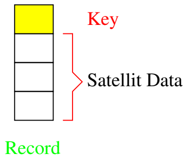
Hashing

Data Structures: Remainder

Given a universe $\mathcal{U}$, a dynamic set of records, where each record:



- Array
- Linked List (and variations)
- Stack (LIFO): Supports push and pop
- Queue (FIFO): Supports enqueue and dequeue
- Deque: Supports push, pop, enqueue and dequeue
- Heaps: Supports insertions, deletions, find Max and MIN
- Hashing

Dynamic Sets.

Given a universe $\mathcal{U}$ and a set of keys $\mathcal{S} \subset \mathcal{U}$, for any $k \in \mathcal{S}$ we can consider the following operations

- Search $(\mathcal{S}, k)$: decide if $k \in \mathcal{S}$
- Insert $(\mathcal{S}, k)$: $\mathcal{S} := \mathcal{S} \cup \{k\}$
- Delete $(\mathcal{S}, k)$: $\mathcal{S} := \mathcal{S} \backslash \{k\}$
- Minimum $(\mathcal{S})$: Returns element of $\mathcal{S}$ with smallest k
- Maximum $(\mathcal{S})$: Returns element of $\mathcal{S}$ with largest k
- Successor $(\mathcal{S}, k)$: Returns element of $\mathcal{S}$ with next larger key to k
- Predecessor $(\mathcal{S}, k)$: Returns element of $\mathcal{S}$ with next smaller key to k.

Recall Dynamic Data Structures

DICTIONARY

Data structure for maintaining $\mathcal{S} \subset \mathcal{U}$ together with operations:

- Search $(\mathcal{S}, k)$: decide if $k \in \mathcal{S}$
- Insert $(\mathcal{S}, k)$: $\mathcal{S} := \mathcal{S} \cup \{k\}$
- Delete $(\mathcal{S}, k)$: $\mathcal{S} := \mathcal{S} \backslash \{k\}$

PRIORITY QUEUE

Data structure for maintaining $\mathcal{S} \subset \mathcal{U}$ together with operations:

- Insert $(\mathcal{S}, k)$: $\mathcal{S} := \mathcal{S} \cup \{k\}$
- Maximum $(\mathcal{S})$: Returns element of $\mathcal{S}$ with largest k
- Extract-Maximum $(\mathcal{S})$: Returns and erase from $\mathcal{S}$ the element of $\mathcal{S}$ with largest k

Priority Queue

Linked Lists:

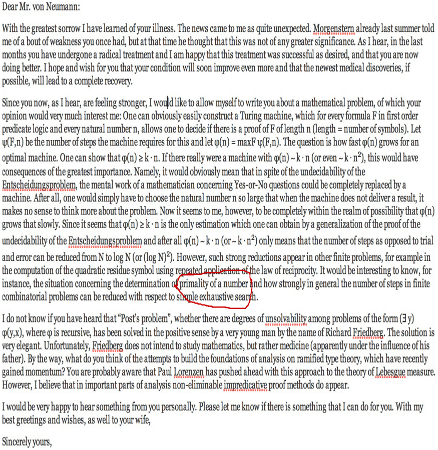
- *INSERT:* $O(n)$
- *EXTRACT-MAX:* $O(1)$

Heaps:

- *INSERT:* $O(\lg n)$
- *EXTRACT-MAX:* $O(\lg n)$

Using a Heap is a good compromise between fast insertion and slow extraction.

String Matching



Dear Mr. von Neumann:

With the greatest sorrow I have learned of your illness. The news came to me as quite unexpected. Morgenstern already last summer told me of a bout of weakness you once had, but at that time he thought that this was not of any greater significance. As I hear, in the last months you have undergone a radical treatment and I am happy that this treatment was successful as desired, and that you are now doing better. I hope and wish for you that your condition will soon improve even more and that the newest medical discoveries, if possible, will lead to a complete recovery.

Since you now, as I hear, are feeling stronger, I would like to allow myself to write you about a mathematical problem, of which your opinion would very much interest me: One can obviously easily construct a Turing machine, which for every formula F in first order predicate logic and every natural number n, allows one to decide if there is a proof of F of length n (length = number of symbols). Let $\psi(F,n)$ be the number of steps the machine requires for this and let $\varphi(n) = \max_F \psi(F,n)$. The question is how fast $\varphi(n)$ grows for an optimal machine. One can show that $\varphi(n) \geq k \cdot n$. If there really were a machine with $\varphi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$), this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. After all, one would simply have to choose the natural number n so large that when the machine does not deliver a result, it makes no sense to think more about the problem. Now it seems to me, however, to be completely within the realm of possibility that $\varphi(n)$ grows that slowly. Since it seems that $\varphi(n) \geq k \cdot n$ is the only estimation which one can obtain by a generalization of the proof of the undecidability of the Entscheidungsproblem and after all $\varphi(n) \sim k \cdot n$ (or $\sim k \cdot n^2$) only means that the number of steps as opposed to trial and error can be reduced from N to log N (or $(\log N)^2$). However, such strong reductions appear in other finite problems, for example in the computation of the quadratic residue symbol using repeated application of the law of reciprocity. It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search.

I do not know if you have heard that "Post's problem", whether there are degrees of unsolvability among problems of the form $(\exists y)\, \varphi(y,x)$, where φ is recursive, has been solved in the positive sense by a very young man by the name of Richard Friedberg. The solution is very elegant. Unfortunately, Friedberg does not intend to study mathematics, but rather medicine (apparently under the influence of his father). By the way, what do you think of the attempts to build the foundations of analysis on ramified type theory, which have recently gained momentum? You are probably aware that Paul Lorenzen has pushed ahead with this approach to the theory of Lebesgue measure. However, I believe that in important parts of analysis non-eliminable impredicative proof methods do appear.

I would be very happy to hear something from you personally. Please let me know if there is something that I can do for you. With my best greetings and wishes, as well to your wife,

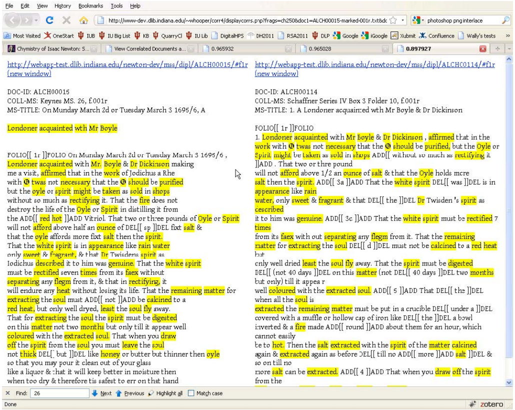
Sincerely yours,

Search: primality of a number

Given a text, find a subtext

- Given two texts, find common subtexts (plagiarism)
- Given two genomes, find common subchains (consecutive characters)

Document similarity



http://webapp-test.dlib.indiana.edu/newton-dev/mss/dipl/ALCH00015/#f1r
(new window)

DOC-ID: ALCH00015
COLL-MS: Keynes MS. 26, f.001r
MS-TITLE: On Munday March 2d or Tuesday March 3 1695/6, A

Londoner acquainted wth Mr Boyle

FOLIO[[1r]]FOLIO On Munday March 2d or Tuesday March 3 1695/6 ,
Londoner acquainted with Mr. Boyle & Dr Dickinson making
me a visit, affirmed that in the work of Jodichus a Rhe
with ⦶ twas not necessary that the ⦶ should be purified
but the oyle or spirit might be taken as sold in shops
without so much as rectifying it. That the fire does not
destroy the life of the Oyle or Spirit in distilling it from
the ADD[[red hot]]ADD Vitriol. That two or three pounds of Oyle or Spirit
will not afford above half an ounce of DEL[[sp]]DEL fixt salt &
that the oyle affords more fixt salt then the spirit.
That the white spirit is in appearance like rain water
only sweet & fragrant, & that Dr Twisdens spirit as
Iodichus described it to him was genuine. That the white spirit
must be rectified seven times from its faex without
separating any flegm from it, & that in rectifying, it
will endure any heat without losing its life. That the remaining matter for
extracting the soul must ADD[[not]]ADD be calcined to a
red heat, but only well dryed, least the soul fly away.
That for extracting the soul the spirit must be digested
on this matter not two months but only till it appear well
coloured with the extracted soul. That when you draw
off the spirit from the soul you must leave the soul
not thick DEL[[but]]DEL like honey or butter but thinner then oyle
so that you may pour it clean out of your glass
like a liquor & that it will keep better in moisture then
when too dry & therefore tis safest to err on that hand

http://webapp-test.dlib.indiana.edu/newton-dev/mss/dipl/ALCH00114/#f1r
(new window)

DOC-ID: ALCH00114
COLL-MS: Schaffner Series IV Box 3 Folder 10, f.001r
MS-TITLE: 1. A Londoner acquainted wth Mr Boyle & Dr Dickinson

FOLIO[[1r]]FOLIO
1. Londoner acquainted with Mr Boyle & Dr Dickinson , affirmed that in the
work with ⦶ twas not necessary that the ⦶ should be purified, but the Oyle or
Spirit might be taken as sold in shops ADD[[without so much as rectifying it
]]ADD . That two or thre pound
will not afford above 1/2 an ounce of salt & that the Oyle holds more
salt then the spirit. ADD[[3a]]ADD That the white spirit DEL[[was]]DEL is in
appearance like rain
water, only sweet & fragrant & that DEL[[the]]DEL Dr Twisden 's spirit as
described
it to him was genuine. ADD[[3c]]ADD That the white spirit must be rectified 7
times
from its faex with out separating any flegm from it. That the remaining
matter for extracting the soul DEL[[d]]DEL must not be calcined to a red heat
but
only well dried least the soul fly away. That the spirit must be digested
DEL[[(not 40 days]]DEL on this matter (not DEL[[40 days]]DEL two months
but only) till it appea r
well coloured with the extracted soul. ADD[[5]]ADD That DEL[[the]]DEL
when all the soul is
extracted the remaining matter must be put in a crucible DEL[[under a]]DEL
covered with a muffle or hollow cap of iron like DEL[[the]]DEL a bowl
inverted & a fire made ADD[[round]]ADD about them for an hour, which
cannot easily
be to hot. Then the salt extracted with the spirit of the matter calcined
again & extracted again as before DEL[[till no ADD[[more]]ADD salt]]DEL &
so on till no
more salt can be extracted. ADD[[4]]ADD That when you draw off the spirit
from the

Finding similar documents in the WWW

- Proliferation of almost identical documents
- Approximately 30% of the pages on the web are (near) duplicates.
- Another way to find plagiarism

Hashing functions

Data Structure that supports *dictionary* operations on an universe of numerical keys.

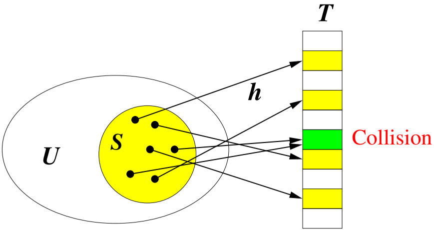
Notice the number of possible keys represented as 64-bit integers is $2^{63} = 18446744073709551616$.

Tradeoff *time/space*

Define a hashing table $T[0, \ldots, m-1]$

a hashing function $h : \mathcal{U} \rightarrow T[0, \ldots, m-1]$

Hans P. Luhn
(1896-1964)



Simple uniform hashing function.

A good hashing function must have the property that $\forall k \in \mathcal{U}$, $h(k)$ must have the same probability of ending in any $T[i]$.

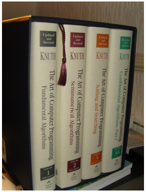
Given a hashing table T with m slots, we want to store $n = |\mathcal{S}|$ keys, as maximum.

Important measure: load factor $\alpha = n/m$, the average number of keys per slot.

The performance of hashing depends on how well h distributes the keys on the m slots: h is simple uniform if it hash any key *with equal probability* into any slot, independently of where other keys go.

How to choose h?

Advice: For an exhaustive treaty on Hashing: D. Knuth, Vol. 3 of *The Art of computing programming*



h depends on the type of key:

- If $k \in \mathbb{R}, 0 \leq k \leq 1$ we can use $h(k) = \lfloor mk \rfloor$.
- If $k \in \mathbb{R}, s \leq k \leq t$ scale by $1/(t-s)$, and use the previous methode: $h(k/(t-s)) = \lfloor mk/(t-s) \rfloor$.

The division method

Choose m prime and as far as possible from a power,

$$\boxed{h(k) = k \mod m}.$$

Fast ($\Theta(1)$) to compute in most languages ($k\%m$)!

Be aware: if $m = 2^r$ the hash does not depend on all the bits of K

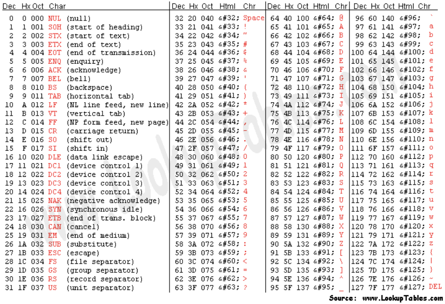
If $r = 6$ with $k = 1011000111\underbrace{011010}_{=h(k)}$

$(45530 \mod 64 = 858 \mod 64)$

- In some applications, the keys may be very large, for instance with alphanumeric keys, which must be converted to ascii:

Example: *averylongkey* is converted via ascii:

$$97 \cdot 128^{11} + 118 \cdot 128^{10} + 101 \cdot 128^{9} + 114 \cdot 128^{8} + 121 \cdot 128^{7} + 108 \cdot 126^{6} + 111 \cdot 128^{5} + 110 \cdot 128^{4} + 103 \cdot 128^{3} + 107 \cdot 128^{2} + 101 \cdot 128^{1} + 121 \cdot 128^{0} = n$$



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

which has 84-bits!

Recall mod arithmetic : for $a, b, m \in \mathbb{Z}$,
$(a+b) \bmod m = (a \bmod m + b \bmod m) \bmod m$
$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$
$a(b+c) \bmod m = ab \bmod m + ac \bmod m$
If $a \in \mathbb{Z}_m$ $(a \bmod m) \bmod m = a \bmod m$

Horner's rule: Given a specific value x_0 and a polynomial $A(x) = \sum_{i=0}^{n} a_i x^i = a_0 + a_1 X + \cdots + a_n x^n$ to evaluate $A(x_0)$ in $\Theta(n)$ steps:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots + x_0(a_{n-1} + a_n x_0)))$$

How to deal with large n

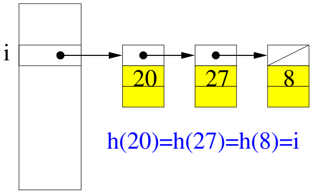
For large n, to compute $h = n \mod m$, we can use mod arithmetic + Horner's method:

$$
\begin{aligned}
&(((((((((97 \cdot 128 + 118) \cdot 128 + 101) \cdot 128 + 114) \cdot 128 + 121) \\
&\cdot 128 + 111) \cdot 128 + 110) \cdot 128 + 103) \cdot 128 + 107) \\
&\cdot 128 + 101) \cdot 128 + 121 \quad \mod m \\
&= ((((((((\underbrace{\underbrace{(97 \cdot 128 + 118 \quad \mod m)} \cdot 128) \quad \mod m + 101)} \cdot \ldots))))))
\end{aligned}
$$

Collision resolution: Separate chaining

For each table address, construct a linked list of the items whose keys hash to that address.

- Every key goes to the same slot
- Time to explore the list = length of the list



h(20)=h(27)=h(8)=i

Cost of average analysis of chaining

The cost of the dictionary operations using hashing:

- Insertion of a new key: $\Theta(1)$.
- Search of a key: $O($ length of the list$)$
- Deletion of a key: $O($ length of the list$)$.

Under the hypothesis that h is *simply uniform hashing*, each key x is equally likely to be hashed to any slot of T, independently of where other keys are hashed

Therefore, the expected number of keys falling into $T[i]$ is $\alpha = n/m$.

Cost of search

For an unsuccessful search (x is not in T) therefore we have to explore the all list at $h(x) \to T[i]$ with an the expected time to search the list at $T[i]$ is $O(1+\alpha)$.
(α of searching the list and $\Theta(1)$ of computing $h(x)$ and going to slot $T[i]$)

For an successful search search, we can obtain the same bound, (most of the cases we would have to search a fraction of the list until finding the x element.)

Therefore we have the following result: Under the assumption of simple uniform hashing, in a hash table with chaining, an unsuccessful and successful search takes time $\Theta(1+\frac{n}{m})$ on the average.

Notice that if $n = \theta(m)$ then $\alpha = O(1)$ and search time is $\Theta(1)$.

Universal hashing: Motivation

For every deterministic hash function, there is a set of bad instances.

An adversary can arrange the keys so your function hashes most of them to the same slot.

Create a set $\mathcal{H}$ of hash functions on $\mathcal{U}$ and choose a hashing function at random and independently of the keys.

Must be careful once we choose one particular hashing function for a given key, we always use the same function to deal with the key.

Universal hashing

Let $\mathcal{U}$ be the universe of keys and let $\mathcal{H}$ be a collection of hashing functions with hashing table $T[0, \dots, m-1]$, $\mathcal{H}$ is universal if $\forall x, y \in \mathcal{U}, x \neq y$, then

$$|\{h \in \mathcal{H} \mid h(x) = h(y)\}| \leq \frac{|\mathcal{H}|}{m}.$$

In an equivalent way, $\mathcal{H}$ is *universal* if $\forall x, y \in \mathcal{U}, x \neq y$, and for any h chosen uniformly from $\mathcal{H}$, we have

$$\mathbf{Pr}\left[h(x) = h(y)\right] \leq \frac{1}{m}.$$

Universality gives good average-case behaviour

Theorem

If we pick a u.a.r. h from a universal $\mathcal{H}$ and build a table using and hash n keys to T with size m, for any given key x let Z_x be a random variable counting the number of collisions with others keys y in T.

$$\mathbf{E}\left[\#\textit{collisions}\right] \leq n/m.$$

Construction of a universal family: $\mathcal{H}$

To construct a family $\mathcal{H}$ for $N = \max\{\mathcal{U}\}$ and $T[0, \dots, m-1]$:

- *$\mathcal{H} = \emptyset$.*
- *Choose a prime p, $N \leq p \leq 2N$. Then $\mathcal{U} \subset \mathbb{Z}_p = \{0, 1, \dots, p-1\}$.*
- *Choose independently and u.a.r. $a \in \mathbb{Z}_p^+$ and $b \in \mathbb{Z}_p$. Given a key x define $h_{a,b}(x) = \underbrace{((ax + b) \mod p)}_{g_{a,b}(x)} \mod m$.*
- $\mathcal{H} = \{h_{a,b} | a, b \in \mathbb{Z}_p, a \neq 0\}$.

Example: $p = 17, m = 6$ we have $\mathcal{H}_{17,6} = \{h_{a,b} : a \in \mathbb{Z}_p^+, b \in \mathbb{Z}_p\}$
if $x = 8$, $a = 3, b = 4$ then
$h_{3,4}(8) = ((3 \cdot 8 + 4) \mod 17) \mod 6 = 5$

Properties of $\mathcal{H}$

1. $h_{ab} : \mathbb{Z}_p \to \mathbb{Z}_m$.
2. $|\mathcal{H}| = p(p-1)$. (We can select a in $p-1$ ways and b in p ways)
3. Specifying an $h \in \mathcal{H}$ requires $O(\lg p) = O(\lg N)$ bits.
4. To choose $h \in \mathcal{H}$ select a, b independently and u.a.r. from $\mathbb{Z}_p^+$ and $\mathbb{Z}_p$.
5. Evaluating $h(x)$ is fast.

Theorem

The family $\mathcal{H}$ is universal.

For the proof:

Chapter 11 of Cormen. Leiserson, Rivest, Stein: *An introduction to Algorithms*

Bloom filter

Given a set of elements S, we want a Data structure for supporting insertions and querying about membership in S.

In particular we wish a DS s.t.

- *minimizes* the use of memory,
- *can check membership* as fast as possible.

Burton Bloom: The Bloom filter data structure. Comm. ACM, July 1970.
A hash data structure where each register in the table is one bit

Query on a list of e-mails

We have a set S of 10^9 e-mail addresses, where the typical e-mail address is 20 bites. Therefore it does not seem reasonable to store S in main memory. We can spare 1 Gigabyte of memory, which is approximately 10^9 bytes or 8×10^9 bites. How can put S in main memory to query it?

Definition Bloom filter

Create a one bit hash table $T[0, \dots, m-1]$, and a hash function h. Initially all m bits are set to 0.

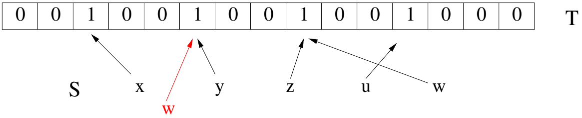
Giving a set $S = \{x_1, \dots, x_n\}$ define a hashing function $h : S \to T$.
For every $x_i \in S$, $h(x_i) \to T[j]$ and $T[j] := 1$.
Given a set S a function $h()$ and a table $T[m]$:

```
Insert ($x$)
$h(x) \to i$
if  $T[i] == 0$  then
    $T[i] = 1$
end if
```

```
inS($y$)
$h(x) \to i$
if  $T[i] == 1$  then
    return  Yes
else
    return  No
end if
```

Notice: once we have hashed S into T we can erase S.

False positives



0	0	1	0	0	1	0	0	1	0	0	1	0	0	0

Bloom filter needs $O(m)$ space and answers membership queries in $\Theta(1)$.

Inconvenience: Do not support removal and may have false positive.

In a query $y \in S$?, a Bloom filter always will report correctly if indeed $y \in S$ $(h(y) \rightarrow T[i]$ with $T[i] = 1)$,
but if $y \notin S$ it may be the case that $h(y) \rightarrow T[i]$ with $T[i] = 1$,
which is called a False positive.

How large is the error of having a false positive?

Probability of having a false positives

Let $|S| = n$, we constructed a BF $(h, T[m])$ with all elements in S. If we query about $y \in S$?, with $y \notin S$, and $h(y) \to T[i]$, what is the probability that $T[i] = 1$?

After all the elements of S are hashed into the Bloom filter, the probability that a specific $T[i] = 0$ is $(1 - \frac{1}{m})^n = e^{-n/m}$

(recall that: $e = \lim_{x\to\infty}(1 + \frac{1}{x})^x$, $e^{-1} = \lim_{x\to\infty}(1 - \frac{1}{x})^x$)

Therefore, for a $y \notin S$, the probability of false positive π:

$$\pi = \mathbf{Pr}\left[h(y) \to T[i] \,|\, \text{where } T[i] = 1\right] = 1 - (1 - \frac{1}{m})^n \sim 1 - e^{-n/m}.$$

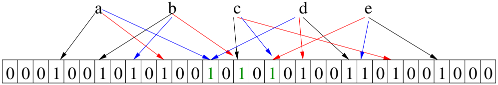
To minimise π, want to maximize $e^{-n/m}$
$\Rightarrow \frac{n}{m}$ has to be small, i.e, $m >> n$.

For ex.: if $m = 100n, \pi = 0.0095$; If $m = n, \pi = 0.632$ and if $m = n/10, \pi = 0.9999$

Alternative: Amplify

Take k different functions $\{h_1, h_2, \ldots, h_k\}$ in the same 2-universal set of functions.

Ex. Bloom filter with 3 hash functions: h_1, h_2, h_3.



When making a query about if $y \in S$, compute $h_1(y), \ldots h_t(y)$, if one of them is 0 we certainty $y \notin S$, else (if all the k hashing go to bits with value 1) $y \in S$ with some probability.
After hashing the n elements k times to T, for an specific $T[i]$:

$$p = \mathbf{Pr}\left[T[i] = 0\right] = (1 - \frac{1}{m})^{kn} = e^{-kn/m}.$$

The probability f of a false positive:

$$f = \left(1 - e^{-kn/m}\right)^k = (1 - p)^k$$

Asymptotic estimations for k and m

To minimize the probability of having a false positive: $\frac{\mathrm{d}p}{\mathrm{d}k} = 0$
Let $f(k) = \ln p$ then $f(k) = k \ln(1 - e^{-kn/m})$
$\Rightarrow f'(k) = \ln(1 - e^{-kn/m}) + \frac{kne^{-kn/m}}{m(1-e^{-kn/m})}$
Making $f'(k) = 0$, we get

$$\boxed{k_{\mathrm{opt}} = \frac{m}{n}\frac{1}{2}\ln 2 = \frac{9}{13}\frac{m}{n}}$$

The probability of having a false positive for k_{opt} is

$$p_0 = (1 - e^{\frac{9}{13}\frac{m}{n}\frac{n}{m}})^{\frac{9}{13}\frac{m}{n}} \sim (\frac{1}{2})^{\frac{9m}{13n}} = 0.619223^{\frac{m}{n}}.$$

Optimizing k

Given n and m we want to find the optimal value of k to minimize the probability of a false positive $f(k) = (1 - e^{-kn/m})^k$

Define $g(k) = \ln f(k) = k \ln(1 - e^{-kn/m})$. Minimizing f is equivalent to minimizing g.

To minimize the probability of having a false positive: $\frac{\mathrm{d}g(k)}{\mathrm{d}k} = 0$

$\Rightarrow \frac{\mathrm{d}g(k)}{\mathrm{d}k} = \ln(1 - e^{-kn/m}) + \frac{kne^{-kn/m}}{m(1-e^{-kn/m})} = 0,$
$\Rightarrow$ when n, m are given, to minimize f is $k_o = (\ln 2)\frac{m}{n}$.

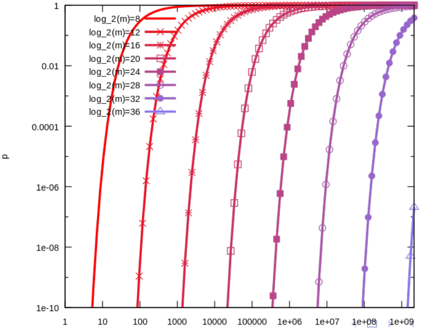
In this case the false positive probability $f_o = 0.6185^{m/n}$.

Bloom filters allow a constant probability of false positive, $m = cn$ for small constant c, i.e. m grows linear wrt n.

For ex.: if $c = 2$ and $k = 6$ the false positive probability is around 2%.

Practical issues

On the other hand although the results shown before are asymptotic, there also work for practical values of n. Figure in the side table give the probability of false positive (y) wrt to n (x), and as function of m, with $k = \ln 2\frac{n}{m}$.



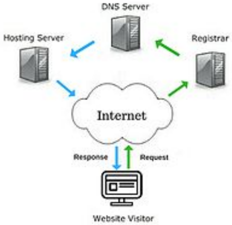
Another application of Bloom filters: Caching structures

Recall: http (Hypertext transfer protocol) basic network protocol to dristributed information on the WWW net. (Tim Berners-Lee (1990))

HTML (HyperText Markup Language) is the standard language for creating web pages and web applications.

URL (Uniform Resource Locator) web address indicating for example web pages. `http://www.cs.upc.edu/∼diaz`

Web server is a computer system that processes requests using http to deliver web pages to clients.

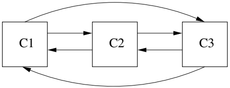


Web cache is a technology for temporary storage of web documents (html pages, images,..) which aim to reduce bandwidth, server load and lag (latency).

Another application of Bloom filters: Caching structures

Suppose we have a set $\mathcal{U}$ with n URL, each one with 100 characters, i.e in total we have 800n bits.

Consider caches C_1, C_2, C_3, each with documents indexed by their URL.
A query for URL x is sent to one of the caches, that cache must determine which of the caches has x (if x is there)

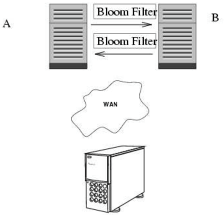


If every C_i stores 10000 documents, that means about 48000000 bits can be exchanged.

Bloom filters may help to reduce the transfer of bits, accepting a small marge of error.

Another application of Bloom filters: Caching structures

- Each proxy all of the URLs in its cache into Bloom Filter.
- Proxies periodically exchange Bloom filters, so queries of other caches can be made locally without sending ICP message.



Cache filtering

Using a Bloom filter to prevent one-hit-wonders from being stored in a web cache decreased the rate of disk writes by nearly one half, reducing the load on the disks and potentially increasing disk performance.

Nearly three-quarters of the URLs accessed from a typical web cache are one-hit-wonders accessed by users only once and never again.

To prevent caching one-hit-wonders, a Bloom filter is used to keep track of all URLs that are accessed by users.

A web object is cached only when it has been accessed at least once before.

Further applications of Bloom filters

Bloom filters are useful when a set of keys is used and space is important.

- The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result)
- Packet routing: Bloom filters provide a means to speed up or simplify packet routing protocols.
- IP Tracebook
- Useful tool for measurement infrastructures used to create data summaries in routers or other network devices.

A. Broder, M. Mitzenmacher: *Network applications of Bloom filters: A survey.* Internet Mathematics, 1,4: 485-509, 2005

Cuckoo Hashing

Pagh, Rodler: *Cuckoo Hashing*. ESA-2001

Cuckoo hashing is a hashing technique where:

- Lookups are $\Theta(1)$ worst-case.
- Deletions are $\Theta(1)$ worst-case.
- Insertions are $O(1)$ in expectation.

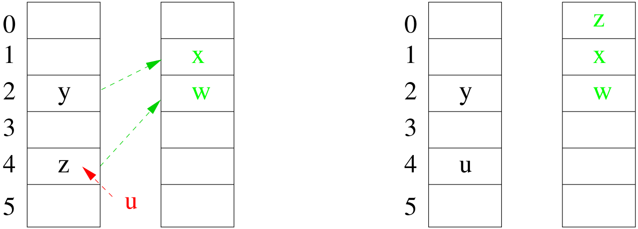
Cuckoo Hashing

- We have two hash tables T_1, T_2 with size m each and two hash functions h_1 for T_1 and h_2 for T_2.
- Can use for instance $h_1(k) = k \mod m$ and $h_2(k) = \lceil k/m \rceil \mod m$
- Every element $k \in \mathcal{U}$ can be only in two positions: at $h_1(k)$ in T_1 or at $h_2(k)$ in T_2.
- Lookups take $\Theta(1)$ because we only need to check 2 positions.
- Deletions take $\Theta(1)$ because we only need to check 2 positions.
- To insert $k \in \mathcal{U}$, try $h_1(k)$, if the slot is empty put k there, if the slot contains k', kick out the k', k stay there, and k' repeats the behavior of k on T_2.
- Repeat this process, bouncing between tables, until all elements stabilize.

Cuckoo Hashing: Long cycles of insertion

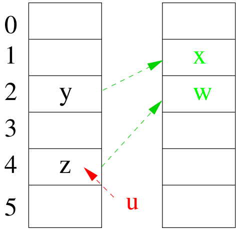
One complication is that the cuckoo may loop for ever. The probability of such an event is small. In such a case choose an upper bound in the number of slot exchanges, and if it exceeds, do a rehash: choose new functions and start .

Example: We have $\{x, y, w, z, u\}$
$h_1(x) = 2; h_1(y) = 2; h_1(w) = 4; h_1(z) = 4, h_1(u) = 4$
$h_2(x) = 1; h_2(y) = 1; h_2(w) = 2; h_2(z) = 0, h_2(u) = 2$



Cuckoo Hashing: Long cycles of insertion

What happens if
$h_1(x) = 2; h_1(y) = 2; h_1(w) = 4; h_1(z) = 4, h_1(u) = 4$
$h_2(x) = 1; h_2(y) = 1; h_2(w) = 2; h_2(z) = 0, h_2(u) = 2$?

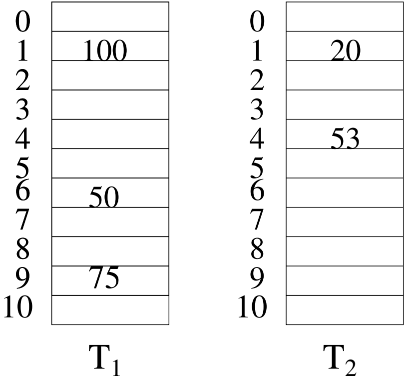


If insertion gets into a cycle, we perform a rehash: choose new h_1, h_2 and insert all elements back into the table.

Cuckoo Hashing: An example

We wish to hash the set of keys:(20, 50, 53, 75, 100, 67, 105, 3, 36, 39, 6) using $h_1(k) = k \mod 11$ and $h_2(k) = \lfloor \frac{k}{11} \rfloor \mod 11$.

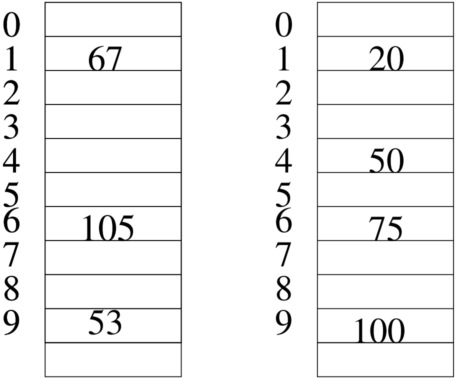
	h_1	h_2
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3
6	6	0



	T_1	T_2
0		
1	100	20
2		
3		
4		53
5		
6	50	
7		
8		
9	75	
10		

Cuckoo Hashing: An example

	h_1	h_2
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3
6	6	0

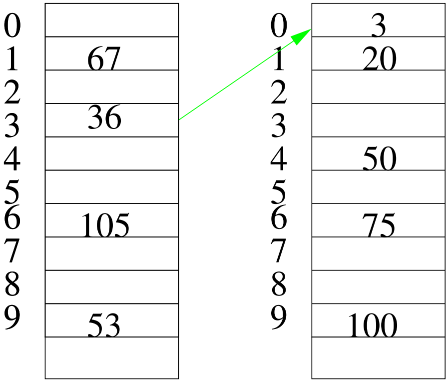


0	
1	67
2	
3	
4	
5	
6	105
7	
8	
9	53

0	
1	20
2	
3	
4	50
5	
6	75
7	
8	
9	100

Cuckoo Hashing: An example

	h_1	h_2
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3
6	6	0

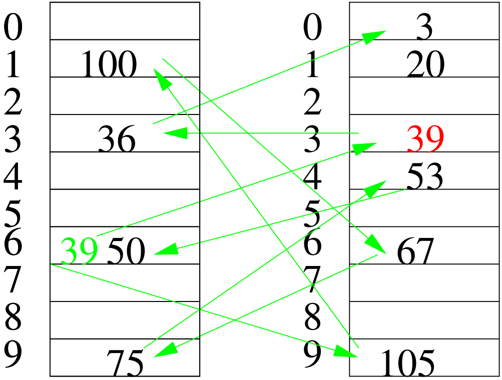


0	
1	67
2	
3	36
4	
5	
6	105
7	
8	
9	53

0	3
1	20
2	
3	
4	50
5	
6	75
7	
8	
9	100

Cuckoo Hashing: An example

	h_1	h_2
20	9	1
50	6	4
53	9	4
75	9	6
100	1	9
67	1	6
105	6	9
3	3	0
36	3	3
39	6	3
6	6	0



With 6 we have to rehash!!!

Complexity

Cuckoo hashing has a complexity:

- *Search an element x:* constant worst case complexity (x only can be in the 2 positions $h_1(x)$ or in $h_2(x)$)
- *Delete an element:* constant worst case complexity (look at the 2 positions and erase the element)
- *Inserte an element:* expected constant complexity.

It is a simple alternative to perfect matching, to implement a dictionary with reasonable space and constant searching time.

Other models, for example d-hashing tables.

String matching

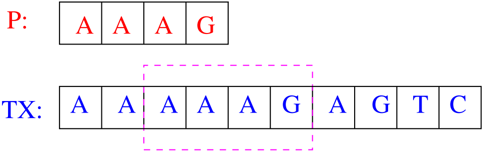
The string matching problem: given a text TX$[1 \ldots n]$ an a pattern P$[1 \ldots \ell]$, where elements of TX and P are draw from the same alphabet Σ, we wish to find all the occurrences of P in TX, together with the position they start to occur.

TX: a b c a b a a b c a b a b a a c b a a b a b
P: a b a a
TX: a b c a b a a b c a b a b a a c b a a b a b

Given a string x and y:
$|x|$ its length
$x||y$ (or xy) its concatenation with length $|x| + |y|$

Naive algorithm

```
**Search** (TX,P)
for $i = 1$ to $n - \ell$ do
    if P $T[1, \ldots, \ell] = \mathrm{TX}[i, \ldots, i + \ell - 1]$ then
        print P occurs at $i$
    end if
end for
```



This algorithm has complexity $\Theta((n - \ell + 1)\ell)$, worst case $O(n^2)$

Rolling Hashing

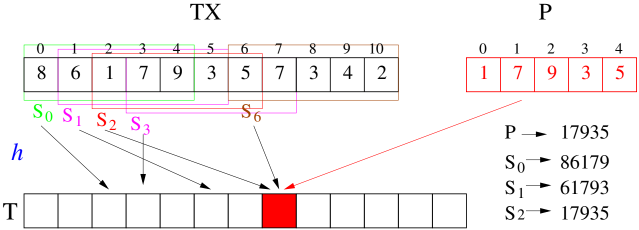
Use Hashing D.Karp, M. Rabin: *Efficient randomized patter matching algorithms*. IBM JRD,1987.

Given TX ($|\mathrm{TX}| = n$) and pattern P ($|\mathrm{P}| = \ell$), want to indicate define a hash function h a table $T[0, \ldots, m-1]$.

Notice each symbol in TX is a key. Wlog consider alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

General idea of Karp-Rabin's hashing algorithm

Idea: Break TX into overlapping substring of length $= \ell$, $S_0, S_1, \ldots S_i, \ldots$ and compute the decimal value of each substring S_i and of P.



Brute force implementation of the algorithm

Let s_i denote the decimal value of S_i and p the decimal value of P. Use Horner's rule to compute p in time $\Theta(\ell)$:

$$p = P[\ell-1] + 10(P[\ell-2] + \cdots + (10P[0]))\cdots)$$

In the same way, use Horner's rule to compute for $0 \le i < n$:

$$\begin{aligned} s_i =& S_i[i]10^{\ell-1} + S_i[i+1]10^{\ell-2} + \cdots + S_i[i+\ell-2]10^1 + S_i[i+\ell-1]10^0 \\ =& S_i[i+\ell-1]1 + 10(S_i[i+\ell-2] + \cdots + 10(S_i[i]1))\cdots). \end{aligned}$$

Brute force implementation

- At the beginning all registers to 0.
- Hash P$\rightarrow$ T $h(p) = p \mod m$, if $h(P) = i$ then $T[i] := 1$
- Run through TX, hashing each set of ℓ consecutive characters into T
- If one of them goes to a $T[i]$ ($T[i] = 1$), double check that the ℓ S_k match P (i.e. $s_k - p = 0$)

Complexity: $O(n\ell)$, where ℓ could be $\Theta(n)$.

Rolling Hash

Instead of looking to $O(n)$ substrings independently, we may take advantage the substrings have a lot of overlap:
$s_i = 79357 \rightarrow s_{i+1} = 93573 \rightarrow s_{i+2} = 35734$

$$s_{i+1} = \underbrace{S_{i+1}[i+1]10^{\ell-1} + S_{i+1}[i+2]10^{\ell-1} + \cdots + S_{i+1}[i+\ell-1]10^{1}}_{(S_i \setminus \{S_i[i]\})*10}$$
$$+ S_{i+1}[i+\ell]10^{0}$$

Knowing s_i to get s_{i+1} with we only have to deal with the element leaving $(S_i[i])$ and the element incorporating $(S_{i+\ell})$:

$$s_{i+1} = (s_i - (S_i[i] * 10^{\ell})) * 10 + S_{i+1}[i+\ell]$$

Rolling Hash

Recall mod magic: for $a, b, m \in \mathbb{Z}$,
$(a + b) \bmod m = (a \bmod m + b \bmod m) \bmod m$
$(a \cdot b) \bmod m = ((a \bmod m) \cdot (b \bmod m)) \bmod m$
If $a, b \in \mathbb{Z}_m$ and $b > a$, $(a - b) \bmod m = (m - (a - b)) \bmod m$
$(a \bmod m) \bmod m = a \bmod m$

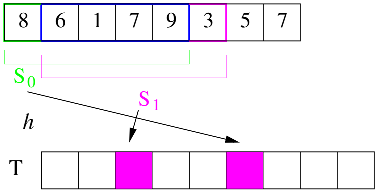
Using the hash function $h(a) = a \bmod m$, for any $a \in \mathbb{N}$

$$h(s_{i+1}) = ((s_i - (S_i[i] * 10^{\ell})) * 10 + S_{i+1}[i + \ell]) \bmod m$$

$$h(s_{i+1}) = ((\underbrace{h(s_i)}_{\text{known}} -(\underbrace{S_i[i]) \bmod m}_{S_i[i]} * \underbrace{10^{\ell}) \bmod m}_{\text{pre-comp.}}))*10+S_{i+1}[i+\ell])) \bmod m$$

Therefore given $h(s_i)$ we can compute $h(s_{i+1})$ in $\Theta(1)$ steps.

Example



TX=861793, $m = 73$,

Preprocess: $h(86179) = 39$ and $10^4 \bmod 73 = 72$.

$$\begin{aligned} h(61793) &= ((86179 - 8 \cdot 10^4) \cdot 10 + 3) \bmod 73 \\ &= (((h(86179) - (8 \cdot 10^4) \bmod 73)10) \bmod 73 + 3) \bmod 73 \\ &= ((47 \cdot 10) \bmod 73 + 3) \bmod 73 = 35 \end{aligned}$$

Karp-Rabin Algorithm

Given a text $|\text{TX}| = n$ pattern $P = \ell$, hash table $|T| = m$, hash function $h = * \mod m$:

```
Karp-Rabin (TX, P, T)
$p = 0; s_0 = 0; q = 10^{\ell-1} \mod m$
for $j = 0$ to $\ell - 1$ do
  $h(p) = (10p + P[j]) \mod m$
  $h(s_0) = (10s_0 + \text{TX}[j] \mod m$
end for
for $i = 0$ to $n - \ell$ do
  if $h(p) == h(s_i)$ then
    if $P[0 \dots \ell - 1] == \text{TX}[i \dots i + \ell - 1]$ then
      return Match at $i$
    end if
  else
    $h(s_{i+1}) = (10(s_i - T[i+1]q) + T[i + \ell + 1]) \mod m$
  end if
end for
```

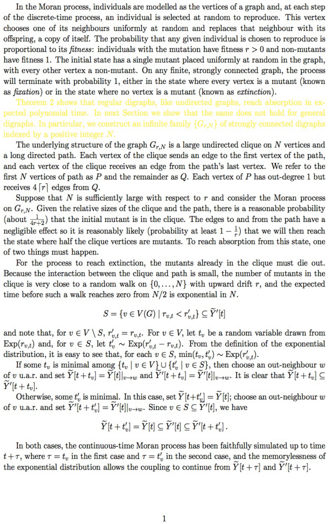
Complexity

- To use any other radix $d \neq 10$ it behaves the same as radix-10. We has to substitute 10 by d.
- Using rolling hash we could speed the computation of the hash function of each ℓ-string to $\Theta(1)$, once we compute the first one in $O(\ell)$
- The total complexity depends of the number of comparisons. Each comparison takes $\Theta(\ell)$.
- If TX and P are such that the algorithm must make $\Theta(n)$ comparisons, the total complexity is $\Theta(n\ell)$
- In most practical applications (genomics, text searching, etc.), string searching using Karp-Rabin takes $O(n + \ell) = O(n)$.

Complexity

- Regarding collisions from hashing different substrings, we must choose m a large prime integer, which fits into a computer word and make sure it keeps basic operations constants. For instance, if $m = O(n)$ then the expected number of collisions is $\Theta(1)$ collision in each slot, if $m = O(n^2)$ we expect $O(1/n)$ number of collisions per cell, which is nice, but at expenses of having a very large T.
- There is a fast algorithm for string matching Knuth-Morris-Pratt $\Theta(n)$. But the simplicity of Karp-Rabin and the easiness to generalize to non-textual applications, makes K-R a good choice, widely used in practice.

Common substring problem



In the Moran process, individuals are modelled as the vertices of a graph and, at each step of the discrete-time process, an individual is selected at random to reproduce. This vertex chooses one of its neighbours uniformly at random and replaces that neighbour with its offspring, a copy of itself. The probability that any given individual is chosen to reproduce is proportional to its *fitness*: individuals with the mutation have fitness $r > 0$ and non-mutants have fitness 1. The initial state has a single mutant placed uniformly at random in the graph, with every other vertex a non-mutant. On any finite, strongly connected graph, the process will terminate with probability 1, either in the state where every vertex is a mutant (known as *fixation*) or in the state where no vertex is a mutant (known as *extinction*).

Theorem 2 shows that regular digraphs, like undirected graphs, reach absorption in expected polynomial time. In next Section we show that the same does not hold for general digraphs. In particular, we construct an infinite family $\{G_{r,N}\}$ of strongly connected digraphs indexed by a positive integer N.

The underlying structure of the graph $G_{r,N}$ is a large undirected clique on N vertices and a long directed path. Each vertex of the clique sends an edge to the first vertex of the path, and each vertex of the clique receives an edge from the path's last vertex. We refer to the first N vertices of path as P and the remainder as Q. Each vertex of P has out-degree 1 but receives $4\lceil r \rceil$ edges from Q.

Suppose that N is sufficiently large with respect to r and consider the Moran process on $G_{r,N}$. Given the relative sizes of the clique and the path, there is a reasonable probability (about $\frac{1}{4r+2}$) that the initial mutant is in the clique. The edges to and from the path have a negligible effect so it is reasonably likely (probability at least $1-\frac{1}{r}$) that we will then reach the state where half the clique vertices are mutants. To reach absorption from this state, one of two things must happen.

For the process to reach extinction, the mutants already in the clique must die out. Because the interaction between the clique and path is small, the number of mutants in the clique is very close to a random walk on $\{0,\ldots,N\}$ with upward drift r, and the expected time before such a walk reaches zero from $N/2$ is exponential in N.

$$S = \{v \in V(G) \mid r_{v,t} < r'_{v,t}\} \subseteq \widetilde{Y}'[t]$$

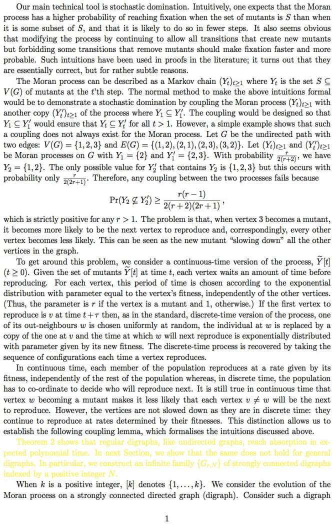
and note that, for $v \in V \setminus S$, $r'_{v,t} = r_{v,t}$. For $v \in V$, let t_v be a random variable drawn from $\mathrm{Exp}(r_{v,t})$ and, for $v \in S$, let $t'_v \sim \mathrm{Exp}(r'_{v,t} - r_{v,t})$. From the definition of the exponential distribution, it is easy to see that, for each $v \in S$, $\min(t_v, t'_v) \sim \mathrm{Exp}(r'_{v,t})$.

If some t_v is minimal among $\{t_v \mid v \in V\} \cup \{t'_v \mid v \in S\}$, then choose an out-neighbour w of v u.a.r. and set $\widetilde{Y}[t+t_v] = \widetilde{Y}[t]|_{v\to w}$ and $\widetilde{Y}'[t+t_v] = \widetilde{Y}'[t]|_{v\to w}$. It is clear that $\widetilde{Y}[t+t_v] \subseteq \widetilde{Y}'[t+t_v]$.

Otherwise, some t'_v is minimal. In this case, set $\widetilde{Y}[t+t'_v] = \widetilde{Y}[t]$; choose an out-neighbour w of v u.a.r. and set $\widetilde{Y}'[t+t'_v] = \widetilde{Y}'[t]|_{v\to w}$. Since $v \in S \subseteq \widetilde{Y}'[t]$, we have

$$\widetilde{Y}[t+t'_v] = \widetilde{Y}[t] \subseteq \widetilde{Y}'[t] \subseteq \widetilde{Y}'[t+t'_v]\,.$$

In both cases, the continuous-time Moran process has been faithfully simulated up to time $t+\tau$, where $\tau = t_v$ in the first case and $\tau = t'_v$ in the second case, and the memorylessness of the exponential distribution allows the coupling to continue from $\widetilde{Y}[t+\tau]$ and $\widetilde{Y}'[t+\tau]$.



Our main technical tool is stochastic domination. Intuitively, one expects that the Moran process has a higher probability of reaching fixation when the set of mutants is S than when it is some subset of S, and that it is likely to do so in fewer steps. It also seems obvious that modifying the process by continuing to allow all transitions that create new mutants but forbidding some transitions that remove mutants should make fixation faster and more probable. Such intuitions have been used in proofs in the literature; it turns out that they are essentially correct, but for rather subtle reasons.

The Moran process can be described as a Markov chain $(Y_t)_{t\geq 1}$ where Y_t is the set $S \subseteq V(G)$ of mutants at the t'th step. The normal method to make the above intuitions formal would be to demonstrate a stochastic domination by coupling the Moran process $(Y_t)_{t\geq 1}$ with another copy $(Y'_t)_{t\geq 1}$ of the process where $Y_1 \subseteq Y'_1$. The coupling would be designed so that $Y_1 \subseteq Y'_1$ would ensure that $Y_t \subseteq Y'_t$ for all $t > 1$. However, a simple example shows that such a coupling does not always exist for the Moran process. Let G be the undirected path with two edges: $V(G) = \{1,2,3\}$ and $E(G) = \{(1,2),(2,1),(2,3),(3,2)\}$. Let $(Y_t)_{t\geq 1}$ and $(Y'_t)_{t\geq 1}$ be Moran processes on G with $Y_1 = \{2\}$ and $Y'_1 = \{2,3\}$. With probability $\frac{r}{2(r+2)}$, we have $Y_2 = \{1,2\}$. The only possible value for Y'_2 that contains Y_2 is $\{1,2,3\}$ but this occurs with probability only $\frac{r}{2(2r+1)}$. Therefore, any coupling between the two processes fails because

$$\Pr(Y_2 \not\subseteq Y'_2) \geq \frac{r(r-1)}{2(r+2)(2r+1)}\,,$$

which is strictly positive for any $r > 1$. The problem is that, when vertex 3 becomes a mutant, it becomes more likely to be the next vertex to reproduce and, correspondingly, every other vertex becomes less likely. This can be seen as the new mutant "slowing down" all the other vertices in the graph.

To get around this problem, we consider a continuous-time version of the process, $\widetilde{Y}[t]$ ($t \geq 0$). Given the set of mutants $\widetilde{Y}[t]$ at time t, each vertex waits an amount of time before reproducing. For each vertex, this period of time is chosen according to the exponential distribution with parameter equal to the vertex's fitness, independently of the other vertices. (Thus, the parameter is r if the vertex is a mutant and 1, otherwise.) If the first vertex to reproduce is v at time $t+\tau$ then, as in the standard, discrete-time version of the process, one of its out-neighbours w is chosen uniformly at random, the individual at w is replaced by a copy of the one at v and the time at which w will next reproduce is exponentially distributed with parameter given by its new fitness. The discrete-time process is recovered by taking the sequence of configurations each time a vertex reproduces.

In continuous time, each member of the population reproduces at a rate given by its fitness, independently of the rest of the population whereas, in discrete time, the population has to co-ordinate to decide who will reproduce next. It is still true in continuous time that vertex w becoming a mutant makes it less likely that each vertex $v \neq w$ will be the next to reproduce. However, the vertices are not slowed down as they are in discrete time: they continue to reproduce at rates determined by their fitnesses. This distinction allows us to establish the following coupling lemma, which formalises the intuitions discussed above.

Theorem 2 shows that regular digraphs, like undirected graphs, reach absorption in expected polynomial time. In next Section, we show that the same does not hold for general digraphs. In particular, we construct an infinite family $\{G_{r,N}\}$ of strongly connected digraphs indexed by a positive integer N.

When k is a positive integer, $[k]$ denotes $\{1,\ldots,k\}$. We consider the evolution of the Moran process on a strongly connected directed graph (digraph). Consider such a digraph

Common substring problem

Given two texts Tx_1 and Tx_2, with $|Tx_1| = |Tx_2| = n$ discover if they share a common substring of length ℓ. Define h and $T[0 \cdots m-1]$ and use rolling hash (notice blancs should be considered as an extra symbol):

1. Hash the first substring of length ℓ in Tx_1 to T. $(O(\ell))$
2. Use rolling hash to compute the subsequent $n-1$ substring in Tx_1, hashing each one to T. $(O(n))$
3. Hash the first substring of length ℓ in Tx_2 to T. $(O(\ell))$
4. Use rolling hash to compute the subsequent $n-1$ substring in Tx_2, hashing each one to T. For each substring, check if there are collisions with substrings from Tx_1. $(O(n))$
5. If a substring of T_1 collide with a substring of T_2 do a string comparison on those substrings. $(O(\ell))$

If the number of collisions should be small the complexity is $O(n)$.
But for large number of collisions it could be $O(n^2)$.