算法第一次作业

李博杰 SA14011026

概率算法

Ex.1

解：若将y=uniform(0,1) 改为y=x，则当且仅当 即 时，k++ 会执行，也就是 ，故 为本算法返回的结果。

Ex.2

解：确定算法（分割法）

|  |  |  |
| --- | --- | --- |
| 分割条数 | 结果 | 偏差 |
| 1000 | 3.143555 | 0.001962 |
| 10000 | 3.141791 | 0.000198 |
| 100000 | 3.141613 | 0.000020 |
| 1000000 | 3.141595 | 0.000002 |

代码：

#include<stdio.h>

#include<math.h>

int main() {

int n = 1000000;

double sum, i;

for (i=0.0; i<1; i+=1.0/n) {

sum += 1.0 / n \* sqrt(1 - i\*i);

}

printf("%lf\n", 4 \* sum);

}

概率算法

|  |  |  |
| --- | --- | --- |
| N | 结果 | 偏差 |
| 1000 | 3.118979 | -0.022613 |
| 10000 | 3.151633 | 0.010040 |
| 100000 | 3.146769 | 0.005176 |
| 1000000 | 3.142897 | 0.001304 |

代码：

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

double f(double x) {

return sqrt(1 - x\*x);

}

double integrate(double a, double b, int n, double (\*f)(double)) {

int i;

double sum = 0;

for (i=0; i<n; i++) {

sum += f(rand() \* 1.0 / RAND\_MAX \* (b-a) + a);

}

return sum \* (b-a) / n;

}

int main() {

printf("%lf\n", 4 \* integrate(0, 1, 1000000, f));

}

Ex.3

|  |  |
| --- | --- |
| N | 结果 |
| 10000 | 0.329808 |
| 100000 | 0.331640 |
| 1000000 | 0.332870 |

代码：

#include<stdio.h>

#include<stdlib.h>

#include<math.h>

double f(double x) {

return x\*x;

}

double integrate(double a, double b, int n, double (\*f)(double)) {

int i;

double sum = 0;

for (i=0; i<n; i++) {

sum += f(rand() \* 1.0 / RAND\_MAX \* (b-a) + a);

}

return sum \* (b-a) / n;

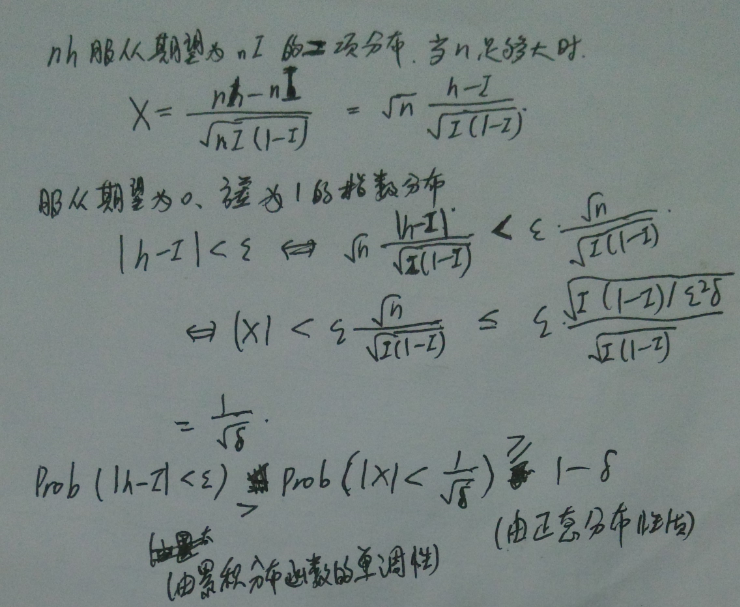
}

int main() {

printf("%lf\n", integrate(0, 1, 1000000, f));

}

Ex.4



概率计数 Ex

估计整数子集 1~n 的大小，并分析 n 对估计值的影响。重复运行10000次取平均值。

|  |  |
| --- | --- |
| n | 估计值 |
| 10 | 15.8 |
| 100 | 136.9 |
| 1000 | 1301.2 |
| 10000 | 12616.0 |
| 100000 | 127576.9 |
| 1000000 | 1289251.3 |
| 10000000 | 12929103.5 |

分析：N 较小时，估计值偏大。N 较大时，有 30% 左右的系统误差（偏大）。

代码：

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<math.h>

#include<time.h>

#define REPEAT\_COUNT 10000

int main(int argc, char\*\* argv) {

int n;

int repeat;

double estimate\_n = 0;

unsigned char \*set;

n = atoi(argv[1]);

srand(time(NULL));

set = (unsigned char\*)malloc(n \* sizeof(unsigned char));

for (repeat = 0; repeat < REPEAT\_COUNT; repeat++) {

int k;

memset(set, 0, n \* sizeof(unsigned char));

for (k=1; ; k++) {

int a = rand() % n;

if (set[a])

break;

set[a] = 1;

}

estimate\_n += 2.0 \* k \* k / 3.1415926;

}

printf("%f\n", estimate\_n \* 1.0 / REPEAT\_COUNT);

return 0;

}

随机的预处理 Ex

DlogRH中随机抽样函数

将输入实例转化为随机化的实例

将随机化实例的输出转化为原实例的输出

原理：

搜索有序表

算法 C：在算法 B 的基础上修改，不是取前 sqrt(n) 个元素作为基准，而是随机取 sqrt(n) 个元素作为基准。

10000 个元素，A、B、D 为确定算法，C 为随机算法。对相同的随机生成的链表实例，每次查询随机的元素，重复 1000 次，结果如下：

Algo A: average 5044.362000, worst 9974

Algo B: average 194.229000, worst 719

Algo C: average 199.987000, worst 771

Algo D: average 3442.302000, worst 9724

10000 个元素，对逆序链表实例（10000=>9999=>9998=>…=>1，修改代码第 66 行为 r=0 即可），每次查询随机的元素，重复 1000 次，结果如下：

Algo A: average 4885.943000, worst 9982

Algo B: average 5049.299000, worst 9982

Algo C: average 199.101000, worst 889

Algo D: average 3322.780000, worst 9537

10000 个元素，对顺序链表实例（1=>2=>…=>10000，修改代码第 66 行为 r=n-i-1 即可），每次查询随机的元素，重复 1000 次，结果如下：

Algo A: average 5050.174000, worst 9997

Algo B: average 4988.882000, worst 9992

Algo C: average 195.875000, worst 760

Algo D: average 3237.751000, worst 9819

10000 个元素，对基本有序的链表实例（随机交换 1000 对元素，修改代码第 65 行的 n 为 n/10，修改代码第 66 行为 r = rand() % n，修改代码第 67 行为 last = rand() % n），每次查询随机的元素，重复 1000 次，结果如下：

Algo A: average 5071.158000, worst 9975

Algo B: average 828.431000, worst 2770

Algo C: average 196.938000, worst 769

Algo D: average 3391.237000, worst 9810

结论：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 算法 | 随机链表 | | 有序链表 | |
| 平均复杂度 | 最坏复杂度 | 平均复杂度 | 最坏复杂度 |
| A | n/2 | n | n/2 | n |
| B | 2\*sqrt(n) | X | n/2 | n |
| C | 2\*sqrt(n) | X | 2\*sqrt(n) | X |
| D | n/3 | n | n/3 | n |

注：2\*sqrt(n) < X < n/2

可见在链表元素基本有序的情况下，随机化的 Sherwood 算法 C 的表现与随机链表相同，而未随机化的算法 B 退化成了 O(n) 的算法。

代码：

#include<stdio.h>

#include<math.h>

#include<stdlib.h>

#include<string.h>

#include<time.h>

#define ALGO\_REPEAT\_COUNT 1000

// global var to count linked list access

int count;

int algo\_a(int n, int val[], int ptr[], int head, int x) {

while (x > val[head]) {

head = ptr[head];

++count;

}

return head;

}

int algo\_b(int n, int val[], int ptr[], int head, int x) {

int sqrtn = sqrt(n);

int i, max = head;

for (i=0; i<sqrtn; i++) {

++count;

if (val[i] <= x && val[i] > val[max])

max = i;

}

return algo\_a(n, val, ptr, max, x);

}

int algo\_c(int n, int val[], int ptr[], int head, int x) {

int sqrtn = sqrt(n);

int i, max = head;

for (i=0; i<sqrtn; i++) {

int r = rand() % n;

++count;

if (val[r] <= x && val[r] > val[max])

max = r;

}

return algo\_a(n, val, ptr, max, x);

}

int algo\_d(int n, int val[], int ptr[], int head, int x) {

int flag = rand() % n;

int y = val[flag];

++count;

if (x < y)

return algo\_a(n, val, ptr, head, x);

else if (x > y)

return algo\_a(n, val, ptr, ptr[flag], x);

else

return flag;

}

void search(int n) {

int val[n], ptr[n], revptr[n];

int i, j, head = 0;

for (i=0; i<n; i++) {

val[i] = i;

ptr[i] = i+1;

revptr[i] = i-1;

}

ptr[n-1] = 0;

revptr[0] = n-1;

for (i=0; i<n; i++) {

int r = rand() % (n-i);

int last = n-i-1;

if (r == last)

continue;

int before\_rand = revptr[r], after\_rand = ptr[r];

int before\_last = revptr[last], after\_last = ptr[last];

int tmp = val[r];

val[r] = val[last];

val[last] = tmp;

if (before\_rand == last) { // adjacent

ptr[before\_last] = r;

ptr[r] = last;

ptr[last] = after\_rand;

revptr[r] = before\_last;

revptr[last] = r;

revptr[after\_rand] = last;

}

else if (before\_last == r) { // adjacent

ptr[before\_rand] = last;

ptr[last] = r;

ptr[r] = after\_last;

revptr[last] = before\_rand;

revptr[r] = last;

revptr[after\_last] = r;

}

else { // ordinary case

ptr[before\_rand] = last;

revptr[last] = before\_rand;

ptr[before\_last] = r;

revptr[r] = before\_last;

ptr[last] = after\_rand;

revptr[after\_rand] = last;

ptr[r] = after\_last;

revptr[after\_last] = r;

}

}

for (i=0; i<n; i++)

if (val[i] == 0) {

head = i;

break;

}

int (\*algo[])(int, int[], int[], int, int) = {algo\_a, algo\_b, algo\_c, algo\_d};

//int repeat\_num[] = {1, 1, ALGO\_REPEAT\_COUNT, 1};

int algo\_num = sizeof(algo) / sizeof(algo[0]);

int worst[algo\_num];

int total[algo\_num];

for (i=0; i<algo\_num; i++) {

worst[i] = total[i] = 0;

for (j=0; j<ALGO\_REPEAT\_COUNT; j++) {

int x = rand() % n;

count = 0;

algo[i](n, val, ptr, head, x);

if (count > worst[i])

worst[i] = count;

total[i] += count;

}

}

for (i=0; i<algo\_num; i++) {

printf("Algo %c: average %f, worst %d\n", i+'A', total[i] \* 1.0 / ALGO\_REPEAT\_COUNT, worst[i]);

}

}

int main(int argc, char\*\* argv) {

int n = atoi(argv[1]);

srand(time(NULL));

search(n);

return 0;

}

第四章 八皇后问题

当放置 (k+1) th皇后时，若有多个位置是开放的，则算法QueensLV选中其中任一位置的概率相等。

证明：设共有 m 个开放的位置。若证QueensLV选中其中任一位置的概率相等，只需证明，对任意 t=1,2…m，第 t 个开放位置被选中的概率为 1/m。

第 t 个开放位置若最终被选中，需要在第 t 轮随机出 uniform(1..t)=1，并且第 t+1…m 轮都没有选中（否则会被覆盖掉）。第 t 轮随机出 uniform(1..t)=1 的概率为 1/t，第 t+1…m 轮没有选中的概率依次为 t/(t+1)… (m-1)/m。

由乘法原理，第 t 个开放位置最终被选中的概率为 1/t \* t/(t+1) \* … \* (m-1)/m = 1/m。证毕。

StepVegas

写一算法，求n=12~20时最优的StepVegas值。

算法重复 1000 次取平均值。每尝试放置一次皇后，记为一个节点。结果如下：

n=12 bestk=5 nodes=78.456000

n=13 bestk=6 nodes=85.967000

n=14 bestk=7 nodes=104.366000

n=15 bestk=8 nodes=106.803000

n=16 bestk=9 nodes=113.846000

n=17 bestk=9 nodes=125.798000

n=18 bestk=10 nodes=138.310000

n=19 bestk=11 nodes=146.336000

n=20 bestk=12 nodes=151.622000

代码：

#include<stdio.h>

#include<stdlib.h>

typedef unsigned char bool;

#define TRIALS 1000

bool queens\_lv(int n, int k, int try[], bool col[], bool diag45[], bool diag135[], int\* try\_count) {

int i, row;

for (i=0; i<n; i++)

col[i] = 0;

for (i=0; i<2\*n; i++)

diag45[i] = diag135[i] = 0;

for (row=0; row<k; row++) {

int open = 0;

int selected = 0;

for (i=0; i<n; i++) {

if (!col[i] && !diag45[n+i-row] && !diag135[i+row]) {

++open;

if (rand() % open == 0)

selected = i;

}

}

if (open == 0)

return 0;

++\*try\_count;

try[row] = selected;

col[selected] = 1;

diag45[n+selected-row] = 1;

diag135[selected+row] = 1;

}

return 1;

}

bool backtrack(int n, int row, int try[], bool col[], bool diag45[], bool diag135[], int\* try\_count) {

int i;

if (row == n)

return 1;

for (i=0; i<n; i++) {

if (!col[i] && !diag45[n+i-row] && !diag135[i+row]) {

++\*try\_count;

try[row] = i;

col[i] = diag45[n+i-row] = diag135[i+row] = 1;

if (backtrack(n, row+1, try, col, diag45, diag135, try\_count))

return 1;

col[i] = diag45[n+i-row] = diag135[i+row] = 0;

}

}

return 0;

}

int search\_queens(int n, int k) {

int try[n];

bool col[n], diag45[2\*n], diag135[2\*n];

int try\_count = 0;

while (1) {

if (queens\_lv(n, k, try, col, diag45, diag135, &try\_count)) {

if (backtrack(n, k, try, col, diag45, diag135, &try\_count))

return try\_count;

}

}

}

int main() {

int n, k, i;

int now, min, bestk;

for (n=12; n<=20; n++) {

bestk = 0;

for (k=0; k<=n; k++) {

now = 0;

for (i=0; i<TRIALS; i++) {

now += search\_queens(n, k);

}

if (bestk == 0 || now < min) {

bestk = k;

min = now;

}

}

printf("n=%d bestk=%d nodes=%f\n", n, bestk, min \* 1.0 / TRIALS);

}

return 0;

}

近似算法

G中最大团的size为α当且仅当Gm里最大团的size是mα。

证明：G中最大团的size为α，等价于G中存在size为α的团，且不存在size为α+1的团。欲证原命题，只需证明：

1. G中有size为α的团时，Gm中有size为mα的团；
2. Gm中有size为mα的团时，G中有size为α的团；
3. G中没有size为α+1的团时，Gm中没有size为mα+1的团；（最大性）
4. Gm中没有size为mα+1的团时，G中没有size为α+1的团。（最大性）

先证明（1）：若G中有size为α的团 {V1, V2, …, Vα}, 其在Gm中对应的顶点为 {V11, V12, …, V1m, V21, …, V2m, …}。下证 {Vij}, i=1…α, j=1…m 是团。

任取 (i1, j1), (i2, j2), 若 j1=j2，则这两个顶点属于原图的同一个副本，由于 {V1…Vα} 是团，这两个顶点相邻。若 j1≠j2，则这两个顶点属于原图的不同副本，由Gm的定义，这两个顶点相邻。

再证明（2）：若Gm中有size为mα的团，设其分布在原图的m个副本中的顶点数分别为 c1, c2, …, cm。由抽屉原理，至少有α个顶点属于原图的同一个副本，这α个顶点在Gm中是团，由Gm的构造知在G中也是团。

（3）等价于Gm中有size为mα+1的团时，G中有size为α+1的团。Gm中构成团的mα+1个顶点，根据抽屉原理，至少有α+1个顶点属于原图的同一个副本，这α+1个顶点在Gm中是团，由Gm的构造知在G中也是团。

（4）等价于G中有size为α+1的团时，Gm中有size为mα+1的团。由（1），G中有size为α+1的团时，Gm中有size为m(α+1)的团，其中任取mα+1个元素，就是一个团。

证毕。