Week 12: Approximation and Randomised Algorithms

Approximation

Approximation for Numerical Problems

2/68

Approximation is often used to solve numerical problems by

- solving a simpler, but much more easily solved, problem
- where this new problem gives an approximate solution
- and refine the method until it is "accurate enough"

Examples:

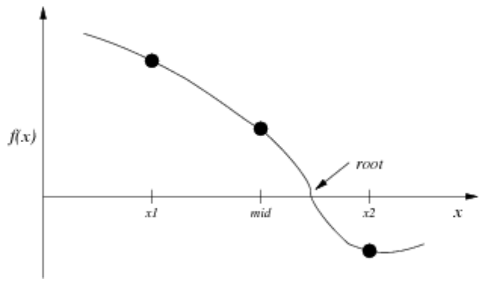
- roots of a function *f*
- length of a curve determined by a function *f*
- … and many more

... Approximation for Numerical Problems

3/68

Example: Finding Roots

Find where a function crosses the x-axis:



Generate and test: move x_1 and x_2 together until "close enough"

... Approximation for Numerical Problems

4/68

A simple approximation algorithm for finding a root in a given interval:

```
bisection(f,x1,x2):
|  Input   function f, interval [x1,x2]
|  Output  x∈[x1,x2] with f(x)≅0
|
|  repeat
|  |  mid=(x1+x2)/2
|  |  if f(x1)*f(mid)<0 then
|  |     x2=mid       // root to the left of mid
|  |  else
|  |     x1=mid       // root to the right of mid
|  |  end if
|  until f(mid)=0 or x2-x1<ε    // ε: accuracy
|  end while
|  return mid
```

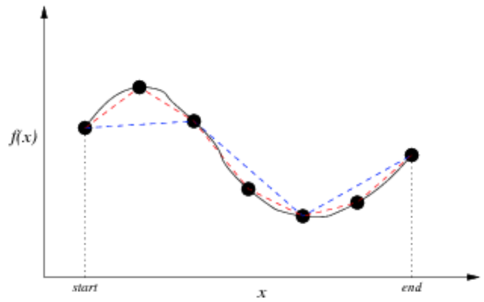
`bisection` guaranteed to converge to a root if *f* continuous on $[x_1,x_2]$ and $f(x_1)$ and $f(x_2)$ have opposite signs

... Approximation for Numerical Problems

5/68

Example: Length of a Curve

Estimate length: approximate curve as sequence of straight lines.



... Approximation for Numerical Problems

6/68

```
curveLength(f,start,end):
|  Input   function f, start and end point
|  Output  curve length between f(start) and f(end)
|
|  length=0, δ=(end-start)/StepSize
|  for each x∈[start+δ,start+2δ,..,end] do
|     length = length + sqrt(δ^2 + (f(x)-f(x-δ))^2)
|  end for
|  return length
```

Sidetrack: Function Pointers

7/68

Function pointers …

- are references to memory address of a function
- are pointer values and can be assigned/passed

Function pointer variables/parameters are declared as:

```
typeOfReturnValue (*fname)(typeOfArguments)
```

... Sidetrack: Function Pointers

8/68

Example:

```
// define a function of type double → double
double myfun(double x) {
   return sqrt(1-x*x);
}

double curveLength(double start, double end, double (*f)(double)) {
   ...
   deltaY  = f(x) - f(x-delta);
   length += sqrt(delta*delta + deltaY*deltaY);
   ...
}

printf("%.10f\n", curveLength(-1, 1, myfun));
```

Approximation for Numerical Problems

9/68

Trade-offs in curve length approximation algorithm:

- large step size …
  - less steps, less computation (faster), lower accuracy
- small step size …
  - more steps, more computation (slower), higher accuracy

However, too many steps may lead to higher rounding error.

Each *f* has an optimal step size …

- but this is difficult to determine in advance

... Approximation for Numerical Problems

10/68

Example: `length = curveLength(0,π,sin);`

Convergence when using more and more steps

```
steps =       0, length = 0.000000
steps =      10, length = 3.815283
steps =     100, length = 3.820149
steps =    1000, length = 3.820197
steps =   10000, length = 3.819753
steps =  100000, length = 3.820198
steps = 1000000, length = 3.820198
```

Actual answer is 3.820197789...

Approximation for NP-hard Problems

11/68

Approximation is often used for NP-hard problems …

- computing a near-optimal solution
- in polynomial time

Examples:

- vertex cover of a graph
- subset-sum problem

Vertex Cover

12/68

Reminder: Graph $G = (V,E)$

- set of vertices V
- set of edges E

Vertex cover C of G …

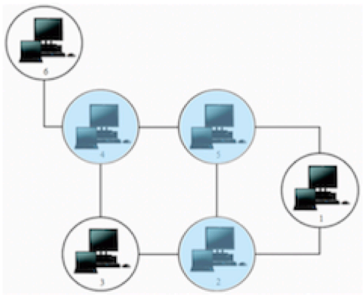
- $C \subseteq V$
- for all edges $(u,v) \in E$ either $v \in C$ or $u \in C$ (or both)

⇒ All edges of the graph are "covered" by vertices in C

... Vertex Cover

13/68

Example (6 nodes, 7 edges, 3-vertex cover):



Applications:

- Computer Network Security
  - compute minimal set of routers to cover all connections
- Biochemistry

... Vertex Cover

14/68

size of vertex cover C … $|C|$ (number of elements in C)

optimal vertex cover … a vertex cover of minimum size

Theorem.
Determining whether a graph has a vertex cover of a given size k is an NP-complete problem.

... Vertex Cover

15/68

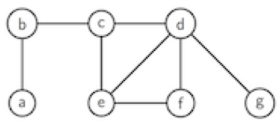
An approximation algorithm for vertex cover:

```
approxVertexCover(G):
|  Input  undirected graph G=(V,E)
|  Output vertex cover of G
|
|  C = ∅
|  unusedE = E
|  while unusedE ≠ ∅
|  |  choose any (v,w)∈unusedE
|  |  C = C∪{v,w}
|  |  unusedE = unusedE\{all edges incident on v or w}
|  end while
|  return C
```

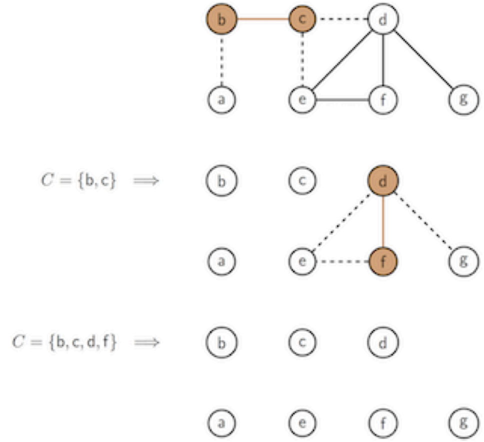
Exercise #1: Vertex Cover

16/68

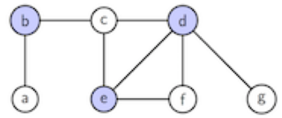
Show how the approximation algorithm produces a vertex cover on:



Possible result:



What would be an optimal vertex cover?



... Vertex Cover

19/68

Theorem.
The approximation algorithm returns a vertex cover *at most twice the size* of an optimal cover.

Cost analysis …

- repeatedly select an edge from E
  - add endpoints to C
  - delete all edges in E covered by endpoints

Time complexity: $O(V+E)$ (adjacency list representation)

Randomisation

Randomised Algorithms

21/68

Algorithms employ randomness to

- improve worst-case runtime
- compute correct solutions to hard problems more efficiently but with low probability of failure
- compute approximate solutions to hard problems

Randomness

Randomness is also useful

- in computer games:
  - may want aliens to move in a random pattern
  - the layout of a dungeon may be randomly generated
  - may want to introduce unpredictability
- in physics/applied maths:
  - carry out simulations to determine behaviour
    - e.g. models of molecules are often assume to move randomly
- in testing:
  - *stress test* components by bombarding them with random data
  - random data is often seen as *unbiased data*
    - gives average performance (e.g. in sorting algorithms)
- in cryptography

Sidetrack: Random Numbers

How can a computer pick a number at random?

- it cannot

Software can only produce *pseudo random numbers*.

- a pseudo random number is one that is predictable
  - (although it may appear unpredictable)

⇒ Implementation may deviate from expected theoretical behaviour

... Sidetrack: Random Numbers

The most widely-used technique is called the *Linear Congruential Generator (LCG)*

- it uses a recurrence relation:
  - $X_{n+1} = (a{\cdot}X_n + c) \bmod m$, where:
    - m is the "modulus"
    - a, $0 < a < m$ is the "multiplier"
    - c, $0 \le c \le m$ is the "increment"
    - X_0 is the "*seed*"
  - if c=0 it is called a *multiplicative congruential generator*

LCG is not good for applications that need extremely high-quality random numbers

- the period length is too short (length of the sequence at which point it repeats itself)
- a short period means the numbers are correlated

... Sidetrack: Random Numbers

Trivial example:

- for simplicity assume c=0
- so the formula is $X_{n+1} = a{\cdot}X_n \bmod m$
- try a=11=X_0, m=31, which generates the sequence:

```
11, 28, 29, 9, 6, 4, 13, 19, 23, 5, 24, 16, 21, 14, 30, 20, 3, 2, 22, 25,
27, 18, 12, 8, 26, 7, 15, 10, 17, 1, 11, 28, 29, 9, 6, 4, 13, 19, 23, 5,
24, 16, 21, 14, 30, 20, 3, 2, 22, 25, 27, 18, 12, 8, 26, 7, 15, 10, 17, 1,
11, 28, 29, 9, 6, 4, 13, 19, 23, 5, 24, 16, 21, 14, 30, 20, 3, 2, 22, 25, 27,
18, 12, 8, 26, 7, 15, 10, 17, 1, 11, 28, 29, 9, 6, 4, 13, 19, 23, 5, 24, 16,
21, 14, 30, 20, 3, 2, 22, 25, 27, 18, 12, 8, 26, 7, 15, 10, 17, 1, 11, 28,
29, 9, 6, 4, 13, 19, 23, 5, 24, 16, 21, 14, 30, 20, 3, 2, 22, 25, 27, 18,
12, 8, 26, 7, 15, 10, 17, 1, ...
```

- all the integers from 1 to 30 are here

... Sidetrack: Random Numbers

Another trivial example:

- again let c=0
- try a=12=X_0 and m=30
  - that is, $X_{n+1} = 12{\cdot}X_n \bmod 30$
  - which generates the sequence:

```
12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6,
12, 24, 18, 6, 12, 24, 18, 6, 12, 24, 18, 6, ...
```

- notice the period length … clearly a terrible sequence

... Sidetrack: Random Numbers

It is a complex task to pick good numbers. A bit of history:

Lewis, Goodman and Miller (1969) suggested

- $X_{n+1} = 7^5{\cdot}X_n \bmod (2^{31}-1)$
- note:
  - 7^5 is 16807
  - 2^{31}-1 is 2147483674
  - $X_0 = 0$ is not a good seed value

Most compilers use LCG-based algorithms that are slightly more involved; see www.mscs.dal.ca/~selinger/random/ for details (including a short C program that produces the exact same pseudo-random numbers as `gcc` for any given seed value)

... Sidetrack: Random Numbers

- Two functions are required:

```
srand(unsigned int seed) // sets its argument as the seed

rand() // uses a LCG technique to generate random
       // numbers in the range 0 .. RAND_MAX
```

where the constant `RAND_MAX` is defined in `stdlib.h`
(depends on the computer: on the CSE network, RAND_MAX = 2147483647)

- The period length of this random number generator is very large
  approximately $16 \cdot ((2^{31}) - 1)$

... Sidetrack: Random Numbers

To convert the return value of `rand()` to a number between 0 .. RANGE

- compute the remainder after division by RANGE+1

Using the remainder to compute a random number is not the best way:

- can generate a 'better' random number by using a more complex division
- but good enough for most purposes

Some applications require more sophisticated, *cryptographically secure* pseudo random numbers

---

Exercise #2: Random Numbers

Write a program to simulate 10,000 rounds of Two-up.

- Assume a $10 bet at each round
- Compute the overall outcome and average per round

```
#include <stdlib.h>
#include <stdio.h>

#define RUNS 10000
#define BET  10

int main(void) {
   srand(1234567);       // choose arbitrary seed
   int coin1, coin2, n, sum = 0;
   for (n = 0; n < RUNS; n++) {
      do {
         coin1 = rand() % 2;
         coin2 = rand() % 2;
      } while (coin1 != coin2);
      if (coin1==1 && coin2==1)
         sum += BET;
      else
         sum -= BET;
   }
   printf("Final result: %d\n", sum);
   printf("Average outcome: %f\n", (float) sum / RUNS);
   return 0;
}
```

... Sidetrack: Random Numbers

Seeding

There is one significant problem:

- every time you run a program with the same seed, you get exactly the same sequence of 'random' numbers (why?)

To vary the output, can give the random seeder a starting point that varies with time

- an example of such a starting point is the current time, ***`time(NULL)`***
  (NB: this is different from the UNIX command `time`, used to measure program running time)

```
#include <time.h>
time(NULL) // returns the time as the number of seconds
           // since the Epoch, 1970-01-01 00:00:00 +0000

// time(NULL) on October 14th, 2018, 12:59pm was 1539482350
// time(NULL) about a minute later was 1539482409
```

Randomised Algorithms

Analysis of Randomised Algorithms

Randomised algorithm to find *some* element with key *k* in an unordered list:

```
findKey(L,k):
|  Input  list L, key k
|  Output some element in L with key k
|
|  repeat
|     randomly select e∈L
|  until key(e)=k
|  return e
```

... Analysis of Randomised Algorithms

Analysis:

- p … ratio of elements in L with key k (e.g. $p = \frac{1}{3}$)
- *Probability of success*: 1 (if $p > 0$)
- *Expected runtime*: $\frac{1}{p}$ $\left(= \lim_{n\to\infty} \sum_{i=1..n} i \cdot (1-p)^{i-1} \cdot p \right)$

- Example: a third of the elements have key $k \Rightarrow$ expected number of iterations = 3

... Analysis of Randomised Algorithms

36/68

If we cannot guarantee that the list contains any elements with key k …

```
findKey(L,k,d):
|  Input  list L, key k, maximum #attempts d
|  Output some element in L with key k
|
|  repeat
|  |  if d=0 then
|  |     return failure
|  |  end if
|  |  randomly select e∈L
|  |  d=d-1
|  until key(e)=k
|  return e
```

... Analysis of Randomised Algorithms

37/68

Analysis:

- p … ratio of elements in L with key k
- d … maximum number of attempts
- *Probability of success*: ~~$1-p^d$~~ 1 - $(1-p)^d$
- *Expected runtime*: $\left(\sum_{i=1..d} i \cdot (1-p)^{i-1} \cdot p\right) + d \cdot (1-p)^{d-1}$
  - $O(1)$ if d is a constant

Non-randomised Quicksort

38/68

Reminder: *Quicksort* applies divide and conquer to sorting:

- Divide
  - pick a *pivot* element
  - move all elements smaller than the *pivot* to its left
  - move all elements greater than the *pivot* to its right
- Conquer
  - sort the elements on the left
  - sort the elements on the right

... Non-randomised Quicksort

39/68

Divide …

```
partition(array,low,high):
|  Input  array, index range low..high
|  Output selects array[low] as pivot element
|         moves all smaller elements between low+1..high to its left
|         moves all larger elements between low+1..high to its right
|         returns new position of pivot element
|
|  pivot_item=array[low], left=low+1, right=high
|  while left<right do
|  |  left  = find index of leftmost element > pivot_item
|  |  right = find index of rightmost element <= pivot_item
|  |  if left<right then
|  |     swap array[left] and array[right]
|  |  end if
|  end while
|  array[low]=array[right] // right is final position for pivot
|  array[right]=pivot_item
|  return right
```

... Non-randomised Quicksort

40/68

... and Conquer!

```
Quicksort(array,low,high):
|  Input  array, index range low..high
|  Output array[low..high] sorted
|
|  if high > low then     // termination condition low >= high
|  |   pivot = partition(array,low,high)
|  |   Quicksort(array,low,pivot-1)
|  |   Quicksort(array,pivot+1,high)
|  end if
```

... Non-randomised Quicksort

41/68

```
3  6  5  2  4  1

3  1  5  2  4  6

3  1  2  5  4  6

2  1   | 3 |   6  4  5

1  2   | 3 |   6  4  5

1  2   | 3 |   5  4   | 6 |

1  2   | 3 |   4  5   | 6 |
```

Worst-case Running Time

42/68

Worst case for Quicksort occurs when the pivot is the unique minimum or maximum element:

- One of the intervals `low..pivot-1` and `pivot+1..high` is of size *n-1* and the other is of size 0
  ⇒ running time is proportional to $n + n\text{-}1 + \ldots + 2 + 1$
- Hence the worst case for non-randomised Quicksort is $O(n^2)$

```
6  5  4  3  2  1

5  4  3  2  1   |   6

4  3  2  1   |   5   |   6

3  2  1   |   4   |   5   |   6

                 ...

1   |   2   |   3   |   4   |   5   |   6
```

Randomised Quicksort

43/68

```
partition(array,low,high):
|  Input   array, index range low..high
|  Output  randomly select a pivot element from array[low..high]
|          moves all smaller elements between low..high to its left
|          moves all larger elements between low..high to its right
|          returns new position of pivot element
|
|  randomly select pivot_index∈[low..high]
|  pivot_item=array[pivot_index], swap array[low] and array[pivot_index]
|  left=low+1, right=high
|  while left<right do
|  |  left  = find index of leftmost element > pivot_item
|  |  right = find index of rightmost element <= pivot_item
|  |  if left<right then
|  |     swap array[left] and array[right]
|  |  end if
|  end while
|  array[low] = array[right], array[right]=pivot_item
|  return right
```

... Randomised Quicksort

44/68

Analysis:

- Consider a recursive call to `partition()` on an index range of size *s*
  - *Good call*:
    both `low..pivot-1` and `pivot+1..high` shorter than ¾·*s*
  - *Bad call*:
    one of `low..pivot-1` or `pivot+1..high` greater than ¾·*s*
- Probability that a call is good: 0.5
  (because half the possible pivot elements cause a good call)

Example of a bad call:

6 3 7 5 8 2 4 1

~~6 3 5 2 4 1 | 7 | 8~~
3 6 5 1 2 4 | 7 | 8

Example of a good call:

3 6 5 2 4 1 | 7 | 8

1 2 | 3 | 5 4 6 | 7 | 8

... Randomised Quicksort

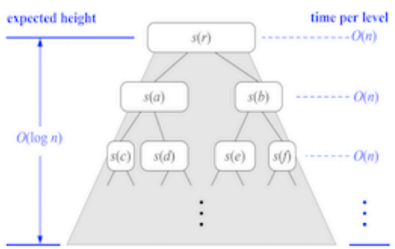
45/68

n … size of array

From probability theory we know that the expected number of coin tosses required in order to get k heads is 2·k

- For a recursive call at depth *d* we expect
  - *d*/2 ancestors are good calls
    ⇒ size of input sequence for current call is $\leq (3/4)^{d/2} \cdot n$
- Therefore,
  - the input of a recursive call at depth $2 \cdot \log_{4/3} n$ has expected size 1
    ⇒ the expected recursion depth thus is *O(log n)*
- The total amount of work done at all the nodes of the same depth is *O(n)*

Hence the expected runtime is *O(n·log n)*



Minimum Cut Problem

46/68

Given:

- undirected graph $G=(V,E)$

Cut of a graph …

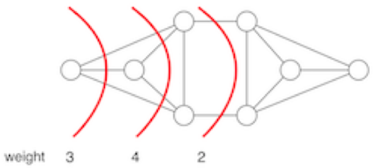
- a partition of V into $S \cup T$
  - S,T disjoint and both non-empty
- its *weight* is the number of edges between S and T:

$$\omega(S,T) = |\ \{ \{s,t\} \in E : s \in S, t \in T \}\ |$$

Minimum cut problem … find a cut of G with minimal weight

... Minimum Cut Problem

Example:



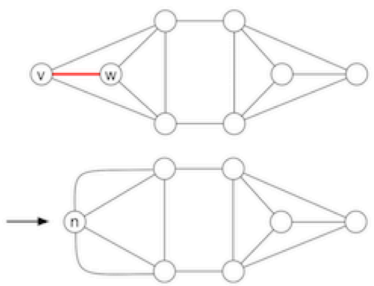
Contraction

Contracting edge $e = \{v,w\}$ …

- remove edge e
- replace vertices v and w by new node n
- replace all edges $\{x,v\}$, $\{x,w\}$ by $\{x,n\}$

… results in a *multigraph* (multiple edges between vertices allowed)

Example:



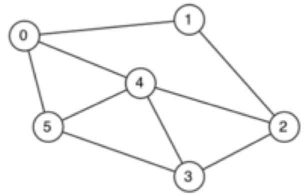
... Contraction

Randomised algorithm for *graph contraction* = repeated edge contraction until 2 vertices remain

```
contract(G):
|  Input   graph G = (V,E) with |V|≥2 vertices
|  Output  cut of G
|
|  while |V|>2 do
|     randomly select e∈E
|     contract edge e in G
|  end while
|  return the only cut in G
```

Exercise #3: Graph Contraction

Apply the contraction algorithm twice to the following graph, with different random choices:



... Contraction

Analysis:

V … number of vertices

- Probability of `contract` to result in a minimum cut:
  $$\geq 1 / \binom{V}{2}$$
- This is much higher than the probability of picking a minimum cut at random, which is
  $$\leq \binom{V}{2} / (2^{V-1} - 1)$$
  because every graph has 2^{V-1}-1 cuts, of which at most $\binom{V}{2}$ can have minimum weight
- Single edge contraction can be implemented in $O(V)$ time on an adjacency-list representation ⇒ total running time: $O(V^2)$

  (Best known implementation uses $O(E)$ time)

Karger's Algorithm

Idea: Repeat random graph contraction several times and take the best cut found

```
MinCut(G):
```

```
Input  graph G with V≥2 vertices
Output smallest cut found

min_weight=∞, d=0
repeat
   cut=contract(G)
   if weight(cut)<min_weight then
      min_cut=cut, min_weight=weight(cut)
   end if
   d=d+1
until d > binomial(V,2)·ln V
return min_cut
```

... Karger's Algorithm

Analysis:

V … number of vertices
E … number of edges

- *Probability of success*: $\geq 1 - \frac{1}{V}$
  - probability of not finding a minimum cut when the contraction algorithm is repeated $d = \binom{V}{2} \cdot \ln n$ times:

$$\leq \left[1 - 1/\binom{V}{2}\right]^d \leq \frac{1}{e^{\ln V}} = \frac{1}{V}$$

- Total running time: $O(E{\cdot}d) = O(E{\cdot}V^2{\cdot}log\ V)$
  - assuming edge contraction implemented in $O(E)$

Sidetrack: Maxflow and Mincut

Given: flow network $G=(V,E)$ with

- edge weights $w(u,v)$
- source $s\in V$, sink $t\in V$

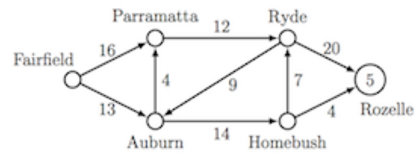
Cut of flow network G …

- a partition of V into $S \cup T$
  - $s\in S$, $t\in T$, S and T disjoint
- its *weight* is the sum of the weights of the edges between S and T:

$$\omega(S,T) = \sum_{s\in S} \sum_{t\in T} w(u,v)$$

Minimum cut problem … find cut of a network with minimal weight

Exercise #4: Cut of Flow Networks

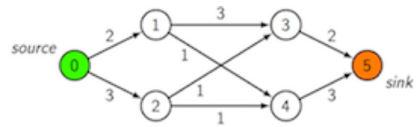


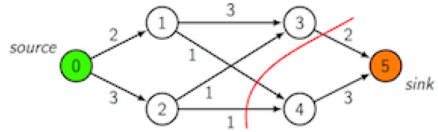
What is the weight of the cut {Fairfield,Parramatta,Auburn}, {Ryde,Homebush,Rozelle}?

12+14 = 26

Exercise #5: Cut of Flow Networks

Find a minimal cut in:





$\omega(S,T) = 4$

... Sidetrack: Maxflow and Mincut

Max-flow Min-cut Theorem.
In a flow network G the following conditions are equivalent:

1. f is a maximum flow in G
2. the residual network G relative to f contains no augmenting path
3. value of flow f = weight of some minimum cut (S,T) of G

Randomised Algorithms for NP-hard Problems

Many NP-hard problems can be tackled by randomised algorithms that

- compute nearly optimal solutions
  - with high probability

Examples:

- travelling salesman
- constraint satisfaction problems, satisfiability
- … and many more

Simulation

Simulation

In some problem scenarios

- it is difficult to devise an analytical solution
- so build a software *model* and run *experiments*

Examples: weather forecasting, traffic flow, queueing, games

Such systems typically require random number generation

- distributions: uniform, numerical, normal, exponential

Accuracy of results depends on accuracy of model.

Example: Gambling Game

Consider the following game:

- you bet $1 and roll two dice (6-sided)
- if total is between 8 and 11, you get $2 back
- if total is 12, you get $6 back
- otherwise, you lose your money

Is this game worth playing?

Test: start with $5 and play until you have $0 or $20.

In fact, this example is reasonably easy to solve analytically.

... Example: Gambling Game

We can get a reasonable approximation by simulation

- set our initial *balance* to $5
- generate two random numbers in range 1..6 (dice)
- adjust *balance* by payout or loss
- repeat above until *balance ≤ $0 or balance ≥ $20*
- run a very large number of trials like the above
- collect statistics on the outcome

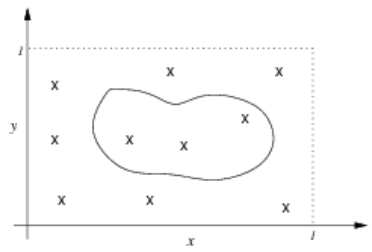
... Example: Gambling Game


```
gameSimulation:
|  Output likelihood of ending with a balance ≥$20
|
|  nwins=0
|  for a large number of Trials do
|  |  balance=$5
|  |  while balance>$0 ∧ balance<$20 do
|  |  |  balance=balance-$1
|  |  |  die1=random number∈[1..6], die2=random number∈[1..6]
|  |  |  if 7≤die1+die2≤11 then
|  |  |     balance=balance+$2
|  |  |  else if die1+die2=12 then
|  |  |     balance=balance+$6
|  |  |  end if
|  |  end while
|  |  if balance≥$20 then
|  |     nwins=nwins+1
|  |  end if
|  end for
|  return nwins/Trials
```

Example: Area inside a Curve

Scenario:

- have a closed curve defined by a complex function
- have a function to compute "X is inside/outside curve?"



... Example: Area inside a Curve

Simulation approach to determining the area:

- determine a region completely enclosing curve
- generate very many random points in this region
- for each point *x*, compute *inside(x)*
- count number of insides and outsides
- areaWithinCurve = totalArea * insides/(insides+outsides)

I.e. we approximate the area within the curve by using the ratio of points inside the curve against those outside

Also known as *Monte Carlo estimation*

Summary

- Approximation
  - factor-2 approximation for vertex cover
- Analysis of randomised algorithms
  - *probability of success*
  - *expected runtime*
- Randomised Quicksort
- Karger's algorithm
- Simulation

- Suggested reading:
  - Approximation … Moffat, Ch.9.4
  - Randomisation, simulation … Moffat, Ch.9.3,9.5

Produced: 16 Oct 2018