MAKALAH
“PERANCANGAN ANALISIS DAN ALGORITMA”

“Disusun sebagai tugas akhir pada mata kuliah Perancangan dan Analisis Algoritma , dengan Dosen Randi Proska Sandra, M.Sc dan Widya Darwin S.Pd., M.Pd.T”

Disusun Oleh :

Nama Anggota	: Mardhyah Fathania Izzati
NIM	: 21346013
Prodi	: Informatika

PROGRAM STUDI INFORMATIKA

JURUSAN TENIK ELEKRONIKA

FAKULTAS TEKNIK

UNIVERSITAS NEGERI PADANG

2023

KATA PENGANTAR

Rasa syukur kita hanya milik Allah SWT atas segala semua rahmatnya, sehingga saya dapat menyelesaikan makalah 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 makalahh 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.

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 makalah ini akan terselesaikan dengan benar.

DAFTAR ISI

BAB I

PENGANTAR ANALISIS ALGORITMA

A. Pengertian Algoritma

Algoritma adalah serangkaian langkah-langkah logis dan terstruktur yang digunakan untuk menyelesaikan masalah atau mencapai tujuan tertentu. Secara umum, algoritma dapat dianggap sebagai resep atau panduan yang mengarahkan langkah-langkah yang harus diambil untuk mencapai hasil yang diinginkan. Algoritma digunakan dalam berbagai bidang, termasuk matematika, ilmu komputer, sains data, dan pemrograman.

Algoritma harus memiliki beberapa karakteristik penting, yaitu:

Terdefinisi dengan baik: Setiap langkah dalam algoritma harus jelas dan dapat dimengerti secara tegas. Instruksi harus eksplisit dan tidak ambigu agar dapat diikuti dengan benar.

Terurut secara sistematis: Langkah-langkah dalam algoritma harus diatur secara logis dan terurut. Setiap langkah harus mengikuti langkah sebelumnya dengan benar sehingga menghasilkan urutan yang koheren.

Efektif: Algoritma harus efisien dan dapat menyelesaikan masalah dalam jumlah waktu yang wajar. Ini berkaitan dengan kompleksitas waktu dan ruang algoritma.

Input dan Output: Algoritma menerima input, yang bisa berupa data atau informasi, dan menghasilkan output yang merupakan hasil dari proses algoritma. Input dan output harus jelas dan sesuai dengan kebutuhan masalah yang ingin diselesaikan.

B. Dasar Algoritma

Dasar-dasar algoritma meliputi beberapa konsep utama yang penting dalam merancang algoritma yang efektif. Beberapa dasar algoritma yang penting meliputi:

1. Pernyataan atau Instruksi: Algoritma terdiri dari serangkaian pernyataan atau instruksi yang menggambarkan tindakan-tindakan yang harus dilakukan. Pernyataan ini bisa berupa operasi matematika, operasi logika, pengulangan (looping), pemilihan (decision), atau pemanggilan fungsi.
2. Variabel: Algoritma menggunakan variabel untuk menyimpan dan memanipulasi data. Variabel dapat digunakan untuk menyimpan angka, teks, atau tipe data lainnya.

Variabel memiliki nama yang unik dan dapat diakses dan dimodifikasi selama eksekusi algoritma.

3. Pengulangan (Looping): Pengulangan memungkinkan eksekusi instruksi yang sama secara berulang berdasarkan kondisi tertentu. Hal ini memungkinkan penanganan data yang berulang atau proses yang perlu diulang beberapa kali hingga kondisi tertentu tercapai.
4. Pemilihan (Decision): Pemilihan memungkinkan algoritma untuk memilih tindakan yang akan diambil berdasarkan kondisi tertentu. Algoritma dapat menggunakan pernyataan if-else atau switch-case untuk memeriksa kondisi dan memilih jalur yang tepat.

C. Problem Solving

Problem Solving adalah kemampuan untuk memecahkan masalah secara efektif dengan menggunakan algoritma. Proses problem solving melibatkan langkah-langkah berikut:

1. Memahami masalah: Langkah pertama dalam problem solving adalah memahami masalah dengan baik. Ini melibatkan mengidentifikasi masalah, memahami persyaratan, dan menentukan tujuan yang ingin dicapai.
2. Merancang algoritma: Setelah memahami masalah, langkah selanjutnya adalah merancang algoritma yang akan menyelesaikan masalah tersebut. Algoritma harus dirancang secara logis dan sistematis, dan dapat menghasilkan hasil yang diinginkan.
3. Implementasi algoritma: Setelah merancang algoritma, langkah selanjutnya adalah mengimplementasikan algoritma dalam bahasa pemrograman yang sesuai. Ini melibatkan menuliskan kode yang mengikuti langkah-langkah yang telah dirancang.
4. Pengujian dan evaluasi: Setelah mengimplementasikan algoritma, algoritma tersebut harus diuji dengan menggunakan berbagai kasus uji untuk memastikan bahwa algoritma berfungsi dengan benar. Jika ada kesalahan atau kekurangan, algoritma perlu diperbaiki.
5. Analisis dan pemeliharaan: Setelah algoritma berfungsi dengan baik, perlu dilakukan analisis untuk mengevaluasi kinerja algoritma, efisiensi, dan skalabilitasnya. Jika diperlukan, algoritma dapat ditingkatkan atau dioptimalkan untuk meningkatkan kinerja atau menangani skala masalah yang lebih besar.
6. Dalam problem solving, penting untuk memiliki pemahaman yang kuat tentang dasar-dasar algoritma dan kemampuan untuk menerapkannya dengan baik. Dengan

menggunakan pendekatan yang sistematis dan logis, kita dapat mengatasi masalah dengan lebih efisien dan efektif.

BAB II

ANALISIS
FRAMEWORK

A. Measuring an Input Size

Pengukuran ukuran input adalah aspek penting dalam menganalisis efisiensi algoritma. Hal ini berkaitan dengan menentukan ukuran atau besaran dari masalah yang dipecahkan oleh algoritma. Ukuran input dapat bervariasi tergantung pada masalahnya dan biasanya diukur berdasarkan jumlah elemen atau jumlah data yang diproses oleh algoritma.

Sebagai contoh, dalam algoritma pengurutan, ukuran input dapat berupa jumlah elemen dalam array yang akan diurutkan. Dalam algoritma penelusuran graf, ukuran input dapat berupa jumlah simpul atau sisi dalam graf. Dengan memperkirakan ukuran input, kita dapat menganalisis bagaimana kinerja algoritma berkembang seiring bertambahnya ukuran input.

Pengukuran ukuran input memungkinkan kita untuk menganalisis efisiensi suatu algoritma dan membuat perbandingan antara algoritma yang berbeda. Hal ini membantu kita memahami bagaimana waktu eksekusi atau kebutuhan ruang algoritma berkembang saat ukuran input meningkat. Informasi ini penting dalam memilih algoritma yang paling sesuai untuk masalah tertentu dan mengoptimalkan kinerjanya.

B. Unit of Measuring Running Time

Unit pengukuran waktu eksekusi adalah cara untuk mengukur waktu yang dibutuhkan oleh suatu algoritma untuk menyelesaikan tugasnya. Waktu eksekusi sering digunakan sebagai indikator utama dalam menganalisis efisiensi algoritma.

Biasanya, waktu eksekusi diukur dalam satuan waktu tertentu, seperti detik, milidetik, mikrodetik, atau nanodetik. Pengukuran waktu eksekusi dapat dilakukan menggunakan perangkat lunak atau alat khusus yang dapat mengukur waktu secara presisi.

Unit pengukuran waktu eksekusi memungkinkan kita untuk membandingkan kinerja algoritma yang berbeda dalam menyelesaikan masalah yang sama. Dengan mengukur waktu eksekusi, kita dapat mengidentifikasi algoritma yang lebih efisien dalam menyelesaikan tugas dengan waktu yang lebih singkat.

Namun, penting untuk diingat bahwa waktu eksekusi dapat dipengaruhi oleh berbagai faktor, seperti kecepatan komputer, lingkungan eksekusi, dan implementasi algoritma. Oleh karena itu, perlu hati-hati dalam menginterpretasikan hasil pengukuran waktu eksekusi dan mempertimbangkan faktor-faktor tersebut.

C. Order of Growth

Tingkat pertumbuhan adalah konsep yang digunakan untuk menganalisis seberapa cepat waktu eksekusi atau kebutuhan ruang algoritma berkembang seiring bertambahnya ukuran input. Hal ini berguna untuk memahami kompleksitas algoritma dan memprediksi bagaimana algoritma akan berkinerja saat dihadapkan pada input yang lebih besar.

Biasanya, tingkat pertumbuhan diukur dalam notasi Big O, yang memberikan perkiraan atas tingkat pertumbuhan algoritma dalam hal waktu eksekusi atau kebutuhan ruang. Notasi Big O mengidentifikasi fungsi terbesar yang mengikuti tingkat pertumbuhan algoritma saat ukuran input mendekati tak terbatas.

Contohnya, jika algoritma memiliki tingkat pertumbuhan O(n^2), itu berarti waktu eksekusinya berkembang secara kuadratik seiring dengan peningkatan ukuran input. Jika algoritma memiliki tingkat pertumbuhan O(n), maka waktu eksekusinya berkembang secara linier seiring dengan ukuran input.

Dengan memahami tingkat pertumbuhan, kita dapat memilih algoritma yang memiliki kompleksitas yang lebih rendah untuk mengoptimalkan kinerja dan efisiensi dalam menyelesaikan masalah.

D. Worst-Case, and Average-Case Efficiency

Efisiensi kasus terburuk dan kasus rata-rata adalah konsep yang digunakan untuk menganalisis kinerja algoritma dalam kondisi terburuk dan rata-rata.

Efisiensi kasus terburuk merujuk pada kinerja terburuk yang dapat terjadi dalam semua kemungkinan input. Ini memberikan batasan atas waktu eksekusi maksimum yang dapat terjadi pada algoritma dalam skenario terburuk. Analisis kasus terburuk membantu mengidentifikasi batasan kinerja algoritma yang paling buruk dan memastikan bahwa algoritma tetap efisien bahkan dalam kondisi yang paling tidak menguntungkan.

Efisiensi kasus rata-rata, di sisi lain, melibatkan analisis kinerja algoritma di atas sejumlah kasus input yang mungkin. Ini memperhitungkan distribusi kemungkinan input dan memberikan estimasi rata-rata waktu eksekusi algoritma dalam kondisi normal.

Analisis efisiensi kasus terburuk dan kasus rata-rata membantu memahami kinerja algoritma secara menyeluruh dan memberikan wawasan tentang bagaimana algoritma akan berperilaku dalam situasi yang berbeda. Dengan memperhitungkan kedua kasus tersebut, kita dapat membuat keputusan yang lebih baik dalam pemilihan algoritma yang paling efisien untuk menyelesaikan suatu masalah.

.

BAB III

BRUTE FORCE DAN EXHAUTIVE SEARCH

A. Selection Sort and Bubble Sort

algoritma pengurutan yang termasuk dalam kategori Brute Force. Keduanya merupakan metode pengurutan sederhana yang digunakan untuk mengurutkan elemen dalam sebuah array.

Selection Sort bekerja dengan cara memilih elemen terkecil dari array dan menukarnya dengan elemen pertama. Kemudian, elemen terkecil kedua dipilih dari sisa array dan ditukar dengan elemen kedua, dan seterusnya. Proses ini terus diulang hingga seluruh array terurut.

Bubble Sort, di sisi lain, bekerja dengan membandingkan pasangan elemen berturut-turut dalam array dan menukar posisinya jika diperlukan. Elemen yang lebih besar akan bergerak ke arah akhir array dalam setiap iterasi. Proses ini diulang secara berulang hingga seluruh array terurut.

Kedua algoritma ini menggunakan pendekatan Brute Force karena secara langsung memeriksa dan memanipulasi setiap elemen dalam array. Mereka tidak menggunakan pendekatan yang lebih efisien seperti membagi array menjadi subarray atau menggunakan teknik pemotongan untuk mengurangi jumlah perbandingan yang diperlukan.

Namun, kedua algoritma ini memiliki kompleksitas waktu yang sama, yaitu O(n^2), di mana n adalah jumlah elemen dalam array. Hal ini berarti waktu eksekusi algoritma meningkat secara kuadratik seiring dengan pertambahan ukuran input. Oleh karena itu, mereka cenderung tidak efisien untuk jumlah data yang besar.

B. Sequential Search and Brute-ForceString Maching

algoritma pencarian yang termasuk dalam kategori Brute Force. Keduanya digunakan untuk mencari pola atau elemen tertentu dalam sebuah array atau string.

Sequential Search adalah metode pencarian sederhana di mana setiap elemen dalam array atau daftar diperiksa satu per satu hingga elemen yang dicari ditemukan atau mencapai akhir array. Proses ini dilakukan secara berurutan dari awal hingga akhir elemen. Sequential Search cocok digunakan untuk array yang tidak terurut atau ketika kita tidak

memiliki informasi tambahan tentang lokasi elemen yang dicari. Namun, kompleksitas waktu Sequential Search adalah O(n), di mana n adalah ukuran array, sehingga bisa menjadi lambat untuk array dengan jumlah data yang besar.

Brute-Force String Matching adalah algoritma pencocokan pola yang sederhana untuk mencari kecocokan pola dalam sebuah teks. Algoritma ini bekerja dengan membandingkan karakter per karakter antara pola yang dicari dan teks yang sedang diuji. Algoritma ini mencoba memindahkan pola melalui teks, karakter demi karakter, dan memeriksa kecocokan pada setiap posisi. Jika ada ketidakcocokan, algoritma akan memindahkan pola ke posisi berikutnya dan melanjutkan pencocokan. Kompleksitas waktu Brute-Force String Matching adalah O(m*n), di mana m adalah panjang pola dan n adalah panjang teks. Algoritma ini efektif untuk pencocokan pola pada teks pendek atau saat kita tidak memiliki informasi tentang pola yang akan dicocokkan.

Meskipun kedua algoritma ini sederhana dan mudah diimplementasikan, mereka memiliki kompleksitas waktu yang tinggi dan tidak efisien untuk jumlah data yang besar. Oleh karena itu, dalam kasus-kasus di mana efisiensi waktu menjadi faktor penting, algoritma lain seperti Binary Search atau algoritma pencocokan pola yang lebih canggih seperti Knuth-Morris-Pratt atau Boyer-Moore dapat digunakan.

C. Closest-Pair and Convex-Hull Problems

masalah-masalah geometri yang melibatkan manipulasi dan analisis titik-titik dalam ruang dua dimensi. Kedua masalah ini seringkali membutuhkan pendekatan Brute Force untuk menemukan solusi optimal.

Closest-Pair Problem adalah masalah mencari dua titik terdekat dalam himpunan titik-titik yang diberikan. Tujuannya adalah untuk menemukan pasangan titik dengan jarak terkecil di antara semua pasangan yang ada. Pendekatan Brute Force untuk masalah ini adalah dengan memeriksa setiap pasangan titik dan menghitung jarak antara mereka. Selanjutnya, pasangan dengan jarak terkecil akan diidentifikasi sebagai solusi. Namun, metode ini memiliki kompleksitas waktu O(n^2), di mana n adalah jumlah titik dalam himpunan, karena harus memeriksa semua pasangan titik.

Convex-Hull Problem melibatkan mencari cangkang terluar (hull) yang melingkupi semua titik dalam himpunan titik-titik. Cangkang terluar ini harus berupa poligon convex. Algoritma Brute Force untuk Convex-Hull Problem akan memeriksa semua kombinasi tiga titik dan memeriksa apakah ketiga titik tersebut membentuk segitiga convex. Jika iya, titik-titik tersebut akan menjadi bagian dari cangkang terluar. Metode ini akan diulang untuk setiap kombinasi tiga titik. Algoritma Brute Force untuk Convex-Hull Problem juga memiliki kompleksitas waktu O(n^3), karena harus memeriksa setiap kombinasi tiga titik.

Meskipun pendekatan Brute Force ini sederhana, mereka dapat digunakan untuk menyelesaikan masalah-masalah tersebut. Namun, untuk jumlah titik yang besar, kompleksitas waktu yang tinggi membuat pendekatan ini kurang efisien. Oleh karena itu, terdapat algoritma-algoritma yang lebih efisien, seperti algoritma divide-and-conquer atau algoritma berdasarkan pemrograman dinamis, yang dapat digunakan untuk menyelesaikan masalah-masalah ini dengan kompleksitas waktu yang lebih rendah.

D. Exhautive Search

Exhaustive Search, juga dikenal sebagai Complete Search atau Brute Force Search, adalah metode pencarian yang mencoba semua kemungkinan solusi untuk menemukan solusi yang optimal. Pada dasarnya, algoritma Exhaustive Search secara sistematis memeriksa setiap kemungkinan yang ada untuk mencari solusi yang diinginkan.

Metode Exhaustive Search dapat digunakan dalam berbagai masalah optimisasi, kombinatorial, atau pemecahan masalah yang memerlukan pengecekan semua kemungkinan solusi. Algoritma ini tidak bergantung pada karakteristik masalah atau struktur data tertentu. Sebagai gantinya, itu memeriksa setiap kemungkinan secara berurutan untuk mencari solusi yang memenuhi kriteria yang ditentukan.

Salah satu kelemahan utama dari Exhaustive Search adalah kompleksitas waktu yang tinggi. Karena algoritma ini memeriksa semua kemungkinan, waktu eksekusi algoritma akan meningkat secara eksponensial seiring dengan bertambahnya ukuran input. Dalam

beberapa kasus, kompleksitas waktu algoritma Exhaustive Search dapat mencapai O(2^n), di mana n adalah jumlah elemen atau variabel yang harus dipertimbangkan.

Namun, meskipun memiliki kompleksitas waktu yang tinggi, Exhaustive Search memiliki keunggulan dalam memberikan solusi yang akurat dan optimal. Karena metode ini mencoba semua kemungkinan, ia menjamin bahwa tidak ada solusi yang terlewatkan. Oleh karena itu, algoritma Exhaustive Search sering digunakan dalam kasus-kasus di mana pencarian solusi optimal penting dan jumlah kemungkinan solusi terbatas.

E. Depth-First Search and Breath-First Search

Depth-First Search (DFS) dan Breadth-First Search (BFS) adalah dua algoritma pencarian yang sering digunakan dalam pemrosesan graf. Keduanya beroperasi pada graf yang terdiri dari simpul-simpul yang saling terhubung melalui tepi atau busur.

DFS adalah algoritma yang melakukan pencarian secara vertikal ke kedalaman terdalam sebelum melanjutkan ke simpul-simpul lain. Algoritma ini menjelajahi graf dengan mengunjungi simpul awal, kemudian beralih ke simpul tetangga yang belum dikunjungi, dan melanjutkan proses ini hingga tidak ada simpul tetangga yang tersisa. Jika ada simpul yang belum dikunjungi, DFS akan mundur ke simpul sebelumnya dan mencari simpul lain yang belum dikunjungi. DFS menggunakan pendekatan tumpukan (stack) untuk melacak simpul-simpul yang akan dikunjungi selanjutnya. DFS biasanya digunakan untuk mencari jalur dari simpul awal ke simpul tujuan, penelusuran graf, atau analisis struktur data seperti pohon.

BFS, di sisi lain, melakukan pencarian secara horizontal dari simpul awal ke semua simpul tetangganya sebelum melanjutkan ke tingkat berikutnya. Algoritma ini menjelajahi graf dengan mengunjungi simpul awal, kemudian mengunjungi semua simpul tetangganya, dan melanjutkan ke simpul-simpul tetangga yang belum dikunjungi pada tingkat berikutnya. BFS menggunakan pendekatan antrian (queue) untuk melacak simpul-simpul yang akan dikunjungi selanjutnya. BFS biasanya digunakan untuk mencari jalur terpendek antara dua simpul, penemuan komponen terhubung dalam graf, atau permasalahan yang membutuhkan pemrosesan berdasarkan tingkat.

Kedua algoritma ini memiliki kelebihan dan kekurangan masing-masing tergantung pada kebutuhan aplikasi dan struktur graf yang ada. Pemilihan antara DFS dan BFS tergantung pada tujuan pencarian, efisiensi waktu, atau penggunaan sumber daya yang tersedia.

BAB IV
DECREASE AND CONQUER

A. Three Major Varian of Decrease-and-Conquer

Three Major Variants of Decrease-and-Conquer (Tiga Varian Utama Decrease-and-Conquer) adalah sebuah pendekatan dalam algoritma yang melibatkan pengurangan atau penyederhanaan masalah secara bertahap untuk memecahkan masalah yang lebih kecil atau lebih sederhana. Terdapat tiga varian utama dalam Decrease-and-Conquer, yaitu:

1. Decrease by a Constant: Dalam varian ini, masalah awal dikurangi secara konstan pada setiap langkah. Misalnya, jika masalah awal memiliki ukuran n, maka pada setiap langkah ukuran masalah dikurangi sebanyak k (misalnya, dikurangi sebanyak 1). Teknik ini biasanya digunakan untuk masalah yang memiliki struktur teratur atau pola yang dapat diatur dalam ukuran konstan.
2. Decrease by a Fraction: Dalam varian ini, masalah awal dikurangi sebanyak sebagian tertentu pada setiap langkah. Misalnya, jika masalah awal memiliki ukuran n, maka pada setiap langkah ukuran masalah dikurangi sebanyak n/k (misalnya, dikurangi sebanyak setengah dari ukuran awal). Teknik ini biasanya digunakan untuk masalah yang dapat dibagi secara merata menjadi submasalah yang lebih kecil.
3. Decrease by a Variable Amount: Dalam varian ini, masalah awal dikurangi sebanyak jumlah yang bervariasi pada setiap langkah. Jumlah pengurangan dapat bergantung pada karakteristik masalah atau pada analisis masalah yang sedang diselesaikan. Teknik ini memungkinkan penyesuaian yang lebih fleksibel dalam proses pengurangan masalah.

B. Sort

Sort (Mengurutkan) adalah sebuah tugas umum dalam algoritma yang melibatkan mengatur kembali elemen-elemen dalam himpunan data menjadi urutan yang teratur atau terurut. Mengurutkan sering digunakan dalam berbagai aplikasi, seperti pemrosesan data, pencarian, analisis, dan lain-lain.

Ada banyak algoritma pengurutan yang berbeda, masing-masing dengan kelebihan dan kelemahan mereka sendiri. Beberapa algoritma pengurutan yang umum digunakan termasuk Mergesort, Quicksort, Insertion Sort, Selection Sort, dan Bubble Sort. Setiap algoritma pengurutan memiliki kompleksitas waktu dan ruang yang berbeda, sehingga pemilihan algoritma pengurutan yang tepat sangat penting tergantung pada sifat masalah yang dihadapi.

Pengurutan dilakukan dengan membandingkan elemen-elemen dalam himpunan data dan mempertukarkan posisi mereka sesuai dengan aturan urutan yang ditentukan. Tujuannya adalah menghasilkan himpunan data yang terurut sesuai dengan kriteria yang ditetapkan, seperti urutan numerik (mengurutkan angka dari yang terkecil ke terbesar) atau urutan alfabetik (mengurutkan kata-kata secara alfabetic).

C. Topological Sorting

Topological Sorting (Pengurutan Topologis) adalah sebuah algoritma yang digunakan untuk mengurutkan simpul-simpul dalam sebuah graf berarah sehingga tidak ada arah dari simpul A ke simpul B jika simpul A berada sebelum simpul B dalam urutan hasil pengurutan. Pengurutan ini digunakan terutama pada graf yang mewakili ketergantungan antara tugas atau kejadian.

Proses Topological Sorting dimulai dengan memilih simpul-simpul yang tidak memiliki tautan masuk (indegree) atau simpul yang tidak memiliki ketergantungan pada simpul lain. Selanjutnya, simpul-simpul ini ditempatkan dalam urutan pengurutan yang benar. Setelah itu, simpul-simpul yang terhubung dengan simpul-simpul tersebut akan kehilangan satu tautan masuk, dan proses ini diulangi hingga semua simpul terurut.

Topological Sorting sering digunakan dalam perencanaan proyek, penjadwalan tugas, analisis dependensi, dan pemrosesan bahasa alami. Algoritma yang umum digunakan untuk Topological Sorting adalah algoritma DFS (Depth-First Search) dan algoritma BFS (Breadth-First Search).

.

BAB V

DEVIDE AND CONQUER

A. Mergesort

Mergesort adalah algoritma pengurutan yang menggunakan pendekatan Devide and Conquer. Algoritma ini bekerja dengan membagi himpunan data menjadi dua bagian yang lebih kecil, mengurutkan masing-masing bagian secara terpisah, dan kemudian menggabungkan kembali bagian-bagian yang telah diurutkan.

Proses Mergesort dimulai dengan membagi himpunan data menjadi dua bagian yang seimbang. Setiap bagian tersebut kemudian diurutkan secara rekursif menggunakan algoritma Mergesort. Selanjutnya, hasil pengurutan kedua bagian tersebut digabungkan dengan menggabungkan elemen-elemen secara terurut dalam urutan yang benar.

Keuntungan utama dari Mergesort adalah memiliki kompleksitas waktu yang stabil dan tidak bergantung pada data input. Algoritma ini memiliki kompleksitas waktu O(n log n), di mana n adalah jumlah elemen dalam himpunan data.

B. Quicksort

Quicksort adalah algoritma pengurutan yang menggunakan pendekatan Devide and Conquer. Algoritma ini bekerja dengan memilih elemen yang disebut pivot dari himpunan data, mempartisi himpunan data menjadi dua subhimpunan berdasarkan pivot, dan kemudian mengurutkan kedua subhimpunan secara terpisah.

Proses Quicksort dimulai dengan memilih pivot dari himpunan data. Setelah itu, himpunan data dipartisi menjadi dua subhimpunan: satu subhimpunan dengan elemen yang lebih kecil dari pivot dan satu subhimpunan dengan elemen yang lebih besar dari pivot. Setiap subhimpunan tersebut kemudian diurutkan secara rekursif menggunakan algoritma Quicksort. Akhirnya, hasil pengurutan kedua subhimpunan digabungkan menjadi himpunan data yang terurut.

Keuntungan utama dari Quicksort adalah memiliki kompleksitas waktu yang cepat dan efisien dalam kasus rata-rata. Namun, dalam kasus terburuk, Quicksort dapat memiliki kompleksitas waktu O(n^2), di mana n adalah jumlah elemen dalam himpunan data. Untuk menghindari kasus terburuk, strategi pemilihan pivot yang cerdas seperti pivot acak atau pivot median dapat digunakan.

C. Binary Tree Traversals and Related Properties

Binary Tree Traversals and Related Properties (Traversal Pohon Biner dan Properti Terkait) adalah topik yang berkaitan dengan pohon biner dan cara mengunjungi atau melintasi setiap simpul dalam pohon. Pada dasarnya, ini adalah teknik yang digunakan untuk memeriksa atau mengakses elemen-elemen dalam pohon biner.

Terdapat beberapa jenis traversal yang umum digunakan dalam pohon biner. Preorder traversal melintasi simpul-simpul dalam urutan root-kiri-kanan. Inorder traversal melintasi simpul-simpul dalam urutan kiri-root-kanan. Postorder traversal melintasi simpul-simpul dalam urutan kiri-kanan-root. Level-order traversal melintasi simpul-simpul dalam urutan tingkat per tingkat, mulai dari root dan berlanjut ke simpul-simpul di level berikutnya.

Traversing pohon biner dapat digunakan untuk berbagai tujuan, seperti mencetak elemen-elemen pohon dalam urutan yang diinginkan, mencari elemen tertentu, menghitung tinggi atau kedalaman pohon, atau melakukan operasi lainnya pada elemen-elemen pohon.

BAB VI
TRANSFORM AND CONQUER

A. Instance Simplification

Instance Simplification (Sederhanaan Instance) adalah sebuah teknik dalam pendekatan Transform and Conquer yang digunakan untuk menyederhanakan sebuah instansi masalah menjadi bentuk yang lebih mudah atau lebih efisien untuk dipecahkan. Teknik ini melibatkan mengubah atau memodifikasi instansi masalah yang kompleks menjadi bentuk yang lebih sederhana tanpa mengubah esensi permasalahan yang ingin diselesaikan.

Dalam instance simplification, beberapa pendekatan yang umum digunakan adalah menghilangkan informasi yang tidak relevan atau redundan dari instansi masalah, membatasi ukuran instansi masalah dengan mengurangi jumlah elemen atau aspek yang harus diperhatikan, atau mempertimbangkan kasus khusus dengan batasan tertentu yang lebih mudah untuk dipecahkan.

Teknik instance simplification dapat meningkatkan efisiensi dan kepraktisan dalam menyelesaikan masalah yang kompleks dengan mengurangi kompleksitas masalah menjadi bentuk yang lebih sederhana atau lebih terkelola.

B. Representation Change

Representation Change (Perubahan Representasi) adalah sebuah teknik dalam pendekatan Transform and Conquer yang melibatkan perubahan representasi atau representasi ulang dari instansi masalah menjadi bentuk yang lebih mudah atau lebih efisien untuk dipecahkan. Teknik ini mencakup konversi dari satu representasi data ke representasi data yang lain, yang dapat memberikan keuntungan dalam efisiensi atau keterampilan algoritma yang digunakan untuk menyelesaikan masalah.

Dalam representation change, perubahan representasi dapat dilakukan dengan mengubah struktur data yang digunakan untuk menyimpan dan mengakses informasi, mengubah representasi visual atau grafis dari masalah, atau mengubah representasi matematis atau logika yang digunakan untuk merumuskan masalah.

Perubahan representasi dapat membantu dalam mengatasi keterbatasan atau kelemahan dari representasi awal yang digunakan dalam masalah yang diberikan. Hal ini dapat

mempercepat pemrosesan atau analisis masalah, mengurangi kompleksitas, atau memungkinkan penerapan algoritma yang lebih efisien.

C. Problem Reduction

Problem Reduction (Reduksi Masalah) adalah sebuah teknik dalam pendekatan Transform and Conquer yang melibatkan pengurangan atau transformasi masalah yang kompleks menjadi masalah yang lebih sederhana atau yang telah diketahui solusinya. Teknik ini memungkinkan penyelesaian masalah kompleks dengan memanfaatkan solusi yang telah ada atau yang lebih mudah dicapai.

Dalam problem reduction, masalah awal diubah atau dikurangi menjadi masalah yang setara atau lebih mudah untuk dipecahkan. Hal ini dapat dilakukan dengan mengidentifikasi submasalah yang dapat dipecahkan secara terpisah, memanfaatkan struktur atau properti tertentu dalam masalah untuk menyederhanakan solusi, atau mengurangi kompleksitas dengan membatasi variabel atau parameter tertentu.

Teknik problem reduction membantu dalam meningkatkan efisiensi dan kemampuan penyelesaian masalah dengan memanfaatkan solusi yang telah ada atau dengan memecah masalah kompleks menjadi submasalah yang lebih terkelola.

BAB VII
SPACE AND 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 suatu himpunan data dengan menggunakan pendekatan trade-off antara waktu dan ruang. Metode ini efektif untuk mengurutkan himpunan data dengan rentang nilai terbatas.

Pada metode Sorting by Counting, pertama-tama dilakukan penghitungan frekuensi kemunculan setiap nilai dalam himpunan data. Kemudian, dilakukan penjumlahan kumulatif dari frekuensi tersebut untuk mendapatkan posisi awal setiap nilai dalam himpunan data yang terurut. Akhirnya, elemen-elemen himpunan data dipindahkan ke posisi yang sesuai berdasarkan posisi awal yang telah dihitung sebelumnya.

Metode Sorting by Counting memiliki kompleksitas waktu yang lebih rendah daripada beberapa metode pengurutan lainnya seperti Quicksort atau Mergesort. Namun, metode ini memiliki keterbatasan dalam hal rentang nilai yang dapat diurutkan.

B. Input Enchantment in String Maching

Input Enhancement in String Matching (Peningkatan Masukan dalam Pencocokan String) adalah teknik yang digunakan untuk meningkatkan efisiensi pencocokan string dalam algoritma pencocokan string brute force. Teknik ini memanfaatkan informasi tambahan tentang pola yang dicari atau teks yang diuji untuk mempercepat proses pencocokan.

Dalam algoritma pencocokan string brute force, pencocokan dilakukan dengan membandingkan setiap karakter dalam pola dengan karakter-karakter yang sesuai dalam teks. Namun, dengan teknik Input Enhancement, informasi tambahan seperti tabel preskips atau heuristik lainnya digunakan untuk menghindari perbandingan yang tidak perlu antara pola dan teks.

Teknik Input Enhancement dapat meningkatkan efisiensi pencocokan string brute force dengan mengurangi jumlah perbandingan yang harus dilakukan, terutama dalam kasus-kasus di mana ada karakter-karakter yang cocok sebelumnya yang telah ditemukan.

C. Hasing

Hashing (Pemetaan) adalah teknik yang digunakan untuk mengonversi data menjadi nilai indeks yang unik dan efisien dalam struktur data yang disebut tabel hash. Teknik ini digunakan untuk mempercepat operasi pencarian, penyisipan, dan penghapusan data dalam kumpulan data yang besar.

Dalam proses hashing, setiap elemen data diberikan nilai hash yang kemudian digunakan sebagai indeks untuk memetakan elemen tersebut ke posisi yang sesuai dalam tabel hash. Nilai hash dihasilkan oleh fungsi hash yang mengubah data menjadi indeks yang unik dan terdistribusi secara merata.

Keuntungan utama dari hashing adalah kemampuannya untuk memberikan akses yang cepat ke elemen-elemen data dengan kompleksitas waktu yang konstan, asalkan tidak ada bentrokan (collision) yang terjadi. Namun, jika terjadi bentrokan, teknik-teknik seperti penyelesaian bentrokan dan penggunaan tabel hash yang lebih besar dapat diterapkan.

BAB VIII
DYNAMIC
PROGRAMMING

A. Three Basic

Three Basic (Tiga Dasar) adalah konsep dasar dalam Dynamic Programming yang membentuk landasan untuk memahami dan menerapkan teknik ini. Tiga konsep dasar tersebut meliputi:

1. Overlapping Subproblems (Submasalah yang Tumpang Tindih): Dalam Dynamic Programming, masalah besar dipecah menjadi submasalah yang lebih kecil. Salah satu sifat penting dari submasalah tersebut adalah adanya tumpang tindih di antara mereka, artinya beberapa submasalah memiliki kesamaan dalam solusi mereka. Dengan mengidentifikasi dan memecahkan submasalah yang tumpang tindih hanya sekali, kita dapat menghindari perhitungan berulang dan meningkatkan efisiensi algoritma.
2. Optimal Substructure (Substruktur Optimal): Jika sebuah masalah dapat dipecahkan menjadi submasalah yang lebih kecil, dan solusi optimal dari masalah tersebut dapat ditemukan dengan menggabungkan solusi optimal dari submasalah yang lebih kecil, maka masalah tersebut memiliki sifat optimal substructure. Ini memungkinkan kita untuk membangun solusi masalah yang lebih besar dari solusi masalah yang lebih kecil secara bertahap.
3. Memoization (Pengingatan): Memoization adalah teknik dalam Dynamic Programming yang digunakan untuk menyimpan hasil perhitungan submasalah yang sudah dipecahkan. Dengan menyimpan hasil ini dalam struktur data, kita dapat menghindari perhitungan berulang saat submasalah yang sama muncul lagi. Memoization memungkinkan kita untuk mengurangi kompleksitas waktu algoritma dengan mengurangi jumlah perhitungan yang diperlukan.

B. The Knapsack Problem and Memory Functions

The Knapsack Problem and Memory Functions (Masalah Knapsack dan Fungsi Memori) adalah salah satu contoh aplikasi dari Dynamic Programming. Masalah Knapsack adalah masalah optimisasi yang melibatkan pemilihan objek dengan bobot dan nilai tertentu untuk dimasukkan ke dalam knapsack dengan kapasitas terbatas, sedemikian rupa

sehingga nilai total yang dimasukkan ke dalam knapsack adalah maksimum. Dalam memecahkan masalah Knapsack, fungsi memori digunakan untuk menyimpan nilai optimal dari submasalah yang sudah dipecahkan sehingga dapat digunakan kembali saat memecahkan submasalah yang lebih besar.

C. Warshall’sand Floyd’s Algorithms

Warshall's dan Floyd's Algorithms (Algoritma Warshall dan Floyd) adalah dua algoritma yang digunakan dalam pemecahan masalah yang melibatkan graf berbobot. Algoritma Warshall digunakan untuk mencari jalur terpendek antara semua pasangan simpul dalam graf berbobot. Algoritma ini berbasis pada prinsip pemrograman dinamis dan menggunakan tabel adjacency matrix untuk menyimpan informasi jarak antara simpul-simpul. Algoritma Floyd, juga dikenal sebagai algoritma Floyd-Warshall, adalah pengembangan dari algoritma Warshall yang dapat menangani graf dengan bobot negatif. Algoritma ini juga menggunakan tabel adjacency matrix untuk menyimpan jarak antara simpul-simpul, namun dengan pendekatan yang sedikit berbeda dalam perhitungan jarak terpendek.

BAB IX

GREEDY TECHNIQUE

A. Prim’s Algorithm

Prim's Algorithm adalah salah satu algoritma Greedy yang digunakan untuk mencari Minimum Spanning Tree (MST) dalam sebuah graf berbobot. MST adalah sebuah subgraf dari graf asli yang terhubung secara lengkap dan memiliki bobot minimum di antara semua subgraf terhubung yang mungkin. Prim's Algorithm dimulai dengan memilih simpul awal dan secara berulang memilih tepi dengan bobot minimum yang terhubung ke simpul yang sudah dipilih sebelumnya, hingga semua simpul terhubung. Algoritma ini memastikan bahwa MST yang dihasilkan akan memiliki bobot minimum.

B. Kruskal’s Algorithm

Kruskal's Algorithm juga digunakan untuk mencari Minimum Spanning Tree (MST). Algoritma ini bekerja dengan mempertimbangkan setiap tepi secara terpisah dan menggabungkannya dengan MST jika tidak membentuk siklus. Algoritma ini dimulai dengan mengurutkan semua tepi berdasarkan bobotnya secara ascending. Kemudian, secara berulang, tepi dengan bobot terkecil ditambahkan ke MST jika tidak membentuk siklus. Algoritma ini berhenti saat MST terbentuk dengan menghubungkan semua simpul.

C. Djikstra’s Algorithm

Dijkstra's Algorithm digunakan untuk mencari jalur terpendek dari satu simpul asal ke semua simpul lain dalam graf berbobot positif. Algoritma ini beroperasi dengan mempertimbangkan simpul yang memiliki jarak terpendek dari simpul asal, kemudian secara bertahap memperluas jarak terpendek ke simpul-simpul lain. Dijkstra's Algorithm menggunakan pendekatan Greedy dengan memilih simpul berikutnya yang memiliki jarak terpendek yang diketahui. Algoritma ini berhenti saat semua simpul telah dijangkau atau saat jarak terpendek ke simpul yang belum dijangkau tidak dapat diperbarui lagi.

D. Huffman Trees and Codes

Huffman Trees and Codes adalah teknik yang digunakan dalam kompresi data. Algoritma Huffman digunakan untuk menghasilkan kode biner dengan panjang variabel untuk

merepresentasikan data dengan frekuensi yang berbeda. Algoritma ini menggunakan pendekatan Greedy dengan membangun pohon Huffman berdasarkan frekuensi kemunculan data. Pada pohon Huffman, data dengan frekuensi tinggi diberikan kode yang lebih pendek, sedangkan data dengan frekuensi rendah diberikan kode yang lebih panjang. Hal ini memungkinkan representasi data yang sering muncul dengan lebih sedikit bit, menghasilkan kompresi data yang efisien.

BAB X

ITERATIVE IMPROVEMENT

A. The Simplex Method

Metode Simpleks adalah algoritma optimisasi yang digunakan untuk menyelesaikan masalah pemrograman linier. Masalah pemrograman linier melibatkan maksimisasi atau minimisasi fungsi linier subjek terhadap sejumlah kendala linier. Metode Simpleks berfungsi dengan menjelajahi sudut-sudut dari himpunan solusi yang mungkin dan secara iteratif memperbaiki solusi saat bergerak ke sudut-sudut yang menghasilkan nilai fungsi yang lebih baik.

Metode Simpleks bekerja dengan memulai dari solusi dasar yang layak dan kemudian melakukan iterasi melalui serangkaian langkah perubahan variabel untuk memperbaiki solusi secara bertahap. Pada setiap iterasi, algoritma memilih variabel yang akan dimasukkan atau dikeluarkan dari solusi dasar untuk mencari perubahan yang meningkatkan nilai fungsi. Proses ini terus berlanjut hingga ditemukan solusi yang optimal atau tidak ada perubahan yang membaik lagi.

Metode Simpleks telah terbukti efektif dalam menyelesaikan berbagai masalah pemrograman linier dalam berbagai bidang, termasuk manajemen rantai pasokan, perencanaan produksi, dan optimisasi keuangan.

B. The Maximum-Flow Problem

Masalah Aliran Maksimum (Maximum-Flow Problem) adalah masalah dalam teori graf yang melibatkan penentuan aliran maksimum yang dapat mengalir melalui jaringan dari satu titik ke titik lain. Jaringan terdiri dari simpul-simpul yang mewakili lokasi dan jalur yang menghubungkan simpul-simpul tersebut.

Tujuan dari masalah aliran maksimum adalah menentukan sejauh mana aliran dapat melewati jaringan dari satu simpul sumber (source) ke simpul tujuan (sink) dengan mematuhi batasan kapasitas setiap jalur. Algoritma yang umum digunakan untuk menyelesaikan masalah ini adalah Algoritma Edmonds-Karp dan Algoritma Dinic, yang berbasis pada algoritma aliran augmentasi.

Masalah aliran maksimum memiliki banyak aplikasi dalam dunia nyata, seperti optimisasi jaringan transportasi, pengaturan jaringan komunikasi, dan perencanaan aliran air dalam sistem irigasi.

C. Maximum Matching in Bipartite Graphs

Masalah Pencocokan Maksimum dalam Graf Bipartit (Maximum Matching in Bipartite Graphs) adalah masalah dalam teori graf yang melibatkan pencarian himpunan pencocokan (matching) terbesar di antara simpul-simpul dua himpunan yang saling berhubungan dalam graf bipartit. Graf bipartit terdiri dari dua himpunan simpul yang tidak memiliki sisi dalam himpunan yang sama.

Tujuan dari masalah ini adalah untuk menemukan himpunan pencocokan terbesar yang tidak saling tumpang tindih, di mana setiap simpul hanya terhubung dengan satu simpul lainnya dalam himpunan lain. Algoritma yang umum digunakan untuk menyelesaikan masalah ini adalah Algoritma Hopcroft-Karp dan Algoritma Berge.

Masalah pencocokan maksimum dalam graf bipartit memiliki aplikasi dalam berbagai bidang, seperti jaringan sosial, pengaturan pernikahan, dan pencocokan pemrograman.

D. The Stable Marriage Problem

Masalah Pernikahan Stabil (Stable Marriage Problem) adalah masalah dalam teori permainan yang melibatkan pencocokan pasangan stabil antara dua himpunan orang dengan preferensi mereka terhadap anggota himpunan lain. Misalnya, terdapat dua himpunan, pria dan wanita, yang masing-masing memiliki preferensi atas anggota himpunan lain. Tujuan dari masalah ini adalah mencari pencocokan yang stabil, di mana tidak ada pasangan yang ingin meninggalkan pasangan saat ini untuk membentuk pasangan baru yang lebih disukai.

Algoritma Gale-Shapley adalah algoritma yang umum digunakan untuk menyelesaikan masalah pernikahan stabil. Algoritma ini beroperasi dengan menganalisis preferensi dan proposal dari setiap anggota himpunan untuk mencapai pencocokan yang stabil dan saling menguntungkan.

Masalah pernikahan stabil memiliki aplikasi dalam pengaturan pernikahan, pengaturan pasangan dalam program pertukaran siswa, dan pengaturan pasangan dalam jaringan sosial.

BAB XI
LIMITATIONS OF
ALGORITHM POWER

A. Lower-Bounf Arguments

Lower-Bound Arguments (Argumen Batas Bawah) adalah konsep dalam analisis algoritma yang digunakan untuk menentukan batas bawah kompleksitas waktu atau ruang yang diperlukan oleh suatu algoritma dalam menyelesaikan suatu masalah. Argumen batas bawah berfokus pada identifikasi batas bawah yang tak terelakkan, yang menunjukkan bahwa tidak mungkin ada algoritma yang lebih baik dari batas yang telah ditetapkan.

Argumen batas bawah digunakan untuk membuktikan bahwa suatu masalah adalah sulit atau tidak dapat diselesaikan secara efisien oleh algoritma tertentu. Melalui analisis matematis, argumen batas bawah dapat memberikan keyakinan bahwa algoritma yang ada sudah mendekati batas kinerja optimal. Misalnya, dalam Teori Komputasi, argumen batas bawah telah digunakan untuk membuktikan bahwa beberapa masalah komputasi, seperti Sorting, memiliki kompleksitas waktu yang terbatas dari bawah.

Dalam analisis argumen batas bawah, terdapat berbagai teknik yang digunakan, seperti reduksi masalah, pemisahan informasi, atau penggunaan konsep seperti informasi bit. Tujuannya adalah untuk mengidentifikasi batasan intrinsik dari suatu masalah yang tidak dapat diatasi oleh algoritma apa pun.

B. Decision Tree

Decision Tree (Pohon Keputusan) adalah struktur representasi grafis yang digunakan untuk memodelkan pengambilan keputusan berdasarkan serangkaian pertanyaan atau kondisi. Pohon keputusan terdiri dari simpul-simpul yang mewakili pertanyaan atau kondisi, serta cabang-cabang yang mewakili kemungkinan jawaban atau hasil dari pertanyaan tersebut. Pada setiap simpul, keputusan diambil berdasarkan jawaban yang diberikan, dan pohon dilanjutkan hingga mencapai simpul daun yang mewakili solusi atau tindakan yang diambil.

Pohon keputusan digunakan dalam berbagai bidang, seperti ilmu data, kecerdasan buatan, dan sistem ekspert. Mereka dapat digunakan untuk mengambil keputusan, melakukan klasifikasi data, atau mengidentifikasi pola dalam data. Keuntungan utama dari pohon

keputusan adalah kemampuan mereka untuk memvisualisasikan alur keputusan dan memahami relasi antara variabel dan keputusan yang diambil.

Proses pembentukan pohon keputusan melibatkan pemilihan fitur atau variabel yang paling informatif untuk memisahkan data dengan cara yang paling efisien. Beberapa metode yang umum digunakan untuk membangun pohon keputusan termasuk ID3, C4.5, dan CART. Metode ini menggunakan algoritma untuk membagi data berdasarkan keuntungan informasi atau kriteria yang sesuai.

C. P, Np and-Complete Problems

P, NP, dan Masalah-NP Selesai adalah konsep yang berkaitan dengan kompleksitas algoritma dalam teori kompleksitas komputasi. P adalah kelas kompleksitas yang terdiri dari masalah yang dapat diselesaikan dalam waktu polinomial oleh sebuah algoritma deterministik. Masalah dalam kelas P adalah yang memiliki solusi yang efisien dan dapat dihitung dalam waktu yang wajar.

Sebaliknya, NP (Non-deterministic Polynomial) adalah kelas kompleksitas yang terdiri dari masalah yang dapat diverifikasi dalam waktu polinomial oleh sebuah algoritma, tetapi solusi yang tepat belum ditemukan secara efisien. Dalam kelas NP, solusi yang diusulkan dapat diverifikasi dengan cepat, tetapi tidak ada algoritma yang diketahui dapat menyelesaikan semua masalah NP dalam waktu polinomial. Masalah NP termasuk Traveling Salesman Problem, Boolean Satisfiability Problem, dan Knapsack Problem.

Masalah-NP Selesai (NP-Complete) adalah kelas masalah dalam NP yang memiliki sifat bahwa jika salah satu masalah NP-Complete dapat diselesaikan dalam waktu polinomial, maka semua masalah dalam kelas NP-Complete juga dapat diselesaikan dalam waktu polinomial. NP-Complete menggambarkan masalah yang sangat sulit dan tidak diketahui solusi efisiennya. Pada saat ini, tidak ada algoritma yang diketahui yang dapat menyelesaikan masalah NP-Complete secara efisien.

D. Challenge of Numerical Algorithms

Tantangan Algoritma Numerik adalah permasalahan yang muncul saat merancang dan mengimplementasikan algoritma untuk menyelesaikan permasalahan matematika yang melibatkan operasi numerik atau komputasi. Algoritma numerik sering digunakan dalam analisis numerik, simulasi komputer, optimisasi, atau pemodelan matematis.

Salah satu tantangan utama dalam algoritma numerik adalah akurasi dan kestabilan perhitungan. Karena operasi numerik melibatkan pembulatan dan pemotongan bilangan, kesalahan numerik dapat terakumulasi selama proses perhitungan. Oleh karena itu, algoritma numerik harus dirancang sedemikian rupa sehingga dapat mengatasi masalah akurasi dan kestabilan dengan menggunakan metode yang tepat, seperti penggunaan bilangan titik mengambang yang tepat atau pemilihan metode iterasi yang stabil.

Selain itu, efisiensi komputasi juga menjadi tantangan dalam algoritma numerik. Algoritma numerik harus dirancang agar dapat menyelesaikan permasalahan dengan cepat dan efisien dalam batasan sumber daya yang ada, seperti waktu dan memori. Optimisasi dan pemilihan metode yang tepat menjadi penting untuk mencapai kinerja yang baik dalam algoritma numerik.

BAB XII
COPYING WITH THE LIMITATION OF ALGORITM POWER

A. Backtracking

Backtracking (Pemunduran Langkah) adalah metode yang digunakan untuk mencari solusi dalam ruang pencarian yang besar dengan menguji setiap kemungkinan secara berurutan dan membatalkan langkah-langkah yang tidak mengarah ke solusi. Algoritma Backtracking memulai dengan langkah awal, kemudian secara rekursif mencoba setiap kemungkinan langkah berikutnya. Jika langkah tersebut tidak menghasilkan solusi yang valid, algoritma akan melakukan pemunduran langkah ke langkah sebelumnya dan mencoba kemungkinan lainnya. Proses ini berlanjut hingga solusi ditemukan atau semua kemungkinan telah dijelajahi.

Metode Backtracking sering digunakan dalam permasalahan yang melibatkan kombinatorial atau optimisasi, seperti penyelesaian Sudoku, pengaturan ratu pada papan catur, atau mencari jalur terpendek dalam graf. Algoritma ini memungkinkan eksplorasi yang efisien pada ruang pencarian yang besar dengan mengurangi jumlah kemungkinan yang harus diuji. Kompleksitas waktu dari algoritma Backtracking tergantung pada banyak faktor, seperti kompleksitas langkah-langkah yang diuji dan jumlah solusi yang mungkin ada.

B. Branch and Bound

Branch and Bound (Cabang dan Batas) adalah metode yang digunakan untuk mencari solusi optimal dalam ruang pencarian dengan mengurangi jumlah kemungkinan yang perlu dieksplorasi. Algoritma Branch and Bound memulai dengan langkah awal dan secara bertahap membentuk cabang-cabang atau subset dari ruang pencarian berdasarkan aturan atau batasan tertentu. Kemudian, algoritma menghitung batas atau estimasi terbaik untuk setiap cabang yang membantu menghilangkan cabang yang tidak akan menghasilkan solusi optimal. Jika batas dari suatu cabang lebih buruk daripada batas solusi terbaik yang ada, cabang tersebut dapat diabaikan.

Metode Branch and Bound sering digunakan dalam permasalahan optimisasi atau pencarian solusi terbaik, seperti TSP (Traveling Salesman Problem) atau Knapsack Problem. Algoritma ini memungkinkan penjelajahan yang efisien pada ruang pencarian dengan membatasi jumlah cabang yang harus dijelajahi. Kompleksitas waktu dari algoritma Branch and Bound tergantung pada kompleksitas perhitungan batas dan jumlah cabang yang dihasilkan.

C. Algorithms for Solving Nonlinear Problems

Algoritma untuk Menyelesaikan Masalah Nonlinear adalah metode yang digunakan untuk menemukan solusi numerik untuk masalah matematika nonlinier. Masalah nonlinier melibatkan fungsi-fungsi yang tidak mematuhi sifat linier, seperti persamaan nonlinear atau optimisasi nonlinear. Algoritma yang digunakan dalam masalah ini sering melibatkan pengulangan iteratif untuk mendekati solusi yang akurat.

Beberapa algoritma yang umum digunakan dalam penyelesaian masalah nonlinier termasuk Metode Newton-Raphson, Metode Levenberg-Marquardt, dan Metode Quasi-Newton. Metode ini berdasarkan pendekatan iteratif untuk memperbaiki perkiraan solusi pada setiap langkah iterasi. Algoritma tersebut menggabungkan konsep dari analisis numerik dan kalkulus untuk memperkirakan solusi dengan tingkat ketepatan tertentu.

Pemilihan algoritma yang tepat tergantung pada sifat dan kompleksitas masalah nonlinier yang dihadapi. Beberapa algoritma lebih efisien untuk masalah yang lebih sederhana, sementara yang lain lebih cocok untuk masalah yang lebih kompleks. Penting untuk memahami karakteristik masalah dan menerapkan algoritma yang sesuai untuk mencapai solusi yang diinginkan.