ALGORITMOS GANANCIOSOS

* Aplicável a problemas de optimização (maximização ou minimização). Em diversos problemas, a optimização local garante também a optimização global, permitindo encontrar a solução óptima de forma eficiente.
* Subestrutura óptima: um problema tem subestrutura óptima se uma solução óptima p/ problema contém soluções óptimas para os seus subproblemas.
* As cinco principais características que suportam essa solução:

1. Um conjunto de candidatos, de onde a solução é criada
2. Uma função de selecção, que escolhe o melhor candidato a ser incluído na solução
3. Uma função de viabilidade, que determina se o candidato poderá ou não fazer parte da solução
4. Uma função objectivo, que atribui um valor a uma solução, ou solução parcial
5. Uma função solução, que determinará se, e quando se terá chegado à solução completa do problema

* Um sistema de moedas diz-se canónico, se o algoritmo ganancioso encontra sempre uma solução ótima para o problema do troco (com stock ilimitado)

ALGORITMOS DE RETROCESSO

* Algoritmos de tentativa e erro
* Contexto geral de aplicação:
  + Explorar um espaço de estados à procura dum estado-objetivo, sem algoritmos eficientes que levem directamente ao objetivo
* Estratégia:
  + Ao chegar a um ponto de escolha (com vários estados seguintes), escolher uma das opções e prosseguir a exploração
  + Chegando a um “beco sem saída”, retroceder até ao ponto de escolha mais próximo com alternativas por explorar, e tentar outra alternativa
* Dado um problema com um conjunto de restrições e/ou uma função objetivo (maximizar ou minimizar), procura-se uma solução que satisfaz as restrições e/ou optimize a função objetivo.
* Caso a solução possa ser construída passo a passo, pode-se representar (pelo menos conceptualmente) o espaço de solução para o problema através de uma árvore de espaço de estados.
  + A raiz da árvore representa o início (0 escolhas)
  + Nós ao nível 1 representam a primeira escolha, nós ao nível 2 representam a segunda escolha, etc…
  + Caminhos da raiz até às folhas representam soluções candidatas
* Algoritmo recursivo de visita em profundidade da árvore de espaço de estados:
  + Explore state/node N:

1. if N is a goal state/node, return “success”
2. for each successor/child C of N,
   1. (optional) set new state
   2. explore state/node C
   3. if exploration was successful, return “success”
   4. (optional) restore previous state 3. return “failure”

* Tempo de execução no pior caso (pesquisa exaustiva do espaço de estados) é determinado pela dimensão do espaço de estados, que muitas vezes é exponencial. Como melhorar?
* *Pruning* - Interromper a pesquisa e retroceder em nós que garantidamente não levam a uma solução viável a.k.a. nós não promissores

DIVIDE AND CONQUER

* *Dividir* o problema em subproblemas que são instâncias mais pequenas do mesmo problema. *Conquistar* os subproblemas resolvendo-os recursivamente; se os subproblemas forem suficientemente pequenos, resolvem-se diretamente. *Combinar* as soluções dos subproblemas para obter a solução do problema original.
* Notas:
  + Subproblemas devem ser disjuntos (senão, usar programação dinâmica).
  + Dividir em subproblemas de dimensão similar para maior eficiência.
  + Para existir divisão, devem existir 2 ou mais chamadas recursivas.
* Exemplos:
  + *Mergesort* - Ordenar 2 subsequências de igual dimensão e juntá-las; T(n) = O(n log n), S(n) = n
  + *Quicksort -* Ordenar elementos menores e maiores que pivot, concatenar ; no pior caso (1 elemento menor, restantes maiores) , T(n) = O(n log n) no melhor caso e no caso médio (escolha aleatória do pivot) , S(n) = 1

PROGRAMAÇÃO DINÂMICA

* Problemas resolúveis recursivamente, mas em que a resolução recursiva directa duplicaria trabalho (resolução repetida do mesmo subproblema).
* Abordagens:

1. Economizar tempo (evitar repetir trabalho), memorizando as soluções parciais dos subproblemas (gastando memória!)
2. Economizar memória, resolvendo subproblemas por ordem que minimiza nº de soluções parciais a memorizar (bottom-up, começando pelos casos-base)

* Caminho mais curto em grafos dirigidos. Se existirem ciclos com peso negativo, o problema não tem solução. Não existindo ciclos com peso negativo, o problema é resolúvel em tempo O(|V||E|) pelo algoritmo de Bellman–Ford.
* O algoritmo de Bellman-Ford é um exemplo de Programação Dinâmica: começa com um vértice inicial e calcula as distâncias a outros vértices que podem ser alcançados com uma aresta, e continua a encontrar caminhos com duas arestas, e assim sucessivamente.
* Em cada iteração i, o algoritmo processa todas as arestas e garante que encontra todos os caminhos mais curtos com até i arestas (e possivelmente alguns mais longos) (invariante do ciclo principal).
* A screenshot of a computer

  Description automatically generated with medium confidenceNo final é executada mais uma iteração para ver se alguma distância pode ser melhorada; se for o caso, significa que há um caminho mais curto com |V| arestas, o que só pode acontecer se existir pelo menos um ciclo de peso negativo.
* *Algoritmo Floyd-Warshall* – com programação dinâmica: Θ(|V|3). Melhor que *Bellman-Ford* se o grafo for denso (|E| ~= (|V|^2) ).

Text

Description automatically generated with medium confidencePROGRAMAÇÃO INTEIRA

onde # ∈ {=, ≤, ≥}.

* A função objetivo é linear e as restrições são lineares:
  + Linear Programming (LP): variáveis reais (contínuas)
  + Integer Programming (IP): variáveis inteiras
  + IP puro: também z e as vars. de desvio em Z (se necessário, multiplicar A, b e c por escalares)
  + Mixed Integer Programming (MIP): algumas variáveis inteiras e outras reais
* Relaxação linear do problema : assume todas as variáveis com valores em R{0+}.
* Se o problema admitir solução ótima então algum dos vértices do espaço de soluções é solução ótima. Os vértices correspondem a soluções básicas admissíveis do sistema de equações.
* Método Simplex: restringe a pesquisa a vértices do espaço de soluções; parte de um vértice; em cada iteração, move-se (por uma aresta) para um vértice adjacente do atual que melhore a função objetivo, se existir.
* Sendo A uma matriz m × n e assumindo que car(A) = m = car(A | b) então se m = n, o sistema Ax = b tem solução única x = A(^-1)b. Se m < n, o sistema tem uma infinidade de soluções.
* Quando alguma das variáveis é 0, a solução diz-se degenerada.
* *Diagram

  Description automatically generatedBranch and Bound –* Quando obtida uma solução não inteira para uma otimização, utilizar a interpretação geométrica e acrescentar restrições de acordo com os valores obtidos (e.g. se x = 8/3 = 2.6(6) -> 2 < x < 3 -> Caso 1: x <=2 ; Caso 2: x>=3)

ALGORITMOS EM GRAFOS

* *Breadth-First Search* – visita s, depois todos os vizinhos de s, a seguir os vizinhos dos vizinhos de s que ainda não tenham sido visitados, e sucessivamente.
* *Depth-First Search* – visita s, depois visita um dos vizinhos de s, a seguir um vizinho desse vizinho de s que ainda não tenha sido visitado, e sucessivamente.

Text

Description automatically generatedText

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence

* A complexidade temporal de *BFS\_Visit* é O(|V| + |A|); a complexidade espacial é O(|V|), se a fila for suportada por um vetor (ou lista ligada com acesso ao primeiro e ao último elemento), além de Θ(|V| + |A|) para G.
* A complexidade temporal de *DFS* é Θ(n + m), com n = |V| e m = |E|
* *Componente conexa de um grafo não dirigido* - é um subgrafo C = (VC, EC) tal que VC é um conjunto máximo de nós acessíveis uns dos outros (máximo significa aqui que não podemos acrescentar mais nós).
* *Componente fortemente conexa* - todos os vértices da componente podem ser acessados ​​de todos os outros vértices
* *Algoritmo de Kosaraju*-Sharir:
  + Usar DFS(G) para ter pilha S com os nós por ordem decrescente de tempo final
  + Para v ∈ G.V fazer cor[v] ← branco;
  + Enquanto (S =/= { }) fazer:
    - v ← POP(S);
    - Se cor[v] = branco ent˜ao DFS Visit(v, G T ) e indica os nós visitados;
* O algoritmo de Kosajaru-Sharir tem complexidade Θ(|V| + |A|), se o grafo for representado por listas de adjacências.
* Uma ordenação topológica de um DAG é uma ordem linear de seus nós em que cada nó vem antes de todos nós para os quais este tenha arestas de saída. Cada DAG tem uma ou mais ordenações topológicas.
* Assumir que, se for necessário escolher adjacente, segue ordem lexicográfica.

Text

Description automatically generatedText, letter

Description automatically generatedGraphical user interface, text, application

Description automatically generatedText, letter

Description automatically generated

* *Problema*: obter um caminho máximo num grafo dirigido acíclico G = (V, A), sendo o comprimento dado pelo número de ramos do caminho.
* Dado G = (V, A, D) em que D((i,j)) ∈ R{0+} é a duração da tarefa (i,j) ∈ V, determinar ES[v], o instante mais próximo em que pode dar início a v (”earliest start”).

Text, letter

Description automatically generated

* Obter a data de conclusão mais afastada para as tarefas com fim no nó v, para todo v, por análise para trás (backward analysis), fixando LF[v] inicialmente como DurMin (a duração mínima do projeto).

Text

Description automatically generated

PROBLEMAS DE FLUXO

* Seja G = (V, E, c) um grafo dirigido, finito, c(u, v) ≥ 0 indica a capacidade do ramo (u, v). A capacidade de um percurso é o mínimo das capacidades dos ramos que constituem o percurso.

Text

Description automatically generated

* Se G for dado por listas de adjacências e a fila de prioridade Q for suportada por uma heap de máximo, tem complexidade temporal O((|V| + |E|) log2 |V|).

Text, letter

Description automatically generated

Text, letter

Description automatically generatedText, letter

Description automatically generatedAtualização do grafo: transversar todas as edges de G e atualizar o valor das capacidades residuais de acordo com a seguinte função:

PROBLEMAS DE EMPARELHAMENTO

* *Grafo Bipartido* – Grafo não dirigido G = (V1 ∪ V2, E), com u ∈ V1 e v ∈ V2, qualquer que seja o ramo (u, v) ∈ E.
* *Emparelhamento* –subconjunto M de E tal que quaisquer ramos em M são incidentes em vértices distintos de V (ou seja, não há dois ramos em M que partilhem algum extremo).
* *Maximal Bipartite Matching Problem –* Dado um grafo bipartido G = (V1 ∪ V2, E) determinar um emparelhamento de cardinal máximo em G.
* A maximum matching is a matching of maximum cardinality, that is, a matching M such that for any matching M’, we have |M| >= |M’|.
* Pode-se reduzir instâncias de ”maximum matching” a fluxo máximo. Basta orientar os ramos, inserir origem s e destino t fictícios, acrescentar ramos de s para os nós de V1 e dos nós de V2 para t, e atribuir capacidades unitárias. Um fluxo máximo na rede corresponde a um emparelhamento de cardinal máximo.

Diagram

Description automatically generated

* Um fluxo máximo na rede corresponde a um emparelhamento de cardinal máximo.
* *The stable marriage problem* – Seja H = {h1, . . . , hn} um conjunto de n homens e seja M = {m1, . . . , mn} um conjunto de n mulheres. Cada elemento ordenou todos os de sexo oposto por ordem de preferência estrita. Determinar um emparelhamento estável M, com |M| = n.
* M é instável se existir um par (h, m) ∉ M tal que h prefere m a M(h) e m prefere h a M(m), sendo M(x) o par de x em M.

*Text

Description automatically generated*

* O algoritmo de Gale-Shapley obtém um emparelhamento estável em tempo O(n^2).
* Foi provado que tal emparelhamento é o melhor emparelhamento estável segundo os homens e, se não for o único emparelhamento estável, é o pior segundo as mulheres.
* *Text

  Description automatically generated*Extensão do Algoritmo de Gale-Shapley para colocar recém licenciados em medicina nos hospitais para o internato. Os hospitais fazem as propostas. O resultado é ótimo do ponto de vista dos hospitais.

ALGORITMOS EM STRINGS

* *Distância de edição* – número de operações necessárias para transformar uma palavra a noutra palavra b.
* Três operações possíveis:
  + Substituir uma letra em a por uma letra em b
  + Apagar letra
  + Inserir letra

Text, letter

Description automatically generated

* *Codificação de Huffman:*

Normally when characters are coded using standard codes like ASCII or the Unicode, each character is represented by a fixed-length codeword of bits. Now, suppose that you knew the frequency of characters in advance. You can use this knowledge to encode strings differently. Frequently occurring characters are encoded using fewer bits and less frequent characters are encoded using more bits.

Prefix Code: Mapping of codewords to characters so that no codeword is a prefix of another.

Diagram, table

Description automatically generated

Diagram, schematic

Description automatically generatedDiagram, schematic

Description automatically generatedThe expected number of bits needed to encode a text with n characters is given in the following formula: TOTAL BITS = Sum of (frequency of char \* number of bits of char codeword), for each char in the text.

Algorithm: We will take two characters x and y, and “merge” them into a single super-character called z, which then replaces x and y in the alphabet. The character z will have a probability equal to the sum of x and y’s probabilities. When the process is completed, we know the code for each char.

Text, letter

Description automatically generated

HARDNESS AND APROXIMATION ALGORITHMS\*

* *NP-complete* problems are problems whose running time is unknown. No polynomial-time algorithm has yet been discovered for an *NP-complete* problem, nor has anyone yet been able to prove that no polynomial-time algorithm can exist for any one of them.
* Three classes of decision problems:
  + The class P consists of those problems that are solvable in polynomial time
  + The class NP consists of those problems that are “verifiable” in polynomial time. Verifiable = if we were somehow given a “certificate” of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem.
  + Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being supplied a certificate.
* A decision problem is a problem whose output is a single boolean value: YES or NO.
* Examples of Decision Problems in NP:
  + CIRCUIT-SAT: Given a combinational circuit built from AND, OR, and NOT gates, is there a way to set the circuit inputs so that the output is 1?
  + EULERIAN CYCLE: Given an undirected graph G = (V, E), does G contain an Eulerian cycle (a cycle that visits all edges exactly once)?
  + HAMILTONEAN CYCLE: Given an undirected graph G = (V, E), does G contain an Hamiltonean cycle (a cycle that visits all nodes exactly once)?
  + TSP (travelling salesperson problem): Given a complete weighted graph G = (V, E, d), with d(e) ∈ Z +, for all e ∈ E, and k ∈ Z +, is there a hamiltonean cycle γ with d(γ) ≤ k? (shortest hamiltonean cycle)

CIRCUITOS DE EULER

* Caminho de Euler: caminho que visita cada aresta exatamente uma vez
* Grafo Euleriano = grafo com um caminho de euler
* Circuito de Euler: caminho de Euler que começa e acaba no mesmo vértice
* Um grafo não dirigido contém um circuito de Euler se

1. é conexo
2. cada vértice tem grau (nº de arestas incidentes) par.

* Um grafo não dirigido contém um caminho de Euler se

1. é conexo
2. todos menos dois vértices têm grau par (estes dois vértices serão os vértices de início e fim do caminho).

* Método baseado em pesquisa em profundidade para encontrar um circuito de Euler
  1. Escolher um vértice qualquer e efetuar uma pesquisa em profundidade a partir desse vértice:
     + Visitar vértice: se tiver arestas incidentes não visitadas, escolher uma dessas arestas, marcá-la como visitada, e visitar vértice adjacente
     + Se o grafo satisfizer as condições necessárias e suficientes, esta pesquisa termina necessariamente no vértice de partida, formando um circuito, embora não necessariamente de Euler
  2. Enquanto existirem arestas por visitor
  3. Procurar o primeiro vértice no caminho (circuito) obtido até ao momento que possua uma aresta não percorrida
  4. Lançar uma sub-pesquisa em profundidade a partir desse vértice (sem voltar a percorrer arestas já percorridas)
  5. Inserir o resultado (circuito) no caminho principal
* Tempo de execução: O(|E| + |V|)

PROBLEMA DO CARTEIRO CHINÊS

* Dado um grafo pesado conexo G=(V,E), encontrar um caminho fechado (i.e., com início e fim no mesmo vértice) de peso mínimo que atravesse cada aresta de G pelo menos uma vez, a um caminho assim chama-se percurso ótimo do carteiro Chinês.p
* A um caminho fechado (não necessariamente de peso mínimo) que atravesse cada aresta pelo menos uma vez chama-se percurso do carteiro.
* Resolúvel em tempo polinomial para grafos dirigidos ou não dirigidos, mas o problema é NP-completo quando se combinam arestas dirigidas com arestas não dirigidas.
* Se o grafo G for Euleriano, a solução é trivial, pois qualquer circuito de Euler é um percurso ótimo do carteiro Chinês.
* Se o grafo G não for Euleriano, pode-se construir um grafo Euleriano G\* duplicando algumas arestas de G, selecionadas por forma a conseguir um grafo Euleriano com peso total mínimo.
* Método para grafos não dirigidos:
  + - 1. Achar todos os vértices de grau ímpar em G. Seja k o nº (par!) destes vértices. Se k=0, fazer G\*=G e saltar para o passo 6.
      2. Achar os caminhos mais curtos e distâncias mínimas entre todos os pares de vértices de grau ímpar em G.
      3. Construir um grafo completo G' com os vértices de grau ímpar de G ligados entre si por arestas de peso igual à distância mínima calculada no passo 2.
      4. Encontrar um emparelhamento perfeito (envolvendo todos os vértices) de peso mínimo em G'.
      5. Para cada par (u, v) no emparelhamento perfeito obtido, adicionar pseudo-arestas (arestas paralelas duplicadas) a G ao longo de um caminho mais curto entre u e v. Seja G\* o grafo resultante.
      6. Achar um circuito de Euler em G\*. Este circuito é um percurso óptimo do carteiro Chinês.
* Método para grafos dirigidos:
  1. No grafo G dado, identificar os vértices com nºs diferentes de arestas a entrar e a sair
  2. Determinar os caminhos mais curtos de vértices que têm défice de saídas para vértices que têm défice de entradas e representar as distâncias respetivas num grafo bipartido G’. Vértices são anotados com multiplicidade (nº de parelhas em que deve participar) igual ao défice absoluto.
  3. Formular problema de emparelhamento óptimo como problema de fluxo máximo de custo mínimo e resolver.
  4. Obter grafo Euleriano G\*, duplicando em G os caminhos mais curtos entre os vértices emparelhados no passo 3, e obter um circuito Euleriano.

STRING MATCHING

* Encontrar todas as ocorrências de um padrão P num texto T. Ocorrências podem ser sobrepostas.
* Soluções possíveis:
* Algoritmo naïve – para cada deslocamento possível, compara desde o início do padrão, O(|P|.|T|)
* Algoritmo baseado em autómato finito:
  + Pré-processamento: gerar autómato finito correspondente ao padrão
  + Permite depois analisar o texto em tempo linear O(|T|), mas tempo e espaço requerido pelo pré-processamento pode ser elevado: O(|P|.|Σ|), em que |Σ| é o tamanho do alfabeto.
* Algoritmo de Knuth-Morris-Pratt:
  + Efetua um pré-processamento do padrão em tempo O(|P|), sem chegar a gerar explicitamente um autómato, seguido de processamento do texto em O(|T|), dando total O(|T|+|P|)
  + função prefixo: π[q] = max {k: 0≤k<q e P[1..k] = P[(q-k+1)..q]}
  + q = 1, …, |P| ; P[i…j] – substring entre índices i e j
  + π[q] é o comprimento do maior prefixo de P que é um sufixo próprio do prefixo de P de comprimento q

A close-up of a document

Description automatically generated with medium confidence A close-up of a document

Description automatically generated with medium confidence

Non-deterministic Polynomial (NP) problemas, são problemas que têm uma solucao (em tempo polinomial) nao deterministica (ou seja tu no meio do algoritmo de resolucao dizes 'epa esta parte vai ser feita em tempo O(1) porque me da jeito que assim seja'). Todos os problemas NP são problemas de decisão (YES or NO).

Existem varios problemas NP que por sua vez variam em dificuldade. Aos problemas que são tão ou mais dificeis que todos os outros problemas NP chamamos de NP-hard (hard = dificil, obvio né, daí o nome). Estes problemas são dificeis de resolver e dificeis de verificar, por isso nao se encaixam na perfeicao com NP, logo nem todos os NP-hard são NP.

A interseçao de problemas NP e NP-hard (problemas tanto ou mais dificeis que todos os outros mas verificaveis em tempo polinomial) é igual a NP-Complete. Daqui vem a noçao de 'reducao', se um dado problema Y for NP-complete e tu conseguires provar que Y é reduzivel para X, entao provas que X é um 'sub-problema' de Y, e portanto é tanto ou mais dificil como qualquer outro problema NP-Complete e, portanto, é NP-hard. Porque é que a reduçao te garante um spot em NP-hard e não em NP-complete? Porque tu só verificaste se ele é tanto ou mais dificil, e nao se é facil/dificil de verificar em tempo polinomial.

Text

Description automatically generatedText

Description automatically generatedTu pegas no X e adaptas o input do X de modo a caber nos moldes de Y (NP-complete), se couber é porque X é (ate certo ponto) igual ao problema Y.