Algorithms: COMP3121/9101

Aleks Ignjatović

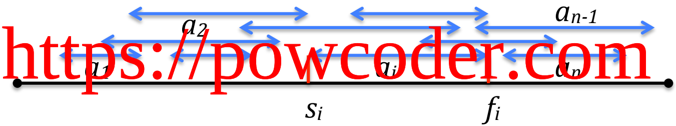
School of Computer Science and Engineering
University of New South Wales

6. THE GREEDY METHOD

Activity selection problem.

Instance: A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

Task: Find a *maximum size* subset of compatible activities.



Attempt 1: always choose the shortest activity which does not conflict with the previously chosen activities, remove the conflicting activities and repeat?

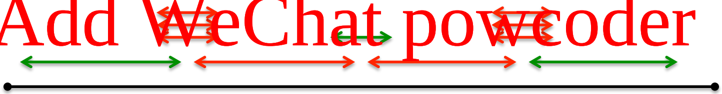
- The above figure shows this does not work... (chosen activities in green, conflicting in red)

Activity selection problem.

Instance: A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times f_i. No two activities can take place simultaneously.

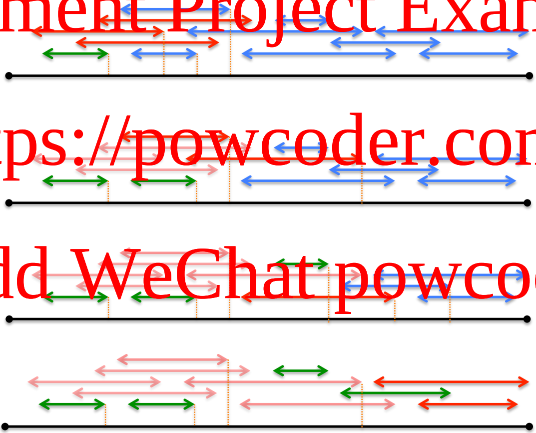
Task: Find a *maximum size* subset of compatible activities.

- **Attempt 2:** Maybe we should always choose an activity which conflicts with the fewest possible number of the remaining activities? It may appear that in this way we minimally restrict our next choice....



- As appealing this idea is, the above figure shows this again does not work ...

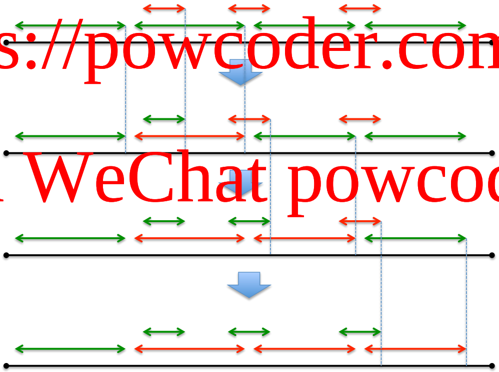
The correct solution: Among the activities which do not conflict with the previously chosen activities always chose the one with the earliest end time.



Proving optimality of a greedy solution

Show that any optimal solution can be transformed into the greedy solution with equal number of activities:

- Find the first place where the chosen activity violates the greedy choice.
- Show that replacing that activity with the greedy choice produces a non conflicting selection with the same number of activities.
- Continue in this manner till you "morph" your optimal solution into the greedy solution, thus proving the greedy solution is also optimal.



- What is the time complexity of the algorithm?
- We represent activities by ordered pairs of their starting and their finishing times and sort them in an increasing order of their finishing time (the second coordinate).

- This takes $O(n \log n)$ time.
- We go through such a sorted list in order, looking for the first activity whose starting time is after the finishing time of the last taken activity.

- Every activity is handled only once, so this part of the algorithm takes $O(n)$ time.
- Thus, the algorithm runs in total time $O(n \log n)$.

Activity selection problem II

- **Instance:** A list of activities a_i, $(1 \leq i \leq n)$ with starting times s_i and finishing times $f_i = s_i + d$; thus, all activities are of the same duration. No two activities can take place simultaneously.

- **Task:** Find a subset of compatible activities of *maximal total duration.*

- **Solution:** Since all activities are of the same duration, this is equivalent to finding a selection with a largest number of non conflicting activities, i.e., the previous problem.

- **Question:** What happens if the activities are not all of the same duration?

- Greedy strategy no longer works - we will need a more sophisticated technique.

More practice problems for the Greedy Method

- Along the long, straight road from Loololong to Goolagong houses are scattered quite sparsely, sometimes with long gaps between two consecutive houses. Telstra must provide mobile phone service to people who live alongside the road, and the range of Telstra's cell base station is 5km. Design an algorithm for placing the minimal number of base stations alongside the road, that is sufficient to cover all houses.

- Once again, along the long, straight road from Loololong (on the West) to Goolagong (on the East) houses are scattered quite sparsely, sometimes with long gaps between two consecutive houses. Telstra must provide mobile phone service to people who live alongside the road, and the range of Telstras cell base station is 5km.

- One of Telstra's engineer started with the house closest to Loololong and put a tower 5km away to the East. He then found the westmost house not already in the range of the tower and placed another tower 5 km to the East of it and continued in this way till he reached Goolagong.

- His junior associate did exactly the same but starting from the East and moving westwards and claimed that his method required fewer towers.

- Is there a placement of houses for which the associate is right?

More practice problems for the Greedy Method

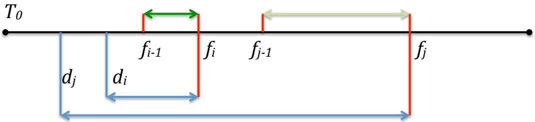
Minimising job lateness

- **Instance:** A start time T_0 and a list of jobs a_i, $(1 \le i \le n)$, with duration times t_i and deadlines d_i. Only one job can be performed at any time; all jobs have to be completed. If a job a_i is completed at a finishing time $f_i > d_i$ then we say that it has incurred lateness $l_i = f_i - d_i$.

- **Task:** Schedule all the jobs so that the lateness of the job with the largest lateness is minimised.

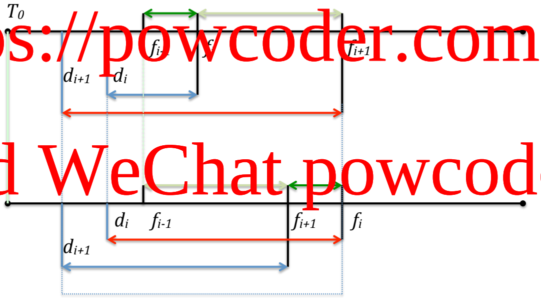
- **Solution:** Ignore job durations and schedule jobs in the increasing order of deadlines.

- **Optimality:** Consider any optimal solution. We say that jobs a_i and jobs a_j form an inversion if job a_i is scheduled before job a_j but $d_j < d_i$.



Minimising job lateness

- We will show that there exists a scheduling without inversions which is also optimal.
- Recall the BubbleSort: if we manage to eliminate all inversions between adjacent jobs, eventually all the inversions will be eliminated.



- Note that swapping adjacent inverted jobs reduces the larger lateness!

Tape storage

- **Instance:** A list of n files f_i of lengths l_i which have to be stored on a tape. Each file is equally likely to be needed. To retrieve a file, one must start from the beginning of the tape and scan it until the file is found and read.

- **Task:** Order the files on the tape so that the average (expected) retrieval time is minimised.

- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$l_1 + (l_1 + l_2) + (l_1 + l_2 + l_3) + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n) = \\ nl_1 + (n-1)l_2 + (n-2)l_3 + \ldots + 2l_{n-1} + l_n$$

- This is minimised if $l_1 \leq l_2 \leq l_3 \leq \ldots \leq l_n$.

Tape storage II

- **Instance:** A list of n files f_i of lengths l_i and probabilities to be needed p_i, $\sum_{i=1}^{n} p_i = 1$, which have to be stored on a tape. To retrieve a file, one must start from the beginning of the tape and scan it until the file is fund and read.
- **Task:** Order the files on the tape so that the **expected** retrieval time is minimised.

- **Solution:** If the files are stored in order $l_1, l_2, \ldots l_n$, then the expected time is proportional to

$$p_1 l_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n$$

- We now show that this is minimised if the files are ordered in a decreasing order of values of the ratio p_i / l_i.

- Let us see what happens if we swap to adjacent files f_k and f_{k+1}.
- The expected time before the swap and after the swap are, respectively,

$$\begin{aligned} E =& l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k)p_k \\ &+ (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_k + l_{k+1})p_{k+1} + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n \end{aligned}$$

and

$$\begin{aligned} E' =& l_1 p_1 + (l_1 + l_2)p_2 + (l_1 + l_2 + l_3)p_3 + \ldots + (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1})p_{k+1} \\ &+ (l_1 + l_2 + l_3 + \ldots + l_{k-1} + l_{k+1} + l_k)p_k + \ldots + (l_1 + l_2 + l_3 + \ldots + l_n)p_n \end{aligned}$$

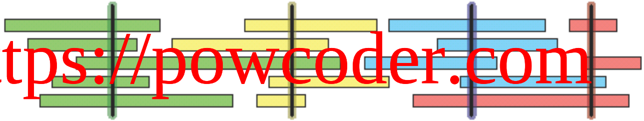
- Thus, $E - E' = l_k p_{k+1} - l_{k+1} p_k$, which is positive just in case $l_k p_{k+1} > l_{k+1} p_k$, i.e., if $p_k/l_k < p_{k+1}/l_{k+1}$.

- Consequently, $E > E'$ if and only if $p_k/l_k < p_{k+1}/l_{k+1}$, which means that the swap decreases the expected time just in case $p_k/l_k < p_{k+1}/l_{k+1}$, i.e., if there is an inversion: a file f_{k+1} with a larger ratio p_{k+1}/l_{k+1} has been put after a file f_k with a smaller ratio p_k/l_k.

- For as long as there are inversions there will be inversions of consecutive files and swapping will reduce the expected time. Consequently, the optimal solution is the one with no inversions.

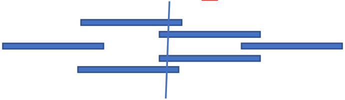
More practice problems for the Greedy Method

- Let X be a set of n intervals on the real line. We say that a set P of points stabs X if every interval in X contains at least one point in P; see the figure below. Describe and analyse an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.



A set of intervals stabbed by four points (shown here as vertical segments)

- Is it a good idea to stab the largest possible number of intervals?



- Hint: the interval which ends the earliest has to be stabbed.
- What is the best place to stab it?

0-1 knapsack problem

- **Instance:** A list of weights w_i and values v_i for **discrete items** a_i, $1 \leq i \leq n$, and a maximal weight limit W of your knapsack.

- **Task:** Find a subset S of all items available such that its weight does not exceed W and its value is maximal.

- Can we always choose the item with the highest value per unit weight?
- Assume there are just three items with weights and values: (10kg, \$60), (20kg, \$100), (30kg, \$120) and a knapsack of capacity W = 50kg.

- Greedy would choose (10kg, \$60) and (20kg, \$100), while the optimal solution is to take (20kg, \$100) and (30kg, \$120)!
- So when does the Greedy Strategy work??
- Unfortunately there is no easy rule...

- Assume you are given n sorted arrays of different sizes.
- You are allowed to merge any two arrays into a single new sorted array and proceed in this manner until only one array is left.

- Design an algorithm that achieves this task and uses minimal total number of moves of elements of the arrays and give an informal justification why your algorithm is optimal.

- This problem is somewhat related to the next application of the Greedy method, which is, arguably, among the most important applications of the greedy method!

The most important applications of the Greedy Method: The Huffman Code

- Assume you are given a set of symbols, for example the English alphabet plus punctuation marks and a blank space (to be used between words).
- You want to encode these symbols using binary strings, so that sequences of such symbols can be decoded in an unambiguous way.
- One way of doing so is to reserve bit strings of equal and sufficient length, given the number of distinct symbols to be encoded; say if you have 26 letters plus 6 punctuation symbols, you would need strings of length 5 ($2^5 = 32$).
- To decode a piece of text you would partition the bit stream into groups of 5 bits and use a lookup table to decode the text.
- However this is not an economical way: all the symbols have codes of equal length but the symbols are not equally frequent.
- One would prefer an encoding in which frequent symbols such as 'a', 'e', 'i' or 't' have short codes while infrequent ones, such as 'w','x' and 'y' can have longer codes.
- However, if the codes are of variable length, then how can we partition a bitstream UNIQUELY into segments each corresponding to a code?
- One way of insuring unique readability of codes from a single bitstream is to ensure that no code of a symbol is a prefix of a code for another symbol.
- Codes with such property are called *the prefix codes*.

The most important applications of the Greedy Method: The Huffman Code

- We can now formulate the problem:

  Given the frequencies (probabilities of occurrences) of each symbol, design an optimal prefix code, i.e., a prefix code such that the expected length of an encoded text is as small as possible.

- Note that this amounts to saying that the *average* number of bits per symbol in an "average" text is as small as possible.

- We now sketch the algorithm informally; please see the textbook for details and the proof of optimality.

- There are n radio towers for broadcasting tsunami warnings. You are given the (x, y) coordinates of each tower and its radius of range. When a tower is activated, all towers within the radius of range of the tower will also activate, and those can cause other towers to activate and so on.

- You need to equip some of these towers with seismic sensors so that when these sensors activate the towers where these sensors are located all towers will eventually get activated and send a tsunami warning.

- The goal is to design an algorithm which finds the fewest number of towers you must equip with seismic sensors.

- Someone has proposed the following two algorithms:
  1. Algorithm 1: Find the unactivated tower with the largest radius (if multiple with the same radius, pick the any of them). Activate this tower. Find and remove all activated towers. Repeat.
  2. Algorithm 2: Find the unactivated tower with the largest number of towers within its range. If there is none, activate the leftmost tower. Repeat.

- Give examples which show that neither Algorithm 1 nor Algorithm 2 solve the problem correctly.
- Design an algorithm which correctly solves the problem.
- Solving this problem involves several important concepts which we now revisit.

- Given a directed graph $G = (V, E)$ and a vertex v, the *strongly connected component* of G containing v consists of all vertices $u \in V$ such that there is a path in G from v to u and a path from u to v.

- How do we find a strongly connected component $C \subseteq V$ containing v?
- Construct another graph $G_{rev} = (V, E_{rev})$ consisting of the same set of vertices V but with the set of edges E_{rev} obtained by reversing the direction of all edges E of G.

- Use BFS to find the set $D \subseteq V$ of all vertices in V which are reachable in G from v.

- Find also the set $R \subseteq V$ of all vertices which are reachable in G_{rev} from v.
- The strongly connected component C of G containing v is just the set $C = D \cap R$.
- Clearly, distinct strongly connected components are disjoint sets.

- Let S_G be the set of all strongly connected components of a graph G.

- We define a graph $\Sigma = (S_G, E^*)$ with vertices S_G and directed edges E^* between the strongly connected components so that if $\sigma_1 \in S_G$ and $\sigma_2 \in S_G$ and $\sigma_1 \neq \sigma_2$ then there exists an edge from σ_1 to σ_2 in E^* just in case there exist a vertex $u_1 \in \sigma_1$ and a vertex $u_2 \in \sigma_2$ so that there exists and edge from u_1 to u_2 in E.

- Clearly, $\Sigma = (S_G, E^*)$ is a directed acyclic graph, and thus allows a topological sorting of vertices.

- Topological sort of a directed acyclic graph is a linear ordering (enumeration) of its vertices $\sigma_i \in S_G$ such that if there exists an edge $(\sigma_i, \sigma_j) \in E^*$ then vertex σ_i precedes σ_j in such an ordering , i.e., $i < j$.

- How do we topologically sort a directed acyclic graphs?

Topological sorting

- $L \leftarrow$ Empty list that will contain ordered elements
- $S \leftarrow$ Set of all nodes with no incoming edge
- **while** S is non-empty **do**
  - remove a node u from S;
  - add u to tail of L;
  - **for** each node v with an edge $e = (u, v)$ from u to v **do**
    - remove edge e from the graph;
    - **if** v has no other incoming edges **then** insert v into S;
- **if** graph has edges left, **then return** error (graph has at least one cycle)

  else return L (a topologically sorted order)

Now it should be easy to use the Greedy Strategy to solve the problem of finding the fewest number of towers which you must equip with seismic sensors, so that all emergency transmission towers get activated.

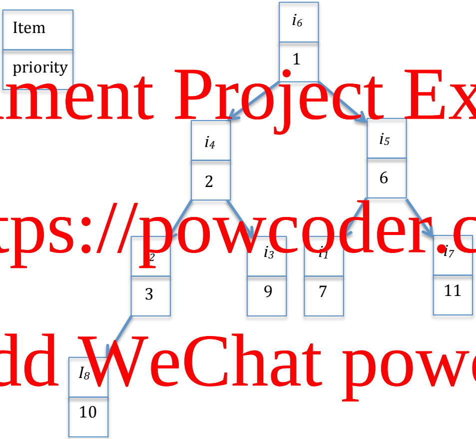
An augmented Priority Queue

- We will need a priority queue which allows an efficient change of the key of an element in the queue, so we first need to extend the Heap data structure.

- We will use heaps represented by arrays; the left child of $A[i]$ is stored in $A[2i]$ and the right child in $A[2i+1]$.
- We will store in heaps vertices of graphs with keys computed in various ways; if a graph has n vertices we will label them with positive integers 1 to n.

- Thus, every element of A is of the form $(i, k(i))$ where $k(i)$ is the key of element i.

- Besides the array A which represents the heap, we will use another array P of the same length which stores the position of elements in the heap; thus $A[P[i]] = (i, k(i))$.
- Changing the key of an element i is now an $O(\log n)$ operation: we look up its position $P[i]$ in A, change the key of the element in $A[P[i]]$ and then perform the Heappify operation to make sure the Heap property is being preserved.



Heap Array

i_6	i_4	i_5	i_2	i_3	i_1	i_7	i_8
1	2	6	3	9	7	11	10
1	2	3	4	5	6	7	8

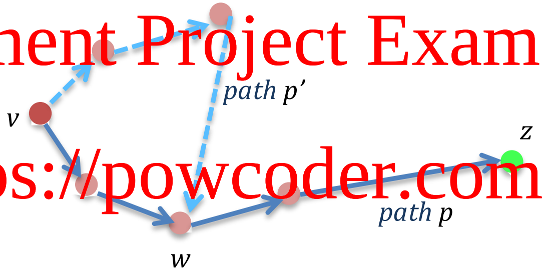
Index Array

i_1	i_2	i_3	i_4	i_5	i_6	i_7	i_8
6	*4*	*5*	*2*	*3*	*1*	*7*	*8*

Greedy Method applied to graphs: Shortest Paths

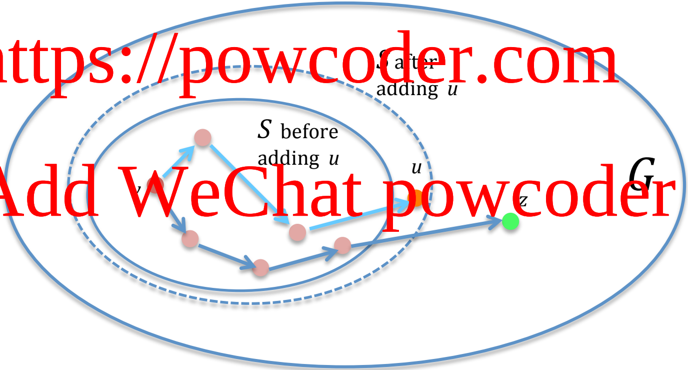
- Some of the most important applications of the Greedy Strategy are in graph algorithms.
- Assume we are given a directed graph $G = (V, E)$ with **non-negative** weight $w(e) \geq 0$ assigned to each edge $e \in E$.
- We are also given a vertex $v \in V$.
- For simplicity, we will assume that for every $u \in V$ there is a path from v to u.
- The task is to find for every $u \in V$ the shortest path from v to u.
- This is accomplished by a very elegant greedy algorithm by Edsger Dijkstra already in 1959!

- We first prove a simple fact about shortest paths.
- Consider a shortest path p in G from a vertex v to a vertex z (shown in dark blue).



- We claim that for every vertex w on that path, the shortest path from v to w is just the truncation of the shortest path from v to z, ending at w.
- Assume the opposite, that there is a shorter path from v to w (shown in dashed light blue) which is not an initial segment of the shortest path from v to z.
- However, in that case we could remove the part of the shortest path between v and z which is between v and w and replaced it with the light blue shorter path from v to w, thus obtaining a shorter path from v to z.
- Contradiction!

- The algorithm builds a set S of vertices for which the shortest path has been already established, starting with a single source vertex $S = \{v\}$ and adding one vertex at a time.
- At each stage of the construction, we add the element $u \in V \setminus S$ which has the shortest path from v to u with all intermediate vertices already in S.

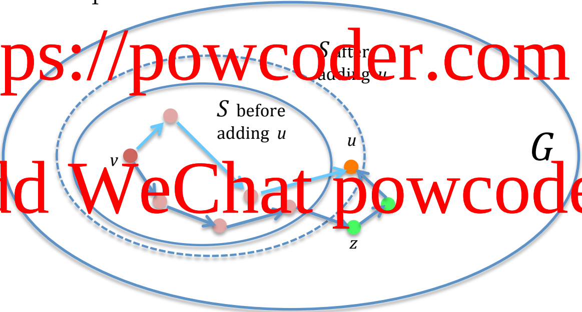


Dijkstra's Shortest Paths Algorithm

- Why does this produce a shortest path from v to u in G?

- Assume the opposite, that there exists a shorter path from v to u in G. By our choice of u such a path cannot be entirely in S.

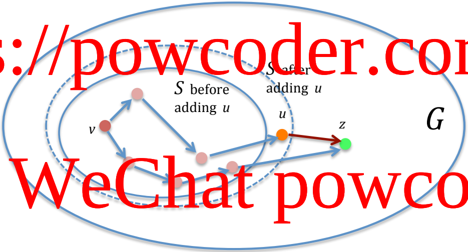
- Let z be the first vertex outside S (as it was just prior to addition of u) on such a shortest path.



- Since there are no negative weight edges the path from v to such z would be shorter than the path from v to u, contradicting our choice of u.

Dijkstra's Shortest Paths Algorithm

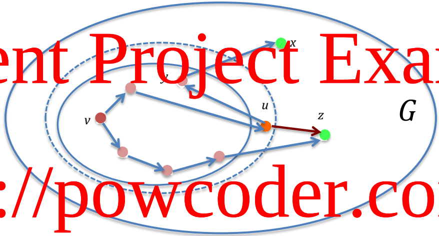
- How is this construction done efficiently?
- Initially all vertices except v are placed in a heap based priority queue with additional Position array, with the weight $w(v, u)$ if $(v, u) \in E$ or ∞ as the key;
- As we progress, the key of each element u will be updated with length $\mathrm{lh}_{\mathrm{S,v}}(u)$ of the shortest path from v to u which has all intermediate vertices on such a path in S.



- We always pop the element u from the priority queue which has the smallest key and add it to set S.
- We then look for all elements $z \in V \setminus S$ for which $(u, z) \in E$ and if $\mathrm{lh}_{\mathrm{S,v}}(u) + w(u, z) < \mathrm{lh}_{\mathrm{S,v}}(z)$ we update the key of z to the value $\mathrm{lh}_{\mathrm{S,v}}(u) + w(u, z)$.

Dijkstra's Shortest Paths Algorithm

- Why is this enough; i.e., why the only shortest paths which could have changed have u as the last vertex in S?



- Assume opposite that the shortest path to a vertex x has changed when u was added, and that instead of u another node y was the last vertex before x on such a new shortest path.
- However, this is not possible because it would produce a shortest path to y with a vertex u on that path not belonging to set S before adding u.

- If graph G has n vertices and m edges, then each edge is inspected only once, when it is an outgoing edge from the most recently added vertex; updating a vertex key takes $O(\log n)$ many steps.

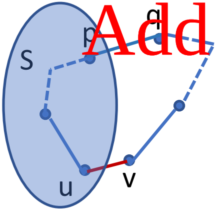
- Thus, in total, the algorithm runs in time $O(m \log n)$.

Greedy Method for graphs: Minimum Spanning Trees

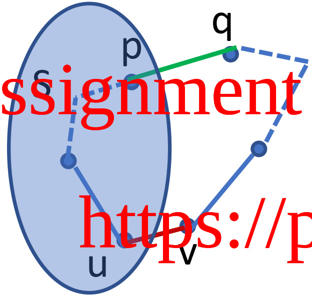
- **Definition:** A minimum spanning tree T of a connected graph G is a subgraph of G (with the same set of vertices) which is a tree, and among all such trees it is of minimal total length of all edges in T.

- **Lemma:** Let G be a connected graph with all lengths of edges E of G distinct and S a non empty proper subset of the set of all vertices V of G. Assume that $e = (u, v)$ is an edge such that $u \in S$ and $v \notin S$ and is of minimal length among all the edges having this property. Then e must belong to every minimum spanning tree T of G.

- **Proof:**



- Assume opposite, that there exists a minimum spanning tree which does not contain such an edge $e = (u, v)$.
- Since T is a spanning tree, there exists a path from u to v within such a tree.



- Since $u \in S$ and $v \notin S$, this path must have left S at a certain point.
- Assume that p is the last vertex on this path which is in S and $q \notin S$ the vertex immediately after p on that path.
- However, the edge (p, q) belongs to T and must have a length larger than the edge (u, v) (which is the minimal length edge with one end in S and one end outside S.
- Adding the edge $e = (u, v)$ to such a path from u to v produces a loop which is removed by removing the edge (p, q).
- However, since by assumption the weight of edge (u, v) is smaller than the weight of edge (p, q), this would result in a spanning tree of smaller weight, contradicting our assumption that T is a minimum spanning tree.

The Kruskal Algorithm

- We order the edges E in a non-decreasing order with respect to their weights.
- We build a tree by adding edges, one at each step of construction.

- An edge e_i is added at a round i of construction whenever its addition does not introduce a cycle in the graph constructed thus far.
- If adding an edge does introduce a cycle, that edge is discarded.

- The process terminates when the list of all edges has been exhausted.

Minimum Spanning Trees: the Kruskal Algorithm

Claim: The Kruskal algorithm produces a minimal spanning tree, and if all weights are distinct, then such a Minimum Spanning Tree is unique.

Proof: We consider the case when all weights are distinct.

- Consider an arbitrary edge $e = (u, v)$ added in the course of the Kruskal algorithm.
- Consider the set S of all vertices w such that there exists a path from u to w using only the subset of edges that have been added by the Kruskal algorithm until just before the edge $e = (u, v)$ has been added.

- Then clearly $u \in S$ but $v \notin S$.
- Until this moment no edge with one end in S and the other outside S has been considered because otherwise it would have been added, not forming a cycle.

- Consequently, edge $e = (u, v)$ is the shortest one with such a property and by the previous lemma it must belong to every spanning tree.
- Thus, the set of edges produced by the Kruskal algorithm is a subset of the set of edges of every spanning tree.
- But the structure produced by the Kruskal algorithm is clearly cycle-free and it spans the graph, otherwise another edge could be added.

- To efficiently implement the Kruskal algorithm, we need a useful data structure for storing sets of elements, called the Union-Find.

- Such a data structure has to support three operations:

1. MakeUnionFind(S), which, given a set S returns a strucure in which all elements are placed into distinct singleton sets. Such an operation should run in time $O(n)$ where $n = |S|$.

2. Find(v), which, given an element v returns the (label of the) set to which v belongs. Such operation should run either in time $O(1)$ or time $O(\log n)$ as we explain later.

3. Union(A, B), which, given (the labels of) two sets A, B changes the data structure by replacing sets A and B with the set $A \cup B$. A sequence of k consecutive Union operations should run in time $O(k \log k)$.

Efficient implementation of the Kruskal Algorithm

- Note that we do not give the run time of a single UNION operation but of a sequence of k consecutive such operations.

- Such time complexity analysis is called *amortized analysis*; it (essentially) estimates average cost of an operation in a sequence of operations, in this case $\log k$.

- We will label each set by one of its elements. Since we will use the Union-Find data structure to keep track of sets of vertices of graphs, we can assume that the underlying set S is of the form $S = \{1, 2, \ldots, n\}$.

- The simplest implementation of the Union-Find data structure consists of:

  1. an array A such that $A[i] = j$ means that i belongs to set labeled j;
  2. an array B such that $B[i]$ contains the number of elements in the set labeled by i (which can be 0) and pointers to the first and last element of a linked list of elements of the set labeled by i.

Efficient implementation of the Kruskal Algorithm

- UNION(i,j) of two sets labeled by i and j, respectively, is defined as follows:
  - if number of elements in the set labeled by i is larger or equal to the number of elements in the set labeled by j then labels in array A of all elements in the set labeled by j is changed to i and array B is updated accordingly;
  - if the set labeled by j has more elements than the set labeled by i, then the labels of all elements in the set labeled by i are changed to j and array B is updated accordingly.
- Note that this definition implies that if UNION(i, j) changes the label of the set containing an element m, then the new set containing m will have at least twice the number of elements of the set which contained m before the UNION(i, j) operation.

- Any sequence of k initial consecutive UNION operations can touch at most $2k$ elements of S (which happens if all UNION operations were applied to singleton sets).

- Thus, the set containing an element m after k initial consecutive UNION must have at most $2k$ elements.

- Since every UNION operation which changes the label of the set containing m at least doubles the size of the set containing that element, the label of the set containing m could change fewer than $\log 2k$ many times.

- Thus, since we have at most $2k$ elements, any sequence of k initial consecutive UNION operations will have in total fewer than $2k \log 2k$ many label changes in A and each UNION operation changes just a few pointers in B and adds up the sizes of sets.

- Thus, every sequence of k initial consecutive Union operations has time complexity of $O(k \log k)$.

- Such Union-Find data structure is good enough to get the sharpest possible bound on the run time of the Kruskal algorithm.

- See the textbook for an Union-Find data structure based on pointers and path compression, which further reduces the amortised complexity of the Union operation at the cost of increasing the complexity of the Find operation from $O(1)$ to $O(\log n)$.

Efficient implementation of the Kruskal Algorithm

- We now use the previously described Union-Find data structure to efficiently implement the Kruskal algorithm on a graph $G = (V, E)$ with n vertices and m edges.

- We first have to sort m edges of graph G which takes time $O(m \log m)$. Since $m \leq n^2$, this step of the algorithm also takes $O(m \log n^2) = O(m \log n)$.

- As we progress through the Kruskal algorithm execution, we will be making connected components which will be merged into larger connected components until all vertices belong to a single connected component. For this purpose we use Union-Find data structure to keep track the connected components constructed thus far.

- For each edge $e = (v, u)$ on the sorted list of edges we use two FIND operations, FIND(u) and FIND(v) to determine if vertices u and v belong to the same component.

- If they do not belong to the same component, i.e., if $\textsc{Find}(u) = i$ and $\textsc{Find}(v) = j$, $j \neq i$, we add edge $e = (u, v)$ to the spanning tree being constructed and perform $\textsc{Union}(i, j)$ to place u and v into the same connected component.

- In total we perform $2m$ $\textsc{Find}$ operations, each costing $O(1)$, in total costing $O(m)$.

- We also perform $n - 1$ $\textsc{Union}$ operations which in total cost $O(n \log n)$.

- Initial sorting of edges takes $O(m \log m) = O(m \log n^2) = O(m \log n)$ operations; thus, we obtain an overall time complexity of $O(m \log n)$.

k-clustering of maximum spacing

- **Instance:** A complete graph G with weighted edges representing distances between the two vertices.

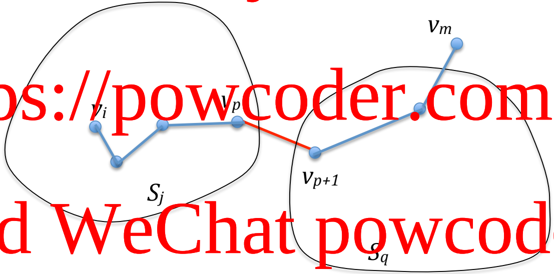
- **Task:** Partition the vertices of G into k disjoint subsets so that the minimal distance between two points belonging to different sets of the partition is as large as possible. Thus, we want a partition into k disjoint sets which are as far apart as possible.

- **Solution:** Sort the edges in an increasing order and start performing the usual Kruskal's algorithm for building a minimal spanning tree, but stop when you obtain k connected components, rather than a single spanning tree.

- **Proof of optimality:** Let d be the distance associated with the first edge of the minimal spanning tree which was not added to our k connected components; it is clearly the minimal distance between two vertices belonging to two of our k connected. Clearly, all the edges included in k many connected components produced by our algorithm are of length smaller or equal to d.

k-clustering of maximum spacing

- Consider any partition $\mathcal{S}$ into k subsets different from the one produced by our algorithm.
- This means that there is a connected component produced by our algorithm which contains vertices v_i and v_m such that $v_i \in S_j$ for some $S_j \in \mathcal{S}$ and $v_m \notin S_j$.



- Since v_i and v_m belong to the same connected component, there is a path in that component connecting v_i and v_m.
- Let v_p and v_{p+1} be two consecutive vertices on that path such that v_p belongs to S_j and $v_{p+1} \notin S_j$.
- Thus, $v_{p+1} \in S_q$ for some $q \neq j$.

- Note that $d(v_p, v_{p+1}) \leq d$ which implies that the distance between these two clusters $S_j, S_q \in \mathcal{S}$ is smaller or equal to the minimal distance d between the k connected components produced by our algorithm.

- Thus, such a partition cannot be a more optimal clustering than the one produced by our algorithm.

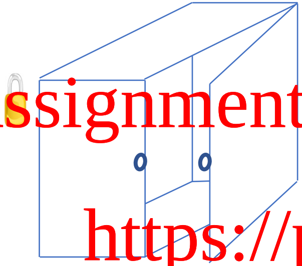
- What is the time complexity of this algorithm?

- We have $O(n^2)$ edges; thus sorting them by weight will take $O(n^2 \log n^2) = O(n^2 \log n)$.

- While running the (partial) Kruskal algorithm we use the UNION-FIND data structure which requires $O(n^2 \log n)$ steps.

- So the grand total for the whole algorithm is $O(n^2 \log n)$ many steps.

PUZZLE!!



Bob is visiting Elbonia and wishes to send his teddybear to Alice who is staying at a different hotel. Both Bob and Alice have boxes like the one shown on the picture as well as padlocks which can be used to lock the boxes. However, there is a problem.

The Elbonian postal service mandates that boxes to be sent, if not empty, must be locked, but they do not allow keys to be sent. The key must remain with the sender. You can send padlocks only if they are locked. How can Bob safely send his teddybear to Alice?

Hint: The way how the boxes are locked (via a padlock) is important as well as that both Bob and Alice have padlocks and boxes. They can also communicate over the phone to agree on the strategy. There are two possible solutions; one can be called the "AND" solution, the other can be called the "OR" solution. The "AND" solution requires 4 mail one way services while the "OR" solution requires only 2.