CSE 589 Part VII

If you try to optimize everything, you will always be unhappy.
-- Don Knuth

Local Search

Local Search Algorithms

General Idea:

- Start with a solution (not necessarily good one)
- Repeatedly try to perform modifications to the current solution to improve it

Use simple local changes.

Local Search Procedure for TSP

Example: Start with TSP tour, repeatedly perform **Swap**, if it improves solution (Swap sometimes called 2-opt)

Call this Greedy Local Search

Does Greedy Local Search Lead Eventually To Optimal Tour?

No.

Solution Spaces

- Solution Space: set of all solutions to a search process and ways one can move from one solution to another.
- Represent process using a graph: a vertex for each possible solution, an edge from solution to solution if a local move can take you from one to other.
- Key question: how to choose moves. Art.
- Tradeoff between small neighborhoods and large neighborhoods.

Other Types of Local Moves For TSP Used

3-Opt

Problem with local search

can get stuck in a local optimum.

To avoid this, perhaps should sometimes allow an operation that takes you to a worst solution.

Hope is to escape the local optima and find global optimum.

Simulated Annealing

Simulated Annealing

Analogy with thermodynamics:

Best crystals grown by "annealing" out their defects.

- First heat or melt material
- Then very very slowly cool to allow system to find its state of lowest energy.

Notation

Solution Space X,   x is a solution in X

Energy (x) -- measure of how good a solution x is.

Each x in X has a neighborhood.

T – temperature

Example: TSP problem; X is all possible tours (permutations). Energy(x): quality of tour (as measured by its length)

Moves for TSP (example)

Section of path removed; replaced with same cities in reverse direction

Section of path removed, placed between 2 cities on another, randomly chosen part of path

Metropolis Algorithm

```
initialize T to hot, choose starting state
do
   generate a random move
   evaluate dE (change in energy)
   if (dE < 0) then accept the move
   else accept the move with probability
            proportional to $e^{-dE/kT}$
   update T
until T is "frozen".
```

What's going on?

T big: more likely to accept big moves.

Theory:

For fixed T, probability of being in state x converges to $e^{-E(x)/T}$

For small T, probability of being in lowest energy state is highest

However, very little known theoretically

Widely used.

Cooling Schedule

Cooling schedule: function for updating T.

Typically,
power law: $a(1+bt)^c$
exponential decay: ae^{-bt}

a -- initial tolerance parameter

b -- scaling parameter, typically << 1

parameter choices chosen by experimentation.

Termination Criteria

Limit the total number of steps.

Step when there has been no improvement in cost of best tour in last m iterations.

An algorithms engineering view of Hashing Schemes and Related Topics

Slides by Andrei Broder
Alta Vista

Engineering

Engineering is the professional art of applying science to the optimum conversion of the resources of nature to the uses of humankind.

[Britannica]

Engineering

An engineer is a man who can do for a dime what any damn fool can do for a dollar.

[Nevil Shute]

Algorithms Engineering

The art and science of crafting cost-efficient algorithms.

Plan

Introduction
Standard hashing schemes
Choosing the hash function
- Universal hashing

Fingerprinting
Bloom filters
Perfect hashing

Reading

Skiena, Sections 2.1.2, 8.1.1

CLR, chapter 12

Some other good books...

Textbook: R. Sedgewick, Algorithms in C, 3rd ed, 1997.

More C: D. Hanson, C Interfaces and Implementations, 1997.

Math bottom line + references & timings: R. Baeza-Yates & G. Gonnet, Handbook of algorithms and Data Structures, 2nd ed, 1991.

THE BOOK on analysis of algorithms: Knuth, Art of Computer Programming. Vol 1, 3rd ed, 1997, Vol 3, 1973.

Dictionaries (Symbol tables)

Dictionaries are data structures for manipulating sets of data items of the form

```
item = [key, info]
```

For simplicity assume that the keys are unique. (Often not true, must deal with it.)

Some examples of dictionaries

Rolodex
- Hash function: first letter
- Supports insertions, deletions

Spelling dictionary
- System word list is fixed.
- Personal word list allows additions.
- Issues: Average case must be very fast, errors allowed, nearest neighbor searches.

Router
- Translate destination into wire number.
- Insertions and deletions are rare.
- Strict limit on the worst case.

Basic operations

`item = [key, info];` Given the `item` the `key` can be extracted or computed.

`Insert(item);`

`Delete(item);`

`Search(key);` (returns `item`)

More operations

`Init(…);`

`Exists(key);` (returns `Boolean`)

`List(…); Sort(…); Iterate(…);` (return the entire list unordered/ordered/one-at-a-time);

`Join(…);` (combine two structures);

`Nearest(key);` (returns `item`)

For our examples

Rolodex
- **`Insert; Delete; Search;`**

? **`Exists; List; Iterate; Join; Nearest;`**

Spelling dictionary (system)
- **`Exists; Nearest;`**

Router
- **`Insert; Delete; Search;`**

Implementing dictionaries

Schemes based on key comparison - keys viewed as elements of arbitrary total order
- Ordered list
- Binary search trees

Schemes based on direct key ⇒ address-in-table translation.
- Hashing
- Bloom filters

Hashing schemes - basics

We want to store N items in a table of size M, at a location computed from the key K.

Two main aspects:

Hash function
- Method for computing table index from key

Collision resolution strategy
- How to handle two keys that hash to the same index

Hash functions

Simple choice:

- Table size `M`
- Hash function `h(K) = K mod M;`

Works fine if keys are random integers.

Example: 20 random keys in [1..100]

```
[56, 82, 87, 39, 98, 86, 69, 22, 99, 61,
 64, 50, 77, 75, 8, 62, 17, 10, 71, 58]
```

…hashed in a table of size 20

```
[16, 2, 7, 19, 18, 6, 9, 2, 19, 1, 4, 10,
 17, 15, 8, 2, 17, 10, 11, 18]
```

Why do collisions happen?

Birthday paradox: expected number of random insertions until the first collision is only

```
sqrt(π*M/2)
```

Examples:

```
M = 100       sqrt(π*M/2) ~  12
M = 1000      sqrt(π*M/2) ~  40
M = 10000     sqrt(π*M/2) ~ 125
```

Separate chaining

Basic method: keep a linked list for each table slot.

Advantages:

- Simple, widely used (maintainability)

Disadvantages:

- Wastes space, must deal with memory allocation.

Example

Input:

```
[56, 82, 87, 39, 98, 86, 69, 22, 99, 61,
 64, 50, 77, 75, 8, 62, 17, 10, 71, 58]
```

Hash table:

```
0: [50, 10]          5: [75]
1: [61, 71]          6: [56, 86]
2: [82, 22, 62]      7: [87, 77, 17]
3: []                8: [98, 8, 58]
4: [64]              9: [39, 69, 99]
```

Performance

Insert cost: `1`
Average search cost (hit): `1+(N-1)/(2 M)`
Average search cost (miss): `1+N/M`
Worst case search cost: `N+1`
Expected worst case search cost (`n=m`):
`~log n/log log n`
Space requirements:

- `(N + M) * |link| + N*|key| + N*|info|`

Deletions: easy
Adaptation (new hash function): easy

Embellishments

Keep lists sorted:

- Average insert cost: `1+N/(2 M)`
- Average search cost (hit): `2+(N-1)/(2 M)`
- Average search cost (miss): `1+N/(2 M)`

Move-to-front / transpose

- Last item accessed in a list becomes the first or moves one closer (Self adjusting hashing)

Store lists as a binary search tree:

- Improves expected worst case

Open addressing

No links, all keys are in the table.

When searching for `K`, check locations `r`$_1$`(K)`, `r`$_2$`(K)`, `r`$_3$`(K)`, … until either

- `K` is found; or
- we find an empty location (`K` not present)

Various flavors of open addressing differ in which probe sequence they use.

Random probing -- each `r`$_i$ is random. (Impractical)

Linear probing

When searching for `K`, check locations `h(K)`, `h(K)+1`, `h(K)+2`, … until either

- `K` is found; or
- we find an empty location (`K` not present)

If table is very sparse, almost like separate chaining.

When table starts filling, we get clustering but still constant average search time.

Full table ⇒ infinite loop.

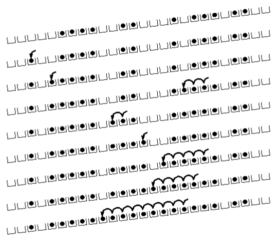
Primary clustering phenomenon

Once a block of a few contiguous occupied positions emerges in table, it becomes a "target" for subsequent collisions

As clusters grow, they also merge to form larger clusters.

Primary clustering: elements that hash to different cells probe same alternative cells

Linear probing -- clustering



[R. Sedgewick]

Performance

Load α = M/N

Average search cost (hit) ~ $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$

Average search cost (miss) ~ $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Very delicate math analysis.

Don't use α above 0.8 .

Performance

Expected worst case search cost:

`O(log n)`

Space requirements:

- `M*(|key| + |info|)`

Deletions:

- What's the problem?

Performance

Deletions:

- By marking
- By deleting the item and reinserting all items in the chain.

Choosing the hash function

What properties do we want from a hash function?

Double hashing

When searching for `K`, check locations h_1`(K)`, h_1`(K)+` h_2`(K)`, h_1`(K)+2*`h_2`(K)`,… until either

- `K` is found; or
- we find an empty location (`K` not present)

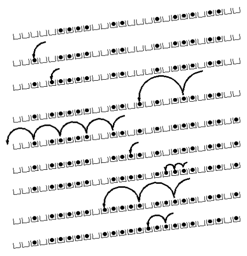
Must be careful about h_2`(K)`

- Not 0.
- Not a divisor of `M`.

Almost as good as random probing.

Very difficult analysis.

Double hashing



[R. Sedgewick]

Performance

Load α = M/N

Average cost (hit) ~ $\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$

Average cost (miss/insert) ~ $\frac{1}{-\alpha}$

Don't use α above 0.95 .

Performance

Expected worst case search cost:

`O(log n)`

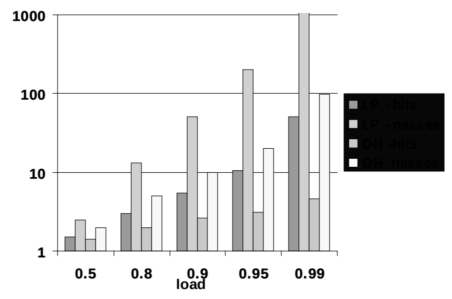
Space requirements:

- `M*(|key| + |info|)`

Deletions:

- Only by marking.
- Eventually misses become very costly!

Open addressing performance



Rules of thumb

Sep chaining is idiot-proof but wastes space…

Linear probing uses space better, is fast when tables are sparse, interacts well with paging

Double hashing is very space efficient, quite fast (get initial hash and increment at the same time), needs careful implementation, …

For average cost t

- Max load for LP (1-1/sqrt(t))
- Max load for DH (1-1/t)

Choosing the hash function

What properties do we want from a hash function?

- Want function to seem random
- Don't want systematic nonrandom pattern in selection of keys to lead to systematic collisions
- Want hash value to depend on all values in entire key and their positions
- Want universe to be distributed randomly

Choosing the hash function

Key = small integer

- For `M` prime `h(K) = K mod M`;
- For `M` non-prime

```
h(K) = floor(M {0.616161*K});

{x} = x - floor(x)

Based on mathematical fact that if A
 is irrational, then for large n
{A}, {2A},…,{nA} distributed uniformly
 across 0..1
```

More hash functions

Key = real in [0,1]

- For any M
  `h(K) = floor(K*M);`

Key = string

- Convert to integer
  - S = a[0] a[1]…. a[n]
  - r -- radix of character code (e.g. 128 or 256)
  - K = a[0] + a[1]*r + …. + a[n]*r ^{n}
  - Can be computed efficiently using Horner's rule
  - Make sure M doesn't divide r ^{k} +/- a for any small a

Caveats

Hash functions are very often the cause of performance bugs.

Hash functions often make the code not portable.

Sometime a poor HF distribution-wise is faster overall.

Always check where the time goes.

Universal hashing

Don't use a fixed hash function; for every run choose a function from a small family.

Example:

```
h(K) = (a*K + b) mod M
```

`a` and `b` chosen u.a.r. in `[1..M]` and `M` prime

Main property

```
Pr(h(K1)=h(K2)) = 1/M
```

Properties

Theory:

- We make no assumptions about input. All proofs are valid wrt our random choices.

Practice:

- If one choice of `a` and `b` turns out to be bad, make a new choice.
- Must use hash schemes that allow re-hashing.
- Useful in critical applications.

Fingerprinting

Fingerprints are short tags for larger objects.

Notations

$\Omega =$ The set of all objects

$k =$ The lenght of the fingerprint

$f : \Omega \to \{0,1\}^k$ A fingerprinting function

Properties

$$f(A) \neq f(B) \Rightarrow A \neq B$$

$$\Pr\left(f(A) = f(B) \middle| A \neq B \right) \approx \frac{1}{2^k}$$

Why fingerprint?

Probability is wrt our choice of a fpr scheme.

- Don't need assumption about input.

Keys are long or there are no keys (need uid's):

- In AltaVista 100M urls @ 90 bytes/url = 9GB
  100M fprs @ 8 byte/fpr = 0.8GB
- Find duplicate pages -- two pages are the same if they have the same fpr.

Fingerprinting schemes

Cryptographically secure:

- MD2, MD4, MD5, SHS, etc
- relatively slow

Rabin's scheme

- Based on polynomial arithmetic
- Very fast (1 table lookup + 1 xor + 1 shift) /byte
- Nice extra-properties

Rabin's scheme

View each string A as a polynomial over Z_2:

$$A = 1\ 0\ 0\ 1\ 1 \quad \Rightarrow \quad A(x) = x^4 + x + 1$$

Let `P(t)` be an irreducible polynomial of degree `k` chosen uar

The fingerprint of `A` is

```
f(A) = A(t) mod P(t)
```

The probability of collision among `n` strings of average length `t (chosen by adversary!)` is about

```
n^2 t / 2^k
```

Nice extra properties

Let ♦ = catenation. Then

f(a ♦ b) = f(f(a) ♦ b)

Can compute extensions of strings easily.

Bloom filters

Want to check only existence of key (e.g. spelling dictionary, stolen credit cards, etc)

Small probability of error is OK.

Simple solution:

- Keep bit-table `B`
- For each `K` turn `B(h(K))` on;
- Say K is in iff `B(h(K))` is on;
- Works if there are no collisions! Must have
  `N = O(sqrt(M))`
- Collisions generate false drops

Better solution:

Use r hash functions.

- For each `K` turn on
  $B(h_1(K)),\ B(h_2(K)), \ldots, B(h_r(K))$
- Say `K` is in iff all hash bits are on.
- Probability of false drop is $\boxed{(1-\exp(-r\alpha))^r}$
- Optimum choice for r is $\boxed{r = \alpha \ln(2)}$
- With this choice, probability of false drop

$$\boxed{0.6185^{1/\alpha}}$$

Example

`/usr/dict/words` -- about `210KB`, `25K` words

Use `30KB` table

Load = `25/(30*8) ~ 0.104`

Optimum `r = 7`

Probability of false drop

- `1%` for `r =7`
- `1.3%` for `r=4`

Perfect hashing

The set of keys is given and never changes.

Find “simple” hash function so that there are no collisions.

Example: reserved words in a compiler.

Hard to do but can be very useful.

Example `(M ~ 6N)`

`(a K mod b) mod M`

Takes time $O(n^3 \log n)$ to compute.