CS 5114
Solutions to Homework Assignment 5

Meghendra Singh

March 17, 2017

[20] 1. CLRS Exercise 23.1-8. Let T be a minimum spanning tree of a graph G, and let L be the sorted list of the edge weights of T . Show that for any other minimum spanning tree T' of G, the list L is also the sorted list of edge weights of T'. First develop good, careful notation. Then use a substitution argument.

Given, T is a MST of a graph $G = (V, D)$, where V and D are the sets of vertices and edges of G. Let E be the sequence of edges in increasing order of their weights, present in T, and let L be the list of edge weights corresponding to each of the edges e_i in E (i.e. sorted list of edge weights of T). So,

$$E = e_1, e_2, e_3, ..., e_n \text{ Such that, } \forall e_i \ w(e_i) < w(e_{i+1})$$
$$\text{and,}$$
$$L = w(e_1), w(e_2), w(e_3), ..., w(e_n)$$

Let T' be another MST for the graph G. Let E' be the sequence of edges in increasing order of their weights, present in T', and let L' be the list of edge weights corresponding to each of the edges e'_i in E' (i.e. sorted list of edge weights of T'). Since both T and T' are MSTs, the number of edges in T would be equal to the number of edges in T' which would be equal to $|V| - 1$, therefore $|E| = |E'| = |L| = |L'| = |V| - 1$. Also, the sum of weights of edges in T and T' needs to be equal, for both of them to be MSTs, i.e. $w(T) = w(T')$, here w represents the weight function. So,

$$E' = e'_1, e'_2, e'_3, ..., e'_n \text{ Such that, } \forall e'_i \ w(e'_i) < w(e'_{i+1})$$
$$\text{and,}$$
$$L' = w(e'_1), w(e'_2), w(e'_3), ..., w(e'_n)$$

Assume, that for the first $k - 1$ positions, all the edges are common between E and E', i.e. $e_i = e'_i \ \forall i$, such that $1 \leq i \leq k - 1$. While, from the k^{th} position onwards the edges are different between E and E', i.e. $e_i \neq e'_i \ \forall i$, such that $k \leq i \leq n$. Lets consider the k^{th} edges in E and E', i.e. e_k and e'_k. This can be written as:

$$E = e_1, e_2, e_3, ..., e_{k-1}, e_k, e_{k+1}, ..., e_n$$
$$E' = e_1, e_2, e_3, ..., e_{k-1}, e'_k, e'_{k+1}, ..., e'_n$$
$$\text{also,}$$
$$L = w(e_1), w(e_2), w(e_3), ..., w(e_{k-1}), w(e_k), w(e_{k+1}), ..., w(e_n)$$
$$L' = w(e_1), w(e_2), w(e_3), ..., w(e_{k-1}), w(e'_k), w(e'_{k+1}), ..., w(e'_n)$$

Now, there can be three possible relationships (by Trichotomy property) between the weights of the e_k and e'_k:

(i) $w(e_k) < w(e'_k)$

(ii) $w(e_k) > w(e'_k)$

(iii) $w(e_k) = w(e'_k)$

Case (i) cannot be true, because this would imply that for the k^{th} edge we could substitute e_k for e'_k in E'. This can be done because, e_k does not form a cycle in T and is a lighter edge than e'_k. So, if e'_k is a part of T', T' is no longer an MST, which is a contradiction to our initial assumption. Similarly, **Case (ii)** cannot be true, as this would imply that for the k^{th} edge we could substitute e'_k for e_k in E, because e'_k does not form a cycle in T' and is a lighter edge than e_k. Hence, if e_k is a part of T, T is no longer an MST, which is once again a contradiction to our initial assumption. This leaves us with **Case (iii)** as the only feasible case, which would not result in a contradiction.

Since, $w(e_k) = w(e'_k)$, we can substitute $w(e'_k)$ in L' with $w(e_k)$, to get a new list L'' for T', given by:

$$L'' = w(e_1), w(e_2), w(e_3), ..., w(e_{k-1}), w(e_k), w(e'_{k+1}), ..., w(e'_n)$$

In a similar way, we can prove that $w(e_{k+1}) = w(e'_{k+1})$, $w(e_{k+2}) = w(e'_{k+2})$, ..., $w(e_n) = w(e'_n)$, by applying the above method of proof by contradiction and substitution into L'' on all of the subsequent edges one by one in E and E'. This would result in:

$$\begin{aligned} L'' &= w(e_1), w(e_2), w(e_3), ..., w(e_{k-1}), w(e_k), w(e_{k+1}), ..., w(e_n) \\ &= L \end{aligned}$$

Hence, we can say that L is the list of sorted edge weights for both T and T'.

---

[20] 2. CLRS Problem 23-1. Second-best minimum spanning tree. Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \to \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct. We define a second-best minimum spanning tree as follows. Let $\mathfrak{T}$ be the set of all spanning trees of G, and let T' be a minimum spanning tree of G. Then a **second-best minimum spanning tree** is a spanning tree T such that $w(T) = \min_{T'' \in \mathfrak{T} - \{T'\}}\{w(T'')\}$.

a. Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.

b. Let T be the minimum spanning tree of G. Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G.

c. Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let $max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T. Describe an $O(V^2)$ time algorithm that, given T, computes $max[u, v]$ for all $u, v \in V$.

d. Give an efficient algorithm to compute the second-best minimum spanning tree of G.

The solutions for each of the questions are as follows:

a. Proof of unique MST: Given, graph $G = (V, E)$ and T' is an MST of G, and all weights of all edges in E are distinct. Let L' be the list of sorted edge weights for T'. Let T'', be another MST of G, and L'' be the list of sorted edge weights for T''. If T' and T'' are both MSTs, $w(T') = w(T'')$. We know from **question 1** above that any two MSTs of the same graph G, have a common sorted list of edge weights, therefore:

$$L' = L'' \tag{1}$$

For, T' and T'', to be distinct there should be at least one edge, that is not common between them. Let this edge be at the k^{th} position in L' and L''. By (1) above, we can write the summation of all the items in the two lists L' and L'' as:

$$\sum L' = \sum L''$$
$$w(e_1) + w(e_2) + ... + w(e_k) + ... + w(e_n) = w(e_1) + w(e_2) + ... + w(e'_k) + ... + w(e_n)$$
$$\implies w(e_k) = w(e'_k)$$

Hence, even-though the k^{th} edges e_k and e'_k are not the same in the two MSTs, the weights of the k^{th} edges are equal, this is a contradiction to the initial condition that all of the edge weights are distinct in G. Therefore, $e_k = e'_k \implies T' = T''$. Hence, the MST T' for the graph G is unique.

Proof of 2^{nd}-best MST may not being unique: Consider the graph G in the top of, **figure 1**, which has 5 vertices $\{a, b, c, d, e\}$ and 7 edges, with weights $\{1, 2, 3, 4, 5, 6, 7\}$. Here, all the edge weights are distinct and we can observe that, although there is a unique MST T (shown with blue edges), there are 3 possible **second-best minimum spanning trees** T', T'' and T''' (shown with red edges), all with the weight of 14. This counter example shows that for a given graph the second-best minimum spanning trees may not be unique.

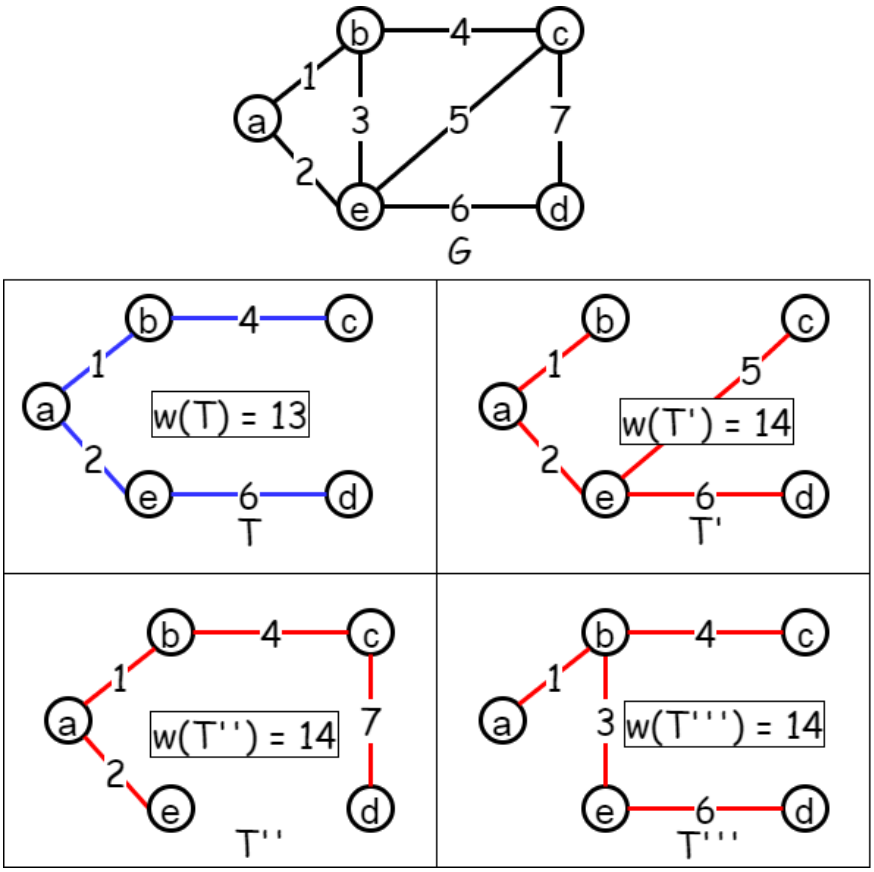

Figure 1: Graph G, its unique MST T and three 2^{nd}-best MSTs T',T'' and T'''

b. Given, T is a MST of $G = (V, E)$. Let T' be a second-best MST of G. Since, both T and T' are spanning trees, the number of edges in them have to be equal and one less than $|V|$, i.e. $|T| = |T'| = |V| - 1$. Also, by the definition of second-best minimum spanning tree we have:

$$w(T') = min_{T'' \in \mathfrak{T} - \{T\}}\{w(T'')\}$$

Here, $\mathfrak{T}$ be the set of all spanning trees of G. This implies that, T and T' should differ by at-least 1 edge, for T' to be a second-best MST, i.e. $|T - T'| \geq 1$. Assume that T' differs from T by one edge and let this edge be (u, v) in T and (x, y) in T'. Therefore, we can write:

$$w(T') = w(T) - w(u, v) + w(x, y)$$

Note, that for T' to be a second-best MST, we need to replace (u, v) with another minimum weight edge that crosses the same cut as (u, v), we call this edge (x, y) and $(x, y) \neq (u, v)$. Assume, there is another second-best MST T'' for G, which differs from T by two edges and let these edges be $(u, v), (u', v')$ in T and $(x, y), (x', y')$ in

T'', respectively. Therefore,

$$\begin{aligned} w(T'') &= w(T) - w(u,v) - w(u',v') + w(x,y) + w(x',y') \\ &= \big[w(T) - w(u,v) + w(x,y)\big] - w(u',v') + w(x',y') \\ &= \big[w(T')\big] + w(x',y') - w(u',v') \end{aligned} \tag{2}$$

Once again, for T'' to be a second-best MST, we need to replace (u',v') with another minimum weight edge that crosses the same cut (say $(S, V-S)$) as (u',v'), we call this edge (x',y') and $(x',y') \neq (u',v')$. Since we know that (u',v') is a light edge (it is present in the MST, which implies it is the minimum weighted edge that crosses the cut $(S, V-S)$), any other edge (say (x',y')) which crosses the same cut $(S, V-S)$ would be heavier than (u',v'). Therefore, $w(x',y') > w(u',v')$ and hence the difference $w(x',y') - w(u',v')$ would be some positive value say C. Replacing $w(x',y') - w(u',v')$ with C in (2), we get:

$$\begin{aligned} & w(T'') = w(T') + C \\ \implies\ & w(T'') > w(T') \end{aligned}$$

Therefore, T'', cannot be a second-best MST, since there exists T', which is another second-best MST, which has lower weight than T''. Hence, if we replace more than one edges from an MST, we will not get a second-best MST. So, if T is an MST of G, G contains edges $(u,v) \in T$ and $(x,y) \notin T$ such that $T - \{(u,v)\} \cup \{(x,y)\}$ is T', which is a second-best minimum spanning tree of G.

c. For constructing the *max* for a spanning tree T, we can modify the breadth first search algorithm. The following algorithm GET-MAX constructs the *max* matrix, given the graph G, its spanning tree T and the weight matrix w:

```
GET-MAX(G, T, w)
 1  n = |G.V|
 2  let max be a 2-dimensional matrix of size [nXn]
 3  for each vertex u ∈ G.V
 4       for each vertex v ∈ G.V
 5            max[u, v] = −∞
 6       Q = φ
 7       ENQUEUE(Q,u)
 8       while (Q ≠ φ)
 9            k = DEQUEUE(Q)
10            for each vertex v ∈ T.Adj[k]
11                 if v ≠ u
12                      if k == u
13                           // look at adjacent vertices of k in T
14                           max[u, v] = w(u, v)
15                      else if k ≠ u and w(k, v) > max[u, k]
16                                // look at non-adjacent vertices of k in T
17                                max[u, v] = w(k, v)
18                           else max[u, v] = max[u, k]
19                      ENQUEUE(Q,v)
20  return max
```

Here, we start with an empty max matrix, and initialize all its elements with $-\infty$. Next, we use a queue Q, to perform breadth first search on the vertices of G, while only considering the edges present in the spanning tree T. Since, $max[u, v]$ has to be the heaviest edge on the simple path between two vertices, we know that this would be equal to the edge weight in case the two vertices are adjacent in T. If they are not adjacent, we keep a record of the maximum edge weight found in the path between the two vertices present in T, and use this for filling the max entry, once the second vertex is reached. This algorithm results in the following max matrix for the MST T shown in **figure 1**:

max	a	b	c	d	e
a	$-\infty$	1	4	6	2
b	1	$-\infty$	4	6	2
c	4	4	$-\infty$	6	4
d	6	6	6	$-\infty$	6
e	2	2	4	6	$-\infty$

In GET-MAX, we are executing a modified BFS, for every vertex in the graph $G = (V, E)$. Since BFS takes $O(|V|)$ (lines 6 through 19 in GET-MAX), and GET-MAX executes the modified BFS $|V|$ times (line 3 in GET-MAX). We can conclude that the running time of GET-MAX will be $O(|V|^2)$.

d. FIND-SECOND-BEST-MST below, specifies an efficient algorithm to compute the sec-

ond best minimum spanning tree for a graph, given the graph G and its weight matrix W.

```
FIND-SECOND-BEST-MST(G, W)
 1  let R be any random vertex in G.V
 2  T = MST-PRIMS(G,W,R)
 3  max = GET-MAX(G,T,W)
 4  diff = ∞
 5  e = NIL
 6  for each edge (u, v) ∈ G.E − T
 7       if max[u, v] − w(u, v) < diff
 8            diff = max[u, v] − w(u, v)
 9            e = (u, v)
10  f = NIL
11  for each edge (x, y) ∈ T
12       if w(x, y) == max[e]
13            f = (x, y)
14  T' = T − {f} ∪ {e}
15  return T'
```

Here, we basically replace an edge in the MST to get the second-best MST. We start by using an efficient MST-PRIMS algorithm to get the MST T. Next, we compute the max table for T. After this we search for an edge $e \notin T$ (i.e. $e \in G.E - T$) that when added to T (and another edge is removed from the cycle created when e is added to T) would result in the second-best MST. This edge e minimizes the difference between its entry in the max matrix for T and its edge weight W(E). Once we find one or more such edges, we need to find the edge $f \in T$ which would be replaced by this new edge e. This edge f is precisely the edge whose weight is equal to the max entry of e. Once found, we construct and return the second-best MST T' by replacing f with e in T.

If we consider the graph $G = (V, E)$ and its corresponding MST T which are shown in **figure 1**. There are 3 possible edges $\notin T$ that can be considered for addition to T in order to get three different second-best MSTs. These 3 edges are $\{(b, e), (c, e), (c, d)\}$, because each of them will result in a $diff$ value of -1, when we execute FIND-SECOND-BEST-MST (this can be easily computed using the max table shown in the last section **c**). The candidate edges that could be replaced by one of these three edges are $\{(a, e), (b, c), (e, d)\}$ since there weights are precisely those that are present in the max matrix entries for edges $\{(b, e), (c, e), (c, d)\}$ respectively. Upon replacement of any one of these three edges in T we get a second-best MST, and as shown in **figure 1** these three second-best MSTs are T', T'' and T'''.

We can use an efficient MST-PRIMS algorithm, (one that uses a Fibonacci heap instead of a min-heap) to get the MST, T required at line 2 of FIND-SECOND-BEST-MST in $O(|V|^2)$ time. Line 3 of FIND-SECOND-BEST-MST would also require $O(|V|^2)$ time as shown in the section **c**. After this the two loops at lines 6 and 11 of FIND-SECOND-BEST-MST, are basically traversing all the edges present in the graph

$G \notin T$ and all the edges present in T. This would result in a running time of $O(|E|) = O(|V|^2)$. Therefore the overall running time of FIND-SECOND-BEST-MST would be $O(|V|^2)$.

---

[20] 3. CLRS Exercise 34.1-3. Give a formal encoding of directed graphs as binary strings using an adjacency-matrix representation. Do the same using an adjacency-list representation. Argue that the two representations are polynomially related.

An encoding of a set S is a one-to-one function: $e : S- > \{0,1\}^*$. In this case the set S consists of an adjacency-matrix and an adjacency-list representing a directed and weighted graph $G = (V, E)$. An encoding of S will be a binary string. Consider the following adjacency-matrix and adjacency-list of a weighted and directed graph G consisting of 3 vertices $\{A, B, C\}$ and 5 edges $\{(A, A), (A, B), (B, B), (B, C), (C, A)\}$ as an example:

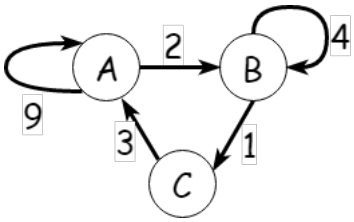

Figure 2: The directed and weighted graph G.

	A	B	C
A	9	2	0
B	0	4	1
C	3	0	0

Table 1: Adjacency-matrix for G.

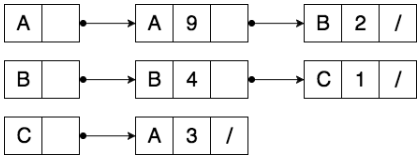

Figure 3: Adjacency-list for G.

We need two different and efficient formal encodings for the adjacency-matrix and the adjacency-list representations. These are as follows:

i. **Adjacency-matrix representation:** In order for the encoding to be efficient, we need to encode the edge weights as Binary coding, while using Unary coding in the beginning of the encoding to specify the maximum number of bits required to be read, so as to identify an edge weight. The maximum number of bits required to be read would depend on the maximum edge weight present in the adjacency-matrix, let this maximum edge weight be: M. Therefore, the binary number representing $M = M_2$, and the number of $1's$ in the Unary code will be equal to $|M_2|$. Let, $k = |M_2|$, and the dimensions of the adjacency-matrix be n. An encoding could hence be written as:

$$e(S_{adjacency-matrix}) = 111...10\ 00..10\ 10..01\ 00..01\ ...$$

i.e. k 1's followed by a 0 (bits shown in red above), followed by base-2 (binary) codes of each of the n^2 edge weights present in the adjacency matrix for G, each binary code will be of length k bits (bits shown in blue above). Note that if the total number of bits in the encoding is l, we can determine the dimensions of the adjacency-matrix (i.e. n, which is also the number of vertices in G, therefore $n = |V|$), by taking the square-root of the number of blocks of bits of size k that are present after the first

$k+1$ Unary code bits, i.e. $n = \sqrt{\frac{l-(k+1)}{k}}$. Hence, we do not need to keep n as a part of the encoding itself. We can write the total number of bits (l) required to encode an adjacency matrix containing n^2 elements as:

$$\begin{aligned} l &= k + 1 + n^2 k \\ &= O(kn^2) \end{aligned}$$

Here, k is the number of bits in the binary encoding of the maximum edge weight present in the graph G, whose adjacency-matrix is being encoded and the space complexity comes out to be $O(kn^2)$. The following is the encoding for the adjacency matrix shown in **Table 1**:

11110 1001 0010 0000 0000 0100 0001 0011 0000 0000

Here, $k = 4$ because the maximum edge weight present in the adjacency-matrix is 9, whose binary representation 1001 requires at-least 4 bits. There are $n^2 = 3^2 = 9$ k-bit binary blocks, each representing an edge weight, following the initial $k+1$ bit unary block.

ii. Adjacency-list representation: Once again, for the encoding to be efficient, we need to encode the edge weights as Binary code, while using Unary coding in the beginning of the encoding to specify the maximum number of bits required to be read, to identify an edge weight. The maximum number of bits required to be read would depend on the maximum edge weight present in the adjacency-list, let this maximum edge weight be: M. Therefore, the binary number representing $M = M_2$, and the number of $1's$ in the Unary code will be equal to $|M_2|$. Let, $k = |M_2|$, and the dimensions of the adjacency-list be n. Since, in this case the total number of elements in the list is not always equal to n^2, we need to keep the number of vertices in the graph as a unary code in the beginning for the encoding to be meaningful. This would take an additional $n+1$ bits after the initial $k+1$ bits. Also, since the adjacency list only contains information (edge-weights) about the adjacent vertices connected to a particular vertex, we need to store this information in the encoding as well. This can be done by a simple bitmap of n bits that shows the number of vertices a particular vertex is connected to. This would take an additional n bits for each of the vertices.An encoding could hence be written as:

$e(S_{adjacency-list})$ = 11...10 11...10 0..1 00..10 10..01 00..01 ...1..0 01..00 00..10 10..11 ...

i.e. k 1's followed by a 0 (bits shown in red above), followed by n 1's followed by a 0 (bits shown in green above), followed by the first block representing the first vertex and its connected vertices. This block (beginning at 0..1 above) has a prefix which is a bitmap of length n (bits shown in orange above), followed by base-2 (binary) codes of k-bits each for each of the edge weights present in the adjacency list for the first vertex (bits shown in blue above). The prefix bitmap has bits set (1's) for each of the vertices the first vertex is connected to by an edge, and unset (0's) for all the vertices that are not connected to the first vertex by an edge. This first block is followed by

$n-1$ similar blocks, for each of the remaining vertices, given in the adjacency-list. We can write the total number of bits (l) required to encode an adjacency list containing x items (edge weights) as:

$$\begin{aligned} l &= k + 1 + n + 1 + n * n + x * k \\ &= 2 + k + n + n^2 + x * k \end{aligned}$$

Here, k is the number of bits in the binary encoding of the maximum edge weight present in the graph G, whose adjacency-list is being encoded. Also, x represents the number of items present in the adjacency-list i.e. number of edges present in G. Since the maximum number of edge-weights (items) present in an adjacency list can be n^2. We can write the number of bits required to encode an adjacency-list as:

$$\begin{aligned} l &= 2 + k + n + n^2 + x * k \\ &= 2 + k + n + n^2 + n^2 * k \\ &= O(kn^2) \end{aligned}$$

Once again the space complexity comes out to be $O(kn^2)$. For this particular encoding, we can also conclude that adjacency list should be the preferred representation when we have sparsely connected graphs. Since in the case the number of vertices n is large and the graph is fully connected, an adjacency matrix would result in a compacter encoding as compared to an adjacency-list. The following is the encoding for the adjacency list shown in **Figure 3**:

11110 1110 110 1001 0010 011 0100 0001 100 0011

Here, $k = 4$ because the maximum edge weight present in the adjacency-matrix is 9, whose binary representation 1001 requires at-least 4 bits. The second unary code has $n = 3$ bits set. This is followed by the bitmap for the first vertex A, which basically means that A is connected to A and B. This bitmap is followed by the binary code for the two edge weights of the edges (A, A) and (A, B). This combined bitmap and binary representation of edge weights forms the first block, which is followed by the second block (which represents vertex B and its connected vertices B and C), and a third block (which represents C and its connected vertex A). The number of set bits in the bitmap tells how many k-bit long binary encoded edge weights are present for the current block. This is 2 in the case of A, 2 in the case of B and 1 in the case of C, in the above example.

iii. Adjacency-matrix and Adjacency-list representations are polynomially related: Let f_1 be the function representing the number of bits required to encode an adjacency-matrix. Let f_2 be the function representing the number of bits required to encode an adjacency-list. We know from the last two sections that:

$$\begin{aligned} f_1 &= O(kn^2) \\ \text{and, } f_2 &= O(kn^2) \\ \therefore f_1 &\in O(f_2) \\ \text{also, } f_2 &\in O(f_1) \end{aligned}$$

Here, n is the number of vertices $|V|$ in the graph G. Hence, the two representations are polynomially related.
