Compressed Graphs

We will next describe a simple representation for graphs.

- Let $G = (V, E)$ be a directed graph, where $V = [0..n)$ and $E \subseteq V \times V$ with $|E| = m$.
- We will use adjency lists to represent the graph. For each $v \in V$, let
$$S_v = (w \in V : (v, w) \in E)$$
be the adjacency list for v.
- Let $S[0..m) = S_0 S_1 \dots S_{n-1}$ be the concatenation of the adjacency lists. Let $L[0..n)$ be the sizes of the adjacency lists, i.e., $L[v] = |S_v|$.
- Now each $e \in [0..m)$ represents an edge:
$$\begin{aligned} \text{target}(e) &= \text{access}_S(e) \\ \text{source}(e) &= \text{search}_L(e) \end{aligned}$$
- The edges incident to a node v can be listed as follows:
$$\begin{aligned} \text{out-edges}(v) &= [\text{sum}_L(v), \text{sum}_L(v+1)) \\ \text{in-edges}(v) &= \{\text{select}_S(v, i) \mid i \in [0..\text{rank}_S(v, m))\} \end{aligned}$$

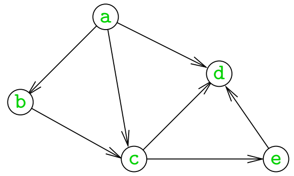
Thus the graph G is represented by:

- A string $S[0..m)$ over the alphabet $V = [0..n)$ with support for operations access, rank and select.
- An array $L[0..n)$ of non-negative integers summing up to m with support for operations sum and search.

Both can be stored in compressed form.

Example 3.16:

$S =$ `bcd c de  d`
$L =$ 3 1 2 0 1



Additional attributes such as weights can be associated to nodes using an array $A[0..n)$ and to edges using an array $B[0..m)$.

Balanced Parentheses

Let $B[0..2n)$ be a bit vector with n 1-bits and n 0-bits. Define
$$\text{excess}_B(i) = \text{rank-1}_B(i) - \text{rank-0}_B(i)$$
B is a *balanced parentheses (BP) sequence* if $\text{excess}_B(i) \geq 0$ for all $i \in [0..2n]$. Then each 1-bit can be interpreted as an opening parenthesis "(" and each 0-bit as a closing parenthesis ")".

Example 3.17:

	(	(	(	)	)	(	(	)	(	)	(	)	)	)		
	1	1	1	0	0	1	1	0	1	0	1	0	0	0		
excess	0	1	2	3	2	1	2	3	2	3	2	3	2	1	0	

Interesting operations on BP sequences include finding the matching parenthesis and the nearest enclosing pair of parentheses:

$$\begin{aligned} \text{find-close}_B(i) &= \min\{j \in [i+1..n) \mid \text{excess}_B(j+1) = \text{excess}_B(i)\} \quad \text{for } B[i] = 1 \\ \text{find-open}_B(j) &= \max\{i \in [0..j) \mid \text{excess}_B(i) = \text{excess}_B(j+1)\} \quad \text{for } B[j] = 0 \\ \text{enclose}_B(i) &= \max\{k \in [0..i) \mid \text{excess}_B(k) < \text{excess}_B(i)\} \quad \text{for } B[i] = 1 \end{aligned}$$

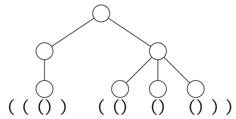
The operations can be supported in constant time using $o(n)$ bits of space in addition to the bit vector. The details are omitted.

Succinct Trees

Any rooted tree of n nodes can be represented as a BP sequence of $2n$ bits:

- A leaf u is represented by $\text{BP}(u) = 10$.
- An internal node v with children $u_1, u_2, \dots, u_k$ is represented by $\text{BP}(v) = 1\text{BP}(u_1)\text{BP}(u_2)\dots\text{BP}(u_k)0$.

Example 3.18: ((())(()()()))



A pointer to a node v is expressed as the starting position of $\text{BP}(v)$ in the whole sequence. Interesting operations include (the ones on the right assume that the requested node exists):

$$\begin{aligned} \text{is-leaf}(v) &= [\text{access}_B(v+1) = 0] & \text{parent}(v) &= \text{enclose}_B(v) \\ \text{depth}(v) &= \text{excess}_B(v) & \text{first-child}(v) &= v + 1 \\ \text{preorder-rank}(v) &= \text{rank-1}_B(v) & \text{next-sibling}(v) &= \text{find-close}_B(v) + 1 \end{aligned}$$

Sparse bit vectors

Many applications involve sparse bit vectors with few 1-bits. The following is a useful result for analysing them:

Lemma 3.19: Let $B[0..u)$ be a bit vector with $n \leq u/2$ 1-bits. Then $uH_0(B) = n\log(u/n) + \mathcal{O}(n)$.

Proof. Since $\ln x \leq x - 1$ for all $x > 0$,

$$\ln(u/(u-n)) \leq (u/(u-n)) - 1 = n/(u-n).$$

Noting that $\log x = (\log e)\ln x$, where $\log e \approx 1.44$, we get

$$\begin{aligned} uH_0(B) &= n\log\frac{u}{n} + (u-n)\log\frac{u}{u-n} \\ &\leq n\log\frac{u}{n} + (u-n)(\log e)\frac{n}{u-n} = n\log\frac{u}{n} + n\log e. \end{aligned}$$

$\square$

Thus such bit vectors with support for rank and select can be stored in $uH_0(B) = n\log(u/n) + \mathcal{O}(n) + o(u)$ bits. We used this result on slide 138.

Gap encoding is another method for compressing sparse bit vectors: Encode gaps between 1-bits using γ or δ encoding. It can be made to support rank and select too.

Summary

- We have seen how data structures with nontrivial functionality can be implemented in small additional space even when the primary data is in compressed form.

- We have seen how complex data structures can be built using a toolbox of basic components and techniques such as bit vectors with rank and select. This is not unlike traditional data structures but the toolbox is different.

- These data structures are practical: they are used in real world applications in bioinformatics, and there are a couple of libraries with implementations of the basic components (see course home page).

- All the data structures we have seen are static: they do not support operations that modify the data. There are dynamic versions of many of the data structures, including dynamic bit vectors, though the dynamicity often comes at a cost in time and/or space.