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 :

IFDAL LISYUKRI
21346012

PROGRAM STUDI S1 TEKNIK INFORMATIKA

JURUSAN TEKNIK ELEKTRONIKA

FAKULTAS TEKNIK

UNIVERSITAS NEGERI PADANG

TAHUN 2023

KATA PENGANTAR

Puji syukur kita hanturkan kehadiran Tuhan Yang Maha Esa, yang telah melimpahkan rahmat dan karunianya kepada kita, sehingga kita dapat menyusun dan menyajikan makalah Perancangan dan Analisis Algoritma ini dengan tepat waktu. Tak lupa pula kita mengucapkan terima kasih kepada berbagai pihak yangtelah memberikan dorongan dan motivasi. Sehingga makalah ini dapat tersusun dengan baik.

Makalah ini dibuat sebagai salah satu syarat untuk memenuhi tugas mata kuliah Perancangan dan Analisis Algoritma. Kami menyadari bahwa dalam penyusunan makalah ini masih terdapat banyak kekurangan dan jauh dari kata sempurna. Oleh karena itu, Kami mengharapkan kritik dan saran untuk menyempurnakan makalah ini dan dapat menjadi acuan dalam menyusun makalah-makalah selanjutnya. Kami muga memohon maaf apabila dalam penulisan makalah ini terdapat kesalahan kata - kata, pengetikan dan kekeliruan, sehingga membingungkan pembaca dalam memahami maksud penulis.

Adapun makalah ini penulis rangkum dari beberapa sumber yang dapat dipercaya yang sajian penulisnya, dirangkum dalam lembar Daftar Pustaka dengan harapan makalah ini dapat menambah pengetahuan kita tentang Pendidikan Di Era Teknologi Informasi Dan Komunikasi. Demikian yang dapat kami sampaikan. Akhir kata, semoga makalah ini dapat menambah wawasan bagi kita semua.

Sago, Mei 2023

Ifdal Lisyukri

DAFTAR ISI

PERANCANGAN & ANALISIS ALGORITMA

BAB I
PENGANTAR ANALISIS ALGORITMA

A. Pengertian Algoritma

Algoritma adalah serangkaian instruksi atau langkah-langkah sistematis yang digunakan untuk menyelesaikan suatu masalah atau tugas dalam komputasi atau matematika. Algoritma biasanya terdiri dari serangkaian langkah yang terorganisir secara logis dan dilakukan dalam urutan tertentu untuk mencapai tujuan tertentu.

Secara umum, algoritma merupakan panduan atau resep untuk menyelesaikan suatu masalah dengan cara yang efektif dan efisien. Algoritma dapat diterapkan pada berbagai bidang, termasuk pemrograman komputer, ilmu data, matematika, dan sains.

Sebuah algoritma biasanya harus memenuhi beberapa kriteria untuk dianggap baik, seperti kejelasan, efektivitas, efisiensi, keunikan solusi, dan korektifitas. Dalam dunia pemrograman, algoritma merupakan dasar dari pembuatan program dan aplikasi. Seorang programmer harus memahami konsep algoritma dengan baik untuk dapat membuat program yang baik dan efisien.

B. Dasar Algoritma

Dasar algoritma terdiri dari beberapa konsep dan teknik yang digunakan untuk merancang algoritma yang efektif dan efisien. Berikut adalah beberapa dasar algoritma yang penting:

1. Struktur Kontrol Aliran Program (Flow Control Structures): Struktur kontrol aliran program digunakan untuk mengatur aliran eksekusi program. Terdapat beberapa jenis struktur kontrol aliran program, seperti struktur seleksi (if-else), struktur perulangan (for, while), dan struktur pemanggilan fungsi.
2. Variabel dan Konstanta: Variabel dan konstanta digunakan untuk menyimpan nilai atau data pada saat program dijalankan. Variabel dapat diubah nilainya selama program berjalan, sedangkan konstanta nilainya tetap sama sepanjang program.
3. Operasi Aritmatika dan Logika: Operasi aritmatika dan logika digunakan untuk melakukan perhitungan matematika atau logika pada nilai atau data yang disimpan dalam variabel atau konstanta.
4. Penggunaan Struktur Data: Struktur data adalah cara untuk menyimpan dan mengelola data yang lebih kompleks dan terstruktur. Beberapa contoh struktur data adalah array, stack, queue, dan linked list.
5. Penggunaan Fungsi: Fungsi digunakan untuk mengelompokkan sejumlah instruksi yang sering digunakan menjadi satu blok yang dapat dipanggil kapan saja selama program berjalan. Hal ini dapat mempermudah dan mempercepat proses pembuatan program.

6. Rekursi: Rekursi adalah teknik di mana suatu fungsi dapat memanggil dirinya sendiri untuk menyelesaikan suatu tugas. Teknik ini sering digunakan pada pemrosesan struktur data yang kompleks, seperti pohon atau graf.

Dengan memahami dasar-dasar algoritma tersebut, seorang programmer dapat merancang algoritma yang lebih efektif dan efisien untuk menyelesaikan suatu masalah atau tugas.

C. Problem Solving

Problem solving (pemecahan masalah) adalah kemampuan untuk menemukan dan mengimplementasikan solusi yang efektif dan efisien terhadap suatu masalah atau situasi yang sulit. Kemampuan ini sangat penting dalam banyak bidang, termasuk teknologi informasi, bisnis, ilmu pengetahuan, dan kehidupan sehari-hari.

Proses problem solving biasanya terdiri dari beberapa langkah, seperti:

1. Identifikasi masalah: Langkah pertama adalah mengidentifikasi masalah atau situasi yang memerlukan solusi. Hal ini melibatkan pengamatan, analisis, dan pemahaman masalah secara menyeluruh.
2. Pengumpulan informasi: Langkah selanjutnya adalah mengumpulkan informasi terkait masalah tersebut, baik dari sumber internal maupun eksternal. Informasi ini dapat membantu dalam memahami lebih lanjut masalah dan menemukan solusi yang tepat.
3. Analisis: Setelah informasi terkumpul, langkah berikutnya adalah menganalisis informasi tersebut dan menentukan penyebab atau akar masalah. Analisis ini dapat membantu dalam menentukan pendekatan yang tepat untuk menyelesaikan masalah.
4. Merancang solusi: Setelah penyebab atau akar masalah telah diidentifikasi, langkah selanjutnya adalah merancang solusi yang tepat. Hal ini melibatkan pemilihan algoritma atau teknik yang paling sesuai untuk menyelesaikan masalah tersebut.
5. Implementasi: Setelah solusi dirancang, langkah berikutnya adalah mengimplementasikannya. Implementasi dapat melibatkan pengembangan program atau aplikasi, perubahan prosedur bisnis, atau tindakan lainnya yang sesuai dengan solusi yang dirancang.
6. Evaluasi: Langkah terakhir adalah mengevaluasi solusi yang diimplementasikan untuk memastikan bahwa masalah telah berhasil dipecahkan dengan benar. Evaluasi dapat dilakukan melalui pengujian dan pengukuran kinerja, dan dapat memberikan umpan balik untuk meningkatkan solusi yang ada.

Kemampuan problem solving sangat penting dalam kehidupan sehari-hari dan dalam dunia bisnis, karena kemampuan ini dapat membantu seseorang untuk menyelesaikan masalah secara efektif dan efisien. Dengan melaksanakan langkah-langkah tersebut, seseorang dapat menjadi lebih terampil dalam menyelesaikan masalah dan menghasilkan solusi yang lebih baik.

BAB II
ANALISIS
FRAMEWORK

A. Measuring an Input Size

Measuring an Input size mengacu pada proses menentukan ukuran atau kompleksitas data masukan yang akan digunakan sebagai masukan untuk suatu algoritma. Ini penting karena performa sebuah algoritma sering tergantung pada ukuran masukan yang diberikan. Mengukur ukuran masukan dapat dilakukan dengan menggunakan berbagai metode tergantung pada jenis data masukan, seperti menghitung jumlah item dalam daftar, mengukur panjang sebuah string, atau menentukan ukuran sebuah file.

Dalam menganalisis kompleksitas waktu suatu algoritma, mengukur ukuran masukan biasanya dinyatakan menggunakan notasi big O, yang menunjukkan batas atas dari laju pertumbuhan algoritma terhadap ukuran masukan. Misalnya, algoritma dengan kompleksitas waktu O(n) berarti waktu yang dibutuhkan oleh algoritma meningkat secara linear dengan ukuran masukan.

Secara ringkas, mengukur ukuran masukan merupakan langkah penting dalam menganalisis performa suatu algoritma, dan dapat dilakukan dengan berbagai metode tergantung pada jenis data masukan. Ukuran masukan biasanya dinyatakan dengan notasi big O dalam menganalisis kompleksitas waktu suatu algoritma.

B. Unit of Measuring Running Time

Unit pengukuran waktu eksekusi (running time) merujuk pada metrik yang digunakan untuk mengukur jumlah waktu yang dibutuhkan oleh sebuah algoritma untuk mengeksekusi atau menyelesaikan sebuah tugas. Mengukur waktu eksekusi sebuah algoritma sangat penting dalam menentukan efisiensi dan performa algoritma tersebut.

Satuan pengukuran yang paling umum digunakan untuk mengukur waktu eksekusi adalah detik, yang mewakili jumlah waktu yang dibutuhkan oleh algoritma untuk mengeksekusi tugas tersebut. Namun, untuk algoritma yang kecil dan dieksekusi dengan cepat, pengukuran dalam detik mungkin kurang akurat, dan oleh karena itu milidetik atau mikrodetik dapat digunakan sebagai pengukuran alternatif.

Satuan pengukuran lain yang digunakan untuk mengukur waktu eksekusi adalah operasi atau instruksi yang dieksekusi oleh algoritma. Ini sering digunakan pada kasus-kasus di mana waktu yang dibutuhkan untuk mengeksekusi setiap operasi konstan, seperti pada operasi aritmatika. Pada kasus tersebut, performa algoritma dapat dianalisis berdasarkan jumlah operasi atau instruksi yang dieksekusi.

Dalam menganalisis kompleksitas waktu sebuah algoritma, satuan pengukuran waktu eksekusi sering dinyatakan menggunakan notasi big O, yang mewakili batas atas dari laju pertumbuhan algoritma terhadap ukuran masukan.

Secara ringkas, satuan pengukuran waktu eksekusi adalah aspek penting dalam mengukur efisiensi dan performa sebuah algoritma. Satuan pengukuran yang paling umum digunakan adalah detik, tetapi satuan lain seperti milidetik atau mikrodetik dapat digunakan untuk algoritma yang kecil. Jumlah operasi atau instruksi yang dieksekusi juga dapat digunakan sebagai satuan pengukuran waktu eksekusi. Dalam menganalisis kompleksitas waktu, notasi big O sering digunakan untuk menyatakan satuan pengukuran waktu eksekusi.

C. Order of Growth

Order of growth, dalam konteks analisis kompleksitas algoritma, merujuk pada laju pertumbuhan dari jumlah operasi atau waktu yang dibutuhkan oleh algoritma dalam menyelesaikan tugasnya seiring dengan meningkatnya ukuran masukan. Order of growth digunakan untuk membandingkan efisiensi dan kinerja berbagai algoritma yang memiliki kompleksitas waktu yang berbeda-beda.

Dalam notasi big O, order of growth dinyatakan sebagai fungsi matematika yang menggambarkan hubungan antara ukuran masukan dan waktu yang dibutuhkan oleh algoritma. Sebagai contoh, jika sebuah algoritma memerlukan waktu O(n) untuk menyelesaikan tugas dengan ukuran masukan n, maka order of growth algoritma tersebut adalah linear, karena waktu yang dibutuhkan meningkat secara linear seiring dengan meningkatnya ukuran masukan.

Ada beberapa kategori order of growth yang umum digunakan dalam analisis kompleksitas algoritma, antara lain:

1. Constant (O(1)): algoritma memiliki waktu eksekusi yang konstan dan tidak tergantung pada ukuran masukan.
2. Linear (O(n)): algoritma memiliki waktu eksekusi yang linier dengan ukuran masukan.
3. Quadratic (O(n^2)): algoritma memiliki waktu eksekusi yang berubah secara kuadratik dengan ukuran masukan.
4. Logarithmic (O(log n)): algoritma memiliki waktu eksekusi yang meningkat secara logaritmik dengan ukuran masukan.
5. Exponential (O(2^n)): algoritma memiliki waktu eksekusi yang meningkat secara eksponensial dengan ukuran masukan.

Dalam memilih algoritma yang tepat untuk menyelesaikan suatu tugas, order of growth menjadi faktor penting dalam menentukan efisiensi dan kinerja algoritma tersebut. Semakin rendah order of growth sebuah algoritma, semakin efisien dan cepat algoritma tersebut dalam menyelesaikan tugas dengan ukuran masukan yang semakin besar.

D. Worst-Case, and Average-Case Efficiency

Efisiensi kasus terburuk (worst-case efficiency) dan kasus rata-rata (average-case efficiency) merujuk pada cara mengukur performa suatu algoritma dalam kondisi yang berbeda.

Efisiensi kasus terburuk adalah pengukuran waktu eksekusi atau jumlah operasi terbanyak yang dibutuhkan oleh algoritma dalam menyelesaikan tugas terberat yang diberikan. Dalam kasus ini, algoritma diuji dengan skenario terburuk, yaitu masukan yang paling sulit atau terburuk. Efisiensi kasus terburuk ini digunakan untuk memberikan jaminan atas waktu maksimum yang dibutuhkan oleh algoritma dalam menyelesaikan tugas pada semua kemungkinan masukan.

Sementara itu, efisiensi kasus rata-rata adalah pengukuran waktu eksekusi atau jumlah operasi rata-rata yang dibutuhkan oleh algoritma dalam menyelesaikan tugas pada berbagai jenis masukan yang mungkin terjadi. Ini dilakukan dengan mengambil rata-rata waktu atau operasi yang dibutuhkan oleh algoritma dalam menyelesaikan tugas pada semua kemungkinan masukan. Efisiensi kasus rata-rata memberikan gambaran yang lebih akurat tentang waktu atau operasi yang dibutuhkan oleh algoritma pada masukan yang umum terjadi.

Kedua metrik ini berguna dalam mengevaluasi performa algoritma, tergantung pada kasus penggunaannya. Jika kasus terburuk dari sebuah algoritma masih memenuhi kebutuhan waktu maksimum, maka efisiensi kasus terburuk digunakan. Namun, jika algoritma diharapkan digunakan pada berbagai jenis masukan, maka efisiensi kasus rata-rata lebih relevan. Perlu diingat bahwa efisiensi kasus terburuk seringkali jauh lebih lambat daripada efisiensi kasus rata-rata pada kasus umum.

Dalam menganalisis kompleksitas waktu algoritma, biasanya fokus pada efisiensi kasus terburuk karena kasus terburuk akan memberikan batas atas dari waktu yang dibutuhkan oleh algoritma pada semua kemungkinan masukan. Namun, efisiensi kasus rata-rata seringkali juga penting, terutama dalam konteks aplikasi yang memerlukan kinerja yang konsisten pada berbagai jenis masukan.

BAB III
BRUTE FORCE DAN EXHAUTIVE SEARCH

A. Selection Sort and Bubble Sort

Selection Sort dan Bubble Sort adalah dua algoritma sorting sederhana yang digunakan untuk mengurutkan data dalam sebuah array atau daftar.

Selection Sort bekerja dengan mencari elemen terkecil dalam array dan menukar posisinya dengan elemen pertama, kemudian mencari elemen terkecil berikutnya dan menukar posisinya dengan elemen kedua, dan seterusnya hingga seluruh array terurut. Algoritma ini memiliki kompleksitas waktu O(n^2) dan tidak cocok untuk mengurutkan array dengan jumlah data yang besar.

Bubble Sort bekerja dengan membandingkan dua elemen sekaligus dalam array, dan menukar posisi elemen tersebut jika mereka tidak terurut, kemudian memindahkan "gelembung" ke posisi berikutnya dan membandingkan kembali elemen di posisi tersebut, dan seterusnya hingga seluruh array terurut. Algoritma ini juga memiliki kompleksitas waktu O(n^2) dan seringkali lebih lambat daripada Selection Sort.

Kedua algoritma ini cukup sederhana dan mudah dipahami, namun efisiensi mereka sangat tergantung pada jumlah data yang diurutkan. Pada array dengan jumlah data yang besar, kedua algoritma ini tidak efisien dan lebih baik diganti dengan algoritma sorting yang lebih cepat seperti QuickSort atau MergeSort.

B. Sequential Search and Brute-ForceString Maching

Sequential Search dan Brute-Force String Matching adalah dua algoritma pencarian sederhana yang digunakan untuk mencari elemen tertentu dalam sebuah array atau string.

Sequential Search, juga dikenal sebagai Linear Search, bekerja dengan mengecek satu per satu elemen dalam array untuk mencari elemen yang dicari. Algoritma ini efektif untuk array dengan ukuran kecil, tetapi kompleksitas waktu nya adalah O(n), di mana n adalah jumlah elemen dalam array. Oleh karena itu, pada array yang besar, Sequential Search kurang efisien dibandingkan dengan algoritma pencarian lainnya seperti Binary Search.

Brute-Force String Matching adalah algoritma pencarian string yang sederhana dan digunakan untuk mencari sebuah pola tertentu dalam sebuah string. Algoritma ini bekerja dengan membandingkan setiap karakter dari pola dengan setiap karakter pada string sampai seluruh string telah diperiksa. Jika tidak ditemukan kecocokan, pola digeser satu karakter ke depan dan proses pencarian diulang hingga ditemukan kecocokan atau sampai pola telah mencapai akhir string. Algoritma ini memiliki kompleksitas waktu O(m*n), di

mana m adalah panjang pola dan n adalah panjang string. Oleh karena itu, pada string yang panjang dan pola yang besar, Brute-Force String Matching juga kurang efisien dan diganti dengan algoritma pencarian string yang lebih cepat seperti Knuth-Morris-Pratt (KMP) atau Boyer-Moore.

Kedua algoritma ini cukup sederhana dan mudah dipahami, namun efisiensi mereka sangat tergantung pada ukuran data yang dicari. Pada data yang besar, kedua algoritma ini tidak efisien dan lebih baik diganti dengan algoritma pencarian yang lebih cepat dan efisien.

C. Closest-Pair and Convex-Hull Problems

Closest-Pair dan Convex-Hull adalah dua masalah geometri komputasi yang sering ditangani dalam ilmu komputer.

Closest-Pair adalah masalah menemukan sepasang titik dalam set data yang memiliki jarak terpendek satu sama lain. Algoritma yang sering digunakan untuk menyelesaikan masalah ini adalah Divide and Conquer. Ide dasarnya adalah membagi set data menjadi dua subset, mencari pasangan terdekat di setiap subset, kemudian mencari pasangan terdekat di antara kedua subset. Algoritma ini memiliki kompleksitas waktu O(n log n) dan sering digunakan dalam masalah pemrosesan citra, pemrosesan sinyal, dan pengenalan wajah.

Convex-Hull adalah masalah menemukan poligon tertutup paling kecil yang mencakup set data. Algoritma yang sering digunakan untuk menyelesaikan masalah ini adalah Graham Scan dan Jarvis March. Ide dasarnya adalah memilih titik paling kiri bawah sebagai titik awal, kemudian mengurutkan semua titik dalam arah berlawanan arah jarum jam, dan membuat poligon yang mengikuti urutan titik tersebut. Algoritma ini memiliki kompleksitas waktu O(n log n) dan sering digunakan dalam pemetaan, pengenalan gambar, dan pengolahan data lainnya.

Kedua masalah ini adalah masalah optimasi dan digunakan dalam banyak aplikasi di bidang ilmu komputer. Meskipun kedua masalah ini sulit untuk dipecahkan, algoritma yang efisien dan cepat dapat menyelesaikan kedua masalah ini dengan baik.

D. Exhautive Search

Exhaustive Search, juga dikenal sebagai Brute Force Search atau Complete Search, adalah teknik algoritma untuk mencari solusi terbaik dari masalah dengan mengecek semua kemungkinan solusi secara sistematis.

Algoritma ini bekerja dengan mencoba semua kemungkinan solusi secara berulang-ulang hingga menemukan solusi yang benar. Proses ini dilakukan dengan membagi masalah menjadi sub-masalah yang lebih kecil dan mengecek semua kemungkinan solusi untuk setiap sub-masalah. Algoritma Exhaustive Search sangat cocok untuk digunakan

pada masalah yang kecil dan sederhana, namun tidak efisien pada masalah yang kompleks dan besar.

Kelebihan dari Exhaustive Search adalah kemampuannya untuk menemukan solusi optimal, karena mencari semua kemungkinan solusi secara menyeluruh. Namun, kelemahan dari Exhaustive Search adalah waktu komputasi yang memakan waktu dan sumber daya yang besar. Oleh karena itu, algoritma ini lebih efektif digunakan pada masalah kecil atau untuk memverifikasi hasil algoritma lain.

Contoh penggunaan Exhaustive Search adalah pada pencarian rute terpendek pada peta, kombinasi kata sandi, atau mencari nilai maksimum dan minimum dari sebuah data. Algoritma Exhaustive Search memiliki kompleksitas waktu O(2^n) atau O(n!), di mana n adalah ukuran masalah, yang artinya waktu komputasi akan meningkat secara eksponensial seiring dengan pertambahan ukuran masalah.

E. Depth-First Search and Breath-First Search

Depth-First Search (DFS) dan Breadth-First Search (BFS) adalah dua algoritma yang sering digunakan dalam pemrograman untuk melakukan pencarian atau penjelajahan pada struktur data seperti graf atau pohon.

DFS bekerja dengan mengeksplorasi sebanyak mungkin cabang atau jalur pada suatu graf atau pohon secara rekursif, sebelum kembali ke cabang atau jalur sebelumnya. Ini berarti algoritma akan menelusuri kedalaman terlebih dahulu sebelum menelusuri lebar. DFS sering digunakan untuk mencari jalur terpendek pada graf atau pohon dan memerlukan kompleksitas waktu O(V + E), di mana V adalah jumlah simpul (node) pada graf dan E adalah jumlah sisi (edge).

Sementara itu, BFS bekerja dengan menelusuri semua simpul pada suatu level sebelum pindah ke level selanjutnya. Algoritma ini akan menelusuri jalur secara lebar terlebih dahulu sebelum menelusuri kedalaman. BFS sering digunakan untuk mencari jalur terpendek antara dua simpul pada graf atau pohon. Algoritma BFS memerlukan kompleksitas waktu O(V + E), sama seperti DFS.

Kedua algoritma ini memiliki kelebihan dan kekurangan masing-masing. DFS lebih cocok digunakan untuk graf yang dalam dan jarang memiliki cabang, sementara BFS lebih cocok digunakan untuk graf yang lebar dan memiliki banyak cabang. DFS juga lebih mudah untuk diimplementasikan secara rekursif, sedangkan BFS membutuhkan struktur data queue untuk memproses simpul pada level yang sama.

Dalam prakteknya, baik DFS maupun BFS sering digunakan secara bersama-sama dalam pemrograman, tergantung pada jenis masalah yang ingin dipecahkan dan struktur data yang digunakan.

BAB IV
DECREASE AND CONQUER

A. Three Major Varian of Decrease-and-Conquer

Decrease-and-Conquer merupakan teknik algoritma yang digunakan untuk menyelesaikan masalah dengan cara memecah masalah menjadi submasalah yang lebih kecil dan menyelesaikan submasalah tersebut secara rekursif. Ada tiga varian utama Decrease-and-Conquer, yaitu sebagai berikut:

1. Decrease-by-One Metode ini dilakukan dengan mengurangi ukuran masalah satu per satu hingga menjadi ukuran masalah yang bisa diselesaikan secara langsung tanpa menggunakan rekursi. Setelah ukuran masalah yang lebih kecil berhasil diselesaikan, solusi untuk masalah asli akan ditemukan secara berurutan. Contoh dari metode Decrease-by-One adalah pengurutan array secara rekursif dengan memisahkan elemen terakhir pada setiap iterasi.
2. Variable-Size-Decrease Metode ini dilakukan dengan mengurangi ukuran masalah secara variatif pada setiap iterasi. Ukuran masalah yang akan diproses pada iterasi berikutnya ditentukan oleh ukuran masalah pada iterasi saat ini. Contoh dari metode Variable-Size-Decrease adalah algoritma binary search, di mana ukuran masalah akan terus berkurang seiring dengan pencarian elemen pada setengah bagian array.
3. Problems-with-Constraints Metode ini dilakukan dengan memecah masalah menjadi submasalah yang lebih kecil dengan batasan tertentu. Batasan tersebut bisa berupa aturan, nilai minimum, atau maksimum. Setelah submasalah berhasil diselesaikan, solusi untuk masalah asli akan ditemukan secara berurutan. Contoh dari metode Problems-with-Constraints adalah pengurutan array dengan batasan waktu dan ruang, di mana elemen array harus diurutkan dalam waktu dan ruang yang terbatas.

Ketiga varian Decrease-and-Conquer memiliki kelebihan dan kekurangan masing-masing. Pemilihan varian yang tepat tergantung pada sifat masalah dan batasan yang diberikan.

B. Sort

Sort atau pengurutan adalah proses menyusun kumpulan data dalam urutan tertentu. Tujuan pengurutan adalah memudahkan proses pencarian, analisis, dan pemrosesan data. Terdapat berbagai algoritma pengurutan yang berbeda-beda, masing-masing memiliki kompleksitas waktu dan ruang yang berbeda pula.

Beberapa jenis algoritma pengurutan yang umum digunakan antara lain:

1. Bubble Sort Algoritma pengurutan dengan membandingkan dua elemen yang bersebelahan pada setiap iterasi, dan menukar posisi jika urutan elemen tersebut tidak sesuai. Algoritma ini relatif mudah dipahami namun memiliki kompleksitas waktu yang cukup tinggi.

2. Selection Sort Algoritma pengurutan dengan memilih elemen terkecil pada setiap iterasi, dan menukar posisi elemen tersebut dengan elemen pada posisi terdepan. Algoritma ini sederhana dan mudah diimplementasikan, namun juga memiliki kompleksitas waktu yang tinggi.
3. Insertion Sort Algoritma pengurutan dengan menyisipkan setiap elemen pada posisi yang tepat pada setiap iterasi. Algoritma ini cocok untuk mengurutkan data yang sudah hampir terurut, namun memiliki kompleksitas waktu yang tinggi untuk data yang tidak terurut.
4. Merge Sort Algoritma pengurutan dengan memecah data menjadi subdata yang lebih kecil, mengurutkan masing-masing subdata secara rekursif, dan menggabungkan subdata tersebut menjadi data yang terurut. Algoritma ini memiliki kompleksitas waktu yang rendah dan cocok untuk mengurutkan data dalam jumlah besar.
5. Quick Sort Algoritma pengurutan dengan memilih pivot, membagi data menjadi dua bagian berdasarkan nilai pivot, dan mengurutkan masing-masing bagian secara rekursif. Algoritma ini cepat dan efisien untuk mengurutkan data dalam jumlah besar, namun rentan terhadap kasus terburuk jika pivot dipilih dengan tidak tepat.

Setiap algoritma pengurutan memiliki kelebihan dan kekurangan masing-masing. Pemilihan algoritma yang tepat tergantung pada sifat data dan kebutuhan aplikasi yang digunakan.

C. Topological Sorting

Topological sorting adalah algoritma pengurutan simpul-simpul pada sebuah graf berarah (directed graph) sedemikian rupa sehingga setiap simpul hanya muncul setelah simpul-simpul yang menuju padanya sudah muncul. Dalam kata lain, topological sorting akan menghasilkan urutan linier simpul-simpul dalam graf yang diurutkan sedemikian rupa sehingga jika terdapat sebuah busur (edge) dari simpul A ke simpul B, maka simpul A harus muncul sebelum simpul B dalam urutan tersebut.

Topological sorting umumnya digunakan pada masalah perencanaan, terutama dalam perencanaan proyek. Contohnya adalah untuk menentukan urutan kegiatan yang harus dilakukan pada suatu proyek, di mana beberapa kegiatan harus dilakukan sebelum kegiatan-kegiatan lainnya. Topological sorting juga sering digunakan pada analisis jaringan, pemrosesan bahasa alami, dan optimasi kompilator.

Algoritma topological sorting bekerja dengan mencari simpul-simpul yang tidak memiliki derajat masuk (in-degree) atau tidak memiliki busur masuk. Kemudian simpul-simpul tersebut dikeluarkan dari graf, bersamaan dengan semua busur yang keluar dari simpul tersebut. Algoritma akan terus mencari simpul-simpul dengan derajat masuk 0 dan mengeluarkannya dari graf sampai tidak ada simpul yang tersisa.

Algoritma topological sorting dapat diimplementasikan dengan menggunakan pendekatan yang mirip dengan algoritma Breadth-First Search (BFS) atau Depth-First Search (DFS). Kompleksitas waktu dari algoritma topological sorting adalah O(|V|+|E|), di mana |V| dan |E| adalah jumlah simpul dan busur dalam graf secara berturut-turut.

BAB V
DEVIDE AND CONQUER

A. Mergesort

Merge sort adalah salah satu algoritma pengurutan (sorting) yang berbasis pada paradigma divide and conquer, yaitu memecah masalah menjadi submasalah yang lebih kecil, menyelesaikan submasalah tersebut secara rekursif, dan kemudian menggabungkan solusi submasalah menjadi solusi untuk masalah aslinya.

Cara kerja merge sort adalah dengan membagi data yang akan diurutkan menjadi dua bagian sama besar secara rekursif sampai tidak bisa dibagi lagi. Setelah itu, dilakukan penggabungan (merge) dua bagian data tersebut dalam urutan yang sudah terurut sehingga menghasilkan kumpulan data yang terurut secara keseluruhan.

Penggabungan pada merge sort dilakukan dengan membandingkan elemen-elemen dari dua bagian data secara berurutan. Elemen-elemen yang lebih kecil akan diambil dan dimasukkan ke dalam kumpulan data hasil penggabungan, kemudian dilanjutkan dengan membandingkan elemen-elemen berikutnya sampai salah satu bagian data telah kosong. Kemudian, semua elemen yang tersisa di bagian data yang masih memiliki elemen akan dimasukkan ke dalam kumpulan data hasil penggabungan.

Merge sort memiliki kompleksitas waktu O(n log n), di mana n adalah jumlah data yang akan diurutkan. Algoritma ini sangat efisien untuk data yang besar dan terkadang digunakan sebagai algoritma pengurutan default pada beberapa bahasa pemrograman dan library.

B. Quicksort

Quick sort adalah salah satu algoritma pengurutan (sorting) yang berbasis pada paradigma divide and conquer, yaitu memecah masalah menjadi submasalah yang lebih kecil, menyelesaikan submasalah tersebut secara rekursif, dan kemudian menggabungkan solusi submasalah menjadi solusi untuk masalah aslinya.

Cara kerja quick sort adalah dengan memilih sebuah elemen dari data yang akan diurutkan sebagai pivot, kemudian membagi data menjadi dua bagian, yaitu bagian yang semua elemennya lebih kecil atau sama dengan pivot (partisi kiri), dan bagian yang semua elemennya lebih besar dari pivot (partisi kanan). Setelah itu, quick sort akan melakukan rekursi pada kedua partisi tersebut secara terpisah sampai hanya tersisa satu elemen atau tidak ada elemen yang perlu diurutkan.

Saat melakukan partisi, algoritma quick sort menggunakan dua pointer, yaitu pointer kiri dan pointer kanan, yang akan bergerak menuju satu sama lain untuk mencari pasangan elemen yang tidak berada pada posisi yang seharusnya di antara partisi kiri dan kanan.

Ketika ditemukan pasangan elemen yang tidak berada pada posisi yang seharusnya, maka kedua elemen tersebut akan ditukar posisinya. Setelah semua pasangan elemen yang tidak berada pada posisi yang seharusnya sudah ditukar, maka pivot akan diposisikan pada tempat yang seharusnya di antara partisi kiri dan kanan.

Quick sort memiliki kompleksitas waktu rata-rata O(n log n), di mana n adalah jumlah data yang akan diurutkan. Namun, pada kasus terburuk ketika pivot dipilih secara tidak optimal, kompleksitas waktu dapat mencapai O(n^2). Untuk menghindari kasus terburuk, penggunaan pivot yang optimal menjadi sangat penting. Beberapa metode yang umum digunakan untuk memilih pivot adalah memilih elemen pertama, terakhir, atau elemen tengah dari data yang akan diurutkan.

C. Binary Tree Traversals and Related Properties

Binary tree traversals adalah suatu cara untuk mengunjungi (traversal) setiap node pada binary tree secara sistematis. Terdapat tiga jenis binary tree traversals yang umum digunakan, yaitu:

1. Inorder Traversal: mengunjungi node secara urut dari kiri ke kanan pada subpohon kiri, kemudian mengunjungi root node, dan terakhir mengunjungi node secara urut dari kiri ke kanan pada subpohon kanan.
2. Preorder Traversal: mengunjungi root node terlebih dahulu, kemudian mengunjungi node secara urut dari kiri ke kanan pada subpohon kiri, dan terakhir mengunjungi node secara urut dari kiri ke kanan pada subpohon kanan.
3. Postorder Traversal: mengunjungi node secara urut dari kiri ke kanan pada subpohon kiri, kemudian mengunjungi node secara urut dari kiri ke kanan pada subpohon kanan, dan terakhir mengunjungi root node.

Selain itu, binary tree memiliki beberapa properti penting, yaitu:

1. Height: jumlah level (tingkat) maksimum pada binary tree.
2. Depth: jarak dari root node ke suatu node pada binary tree.
3. Degree: banyaknya anak (child) yang dimiliki oleh suatu node pada binary tree.
4. Full Binary Tree: binary tree di mana setiap node memiliki tepat dua anak.
5. Complete Binary Tree: binary tree di mana setiap level kecuali mungkin level terakhir terisi penuh, dan pada level terakhir semua node berada pada posisi terkiri.
6. Perfect Binary Tree: binary tree yang merupakan full binary tree dan semua level-nya terisi penuh.

Binary tree traversals dan properti-properti di atas dapat digunakan dalam berbagai aplikasi, seperti pencarian dan penyimpanan data, pembuatan struktur data baru, serta optimisasi algoritma.

BAB VI
TRANSFORM AND CONQUER

A. Instance Simplification

Instance simplification merupakan salah satu teknik dalam algoritma decrease-and-conquer yang bertujuan untuk menyederhanakan sebuah instance (masalah) sehingga dapat diselesaikan dengan lebih efisien. Teknik ini bekerja dengan mengurangi ukuran instance menjadi instance yang lebih kecil, namun tetap memiliki solusi yang sama dengan instance asli.

Contohnya, dalam masalah pengurutan (sorting), teknik instance simplification dapat digunakan dengan membagi instance menjadi dua sub-instance yang lebih kecil dengan menggunakan pivot element. Kemudian, kedua sub-instance tersebut dapat diurutkan secara terpisah dan digabungkan kembali untuk mendapatkan solusi untuk instance asli.

Teknik instance simplification juga dapat diterapkan pada masalah-masalah lainnya, seperti searching, graph traversal, dan lain sebagainya. Dengan mengurangi ukuran instance, waktu dan sumber daya yang diperlukan untuk menyelesaikan masalah dapat dikurangi sehingga algoritma menjadi lebih efisien dan efektif.

B. Representation Change

Representation change adalah teknik dalam algoritma divide-and-conquer yang digunakan untuk mengubah representasi dari sebuah masalah menjadi bentuk yang lebih mudah dan cepat untuk dipecahkan. Teknik ini dilakukan dengan cara mengubah masalah dari suatu representasi ke representasi lain yang lebih mudah dikerjakan, kemudian masalah tersebut dapat dipecahkan secara lebih efisien.

Contohnya, dalam masalah perkalian matriks, jika kita ingin mengalikan dua buah matriks dengan ukuran yang besar, maka proses perkalian akan membutuhkan waktu yang lama dan sumber daya yang besar. Namun, dengan menggunakan teknik representation change, kita dapat mengubah representasi matriks tersebut menjadi representasi lain yang lebih efisien, seperti representasi bit, dan kemudian melakukan perkalian pada representasi bit tersebut.

Teknik representation change juga dapat digunakan pada masalah-masalah lainnya, seperti pengurutan, searching, dan graph traversal. Dengan mengubah representasi masalah menjadi bentuk yang lebih efisien dan cepat dikerjakan, waktu dan sumber daya yang dibutuhkan untuk menyelesaikan masalah dapat dikurangi sehingga algoritma menjadi lebih efisien dan efektif.

C. Problem Reduction

Problem reduction adalah teknik dalam algoritma divide-and-conquer yang digunakan untuk menyelesaikan sebuah masalah dengan cara mengurangi (reduksi) masalah tersebut menjadi masalah yang lebih sederhana atau lebih mudah untuk dipecahkan. Dalam teknik ini, masalah asli (yang sulit) diubah menjadi masalah baru yang lebih mudah dikerjakan, dan kemudian solusi untuk masalah baru tersebut digunakan untuk menemukan solusi untuk masalah asli.

Contohnya, dalam masalah 3-SAT (satisfiability), yaitu masalah menentukan apakah suatu formula boolean dapat dipenuhi atau tidak, teknik problem reduction dapat digunakan dengan cara mengubah masalah 3-SAT menjadi masalah graph coloring. Dalam masalah graph coloring, tujuannya adalah untuk menentukan cara memberi warna pada simpul-simpul (node) dalam sebuah graf, sedemikian rupa sehingga dua simpul yang terhubung dengan edge tidak boleh memiliki warna yang sama. Masalah graph coloring lebih mudah dikerjakan dibandingkan 3-SAT, sehingga jika solusi untuk masalah graph coloring dapat ditemukan, maka solusi untuk masalah 3-SAT juga dapat ditemukan.

Teknik problem reduction juga dapat digunakan pada masalah-masalah lainnya, seperti pengurutan, searching, dan graph traversal. Dengan mengurangi kompleksitas masalah, waktu dan sumber daya yang dibutuhkan untuk menyelesaikan masalah dapat dikurangi sehingga algoritma menjadi lebih efisien dan efektif.

BAB VII
SPACE AND TIME TRADE-OFFS

A. Sorting by Counting

Sorting by counting (counting sort) adalah algoritma sorting yang sangat efisien untuk mengurutkan daftar elemen numerik dengan nilai yang terbatas. Algoritma ini beroperasi dengan cara menghitung berapa kali setiap elemen muncul dalam daftar, dan kemudian menempatkan elemen tersebut pada posisi yang tepat dalam hasil pengurutan.

Cara kerja counting sort adalah dengan menghitung frekuensi kemunculan setiap elemen dalam daftar. Frekuensi ini kemudian diakumulasikan sehingga setiap nilai yang unik dalam daftar memiliki nilai akumulasi yang menunjukkan posisi awal di hasil pengurutan. Setiap elemen dalam daftar kemudian ditempatkan pada posisi yang sesuai dalam hasil pengurutan dengan cara memanfaatkan nilai akumulasi yang telah dihitung sebelumnya.

Counting sort sangat efektif jika nilai maksimum dan minimum dalam daftar diketahui dan nilai-nilai tersebut tidak terlalu jauh terpisah. Algoritma ini memiliki kecepatan yang sangat cepat, yaitu O(n+k) di mana n adalah jumlah elemen dalam daftar dan k adalah jumlah nilai yang mungkin dalam daftar.

Namun, counting sort memiliki kelemahan pada penggunaannya yang terbatas pada daftar dengan nilai terbatas, karena memerlukan array yang cukup besar untuk menghitung frekuensi kemunculan setiap elemen. Selain itu, counting sort tidak efektif jika nilai maksimum dalam daftar sangat besar.

B. Input Enchantment in String Maching

Input Enchantment pada String Matching adalah teknik untuk mengubah teks pencarian (pattern) dengan menambahkan karakter khusus (disebut wildcard) untuk meningkatkan kemampuan pencarian yang lebih fleksibel. Teknik ini dilakukan dengan menambahkan karakter khusus ke dalam teks pencarian (pattern), yang menunjukkan bahwa karakter pada posisi tersebut bisa digantikan oleh karakter apa saja dalam teks yang dicari (text).

Contohnya, dalam teks pencarian "anl", simbol "" merupakan wildcard yang menggantikan karakter apa saja pada posisi tersebut. Sehingga pencarian akan menghasilkan output untuk kata-kata seperti "anal", "anil", "angel", dan sebagainya.

Teknik ini berguna dalam pencarian teks yang tidak memiliki pola yang konsisten atau terdapat kesalahan ketik pada teks yang dicari, sehingga meningkatkan kemungkinan berhasilnya pencarian. Namun, penggunaan teknik ini dapat mempengaruhi kinerja

algoritma pencarian, tergantung pada jenis wildcard yang digunakan dan seberapa sering wildcard tersebut muncul dalam teks pencarian.

Beberapa algoritma pencarian string yang mendukung Input Enchantment termasuk algoritma Boyer-Moore dan algoritma Knuth-Morris-Pratt (KMP).

C. Hasing

Hashing adalah teknik yang digunakan dalam komputasi untuk memetakan data atau nilai tertentu ke dalam nilai acak yang lebih kecil atau singkat, yang disebut hash code. Hashing digunakan dalam berbagai aplikasi seperti database, pencarian, kriptografi, dan lain-lain.

Dalam hashing, sebuah fungsi hash digunakan untuk memetakan data masukan ke dalam nilai hash, yang biasanya jauh lebih kecil dari ukuran data masukan. Fungsi hash ini harus memiliki sifat deterministik, artinya setiap kali diberikan data masukan yang sama, fungsi akan menghasilkan nilai hash yang sama pula. Selain itu, fungsi hash juga harus memiliki sifat sebaliknya, yaitu setiap kali diberikan nilai hash yang sama, fungsi harus mampu mengembalikan data masukan yang benar.

Setelah nilai hash diperoleh, hash code ini kemudian digunakan untuk mencari dan membandingkan data di dalam struktur data seperti tabel hash, dalam operasi seperti pencarian, pengurutan, dan penghapusan data. Keuntungan dari menggunakan hashing adalah pengurangan waktu pencarian data dalam struktur data, karena hanya perlu mencocokkan nilai hash code saja, bukan seluruh nilai data.

Namun, hashing juga memiliki beberapa kekurangan seperti kemungkinan terjadinya tabrakan hash code, yaitu ketika dua nilai data yang berbeda memiliki hash code yang sama. Hal ini dapat memperlambat kinerja algoritma dan menurunkan efisiensi pencarian. Untuk mengatasi hal ini, beberapa teknik seperti open addressing, chaining, dan probing dapat digunakan dalam implementasi tabel hash.

BAB VIII
DYNAMIC PROGRAMMING

A. Three Basic

Tiga dasar dalam Dynamic Programming adalah sebagai berikut:

1. Overlapping Subproblems: Subproblem-subproblem yang muncul dalam permasalahan harus terlihat secara berulang kali. Oleh karena itu, solusi dari subproblem yang sama harus disimpan dan digunakan kembali.
2. Optimal Substructure: Solusi optimal dari permasalahan harus memiliki solusi optimal dari subproblem-subproblem yang lebih kecil. Dengan kata lain, solusi global dapat dicapai dengan menggabungkan solusi dari subproblem yang lebih kecil.
3. Memoization: Teknik ini digunakan untuk menyimpan hasil pemrosesan pada subproblem-subproblem yang telah diselesaikan. Hasil tersebut akan digunakan kembali pada saat subproblem yang sama muncul lagi. Dengan memanfaatkan teknik memoization, waktu eksekusi program dapat dipercepat.

B. The Knapsack Problem and Memory Functions

Masalah Knapsack adalah masalah terkenal dalam ilmu komputer dan optimisasi. Ini melibatkan kumpulan item, masing-masing dengan bobot dan nilai, dan knapsack yang dapat menampung bobot terbatas. Tujuannya adalah untuk menemukan kombinasi item yang memaksimalkan nilai total tanpa melebihi batas bobot dari knapsack.

Pemrograman Dinamis sering digunakan untuk menyelesaikan Masalah Knapsack. Salah satu teknik yang digunakan dalam Pemrograman Dinamis adalah penggunaan fungsi memori, yang digunakan untuk menyimpan hasil yang telah dihitung sebelumnya untuk menghindari perhitungan yang redundan.

Dalam konteks Masalah Knapsack, fungsi memori dapat digunakan untuk menyimpan nilai maksimum yang dapat diperoleh dengan memilih subset item dengan total bobot kurang dari atau sama dengan batas tertentu. Fungsi ini disebut "fungsi nilai". Fungsi nilai diinisialisasi dengan nilai nol untuk semua kombinasi indeks bobot dan item yang mungkin, dan kemudian diperbarui secara rekursif untuk setiap kombinasi yang mungkin. Nilai akhir dari fungsi untuk batas bobot yang diinginkan adalah solusi untuk Masalah Knapsack.

Menggunakan fungsi memori untuk menyelesaikan Masalah Knapsack adalah contoh bagaimana Pemrograman Dinamis dapat digunakan untuk mengoptimalkan masalah kompleks dengan memecahnya menjadi submasalah yang lebih kecil dan menyimpan hasilnya untuk digunakan nanti.

C. Warshall’sand Floyd’s Algorithms

Algoritma Warshall dan Floyd adalah dua algoritma umum yang digunakan untuk menyelesaikan masalah jalur terpendek antar semua pasang titik pada sebuah graf berbobot.

Algoritma Warshall menggunakan pemrograman dinamis untuk menemukan jalur terpendek antara setiap pasang verteks pada sebuah graf berarah dengan tepian bermuatan. Algoritma ini membangun matriks jalur terpendek antara setiap pasang verteks menggunakan ide penutupan transitif. Matriks tersebut diinisialisasi dengan bobot tepian, dan kemudian diperbarui secara iteratif dengan memeriksa apakah terdapat jalur yang lebih pendek melalui suatu titik ketiga.

Algoritma Floyd, juga dikenal sebagai algoritma Floyd-Warshall, adalah pendekatan yang mirip untuk menyelesaikan masalah jalur terpendek antar semua pasang titik pada sebuah graf berbobot. Seperti algoritma Warshall, algoritma Floyd juga menggunakan pemrograman dinamis untuk membangun matriks jalur terpendek antara setiap pasang verteks. Namun, alih-alih menggunakan penutupan transitif, algoritma Floyd menggunakan ide pemrograman dinamis untuk membangun matriks secara iteratif dengan mempertimbangkan semua titik tengah yang mungkin.

Kedua algoritma Warshall dan Floyd memiliki kompleksitas waktu O(n^3), sehingga efisien untuk menyelesaikan masalah jalur terpendek antar semua pasang titik pada graf berukuran kecil hingga menengah.

BAB IX
GREEDY TECHNIQUE

A. Prim's Algorithm

Algoritma Prim adalah algoritma yang digunakan untuk mencari Minimum Spanning Tree (MST) dalam sebuah graf berbobot. MST adalah subgraf dari graf asli yang terhubung, tidak mengandung siklus, dan memiliki bobot total yang minimum.

Algoritma Prim dimulai dengan memilih satu titik awal sebagai simpul awal. Kemudian, secara iteratif, simpul-simpul yang terhubung dengan MST yang sedang dibangun ditambahkan satu per satu berdasarkan bobot terkecil dari tepian yang menghubungkan simpul-simpul tersebut dengan simpul-simpul yang sudah ada di MST.

Selama proses iterasi, setiap simpul baru yang ditambahkan ke MST akan terhubung dengan tepian yang memiliki bobot terkecil yang menghubungkannya dengan simpul-simpul yang sudah ada di MST. Ini dikenal sebagai kriteria Greedy, di mana keputusan yang diambil pada setiap langkah hanya mempertimbangkan informasi lokal untuk membangun MST secara keseluruhan.

Algoritma Prim akan berlanjut hingga semua simpul dalam graf terhubung dengan MST yang sedang dibangun, atau hingga mencapai jumlah simpul yang diinginkan untuk MST. Algoritma ini menghasilkan MST dengan bobot minimum jika graf tersebut terhubung.

Algoritma Prim memiliki kompleksitas waktu O(E log V) dengan menggunakan representasi graf dengan heap biner, di mana E adalah jumlah tepian (edge) dan V adalah jumlah simpul (vertex) dalam graf.

B. Kruskal's Algorithm

Algoritma Kruskal adalah algoritma yang digunakan untuk mencari Minimum Spanning Tree (MST) dalam sebuah graf berbobot. MST adalah subgraf dari graf asli yang terhubung, tidak mengandung siklus, dan memiliki bobot total yang minimum.

Algoritma Kruskal dimulai dengan mengurutkan semua tepian (edge) graf berdasarkan bobotnya secara menaik. Kemudian, tepian dengan bobot terkecil dipilih satu per satu dan ditambahkan ke MST jika penambahan tersebut tidak membentuk siklus dengan tepian yang sudah ada di MST.

Selama proses iterasi, setiap tepian baru yang ditambahkan akan memperluas jangkauan MST yang sedang dibangun dengan menyambungkan dua simpul yang sebelumnya tidak terhubung. Algoritma ini menggunakan pendekatan Greedy, di mana

keputusan yang diambil pada setiap langkah hanya mempertimbangkan informasi lokal untuk membangun MST secara keseluruhan.

Algoritma Kruskal akan terus berlanjut hingga semua simpul dalam graf terhubung atau hingga mencapai jumlah tepian yang diinginkan untuk MST. Algoritma ini menghasilkan MST dengan bobot minimum jika graf tersebut terhubung.

Algoritma Kruskal memiliki kompleksitas waktu O(E log V) dengan menggunakan representasi graf dengan struktur data seperti Union-Find, di mana E adalah jumlah tepian (edge) dan V adalah jumlah simpul (vertex) dalam graf.

C. Djikstra's Algorithm

Algoritma Dijkstra, yang dinamai dari ilmuwan Edsger W. Dijkstra, adalah algoritma yang digunakan untuk mencari jalur terpendek antara dua simpul dalam sebuah graf berbobot. Algoritma ini bekerja dengan asumsi bahwa graf tidak memiliki tepian dengan bobot negatif.

Algoritma Dijkstra dimulai dengan memilih simpul awal dan memberikan bobot awal 0 ke simpul tersebut. Kemudian, secara iteratif, simpul dengan bobot terkecil yang belum dikunjungi dipilih dan dikunjungi. Setelah mengunjungi simpul tersebut, bobot terpendek ke simpul-simpul yang terhubung diperbarui jika ada jalur dengan bobot yang lebih kecil melalui simpul yang baru dikunjungi.

Selama proses iterasi, algoritma Dijkstra menggunakan pendekatan Greedy dengan memilih simpul berikutnya berdasarkan bobot terkecil untuk memperbarui bobot terpendek ke simpul-simpul lainnya. Ini memastikan bahwa setiap simpul yang dikunjungi memiliki bobot terpendek yang diketahui hingga saat ini.

Algoritma Dijkstra akan terus berlanjut hingga semua simpul telah dikunjungi atau sampai jalur terpendek ke simpul tujuan ditemukan. Algoritma ini menghasilkan jalur terpendek dari simpul awal ke simpul tujuan serta bobot total dari jalur tersebut.

Algoritma Dijkstra memiliki kompleksitas waktu O((V + E) log V), di mana V adalah jumlah simpul (vertex) dan E adalah jumlah tepian (edge) dalam graf.

D. Huffman Trees and Codes

Pohon Huffman dan Kode Huffman adalah teknik kompresi data yang digunakan untuk mengurangi ukuran data dengan memanfaatkan frekuensi kemunculan karakter dalam data tersebut.

Pohon Huffman adalah sebuah pohon biner khusus yang digunakan untuk menghasilkan kode Huffman. Pohon ini dibangun dengan mempertimbangkan frekuensi kemunculan karakter dalam data. Karakter-karakter dengan frekuensi kemunculan yang lebih tinggi akan diberikan posisi lebih dekat dengan akar pohon, sedangkan karakter-

karakter dengan frekuensi yang lebih rendah akan ditempatkan lebih jauh dari akar. Setiap cabang dalam pohon mewakili pergeseran bit dalam kode Huffman, dengan cabang kiri mewakili pergeseran bit ke arah 0 dan cabang kanan mewakili pergeseran bit ke arah 1.

Kode Huffman adalah representasi biner dari karakter dalam data menggunakan pohon Huffman. Kode Huffman adalah kode biner yang unik untuk setiap karakter, di mana karakter yang lebih sering muncul memiliki kode yang lebih pendek, dan karakter yang jarang muncul memiliki kode yang lebih panjang. Dalam kompresi data, karakter-karakter dalam data asli digantikan dengan kode Huffman mereka yang sesuai, sehingga mengurangi ukuran total data yang perlu disimpan atau ditransmisikan.

Penggunaan Pohon Huffman dan Kode Huffman dalam kompresi data memungkinkan efisiensi yang tinggi dalam mengurangi ukuran data. Algoritma Huffman merupakan algoritma greedy, di mana keputusan yang diambil pada setiap langkah hanya mempertimbangkan informasi lokal (frekuensi kemunculan karakter) untuk membangun pohon Huffman dan kode Huffman secara keseluruhan.

BAB X
ITERATIVE IMPROVEMENT

A. The Simplex Method

Metode Simpleks adalah algoritma yang digunakan untuk menyelesaikan masalah optimasi linear, yaitu masalah mencari solusi terbaik untuk fungsi tujuan linear yang terbatas oleh sejumlah kendala linear.

Algoritma Metode Simpleks dimulai dengan membangun suatu titik awal dalam ruang solusi yang memenuhi semua kendala. Pada setiap iterasi, algoritma memperbaiki solusi saat ini dengan memindahkan titik awal ke tetangga yang lebih baik dalam ruang solusi. Hal ini dilakukan dengan mempertimbangkan arah peningkatan fungsi tujuan serta batasan kendala yang harus dipenuhi.

Pada setiap langkah iterasi, algoritma Simpleks memilih suatu titik yang berada pada batas kendala (yang dikenal sebagai titik sudut) dan mempertimbangkan perpindahan ke tetangga yang memperbaiki nilai fungsi tujuan. Jika tidak ada perpindahan yang menghasilkan peningkatan, maka solusi saat ini dianggap sebagai solusi optimal.

Metode Simpleks terus berlanjut hingga solusi optimal ditemukan atau hingga tercapai batasan waktu atau iterasi yang ditentukan. Algoritma ini dapat menangani masalah dengan banyak variabel dan kendala, dan telah terbukti efektif dalam mencari solusi optimal untuk masalah optimasi linear.

Meskipun Metode Simpleks secara umum adalah metode yang efisien dalam menyelesaikan masalah optimasi linear, ada beberapa kasus di mana kompleksitas waktu algoritma dapat menjadi tinggi. Oleh karena itu, terdapat pengembangan dan variasi Metode Simpleks, seperti Metode Simpleks Revisi, yang dapat meningkatkan efisiensi algoritma dalam beberapa kasus khusus.

B. The Maximum-Flow Problem

Masalah Aliran Maksimum (Maximum-Flow Problem) adalah sebuah masalah dalam teori graf yang bertujuan untuk mencari aliran maksimum yang dapat melewati jaringan graf dengan kapasitas yang ditentukan dari simpul sumber (source) ke simpul tujuan (sink).

Dalam masalah aliran maksimum, jaringan graf direpresentasikan sebagai graf berarah dengan tepian (edge) yang memiliki kapasitas. Simpul sumber merupakan simpul awal dari aliran, sedangkan simpul tujuan adalah simpul akhir yang ingin dicapai aliran maksimum. Setiap tepian memiliki batasan kapasitas yang menunjukkan jumlah maksimum aliran yang dapat mengalir melalui tepian tersebut.

Tujuan dari masalah ini adalah menemukan cara untuk mengalokasikan aliran sehingga jumlah aliran yang keluar dari simpul sumber ke simpul tujuan adalah maksimum. Aliran harus mematuhi batasan kapasitas tepian dan memenuhi hukum kekekalan aliran, yaitu jumlah aliran yang masuk ke suatu simpul harus sama dengan jumlah aliran yang keluar dari simpul tersebut.

Untuk menyelesaikan masalah aliran maksimum, digunakan berbagai algoritma seperti Algoritma Ford-Fulkerson dan Algoritma Edmonds-Karp. Algoritma-algoritma ini menggunakan pendekatan iteratif untuk menemukan jalur-jalur yang memungkinkan aliran untuk meningkatkan aliran maksimum secara bertahap.

Masalah aliran maksimum memiliki banyak aplikasi dalam berbagai bidang, seperti perencanaan rute transportasi, optimasi jaringan komunikasi, dan pemodelan aliran dalam sistem logistik.

C. Maximum Matching in Bipartite Graphs

Matching maksimum dalam graf bipartit adalah masalah yang mencari pasangan tepi terbesar yang tidak saling tumpang tindih antara dua himpunan simpul dalam sebuah graf bipartit.

Graf bipartit terdiri dari dua himpunan simpul yang disebut sebagai himpunan simpul kiri (L) dan himpunan simpul kanan (R). Tepi dalam graf bipartit hanya menghubungkan simpul dari himpunan L ke simpul himpunan R, atau sebaliknya. Tujuan dari matching maksimum adalah menemukan himpunan tepi terbesar yang tidak memiliki simpul bersama antara simpul-simpul yang dipasangkan.

Algoritma yang umum digunakan untuk menyelesaikan masalah matching maksimum dalam graf bipartit adalah Algoritma Hopcroft-Karp. Algoritma ini menggunakan pendekatan pemodelan sebagai masalah pencarian jalur melalui augmenting paths (jalur peningkatan) dalam graf. Dalam setiap langkah, algoritma mencoba menemukan jalur peningkatan dengan menggunakan algoritma Breadth-First Search (BFS) dan memperbarui pasangan tepi yang ada. Algoritma berlanjut hingga tidak ada jalur peningkatan yang dapat ditemukan lagi.

Hasil dari matching maksimum adalah himpunan tepi terbesar yang tidak saling tumpang tindih antara simpul-simpul dari himpunan L dan R. Hasil ini dapat digunakan dalam berbagai aplikasi, seperti pencocokan pasangan dalam jadwal penugasan, pencarian pasangan dalam jaringan sosial, atau penugasan optimal dalam bidang pemrosesan data.

D. The Stable Marriage Problem

Masalah Pernikahan yang Stabil (Stable Marriage Problem) adalah sebuah masalah yang mencari pencocokan pasangan yang stabil antara dua himpunan orang dengan preferensi pasangan mereka.

Dalam masalah ini, terdapat dua himpunan orang, biasanya disebut sebagai himpunan pria dan himpunan wanita. Setiap orang memberikan preferensi mereka terhadap pasangan dari himpunan lawan jenis. Tujuan dari masalah ini adalah mencari pencocokan pasangan yang stabil, di mana tidak ada pasangan yang lebih suka memiliki pasangan lain di luar pasangan yang telah dicocokkan.

Stabilitas dalam pencocokan pasangan didefinisikan dengan adanya pasangan yang "melanggar" pencocokan, yaitu pasangan yang lebih suka memiliki pasangan lain di luar pasangan yang telah dicocokkan. Sebuah pencocokan pasangan dikatakan stabil jika tidak ada pasangan yang melanggar pencocokan.

Algoritma yang umum digunakan untuk menyelesaikan masalah Pernikahan yang Stabil adalah Algoritma Gale-Shapley. Algoritma ini menggunakan pendekatan berbasis preferensi, di mana setiap pria secara berurutan mengajukan proposal kepada wanita berdasarkan preferensinya. Wanita kemudian mempertimbangkan proposal yang diterimanya dan memilih pasangan yang paling disukainya. Pria yang ditolak akan mengajukan proposal kepada wanita lain, dan proses ini berlanjut hingga semua pasangan terbentuk.

Hasil dari masalah Pernikahan yang Stabil adalah pencocokan pasangan yang stabil, di mana tidak ada pasangan yang melanggar pencocokan. Hasil ini memastikan bahwa setiap orang memiliki pasangan yang mereka sukai lebih dari pasangan lain yang mungkin tersedia. Masalah ini memiliki aplikasi dalam berbagai konteks, seperti pencocokan pasangan dalam jadwal penugasan, pemasaran online, dan analisis pasar.

BAB XI
LIMITATIONS OF ALGORITHM POWER

A. Lower-Bounf Arguments

Argumen Batas Bawah (Lower-Bound Arguments) adalah teknik yang digunakan dalam analisis algoritma untuk menentukan batas bawah (lower bound) dari kompleksitas waktu yang diperlukan untuk memecahkan suatu masalah.

Dalam analisis algoritma, kita tertarik untuk mengetahui seberapa efisien suatu algoritma dalam menyelesaikan masalah tertentu. Batas bawah adalah batas terendah dari kompleksitas waktu yang mungkin diperlukan oleh algoritma untuk memecahkan masalah tersebut. Jika kita dapat membuktikan bahwa tidak ada algoritma yang dapat menyelesaikan masalah dalam waktu yang lebih cepat dari batas bawah yang ditentukan, maka batas bawah tersebut menjadi indikator kinerja optimal.

Argumen batas bawah biasanya melibatkan pemodelan masalah dengan struktur matematis yang sesuai. Melalui analisis matematika dan logika, kemudian dibuktikan bahwa solusi optimal memerlukan waktu setidaknya sebanyak batas bawah yang ditentukan.

Metode yang umum digunakan dalam argumen batas bawah adalah pengurangan masalah (problem reduction) dan pemakaian teknik kompleksitas waktu yang sudah diketahui sebelumnya. Dengan memanfaatkan informasi tentang kompleksitas waktu dari masalah yang sudah terbukti sulit, kita dapat memperoleh batas bawah untuk masalah yang sedang dianalisis.

Argumen batas bawah merupakan alat penting dalam analisis algoritma untuk memahami batasan kinerja algoritma dan membandingkan keefisienan berbagai pendekatan dalam menyelesaikan masalah.

B. Decision Tree

Decision Tree (Pohon Keputusan) adalah salah satu metode dalam machine learning yang digunakan untuk pengambilan keputusan atau klasifikasi berdasarkan serangkaian aturan keputusan yang terstruktur dalam bentuk pohon.

Pohon Keputusan terdiri dari simpul-simpul yang merepresentasikan keputusan atau pengamatan, tepian (edge) yang merepresentasikan hubungan antara keputusan atau pengamatan, dan daun-daun yang merepresentasikan hasil atau klasifikasi akhir. Pada setiap simpul, suatu atribut atau fitur dipilih sebagai kondisi yang membagi data ke dalam kelompok yang lebih kecil. Proses ini berlanjut hingga mencapai daun-daun pohon, di

mana keputusan atau klasifikasi akhir diambil berdasarkan nilai atribut daun tersebut.

Pohon Keputusan dibangun dengan menggunakan algoritma pembelajaran yang mengoptimalkan pemilihan atribut atau fitur yang paling informatif dalam membagi data. Beberapa algoritma terkenal untuk membangun Pohon Keputusan termasuk algoritma ID3, C4.5, dan CART.

Keuntungan utama dari Pohon Keputusan adalah kemampuannya untuk secara intuitif menggambarkan proses pengambilan keputusan dan memberikan pemahaman yang lebih baik tentang faktor-faktor yang mempengaruhi klasifikasi atau keputusan. Selain itu, Pohon Keputusan juga dapat mengatasi masalah klasifikasi dengan data yang tidak linier atau tidak terstruktur.

Namun, Pohon Keputusan juga memiliki beberapa kelemahan, seperti kecenderungan overfitting (terlalu cocok) terhadap data pelatihan dan kerentanan terhadap perubahan kecil dalam data input. Untuk mengatasi masalah ini, teknik seperti pruning (pemangkasan) dan ensemble learning (pembelajaran gabungan) dapat diterapkan.

Pohon Keputusan digunakan dalam berbagai aplikasi, termasuk pengenalan pola, klasifikasi data, analisis risiko, dan pengambilan keputusan dalam berbagai bidang seperti kedokteran, keuangan, dan industri.

C. P, Np and-Complete Problems

Masalah P, NP, dan NP-Complete adalah konsep penting dalam teori kompleksitas komputasional yang berkaitan dengan kesulitan pemecahan suatu masalah menggunakan algoritma komputer.

- Kelas P (Polynomial Time): Kelas P terdiri dari masalah yang dapat diselesaikan oleh algoritma dengan kompleksitas waktu polinomial. Ini berarti algoritma dapat menyelesaikan masalah dengan efisien dalam skala waktu yang wajar saat ukuran masalahnya meningkat. Contoh masalah dalam kelas P termasuk penjumlahan, pengurangan, perkalian matriks, dan pencarian linier dalam array terurut.
- Kelas NP (Nondeterministic Polynomial Time): Kelas NP terdiri dari masalah yang dapat diverifikasi dalam waktu polinomial. Artinya, jika ada solusi yang diajukan, kebenaran solusi tersebut dapat dengan cepat diverifikasi oleh algoritma dalam waktu polinomial. Namun, belum diketahui apakah masalah-masalah dalam kelas NP juga dapat diselesaikan dalam waktu polinomial oleh algoritma yang efisien. Contoh masalah dalam kelas NP termasuk penyelesaian Sudoku, problem traveling salesman, dan problem satisfiability (SAT).
- Masalah NP-Complete: Masalah NP-Complete adalah subkelas dalam NP yang memiliki sifat khusus. Sebuah masalah dikatakan NP-Complete jika masalah tersebut adalah dalam kelas NP dan setiap masalah dalam NP dapat direduksi (dalam waktu polinomial) menjadi masalah NP-Complete. Dengan kata lain, jika kita dapat menemukan algoritma efisien untuk menyelesaikan masalah NP-Complete, maka kita dapat menyelesaikan semua masalah dalam kelas NP dalam waktu polinomial.

Beberapa contoh masalah NP-Complete termasuk problem knapsack, problem traveling salesman, dan problem satisfiability (SAT).

Penentuan apakah suatu masalah termasuk dalam kelas P, NP, atau NP-Complete merupakan salah satu fokus utama dalam teori kompleksitas komputasional. Buktikan P = NP atau P != NP merupakan salah satu dari tujuan yang belum tercapai dalam teori ini, dan memiliki implikasi besar terhadap kompleksitas pemecahan masalah dalam berbagai bidang seperti optimasi, kriptografi, dan kecerdasan buatan.

D. Challenge of Numerical Algorithms

The challenge of numerical algorithms lies in designing efficient and accurate methods for solving mathematical problems that involve numerical computations. These challenges can arise due to various factors:

1. Accuracy: Numerical algorithms must provide accurate results within acceptable tolerances. Errors can accumulate during calculations, leading to loss of precision or incorrect results. Managing and minimizing numerical errors is crucial in ensuring the reliability of numerical algorithms.
2. Efficiency: Numerical algorithms should be designed to execute computations efficiently, especially for large-scale problems. Optimizing computational complexity, minimizing memory usage, and utilizing parallel computing techniques are essential for achieving high-performance numerical algorithms.
3. Stability: Some numerical algorithms can be sensitive to small changes in the input data, leading to unstable or unreliable results. Stability issues can arise when dealing with ill-conditioned problems, where small perturbations in the input can result in significant changes in the output. Ensuring the stability of numerical algorithms is crucial for obtaining robust and consistent results.
4. Scalability: Numerical algorithms should be able to handle problems of varying sizes and dimensions. As the problem size increases, the algorithm should scale efficiently without excessive computational requirements. Scaling up numerical algorithms to handle large-scale problems can pose significant challenges in terms of memory management, parallelization, and load balancing.
5. Convergence: Many numerical algorithms rely on iterative methods to approximate solutions. Ensuring convergence, i.e., the iterative process reaches a stable and accurate solution, can be challenging. Choosing appropriate termination criteria, handling convergence failures, and improving convergence speed are important considerations in designing reliable numerical algorithms.
6. Adaptability: Numerical algorithms should be adaptable to different problem domains and scenarios. They should be able to handle diverse types of data, account for various boundary conditions, and provide flexible parameterization options. Developing versatile numerical algorithms that can accommodate different problem specifications adds to the complexity of their design.

Addressing these challenges requires a deep understanding of the underlying mathematical principles, careful analysis of algorithmic properties, and continual refinement through theoretical analysis and practical experimentation. It involves a combination of mathematical expertise, computational skills, and domain-specific knowledge to develop robust and efficient numerical algorithms.

BAB XII
COPYING WITH THE LIMITATION OF ALGORITM POWER

A. Backtracking

Backtracking (penelusuran mundur) adalah teknik yang digunakan dalam pemrograman untuk mencari solusi secara sistematis melalui pencarian berbasis pohon atau graf. Teknik ini berguna untuk memecahkan masalah yang melibatkan pemilihan opsi dari sekumpulan opsi yang tersedia.

Pada dasarnya, backtracking bekerja dengan mencoba solusi secara berurutan, memeriksa apakah solusi tersebut memenuhi semua kriteria atau tidak. Jika solusi tersebut memenuhi kriteria, maka solusi diterima. Jika tidak, maka dilakukan langkah mundur (backtrack) untuk mencoba solusi alternatif yang mungkin.

Proses backtracking biasanya melibatkan tiga langkah utama:

1. Pemilihan: Pada setiap langkah, pilihan atau opsi yang tersedia dipertimbangkan. Salah satu opsi dipilih untuk dievaluasi sebagai bagian dari solusi sementara.
2. Verifikasi: Setelah pemilihan opsi, dilakukan verifikasi untuk memeriksa apakah opsi tersebut memenuhi semua kriteria atau tidak. Jika opsi memenuhi kriteria, solusi sementara dipertahankan. Jika tidak memenuhi, dilakukan langkah backtrack untuk mencoba opsi lain.
3. Langkah Mundur (Backtrack): Jika opsi yang dipilih tidak menghasilkan solusi yang memenuhi semua kriteria, maka dilakukan langkah mundur (backtrack) untuk mencoba opsi lain yang belum dieksplorasi. Langkah mundur melibatkan pembatalan pilihan sebelumnya dan mencoba opsi berikutnya.

Proses ini berlanjut hingga semua kemungkinan solusi dieksplorasi atau solusi yang memenuhi semua kriteria ditemukan. Backtracking sangat efektif dalam menemukan solusi yang optimal atau semua solusi yang memenuhi kriteria tertentu.

Contoh masalah yang dapat diselesaikan dengan menggunakan backtracking termasuk permainan seperti Sudoku, pemecahan masalah lintasan (maze), pemetaan warna (graph coloring), dan banyak lagi.

Penting untuk mencatat bahwa penggunaan backtracking dapat memakan waktu yang cukup besar karena algoritma secara eksponensial mempertimbangkan semua kemungkinan solusi. Oleh karena itu, optimisasi dan strategi pemangkasan (pruning) sering kali diperlukan untuk mengurangi ruang pencarian dan meningkatkan efisiensi algoritma backtracking.

B. Branch and Bound

Branch and Bound (Cabang dan Batasan) adalah sebuah metode yang digunakan dalam pemecahan masalah pemrograman matematis untuk mencari solusi optimal dengan membatasi ruang pencarian yang dibutuhkan.

Metode Branch and Bound bekerja dengan membagi masalah menjadi serangkaian submasalah yang lebih kecil dan kemudian melakukan pemecahan secara berulang pada setiap submasalah tersebut. Pada setiap langkah, metode ini menggunakan batasan (bound) pada nilai solusi untuk memutuskan apakah submasalah tersebut harus terus dieksplorasi atau bisa diabaikan.

Langkah utama dalam metode Branch and Bound adalah sebagai berikut:

1. Pemisahan (Branching): Pada awalnya, masalah dibagi menjadi beberapa submasalah yang lebih kecil dengan melakukan pemilihan atau pemisahan pada variabel atau komponen yang relevan. Setiap submasalah menghasilkan cabang atau percabangan baru dalam pohon pencarian.
2. Batasan (Bounding): Setiap submasalah diberi batasan pada nilai solusinya menggunakan batasan atas (upper bound) dan batasan bawah (lower bound). Batasan atas mengacu pada solusi terbaik yang telah ditemukan sejauh ini, sedangkan batasan bawah merupakan batasan yang diperoleh dari pembatasan nilai solusi yang mungkin. Batasan ini digunakan untuk mengurangi ruang pencarian dengan memotong cabang-cabang yang dijamin tidak akan menghasilkan solusi yang lebih baik dari solusi terbaik yang telah ditemukan.
3. Penjelajahan (Exploration): Submasalah-submasalah dieksplorasi secara rekursif dengan mencoba semua kemungkinan solusi dalam setiap cabang. Jika batasan menunjukkan bahwa cabang tersebut tidak mungkin menghasilkan solusi yang lebih baik dari solusi terbaik yang telah ditemukan, maka cabang tersebut diabaikan (pruning).
4. Pembaruan (Update): Saat penjelajahan dilakukan, solusi terbaik yang telah ditemukan diperbarui jika ditemukan solusi yang lebih baik.

Proses ini berlanjut hingga seluruh pohon pencarian dieksplorasi atau tidak ada lagi cabang yang mungkin menghasilkan solusi yang lebih baik dari solusi terbaik yang telah ditemukan.

Metode Branch and Bound berguna dalam mencari solusi optimal dalam berbagai masalah pemrograman matematis seperti penjadwalan, pemetaan jaringan, dan optimasi kombinatorial. Dengan menggunakan batasan dan pemangkasan yang cerdas, metode ini dapat mengurangi ruang pencarian yang harus dieksplorasi, menghemat waktu dan sumber daya komputasi.

C. Algorithms for Solving Nonlinear Problems

Terdapat beberapa algoritma yang umum digunakan untuk menyelesaikan masalah nonlinear, antara lain:

1. Metode Newton: Metode Newton adalah algoritma iteratif optimisasi yang digunakan untuk mencari akar persamaan nonlinear atau meminimalkan/maksimalkan fungsi objektif nonlinear. Metode ini menggunakan aproksimasi linier lokal untuk secara iteratif memperbarui solusi hingga mencapai tingkat akurasi yang diinginkan.
2. Metode Quasi-Newton: Metode Quasi-Newton adalah kelas algoritma optimisasi yang mengaproksimasi matriks Hessian (turunan kedua) dari fungsi objektif tanpa menghitungnya secara eksplisit. Contohnya termasuk metode Broyden-Fletcher-Goldfarb-Shanno (BFGS) dan metode Limited-memory BFGS (L-BFGS). Metode ini efisien untuk masalah dengan skala besar.
3. Gradient Descent: Gradient descent adalah algoritma iteratif optimisasi yang mengandalkan gradien (turunan pertama) dari fungsi objektif untuk mencari minimum/maksimum. Algoritma ini memperbarui solusi ke arah berlawanan dengan gradien hingga mencapai konvergensi.
4. Conjugate Gradient: Conjugate gradient adalah algoritma iteratif optimisasi yang cocok untuk menyelesaikan masalah linear dan nonlinear dengan skala besar. Algoritma ini menggabungkan konsep gradient descent dan arah konjugat untuk mencari solusi optimal secara efisien.
5. Metode Levenberg-Marquardt: Metode Levenberg-Marquardt adalah algoritma iteratif yang umum digunakan untuk masalah kuadrat terkecil nonlinear. Metode ini menyesuaikan parameter dari model nonlinear untuk meminimalkan jumlah residual kuadrat antara prediksi model dan data yang diamati.
6. Algoritma Genetika: Algoritma genetika adalah algoritma pencarian berbasis populasi yang terinspirasi dari proses seleksi alam. Algoritma ini menggunakan kombinasi operator genetika, seperti seleksi, crossover, dan mutasi, untuk secara iteratif mengembangkan populasi solusi kandidat menuju solusi optimal.
7. Simulated Annealing: Simulated annealing adalah algoritma optimisasi probabilitas yang terinspirasi oleh proses annealing dalam metalurgi. Algoritma ini memungkinkan eksplorasi ruang solusi dengan menerima langkah suboptimal dengan probabilitas tertentu yang berkurang seiring waktu. Algoritma ini efektif untuk menemukan optima global dalam masalah nonlinear yang kompleks.

Algoritma-algoritma ini memiliki karakteristik yang berbeda dan cocok untuk masalah nonlinear yang berbeda pula. Pemilihan algoritma tergantung pada sifat masalah, dimensionalitas, batasan, dan tingkat akurasi yang diinginkan. Selain itu, umumnya digunakan kombinasi dari beberapa algoritma atau penyesuaian algoritma untuk struktur masalah tertentu guna meningkatkan konvergensi dan efisiensi.