Algorithms

Main source: Prof. Reuven Bar-Yehuda lectures (Technion, 2013) and lecture notes (Technion, 2015).

The last section briefly covers additional materials from TAU, HUJI, Udacity and more.

Summarized by Ido Greenberg in 2018.

Contents

Exploration in graphs 2
Depth-First Search (DFS) 2
Separating nodes and unbreakable components 2
DFS in directed graphs 3
Minimum Spanning Tree (MST) 4
MST in directed graphs 4
Flow in networks 5
Maximum flow problem 5
Flow problem: variations 6
Flow problem: applications 7
Encoding 8
Prefix code 8
Huffman code 9
Further topics 10
List of algorithms and complexities 10
TAU 10
Euler path 11
Dynamic programming 11
Linear programming 11
HUJI 12
Greedy algorithms 12
Computational aspects in linear algebra 13
Udacity: intro to graduate algorithms 13
Median in linear time 13
Spatial search 14

Exploration in graphs

Terminology: **search** / **exploration** / **traversal**.

- BFS w=1 s→f/all
- Dijkstra w>=0 s→f/all
- Ford $w \in R$ s→all
- Floyd $w \in R$ all→all

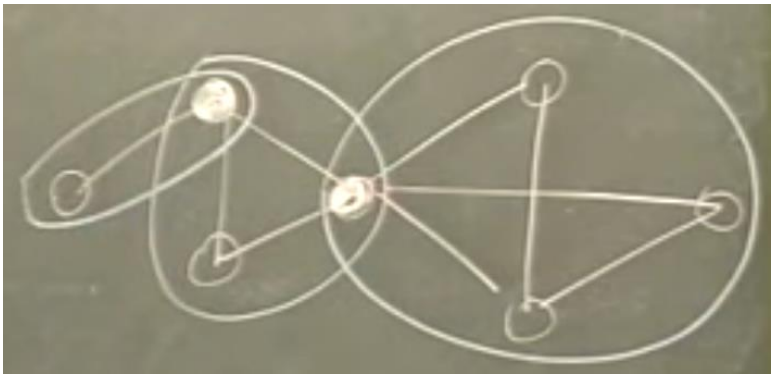
Note: whenever negative weights are permitted, it is assumed that there are no negative cycles.

Depth-First Search (DFS)

- Ancient algorithm of **Tremaux**, originally used for **actual navigation within a maze**.
- It's basically scanning the graph as a maze, and leaving a sign wherever we've already passed.
- In **physical search**, DFS is more natural than BFS, since there's a **cost to going back** towards the root (as done each iteration in BFS).
  - It also doesn't require knowledge of the edges' lengths in advance.
- For every scanned node, we will always eventually go back through the first edge led to this node (implemented "physically" by a unique sign for first arrival to node, and conceptually by holding the frontier in a stack).
  - Consequentially, the whole frontier forms a path from the root to the current node, and the whole algorithm can be implemented as a **recursion**.
- Note that in all the conventional exploration algorithms which hold a frontier and go on from the frontier to the next nodes (with accordance to the data structure of the frontier), **the scanned nodes form a tree** rooted at the starting point.
  - Note: the DFS-tree and BFS-tree of a graph are two different spanning trees.
- The modern implementation of DFS (in terms of stack-frontier) is attributed to **Tarjan & Hopcroft**.
- Time complexity: **O(|E|)**.

Separating nodes and unbreakable components

- **Separating node** := node in a connected graph without which the graph becomes disconnected.
- **Breakable graph** := connected graph with separating node.
- **Unbreakable component** := unbreakable subgraph, such that any subgraph containing it is breakable (or equivalently, unbreakable subgraph which is maximal wrt containing).



- Looking for Separating nodes and unbreakable components:

- Naïve algorithms may remove each node in a turn (O(|V|)) and run a search algorithm to test connectivity (O(|E|)), in time O(|V|*|E|).
- **The local structure of DFS-scan** – in which we never "jump" more than one edge at a time but always go forward to the next node or backward to the previous one – makes the algorithm suitable for separating-nodes detection.
- In particular, one can just run DFS with a twist:
  - For any $x \in V$ keep **L(x) := low-point(x)** := lowest-degree node in the DFS-tree that can be reached from x without going backward on x's path of the tree.
    - Note: "low" here means low-degree, i.e. "closer to root" – which is actually higher in the tree representation.
    - Initialized to L(x):=k(x) when reaching the k-degree x, and updated whenever we find an edge from x to an already-scanned node with k(y)<k(x), and after that when we go backward from x to its parents in the tree (using the frontier stack).
  - A node x is separating if it connects to a child y which cannot reach backward in the tree without going back through x, i.e. **x is separating iff there exists x→y with $\boldsymbol{L(y) \geq k(x)}$**.
  - Each separating node is detected when it is reached in the way backward in the tree, and whatever after it in the tree is the unbreakable component(s) that it separates from the component of the root. Hence, the **unbreakable components can be found by looking down the stack whenever a separating-node is detected**.
- Thus **DFS can solve the separating-nodes & unbreakable components problem in O(|E|)**.

DFS in directed graphs

- Similar to standard DFS, but in this case the root cannot be chosen arbitrarily due to the directionality in the graph. Thus, we begin a standard DFS, and whenever we get stuck we start again from an arbitrary not-yet-scanned node.
- **Topological sort**: given a **Directed Acyclic Graph** (**DAG**) (i.e. no cycles, possibly not connected), sort the nodes such that **a<b iff there's a path from a to b**.
  - Example: **sort clothes** (e.g. shirt, shorts, underpants, socks, shoes) according to which item must be wore before the other. The constraints can be represented by an acyclic graph, and the sorted nodes will be a possible order of wearing the clothes with respect to the constraints.
  - Such sort **can be achieved by DFS over the directed graph if we record the order of the nodes that leave the frontier**, exploiting again the unique structure of DFS (all the descendants of a node die before the ancestor, thus the last to die is the first in the sort).
  - In particular, **cyclicity of a graph can be tested** by applying DFS for topological sort.
- **Well-connected components**: given directed graph, find maximal sub-graphs which are well-connected, i.e. for every x,y there's a path from x to y.
  - In indirect graphs, there's no difference between connected and well-connected.
  - Well-connected components can be seen as equivalence-classes of the relation "there're paths both from x to y and from y to x" in a graph.
  - **Well-connected components can be found by DFS**, using its local nature (requires proof).

- Note: shrinking each well-connected component into a node makes the directed graph acyclic (i.e. DAG). Hence, **every directed graph is a DAG of well-connected components**.

Minimum Spanning Tree (MST)

- **Spanning Tree** := (V,E') such that $E'CE$ and it is a connected graph without cycles (or equivalently, with minimal |E'|).
  - In particular, spanning tree exists only for connected graphs.
- Fundamental lemma: if (X,Y) is a partition of V, and $e \coloneqq argmin\{l((x,y))|x \in X, y \in Y, (x,y) \in E\}$, then there exists MST with e.
  - Proved by replacing [an edge that connects X to Y] with e…
- **Prim** algorithm for MST: each iteration go over all edges from X to V\X, and choose the shortest one (x,y), and add y to X.
  - Correctness: X is always connected, never has any cycles, and eventually X=V. in addition, it only contains edges corresponding to the lemma above.
  - Time complexity: **O(|V|*|E|)**
- An alternative algorithm (also named after Prim?) keeps the distance (i.e. shortest edge) of each $y \in Y$ from X (goes over all edges once in O(|E|), then updates it every iteration in O(1)). This way, every iteration it goes over all nodes rather than all edges, which leads to $\boldsymbol{T = O(|V|^2) + O(|E|)}$.
- **Kruskal algorithm**: since the shortest edge e=(x,y) in E is included in a MST, then it can be added to the basket of E'. After that, x &y can be shrunk into one node, yielding a new graph. Recursively, it's easy to see that in the end (when the whole graph is shrunk to a single point), (V,E') is indeed a MST.
  - Time complexity: wasn't mentioned, but it seems that naïve implementation would yield O(|V|*|E|), whereas sorting the edges in advance would yield O(|V|+|E|log|E|) = **O(|E|log|E|)**.

MST in directed graphs

- **Directed spanning tree** := spanning tree with root $r \in V$, such that there's a path from r to any $x \in V$.
- Note: for any $x \neq v$, the input degree of x is $d_{in}(x) = 1$.
- Algorithm:
  - $\forall x \neq r$: choose the shortest input edge of x ("critical edge").
  - The critical graph's connected components are a tree with root r, and simple cycles (since each node has one input edge except for r).
  - Lemma (proved by the lecturer Bar-Yehuda along with 2 friends): for any such cycle C, there exists a MST with |C|-1 edges of C.
    - The proof looks at a MST and studies the possible statuses of any critical edge which is not in the tree.
  - Following the lemma, one just needs to find the cheapest way to "break" the cycle, which is exactly the node x whose second-best input edge is closest (in length) to the critical edge.

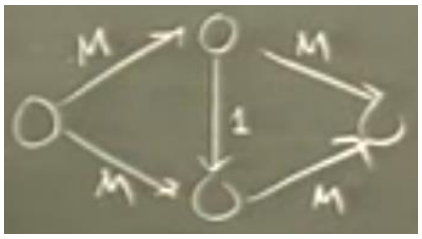
- Every broken cycle becomes a part of either a larger cycle or the tree with the root r – until eventually it all becomes one spanning tree which is minimal.
  - Note that this can be phrased as a **recursion**, by shrinking each cycle into a node.

Flow in networks

Maximum flow problem

- Flow problem: given (positive) **capacity** for each edge, **find the strongest possible flow from s to t**.
  - Example: **maximizing cars traversal from A to B**.
  - Constraints on the solution:
    - **Channel capacity**: flow through an edge doesn't exceed its capacity.
    - **Flow-conservation**: the flow into a node equals the flow out from the node.
- Formally we denote:
  - Capacity of an edge: $c(e)$
  - Flow in an edge: $f(e) \leq c(e)$
  - Flow into node x_0: $\sum f(x \to x_0) - \sum f(x_0 \to x)$ (=0 for any $x \neq s,t$)
  - The flow from s to t: $F \coloneqq \sum f(s \to x) - \sum f(x \to s) = \sum f(x \to t) - \sum f(t \to x)$
- **Cut of graph**: the set of edges between two complement sub-graphs S and V\S.
  - In context of the flow problem, we demand $s \in S,\ t \in \bar{S} = V \backslash S$.
  - The flow through any cut equals the flow from s to f: $\boldsymbol{F = \sum_{e \in (S:\bar{S})} f(e) - \sum_{e \in (\bar{S}:S)} f(e)}$
  - We denote $c(S:\bar{S}) \coloneqq \sum_{e \in (S:\bar{S})} c(e)$.
  - For any cut, $F \leq C(S:\bar{S})$.
    - Hence, **if $\boldsymbol{F = C(S:\bar{S})}$ then the flow F is maximal and the cut S is minimal, and also saturated ("bottle neck")**.
- **Ford-Fulkerson** algorithm for flow maximization:
  - **While there exists a path** from s to t:
    - **Add a path** to the flow.
    - Choose the flow of the path to be the highest possible, i.e. min(c(e)) in the path.
    - **Reduce the flow from the capacity of each edge** in the path, **and add it to a fictive capacity of the flow in the opposite direction**.
      - The graph with the remaining capacities is called **residual-capacities-graph**.
      - Note: the bottleneck of the path wastes all its capacity.
  - The abstract negative capacity allows multiple paths to additively use the same edge in possibly different directions. Equivalently, it allows cancellation of the greedy exploitations of the edges. Hence, **the bi-directional-capacity concept essentially allows the factorization of the flow to additive paths**.
  - When the algorithm stops, **the sub-graph which is still connected to s defines a minimum cut**, since its residual capacity is 0.
- **Variants of F&F**: most flow-maximization algorithms follow F&F approach of incrementally adding flow-paths until no additional flow is possible.
  - **Naive implementation** may be prone to long running time or even failure.

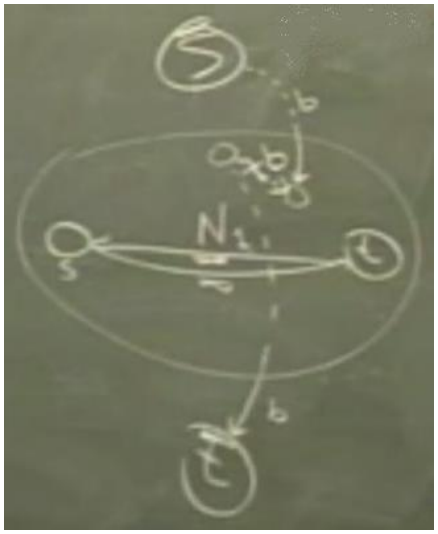
- Pathological example involving irrational capacities causes the algorithm to possibly run infinitely (depending on the order of the edges in the graph search), even without converging to F.
- Even conventional example may cause exponential running time ($\boldsymbol{O(exp(|E|))}$):



- **Edmonds-Karp**: F&F with BFS prevents the exponential running time, but is still too complex ($\boldsymbol{O(|V|^3|E|)}$).
- **Dinic**: similar to E&K, but the capacities are updated only after the currently-maximal flow is found. Implemented with DFS and its layers-graph, this allows addition of multiple paths every single DFS, which reduces the time complexity to $\boldsymbol{O(|V|^2|E|)}$ (or even $\boldsymbol{O(|V|log(|V|)|E|)}$ using smarter repeating searches).
- Many other implementations of F&F approach are currently available with other running times (e.g. $\boldsymbol{O(|V|\text{^}3)}$).

Flow problem: variations

- **Flow with lower bounds**:
  - In addition to edge capacity $f(e) \leq c(e)$, now we also require $b(e) \leq f(e)$.
  - Cut-capacity generalization – reduce the opposite-direction lower bound: $c(S) \coloneqq \sum_{e\in(S,\bar{S})} c(e) - \sum_{e\in(\bar{S},S)} b(e)$.
  - F&F incremental approach can be easily generalized, except for the need for a valid initial flow.
    - Note: given $\{b(e)\}$, $f(e) \equiv 0$ is no longer valid, neither $f(e) \coloneqq b(e)$ (which does not necessarily respect conservation in nodes).
    - In particular, valid flow does not necessarily exist (e.g. $f(e_1) \in [1,2]$ and $f(e_2) \in [3,4]$ in a row are not mutually-satisfiable).
  - To find a valid initial flow, one can transform G to a different graph by:
    - Change from lower-bound edges to constant-flow edges, by replacing each e_{xy} with $c(e'_{xy}) \coloneqq c(e_{xy}) - b(e_{xy})$ and $\boldsymbol{f(e'_{yx})} \coloneqq b(e_{xy})$.
    - Change from constant-flow edges to the original problem by adding s' and t', and replacing e'_{yx} with $c(e_{s'x}) \coloneqq c(e_{yt'}) \coloneqq f(e'_{yx})$.
      - Note: to prevent excess constraint over the original s & t, we also add $c(e_{st}) \coloneqq c(e_{ts}) \coloneqq \infty$.
      - The new graph is demonstrated below (b denotes a removed constant-flow edge, which is replaced by edges from $\bar{s}$ and to $\bar{t}$).



- Note: the flow in the new graph is bounded by the capacities $F_{s't'} \leq \sum c(e_{s'x}) = \sum c(e_{yt'}) = \sum_{original\ graph} b(e)$.
- Theorem: there exists a valid flow in G iff the maximum flow F' in the new graph reaches the boundary $\sum_{original\ graph} b(e)$.
  - Explanation: in such maximum flow, for each $b(e_{xy})$ in G we have $f(e_{s'x}) = f(e_{yt'}) = b(e_{xy})$ in the new graph, which corresponds to $f(e'_{xy}) = b(e_{xy})$ in the constant-flow-edges graph, which corresponds to $b \leq f(e_{xy}) \leq c$ in the lower-bound-edges graph G.
- By applying F&F approach on the new graph and finding F', we find whether there is a valid flow in G and what it is.

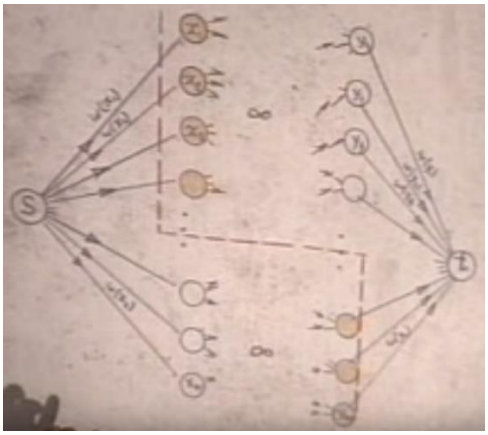
- **Summary** of how to find a valid initial flow:
  - **G with lower bounds → G' with constant flows → G'' with upper bounds only → apply F&F → maximum flow in G'' is a valid flow in G (if existing)**.

- **Minimum flow**: just look for the maximum flow from t to s.
- **Multiple sources** $\{s_i\}$ and targets $\{t_i\}$: just add s_0 with $c(e_{s_0 s_i}) = \infty$, and t_0 with $c(e_{t_i t_0}) = \infty$.

Flow problem: applications

- **Maximum bipartite matching**:
  - **Bipartite graph**: (X,Y,E) where $E \subseteq X \times Y$.
  - **Pairs-match** in graph: $M \subseteq E$ such that each node appears in M at most once.
    - E.g. relationship matching.
  - **Maximum matching problem**: given a bipartite graph of possible matches between X and Y, find a maximum match (i.e. as many pairs as possible under the matching constraints, represented by the edges).
  - **Solution**: add s with $\{c(s,x) \coloneqq 1\}_{x \in X}$ and t with $\{c(y,t) \coloneqq 1\}_{y \in Y}$, and find the maximum flow s→t.
- **Hall theorem**:
  - Full match of X to Y: a match in which each x has a matched y.
  - Hall condition: $\forall A \subseteq X$: $|\Gamma(A)| \geq |A|$ (where $\Gamma(A)$ is all y's connected to some $x \in A$)

  - Hall theorem: (X,Y,E) satisfies hall condition ↔ there's a full match of X to Y.
    - ←: trivial since the there're |A| y's matched to A, and they're all in $\Gamma(A)$.
    - →: in the absence of a full match, one can use the corresponding extended bipartite graph (as built above) and a certain cut of that graph to find a contradiction to hall condition.
- **Maximum independent set** problem:
  - **Independent Set (IS)**: $U \subseteq V$ such that there are no edges in U.
  - **Vertex Cover (VC)**: $\widetilde{U} \subseteq V$ such that every edge has a node in $\widetilde{U}$.
  - Maximum IS U is the complementary of a minimal VC $\widetilde{U} = U^C$.
  - Maximum IS in a general graph G is NP-hard.
  - In a bipartite graph it is solvable by the same extension as built above (and with $c(e_{xy}) \coloneqq \infty$), and finding the maximum flow – or equivalently, the minimum cut (V1,V2). The independent set then is $(V_1 \cap X) \cup (V_2 \cap Y)$ (due to the saturation of the minimum cut, there can't be an edge from V_1 to V_2).



Encoding

Prefix code

- Code / Language: a set of words in some alphabet $\sigma = |\Sigma|$.
- Message: a sequence of words in a certain code.
- **Uniquely-decodable code**: code in which every message has a unique partition to words.
- **Prefix-code**: code with the **prefix property** – no word is a prefix of another word.
  - Note: in popular languages, each word practically ends with a space, which satisfies the prefix condition (e.g. "I" is a prefix of "In", but "I " isn't a prefix of "In ").
- **Tree representation**:
  - A language can be represented by a tree, where each edge is a letter, and each word corresponds to a path.
  - Given the prefix property, each word ends in a leaf in the corresponding tree.
  - On the opposite direction: any tree can be converted to a prefix code, as long as the splits are not larger than the alphabet, and the weights correspond to valid probabilities.
- **Theorem**: given a set of **lengths** $\{\boldsymbol{l_i}\}$ **with** $\sum \boldsymbol{\sigma^{-l}} \leq \mathbf{1}$, **there exists a valid prefix-code** whose words lengths are $\{l_i\}$.

- One possible proof builds the language tree explicitly, using DFS approach on the whole alphabet with lexicographic-order-exploration.

Huffman code

- **Optimal code problem**: given a language L_0 of words with known frequencies P_i, find a new code L (over a possibly-different alphabet) with **minimum expected word length**.
  - Simplified application: encode the letters of L_0 rather than its words.
- Naïve (uniform) code: use $l = \log_\sigma(|L_0|)$ σ-digits (e.g. 3 binary-digits to encode up to 8 words/letters).
- Claim: in the binary case $\sigma = 2$, there exists optimal code whose representing binary tree:
  - Is full, i.e. each non-leaf node has 2 children.
    - Trivial, otherwise just cut the spare edge=letter which doesn't add information to the word.
    - Hence, in particular, there's an even number of leaves in the lowest level.
  - Where the 2 least-frequent letters are siblings.
    - If they're not in the lowest level, then the mean length can be reduced by replacing them with someone of the lowest level; and if they are, then they can be moved to the same parent without any cost.
- Following the claim, **Huffman code recursively builds a tree by replacing the 2 currently least-frequent letters with a new letter** whose frequency is their sum (and that will be their common parent in the tree).

Further topics

List of algorithms and complexities

סיבוכיות של אלגוריתמים

בעיה	אלגוריתם	סיבוכיות	הערות
מיון טופולוגי	מבוסס מחיקת מקורות	$O(V+E)$	בלבד DAG
מסלולים קצרים ביותר	BFS	$O(V+E)$	
	DFS	$O(V+E)$	
גרף הרכיבים קשירים היטב	מבוסס DFS	$O(V+E)$	
עץ פורש מינימום	פרים	מבוסס מערך $O(V^2)$ מבוסס ערימה $O(ElogV)$	
עץ פורש מינימום	קרוסקל	מבוסס UF $O(ElogV)$	
מסלולים קלים ביותר ממקור יחיד	בלמן פורד	$O(VE)$	מוצא מעגלים שליליים אם יש
מסלולים קלים ביותר ממקור יחיד	דיקסטרה	$O(ElogV)$	קשתות במשקל אי שלילי בלבד
מסלולים קלים ביותר בין כל זוג	ג'ונסון	$O(VElogV)$	מוצא מעגלים שליליים אם יש
מסלולים קלים ביותר בין כל זוג	פלויד-וורשל	$O(V^3)$	
מספר מקסימלי של קטעים זרים	חמדן	$O(nlogn)$	
צביעת גרף ב-d_1+1 צבעים	חמדן	$O(V+E)$	
בניית עץ Huffman	רקורסיבי	$O(nlogn)$	
קבוצה בלתי תלויה של קטעים בעלת משקל מקסימלי	מבוסס תכנות דינמי	$O(nlogn)$	
Knapsack	מבוסס תכנות דינמי	$O(nW)$	
זרימת מקסימום	כל אלגוריתם המתאים לאלגוריתם הגנרי של פורד פלקרסון	$O(Ef^*)$	f^* הוא ערך הזרימה המקסימלית ברשת. התכנסות מובטחת רק במקרה של זרימה בשלמים.
זרימת מקסימום	אדמונדס-קרפ	$O(VE^2)$	
שידוך מקסימום בגרף דו-צדדי	מבוסס זרימה	$O(VE)$	
חישוב קונבולוציה	טרנספורם פורייה המהיר	$O(nlogn)$	

TAU

שנה"ל תשע"ח

מספר הקורס	שם הקורס
0368-2160	**אלגוריתמים** **Algorithms**

סילבוס מקוצר

אלגוריתמים יעילים בגרפים: סריקה של גרף, מעגל אוילר, עץ פורש מינימלי, מסלולים קצרים ביותר, זרימה ברשתות; טכניקות אלגוריתמיות נוספות לרבות תכנות דינאמי ותכנות לינארי.

Euler path

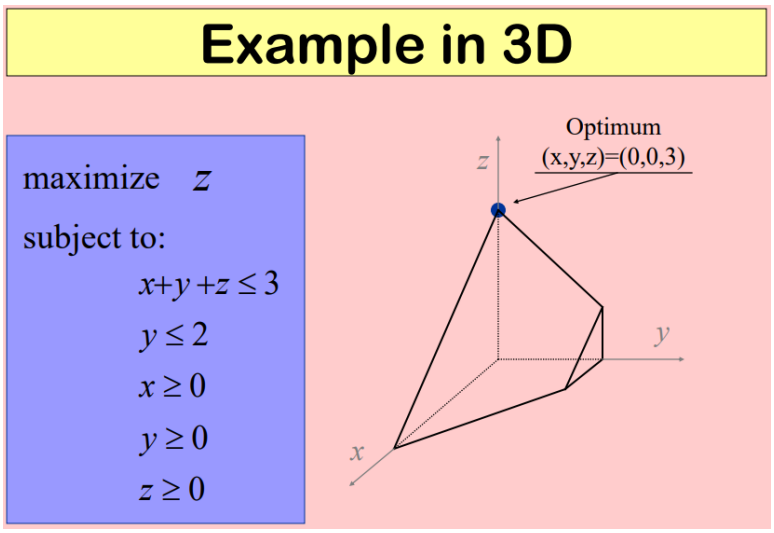
- Euler path := path that passes exactly once through each node.
- Euler cycle := Euler path which is a cycle.
- Theorem: there exists an Euler cycle in a connected graph iff all its node's degrees are even.

Dynamic programming

- Put problems in **recursive** terms, and solve the recursion **bottom-up rather than top-down**, so that **each sub-problem needs to be solved only once**.
- Extremely useful in cases where the same sub-problems occur repeatedly.
- Examples:
  - **Fibonacci**: for n=5, classic recursion would compute f(4) once, f(3) twice, f(2) 3 times and f(1) twice – rather than just once each.
  - **Floyd-Warshall algorithm for shortest distances in a graph** (with real weights, without negative cycles): for every n<|V|, find the shortest path between each pair of nodes, using only paths of up to n edges – by assuming that the problem is already solved for any k<n.
  - **Maximum subset of disjoint intervals**: instead of trying all subsets exponentially – the intervals can be sorted by finish time, then the problem can be solved for each sub-problem of intervals $\{I_i\}_{i=1}^{n}$ (n<N), based on the previous solutions of $\{I_i\}_{i=1}^{k}$ for each k<n.
  - **Matrices-chain minimum multiplication**: find the order of matrices multiplication (e.g. ((AB)(CD)) or (A(B(CD)))) with minimum number of scalar multiplications: there are N-1 matrix-multiplications with $(N-1)!$ Possible orders of the multiplications. Instead of trying all $(N-1)!$, one can look for the best order in each threesome, then in each quartette, etc., which turns out to be just $O(N^3)$.
  - **Sequence alignment**: given two sequences (strings) $x_1 \dots x_m, y_1 \dots y_n$, and few valid operations to modify them (each associated to some cost), find the cheapest way to change one string into the other ("shortest distance" between them): can be solved recursively using the solutions of $(m-1,n-1), (m-1,n), (m,n-1)$.

Linear programming

- **Linear Programming** (**LP**) problem: **maximize $\boldsymbol{c^T x}$ subject to $\boldsymbol{Ax \leq b}$ and $\boldsymbol{x \geq 0}$**.
  - Goal function and constraints are linear.
  - All scalars are supposed to be real.
    - For $x \in N^n$ (integer solution) the problem is called **Integer-LP** (**ILP**) and is NP-hard.
  - Constraints may leave null set of valid solutions, or leave the goal function unbounded.
- **Examples**:
  - Maximize the profits of bank's loans under regulation constraints (limitations on different types of loans).
  - Minimize calories of food while consuming essential basic food types.
  - Minimize number of shifts workers under shifts requirements.
  - Maximum flow problem in a graph can be represented as LP problem.
- Geometrical interpretation of the linear constraints:



  - Note: due to linearity, the solution must be on a vertex (or at least an equal-cost face).
- **Standardization** of LP problems:
  - Equalities in constraints (ax=b) can be implemented through 2 weak inequalities ($\leq$ & $\geq$).
  - Minimum goal function can be implemented by inverting c.
  - Negative values ($x \in R$) can be permitted through $x = x' - x''$ with $x', x'' \geq 0$.
- **Slack form**:
  - $\boldsymbol{Ax = b}$ constraints instead of $\leq$.
  - Achieved using new auxiliary variables with the constraints $x_{n+i} = b - Ax$.
  - Slack form of a LP problem is not unique.
  - Each slack form can be associated with a simple valid assignment of values to x, which corresponds to a vertex in the geometric representation of the constraints.
- **Simplex algorithm**:
  - Convert problem into slack form.
  - Every step ("pivot step"): transform to slightly simpler slack form (which also turns out to correspond to a better assignment of x in terms of cost).
  - For n variables and m constraints, we get $O(mn \cdot 2^{m+n})$, but in practice it was found empirically that there're significantly fewer iterations than 2^{m+n}.
- Other algorithms: ellipsoid, interior point – both are polynomial.
- Sources: Wikipedia, TAU.

HUJI

סילבוס

67504 - אלגוריתמים

בנייה וניתוח של אלגוריתמים יעילים עבור בעיות חישוב שימושיות. הנושאים יכללו אלגוריתמים חמדניים, תכנון דינמי, קירובים לבעיות חישוב קשות, זרימה ברשתות, טרנספורם פורייה המהיר ושימושיו, אלגוריתמים בתורת המספרים ויסודות בתורת ההצפנה, היבטים חישוביים של אלגברה לינארית.

Greedy algorithms

- Looks quite shallow. In some summaries of the Technion there were some rough explanation, several examples, and a note about the fact that greedy may or may not be optimal.
- Example: Minimal Spanning Trees (solved above).

- Example: Maximum Disjoint Intervals (solved by sorting the ends of the intervals and going over all of them once, inserting any interval which doesn't cause conflict with the previous added one).
- No formal definition of greedy was suggested, so I have to define it myself:
  - **A greedy algorithm for problem P wrt goal function F**, is an **iterative algorithm such that each iteration generates an explicit solution y_i, and** $\forall i$: $\boldsymbol{F(y_{i-1}) \leq F(y_i)}$.
  - A greedy algorithm with N iterations (where N is possibly dependent on the input) is **globally optimal** if $\forall y$: $F(y) \leq F(y_N)$.

Computational aspects in linear algebra

- Essentially, it seems to deal with **efficient computations involving matrices**.
- Classical examples are **matrices multiplication**, **inverting a matrix**, and **solving a linear system**.
- Another example was mentioned above – see ***matrices-chain minimum multiplication***.
- Another example is **polynomials multiplication**:
  - Naively, $\forall i$: $c_i \coloneqq \sum_j a_j b_{i-j}$ (the multiplication polynomial coefficients are convolution of the original polynomials coefficients), which takes $\boldsymbol{O(n^2)}$.
  - It turns out that FFT on polynomial coefficients ($O(nlogn)$) yields a representation of the polynomial in terms of its values in the unit roots $\left\{e^{\frac{ik}{n}}\right\}_k$, which can be multiplied in $O(n)$ and transformed back with inverse FFT – totally $\boldsymbol{O(nlogn)}$.
  - See more in pages 55-60 in the lecture notes of 2015.
- More examples are available here.

Udacity: intro to graduate algorithms

Median in linear time

- Here is a commented python-like pseudo code which totally ignores rounding issues:

```
def quantile(A, K):
	'''
	Find the K'th element of a non-sorted n-sized array in time complexity O(n).
	Based on quick-sort outline with:
		1. No need to calculate both sides – only the one with the K'th element.
		2. Pivot is smartly chosen to be in quantiles 0.25-0.75
		   (hence removes at least 25% of the entries each iteration).
	Time complexity is linear as derived from T(n) = T(0.75*n) + T(n/5) + O(n).
	'''
	if len(A)<=1: return A[0]                      # O(1)
	S = fifths_medians(A)                          # O(n)
	P = quantile(S,len(S)//2)                      # T(n/5)
	Al = [x for x in A if x<P]                     # O(n)
	Ar = [x for x in A if x>P]                     # O(n)
	if len(Al)>K: return quantile(Al,K)            # <=T(3/4*n)
	if len(Ar)>K: return quantile(Ar,K-len(Al)-1) # <=T(3/4*n)
	return P

def fifths_medians(A):
	'''
	Split A to groups of 5 elements and find the median of each group.
	Input: n-sized array A.
	Output: n/5-sized array of medians of fifths of A.
	Time: O(n)
	'''
	return [median(A[range(i,i+5)]) for i in range(len(A)/5)]
```

Spatial search

- Goal: organize n points in R^d such that the following queries will be possible in logarithmic time:
  - Return KNN of a new point x.
  - Return all points in a given range (rectangle) or radius (circle).
- Two common spatial tree structures (AKA spatial index):
  - **R tree**: recursively divide the space into m rectangles containing (nearly) equal number of data points.
  - **K-d tree**: recursively find a hyperplane (usually bounded to be aligned to the axes) which goes through a single point and splits the data into two (nearly) equal subsets, preferably with large distance between the hyperplane and the other data points.
- Usage example – 1-nearest-neighbor query in k-d tree:
  - Given new x, go through the tree with x. At each node:
    - Check if it's closer to x than the previously closest node and update if need to.
    - Choose which way to go on according to the location of x wrt the corresponding hyperplane.
    - Verify that the hyperplane is far from x more than the currently closest node – otherwise the other side of the hyperplane needs to be explored as well.
- Reference: https://blog.mapbox.com/a-dive-into-spatial-search-algorithms-ebd0c5e39d2a