lainnya, yang dapat melibatkan perhitungan atau manipulasi khusus. Misalnya, mengubah data dari representasi spasial ke representasi frekuensi melalui transformasi Fourier, atau mengubah data dari representasi mentah menjadi representasi normalisasi.

4. Agregasi Data: Ini melibatkan menggabungkan atau mengelompokkan data dalam bentuk yang berbeda menjadi satu kesatuan. Misalnya, menggabungkan beberapa file data menjadi satu file, atau mengelompokkan data dalam bentuk hierarki untuk analisis yang lebih terstruktur.

Perubahan representasi dapat membantu dalam meningkatkan efisiensi pemrosesan data, mengurangi kompleksitas, atau memungkinkan pemrosesan atau analisis lebih lanjut. Hal ini sering digunakan dalam berbagai bidang, termasuk pengolahan data, analisis data, kecerdasan buatan, komputasi grafis, dan banyak lagi.

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 salah satu metode pengurutan yang digunakan untuk mengurutkan elemen-elemen dalam sebuah rangkaian data dengan memanfaatkan perhitungan frekuensi kemunculan setiap elemen. Metode ini terutama efisien ketika rentang nilai elemen terbatas dan ketika jumlah elemen relatif besar.

Langkah-langkah dalam sorting by counting adalah sebagai berikut:

1. Menghitung Frekuensi Kemunculan Setiap Elemen: Langkah pertama adalah menghitung berapa kali setiap elemen muncul dalam rangkaian data. Hal ini dapat dilakukan dengan membentuk sebuah array atau struktur data lain yang digunakan untuk menyimpan frekuensi kemunculan setiap elemen.

2. Menghitung Jumlah Kumulatif: Selanjutnya, menghitung jumlah kumulatif frekuensi kemunculan setiap elemen. Hal ini dilakukan dengan menjumlahkan frekuensi kemunculan setiap elemen dengan frekuensi elemen sebelumnya. Jumlah kumulatif ini digunakan untuk menentukan posisi akhir setiap elemen dalam hasil pengurutan.

3. Menempatkan Elemen pada Posisi yang Tepat: Setelah menghitung jumlah kumulatif, elemen-elemen ditempatkan pada posisi yang tepat dalam rangkaian hasil pengurutan. Hal ini dilakukan dengan memindahkan setiap elemen dari rangkaian data awal ke posisi yang ditentukan berdasarkan frekuensi kemunculan dan jumlah kumulatif.

4. Membentuk Rangkaian Pengurutan yang Terurut: Setelah elemen-elemen ditempatkan pada posisi yang tepat, membentuk rangkaian data yang terurut dengan menggabungkan elemen-elemen tersebut.

Metode pengurutan dengan menghitung efektif digunakan dalam kasus-kasus di mana rentang nilai elemen terbatas dan jumlah elemen relatif besar. Namun, metode ini memiliki keterbatasan, yaitu hanya dapat digunakan untuk mengurutkan elemen-elemen yang dapat dihitung atau diindeks secara langsung.

Pengurutan dengan menghitung sering digunakan dalam kasus-kasus khusus, seperti pengurutan karakter dalam teks atau pengurutan elemen-elemen non-negatif dalam rentang nilai yang terbatas.

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:

1. 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.

2. 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.

3. 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.

4. 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.

5. 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

The Knapsack Problem (Masalah Knapsack) adalah salah satu masalah optimasi yang umum dalam ilmu komputer dan matematika. Masalah ini melibatkan pemilihan item dari sekumpulan item yang tersedia untuk dimasukkan ke dalam knapsack (tas) dengan kapasitas terbatas. Setiap item memiliki nilai tertentu dan membutuhkan ruang dalam knapsack. Tujuannya adalah memaksimalkan nilai total item yang dimasukkan ke dalam knapsack tanpa melampaui kapasitasnya.

Ada dua versi utama dari Knapsack Problem:

1. 0/1 Knapsack Problem (Masalah Knapsack 0/1): Pada versi ini, setiap item hanya dapat dipilih secara biner, yaitu entah item tersebut diambil sepenuhnya atau tidak diambil sama sekali. Tidak ada opsi parsial untuk item.

2. Knapsack Problem dengan Pengisian Parsial (Fractional Knapsack Problem): Pada versi ini, setiap item dapat dipilih secara parsial, artinya dapat diambil sebagian dengan bobot yang sesuai.

Untuk memecahkan Knapsack Problem, dapat digunakan berbagai algoritma, termasuk algoritma brute-force, algoritma dinamik, atau algoritma greedy. Algoritma yang paling efisien tergantung pada batasan dan tujuan masalah yang spesifik.

Memory Functions (Fungsi Memori) adalah fungsi yang digunakan dalam algoritma pemrograman dinamik untuk mengingat atau menyimpan hasil perhitungan

sebelumnya. Dalam konteks Knapsack Problem, fungsi memori dapat digunakan untuk menyimpan hasil perhitungan submasalah yang telah diselesaikan, sehingga dapat digunakan kembali saat memecahkan masalah yang lebih besar.

Dalam algoritma pemrograman dinamik untuk Knapsack Problem, fungsi memori sering direpresentasikan dengan menggunakan tabel atau matriks. Setiap sel dalam tabel memori menyimpan hasil perhitungan untuk submasalah tertentu. Dengan menggunakan fungsi memori, algoritma dapat menghindari pengulangan perhitungan yang tidak perlu dan menghemat waktu komputasi.

Fungsi memori juga digunakan dalam berbagai masalah optimasi lainnya untuk menyimpan hasil perhitungan yang dapat digunakan kembali. Hal ini memungkinkan algoritma untuk mempercepat waktu eksekusi dengan menghindari perhitungan yang redundan.

Penggunaan fungsi memori dalam algoritma pemrograman dinamik sangat penting untuk meningkatkan efisiensi dan kinerja algoritma, terutama dalam kasus masalah dengan kompleksitas waktu yang tinggi.

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 Prim (Prim's Algorithm) adalah algoritma yang digunakan untuk menemukan Minimum Spanning Tree (MST) atau pohon rentang minimum dalam sebuah graf berbobot. Minimum Spanning Tree adalah subset pohon yang menghubungkan semua simpul dalam graf dengan bobot total minimum.

Langkah-langkah utama dalam Algoritma Prim adalah sebagai berikut:

1. Pilih simpul awal secara acak dari graf sebagai simpul awal MST.

2. Inisialisasi himpunan MST dengan simpul awal dan himpunan pengunjung (visited set) dengan simpul awal tersebut.

3. Selama himpunan MST belum mencakup semua simpul dalam graf:
   - Temukan tepi dengan bobot minimum yang menghubungkan simpul dalam MST dengan simpul di luar MST.
   - Tambahkan simpul tersebut ke dalam MST dan tandai sebagai sudah dikunjungi.
   - Perbarui himpunan pengunjung dengan simpul yang baru ditambahkan.

4. Ulangi langkah 3 hingga himpunan MST mencakup semua simpul dalam graf.

Pada setiap langkah, algoritma memilih tepi dengan bobot minimum yang menghubungkan simpul dalam MST dengan simpul di luar MST. Dengan cara ini, algoritma membangun MST secara bertahap dengan memilih tepi terkecil pada setiap langkah. Algoritma ini berhenti saat semua simpul telah dimasukkan ke dalam MST.

Algoritma Prim dapat digunakan pada graf berarah atau tidak berarah dengan bobot positif atau negatif, tetapi tidak boleh ada siklus negatif. Algoritma ini memberikan solusi optimal dalam hal membangun MST dengan bobot total minimum.

Algoritma Prim memiliki kompleksitas waktu O(V^2) dalam implementasi sederhana menggunakan matriks kedekatan (adjacency matrix), di mana V adalah jumlah simpul dalam graf. Namun, dengan implementasi yang lebih efisien menggunakan struktur data seperti Heap biner atau Fibonacci Heap, kompleksitas waktu dapat ditingkatkan menjadi O(E log V), di mana E adalah jumlah tepi dalam graf.

Algoritma Prim sangat penting dalam teori graf dan memiliki berbagai aplikasi, termasuk dalam jaringan komunikasi, desain jaringan, pemetaan jaringan, pemrograman linier, dan optimasi jaringan.

B. Kruskal's Algorithm

Algoritma Kruskal (Kruskal's Algorithm) adalah algoritma yang digunakan untuk menemukan Minimum Spanning Tree (MST) atau pohon rentang minimum dalam sebuah graf berbobot. Minimum Spanning Tree adalah subset pohon yang menghubungkan semua simpul dalam graf dengan bobot total minimum.

Langkah-langkah utama dalam Algoritma Kruskal adalah sebagai berikut:

1. Urutkan semua tepi dalam graf berdasarkan bobotnya secara tidak menurun.
2. Inisialisasi himpunan MST kosong.

3. Iterasi melalui setiap tepi dalam urutan terurut:
   - Jika menambahkan tepi tersebut ke dalam MST tidak membentuk siklus, tambahkan tepi tersebut ke dalam MST.
   - Jika menambahkan tepi tersebut membentuk siklus, abaikan tepi tersebut dan lanjutkan ke tepi berikutnya.
4. Ulangi langkah 3 hingga semua simpul terhubung dalam MST atau semua tepi telah diproses.

Pada setiap langkah, algoritma memilih tepi dengan bobot terkecil yang tidak membentuk siklus dalam MST yang sedang dibangun. Dengan cara ini, algoritma secara bertahap membangun MST dengan memilih tepi terkecil pada setiap langkah. Algoritma ini berhenti saat semua simpul terhubung dalam MST atau semua tepi telah diproses.

Algoritma Kruskal dapat digunakan pada graf berarah atau tidak berarah dengan bobot positif atau negatif, tetapi tidak boleh ada siklus negatif. Algoritma ini memberikan solusi optimal dalam hal membangun MST dengan bobot total minimum.

Algoritma Kruskal memiliki kompleksitas waktu O(E log V), di mana E adalah jumlah tepi dalam graf dan V adalah jumlah simpul dalam graf. Kompleksitas ini disebabkan oleh langkah pengurutan tepi dan penggunaan struktur data disjoint set (union-find) untuk mendeteksi siklus. Struktur data ini memungkinkan deteksi siklus dengan waktu konstan.

Algoritma Kruskal sangat penting dalam teori graf dan memiliki berbagai aplikasi, termasuk dalam jaringan komunikasi, desain jaringan, pemetaan jaringan, pemrograman linier, dan optimasi jaringan.

C. Djikstra’s Algorithm

Algoritma Dijkstra (Dijkstra's Algorithm) adalah algoritma yang digunakan untuk menemukan jalur terpendek antara simpul awal dan semua simpul lainnya dalam graf berbobot non-negatif. Algoritma ini dapat diterapkan pada graf berarah atau tidak berarah.

Langkah-langkah utama dalam Algoritma Dijkstra adalah sebagai berikut:

1. Inisialisasi:
   - Tetapkan simpul awal sebagai simpul sumber.
   - Atur jarak awal ke simpul awal sebagai 0 dan jarak ke semua simpul lainnya sebagai tak terbatas.
   - Tetapkan simpul awal sebagai simpul saat ini.

2. Selama masih ada simpul yang belum dikunjungi:
   - Periksa semua tetangga simpul saat ini yang belum dikunjungi.
   - Hitung jarak baru ke setiap tetangga melalui simpul saat ini.
   - Jika jarak baru lebih kecil dari jarak sebelumnya, perbarui jarak ke tetangga tersebut.
   - Tandai simpul saat ini sebagai sudah dikunjungi.

3. Ulangi langkah 2 hingga semua simpul telah dikunjungi atau semua jarak terendah telah ditentukan.

Setelah algoritma selesai, jarak terpendek dari simpul awal ke setiap simpul lainnya dapat ditemukan. Algoritma Dijkstra juga menghasilkan informasi jalur terpendek, yang dapat digunakan untuk merekonstruksi jalur terpendek antara simpul awal dan simpul tujuan.

Algoritma Dijkstra bekerja dengan asumsi bahwa bobot tepi dalam graf non-negatif. Jika ada tepi dengan bobot negatif, maka algoritma ini tidak dapat menghasilkan hasil yang benar. Untuk mengatasi graf dengan bobot negatif, dapat digunakan algoritma seperti Algoritma Bellman-Ford.

Kompleksitas waktu Algoritma Dijkstra tergantung pada implementasinya. Dalam implementasi sederhana menggunakan matriks kedekatan (adjacency matrix), kompleksitasnya adalah O(V^2), di mana V adalah jumlah simpul dalam graf. Namun, dengan implementasi yang lebih efisien menggunakan struktur data seperti Heap biner atau Fibonacci Heap, kompleksitas waktu dapat ditingkatkan menjadi O((E + V) log V), di mana E adalah jumlah tepi dalam graf.

Algoritma Dijkstra sangat penting dalam teori graf dan memiliki berbagai aplikasi, termasuk dalam perencanaan jaringan, rute terpendek, navigasi, dan pemetaan jalan.

D. Huffman Tress and Codes

Pohon Huffman (Huffman Tree) dan Kode Huffman (Huffman Codes) adalah teknik kompresi data yang efisien yang digunakan dalam teori informasi dan komputasi.

Pohon Huffman:

Pohon Huffman adalah sebuah pohon biner khusus yang digunakan untuk menghasilkan kode Huffman. Pohon ini dibangun menggunakan algoritma Huffman yang membangun pohon berdasarkan frekuensi kemunculan karakter dalam sebuah teks atau data. Pohon Huffman memiliki sifat khusus di mana karakter dengan frekuensi yang lebih tinggi memiliki jalur yang lebih pendek dalam pohon, sedangkan karakter dengan frekuensi yang lebih rendah memiliki jalur yang lebih panjang.

Langkah-langkah untuk membangun pohon Huffman:

1. Hitung frekuensi kemunculan setiap karakter dalam teks atau data.

2. Buat simpul untuk setiap karakter dengan frekuensi dan tambahkan ke dalam sebuah antrian prioritas.

3. Ambil dua simpul dengan frekuensi terendah dari antrian prioritas, dan gabungkan mereka menjadi satu simpul baru dengan frekuensi yang merupakan penjumlahan frekuensi kedua simpul tersebut. Simpan simpul baru ke dalam antrian prioritas.

4. Ulangi langkah 3 hingga hanya tersisa satu simpul dalam antrian prioritas, yang akan menjadi akar dari pohon Huffman.

Kode Huffman:

Kode Huffman adalah kode biner yang digunakan untuk mewakili karakter dalam teks atau data dengan panjang yang bervariasi berdasarkan frekuensi kemunculan karakter. Kode Huffman diperoleh dengan melakukan traversal dari akar pohon Huffman ke setiap daun, dan atributkan 0 atau 1 untuk setiap tepi yang diambil saat melakukan traversal.

Sifat khusus pohon Huffman memastikan bahwa tidak ada kode yang merupakan prefiks dari kode lainnya, sehingga kode Huffman bersifat unik dan dapat didekode dengan jelas. Ini disebut sebagai kode prefiks atau kode kata kata berimbang.

Kode Huffman memberikan efisiensi kompresi data yang tinggi untuk karakter yang memiliki frekuensi kemunculan yang tinggi, karena karakter tersebut diberikan kode yang lebih pendek. Sedangkan karakter dengan frekuensi yang lebih rendah diberikan kode yang lebih panjang.

Pohon Huffman dan Kode Huffman digunakan dalam berbagai aplikasi kompresi data seperti kompresi teks, audio, dan video. Mereka juga digunakan dalam protokol komunikasi, penyimpanan data, dan dalam implementasi format file yang dioptimalkan.

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.

Metode Simpleks telah menjadi alat penting dalam pemrograman linier dan digunakan dalam berbagai aplikasi seperti perencanaan produksi, alokasi sumber daya, pemodelan ekonomi, dan optimasi bisnis.

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).

Matching maksimum dalam graf bipartit memiliki berbagai aplikasi, seperti dalam pencocokan pasangan dalam jadwal kursus, pencocokan iklan dengan pengguna, pencocokan pasangan dalam situs kencan online, dan dalam pemodelan dan optimasi masalah yang melibatkan himpunan dan relasi dua sisi.

D. The Stable Marriage Problem

Masalah pernikahan stabil (Stable Marriage Problem) adalah masalah yang melibatkan pencocokan pasangan antara dua himpunan orang yang memiliki preferensi terhadap pasangan dari himpunan lainnya. Dalam konteks ini, terdapat himpunan pria dan himpunan wanita yang memiliki daftar preferensi terhadap calon pasangan dari himpunan lainnya.

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.

Langkah-langkah dalam menyelesaikan masalah pernikahan stabil adalah sebagai berikut:

1. Setiap pria dan wanita memberikan peringkat terhadap anggota dari himpunan lainnya berdasarkan preferensinya.

2. Setiap pria dan wanita awalnya belum dipasangkan.

3. Ulangi langkah-langkah berikut hingga semua orang dipasangkan:

- Pilih seorang pria yang belum dipasangkan.

- Pria tersebut mengajukan tawaran kepada wanita pilihannya yang belum menerima tawaran dari pria lain atau lebih memilih pria saat ini daripada pria yang mengajukan tawaran.

- Jika wanita tersebut belum dipasangkan, mereka menjadi pasangan. Jika wanita tersebut sudah dipasangkan, maka perlu dilakukan pengecekan preferensi wanita tersebut:

* Jika wanita lebih memilih pria saat ini daripada pria yang mengajukan tawaran, maka tidak ada perubahan pasangan.* Jika wanita lebih memilih pria yang mengajukan tawaran daripada pria saat ini, maka pasangan sebelumnya harus diputuskan dan wanita tersebut dipasangkan dengan pria yang mengajukan tawaran.

4. Setelah semua orang dipasangkan, pencocokan yang dihasilkan akan menjadi pencocokan pernikahan stabil.

Masalah pernikahan stabil memiliki aplikasi dalam berbagai konteks kehidupan nyata, seperti pencocokan pasangan dalam pernikahan, pencocokan dokter dengan rumah sakit, pencocokan mahasiswa dengan pembimbing akademik, dan pengaturan penugasan pekerjaan. Algoritma Gale-Shapley adalah salah satu algoritma terkenal yang digunakan untuk memecahkan masalah pernikahan stabil.

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 (pohon keputusan) adalah salah satu metode dalam pembelajaran mesin yang digunakan untuk mengambil keputusan atau membuat prediksi berdasarkan serangkaian keputusan atau aturan yang terstruktur dalam bentuk pohon. Pohon keputusan menggambarkan aliran keputusan yang dimulai dari simpul akar (root node) dan berakhir pada simpul daun (leaf node) yang berisi keputusan atau prediksi.

Pada setiap simpul dalam pohon keputusan, terdapat kondisi atau fitur yang digunakan untuk membagi data menjadi kelompok yang lebih kecil. Setiap cabang atau anak simpul menggambarkan nilai atau kondisi yang berbeda dari fitur yang digunakan. Proses ini berlanjut hingga mencapai simpul daun, di mana keputusan atau prediksi

diberikan berdasarkan kelompok data yang telah melewati serangkaian keputusan.

Beberapa istilah yang terkait dengan pohon keputusan adalah sebagai berikut:

1. Simpul Akar (Root Node): Simpul pertama dalam pohon keputusan yang mewakili kondisi awal atau fitur utama yang digunakan untuk membagi data.

2. Simpul Cabang (Branch Node): Simpul yang terletak di antara simpul akar dan simpul daun, mewakili kondisi atau aturan tambahan yang membagi data lebih lanjut.

3. Simpul Daun (Leaf Node): Simpul terakhir dalam pohon keputusan yang menghasilkan keputusan atau prediksi. Setiap simpul daun mewakili kelas atau nilai target yang diharapkan.

4. Kondisi atau Aturan: Fitur atau variabel yang digunakan untuk membagi data menjadi kelompok yang lebih kecil di setiap simpul.

Proses pembuatan pohon keputusan melibatkan pemilihan fitur terbaik untuk membagi data pada setiap simpul, yang ditentukan berdasarkan metrik seperti gain informasi atau indeks Gini. Algoritma yang umum digunakan untuk membangun pohon keputusan adalah algoritma ID3 (Iterative Dichotomiser 3), C4.5, CART (Classification and Regression Trees), dan Random Forest.

Pohon keputusan memiliki kelebihan dalam kemudahan interpretasi, kemampuan menangani data kategorikal dan numerikal, serta kemampuan menangani masalah klasifikasi dan regresi. Namun, pohon keputusan juga dapat cenderung overfitting dan sensitif terhadap perubahan kecil dalam data. Oleh karena itu, teknik pruning dan ensemble learning sering digunakan untuk meningkatkan kinerja dan mengurangi overfitting pada pohon keputusan.

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.

Masalah NP-complete merupakan tantangan yang signifikan dalam teori kompleksitas komputasi. Hingga saat ini, belum ditemukan algoritma yang dapat menyelesaikan semua masalah NP-complete dalam waktu polinomial. P versus NP merupakan salah satu masalah terbuka terpenting dalam teori komputasi yang mencoba menjawab apakah P = NP atau P ≠ NP. Jika P = NP, maka semua masalah dalam NP dapat dipecahkan dengan cepat dan efisien. Namun, jika P ≠ NP, maka ada masalah yang tidak memiliki solusi yang efisien.

D. Challenge of Numerical Algorithms

Tantangan dalam Algoritma Numerik

Algoritma numerik menghadapi beberapa tantangan karena sifat perhitungan numerik dan keterbatasan sistem komputer. Beberapa tantangan utama meliputi:

1. Ketelitian: Algoritma numerik sering melibatkan pendekatan dan kesalahan pembulatan akibat presisi terbatas dari aritmetika komputer. Kesalahan ini dapat terakumulasi dan mempengaruhi akurasi hasil akhir. Merancang algoritma yang meminimalkan dan mengendalikan kesalahan ini sangat penting untuk mendapatkan solusi numerik yang akurat.

2. Stabilitas: Stabilitas mengacu pada kemampuan suatu algoritma untuk menghasilkan hasil yang konsisten dan dapat diandalkan bahkan dalam keberadaan gangguan kecil pada data masukan. Algoritma numerik bisa sensitif terhadap perubahan kecil, dan ketidakstabilan dapat mengakibatkan kesalahan signifikan atau solusi yang salah. Memastikan stabilitas numerik sangat penting untuk mendapatkan hasil yang dapat diandalkan.

3. Efisiensi: Algoritma numerik umumnya melibatkan perhitungan matematis yang kompleks, dan efisiensi mereka menjadi faktor penting. Tantangannya adalah merancang algoritma yang memberikan hasil yang akurat sambil meminimalkan sumber daya komputasi seperti waktu dan memori. Menyeimbangkan akurasi dan efisiensi seringkali merupakan trade-off yang perlu dipertimbangkan dengan hati-hati.

4. Skalabilitas: Algoritma numerik harus mampu menangani masalah dalam skala besar dengan efisien. Seiring bertambahnya ukuran data masukan, kebutuhan komputasi dapat tumbuh dengan cepat. Mengembangkan algoritma yang skalabel untuk menangani kumpulan data besar dan masalah kompleks adalah tantangan utama.

5. Konvergensi: Banyak algoritma numerik melibatkan proses iteratif, di mana solusi membaik dengan setiap iterasi hingga mencapai tingkat akurasi yang diinginkan. Memastikan konvergensi algoritma iteratif ini penting untuk menjamin bahwa mereka akan konvergen ke solusi yang benar dan tidak divergen atau terjebak dalam optimum lokal.

6. Ketangguhan: Algoritma numerik harus tangguh dan mampu menangani berbagai macam masukan, termasuk kasus baik dan kasus yang ekstrem. Mereka harus dapat menangani berbagai jenis distribusi data, menangani nilai ekstrem, dan mengatasi kasus-kasus di mana masalah memiliki karakteristik atau kompleksitas tertentu.

7. Paralelisasi: Dengan ketersediaan sumber daya komputasi paralel yang semakin meningkat, merancang algoritma numerik yang dapat memanfaatkan pemrosesan paralel menjadi krusial. Mendistribusikan beban komputasi secara efisien di antara beberapa prosesor atau inti sambil mempertahankan akurasi dan meminimalkan biaya komunikasi menjadi tantangan tersendiri.

Menghadapi tantangan ini membutuhkan pemahaman mendalam tentang metode numerik

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 metode yang digunakan dalam pemrograman dan optimasi untuk menyelesaikan masalah pencarian solusi secara sistematis dengan membagi masalah menjadi submasalah yang lebih kecil dan menghindari penjelajahan ruang pencarian yang tidak perlu.

Metode BnB melibatkan dua tahap utama:

1. Penciptaan cabang (Branching):
   - Tahap ini melibatkan pembagian masalah awal menjadi beberapa submasalah yang lebih kecil. Setiap submasalah mewakili suatu cabang dalam pohon pencarian.

- Biasanya, cabang dibuat dengan memilih suatu variabel dan menghasilkan beberapa kemungkinan nilai atau keputusan yang mungkin.
- Dengan melakukan branching, pohon pencarian terbentuk, dengan setiap cabang mewakili solusi potensial.

2. Batasan (Bounding):
- Tahap ini melibatkan penggunaan batasan atau pembatasan untuk menghilangkan submasalah yang tidak mungkin menghasilkan solusi optimal.
- Dalam setiap cabang, batasan digunakan untuk menentukan apakah submasalah tersebut perlu dijelajahi lebih lanjut atau tidak.
- Batasan dapat berupa batasan bawah (lower bound) dan batasan atas (upper bound). Batasan bawah digunakan untuk membatasi solusi terbaik yang telah ditemukan, sementara batasan atas digunakan untuk membatasi pencarian ruang solusi yang mungkin lebih buruk dari solusi terbaik yang telah ditemukan.

Proses BnB dilakukan secara rekursif, dimulai dari akar pohon pencarian, dan dilanjutkan ke setiap cabang dan subcabang yang dihasilkan. Pada setiap langkah, metode BnB memilih cabang terbaik berdasarkan batasan dan keputusan yang diambil. Jika submasalah tidak memenuhi batasan atau dianggap tidak optimal, cabang tersebut dapat dipotong (pruned), dan penjelajahan ruang pencarian hanya dilakukan pada cabang yang menjanjikan.

Metode BnB efektif dalam menyelesaikan masalah optimasi dan pencarian solusi dengan ruang pencarian yang besar. Dengan menggunakan teknik pruning dan batasan, metode BnB dapat menghindari penjelajahan yang tidak perlu dan mendekati solusi optimal dengan cepat. Namun, kompleksitas waktu dari metode BnB tetap tergantung pada karakteristik masalah dan kualitas batasan yang digunakan.

C. Algorithms for Solving Nonlinear Problems

Algoritma untuk Menyelesaikan Masalah Nonlinear

Terdapat beberapa algoritma yang tersedia untuk menyelesaikan masalah nonlinear, masing-masing dengan pendekatan dan karakteristiknya sendiri. Beberapa algoritma yang umum digunakan untuk menyelesaikan masalah nonlinear meliputi:

1. Metode Newton: Metode Newton adalah metode numerik iteratif yang digunakan untuk mencari akar persamaan nonlinear. Metode ini menggunakan tebakan awal dan secara berulang memperbaikinya untuk konvergen ke solusi. Metode ini bergantung pada perhitungan turunan untuk memperbarui tebakan pada setiap iterasi.

2. Descent Gradien: Descent gradien adalah algoritma optimisasi yang digunakan untuk mencari nilai minimum (atau maksimum) dari fungsi nonlinear. Algoritma ini secara iteratif memperbarui parameter dalam arah penurunan gradien fungsi. Descent gradien banyak digunakan dalam pembelajaran mesin dan masalah optimisasi.

3. Algoritma Levenberg-Marquardt: Algoritma Levenberg-Marquardt adalah algoritma optimisasi yang umum digunakan untuk masalah kuadrat terkecil nonlinear. Algoritma ini menggabungkan metode Gauss-Newton dengan istilah regulasi untuk memastikan stabilitas dan konvergensi.

4. Metode Broyden: Metode Broyden adalah metode iteratif yang digunakan untuk memecahkan sistem persamaan nonlinear. Metode ini memperkirakan matriks Jacobian dengan menggunakan pembaruan berdasarkan perbedaan nilai fungsi. Metode ini sangat berguna ketika menghitung Jacobian yang tepat membutuhkan komputasi yang mahal.

5. Metode Trust-Region: Metode trust-region adalah algoritma optimisasi yang secara iteratif menyelesaikan submasalah dalam wilayah kepercayaan yang ditentukan. Metode ini menggunakan model kuadrat lokal untuk mengaproksimasi fungsi tujuan dan memperbarui solusi dalam batasan wilayah kepercayaan.

6. Algoritma Genetika: Algoritma genetika adalah algoritma pencarian berbasis populasi yang terinspirasi dari proses seleksi alam. Algoritma ini menggunakan kombinasi operasi seleksi, crossover, dan mutasi untuk mengembangkan populasi solusi potensial dan konvergen ke solusi optimal.

7. Simulated Annealing: Simulated annealing adalah algoritma optimisasi stokastik yang meniru proses anil pada logam. Algoritma ini dimulai dengan solusi awal dan mengeksplorasi ruang solusi dengan menerima pergerakan suboptimal dengan probabilitas tertentu. Seiring waktu berjalan, probabilitas penerimaan berkurang, sehingga algoritma dapat konvergen ke optimum global.

Ini hanya beberapa contoh algoritma yang digunakan untuk menyelesaikan masalah nonlinear. Pemilihan algoritma tergantung pada karakteristik spesifik masalah, seperti fungsi tujuan, batasan, dan sumber daya komputasi yang tersedia. Seringkali diperlukan eksperimen dengan berbagai algoritma dan penyesuaian sesuai dengan masalah yang dihadapi untuk mencapai hasil terbaik.

DAFTAR PUSTAKA

- 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).
- 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.