Department of Computer Science
Technical University of Cluj-Napoca

Introduction to Artificial Intelligence

Laboratory activity 2021-2022

Name: Pop Alexandru si Vilcelean Dalia

Group: 30232

Contents

Chapter 1

Algoritmi de cautare: Jocul Bilei in Labirint

1.1 Introducere

Care a fost scopul? Scopul acestei sarcini a fost să ne familiarizăm cu Limbajul Python prin Proiectul Berkeley Pacman, prin modificarea codului existent și adăugarea unor algoritmi și fragmente proprii, pentru a ne crea propriul nostru joc functional. Pentru a realiza acest lucru, am decis să ne împărțim proiectul în două părți: algoritmi de probleme de căutare, precum și o interfata a jocului Labirinth, cu care utilizatorul poate interacționa.
Am ales să facem asta pentru a putea explora cât mai multe, atât programarea limbajului cat și a proiectului Pacman, dezvoltat ulterior in jocul nostru personalizat. Problemele de căutare pun accent pe versatilitatea jocului si il face sa functioneze cat mai corect.
Din păcate, caracteristicile jocului nu au putut fi încorporate în problemele de căutare. Asta ar fi însemnat redefinirea unei părți foarte mari a proiectului într-o perioadă scurtă de timp.

1.2 Keyboard Agent

1.2.1 Jocul

Experienta user-ului Ideea prncipala a jocului nostru este simpla si a ramas aceeasi: O bila se afla intr-o pozitie pe o harta cu un labirint plin de gaur. Scopul acestei bile este sa ajunga pe partea cealalta a hartii fara sa cada in nicio groapa. Aceasta bila va fi controlata de catre jucator folosind tastatura.

Ce am Adaugat nou? Cu fiecare miscare mai aproape de linia de Finish, bila se va misca din ce in ce mai rapid, iar in comparative cu jocul PacMan, aceasta bila (actorul principal va trebui sa se fereasca de gauri care vor fi create ca s niste “fantome fixe”.

1.2.2 Logica din spatele lui

In aceasta sectiune va vom explica modul in care am gandit si am implementat acest joc.

clasa Cons - se afla valorile constante care se vor folosi in interiorul altor clase.

clasa Board - se initializeaza interfata grafica, se deseneaza labirintul pe ecrean, se verifica coliziunile, se misca bila in functie de tasta apasata si se verifica stadiul in care se afla jocul.

Clasele Stack, Queue si PriorityQueue - sunt clase in care sunt implementate functionalitatile unei stive, a unei cozi si a unei cozi prioritare.

clasa Search- se afla algoritmii de cautare, iar in clasa GameStart se incepe jocul si se aplica cautarea selectata.

1.2.3 Cum rulam jocul?

Pentru a rula jocul, userul va decomenta la inceputul proiectului apelul functiei self.initGame1() pentru labirintul usor, self.initGame2() pentru labirintul mediu sau self.initGame3() pentru labirintul greu.

```
class Board(Canvas):

    def __init__(self):
        super().__init__(width=Cons.BOARD_WIDTH, height=Cons.BOARD_HEIGHT,
            background="white", highlightthickness=0)#, ballX=Cons.BALLX,
        #easy mode
        self.initGame1()
        #medium mode
#        self.initGame2()
        #hard mode
#        self.initGame3()
        self.pack()
```

Figure 1.1: Rularea

1.3 Interfata

Interfata este formata din 3 harti cu dificultati diferite: easy, medium si hard.

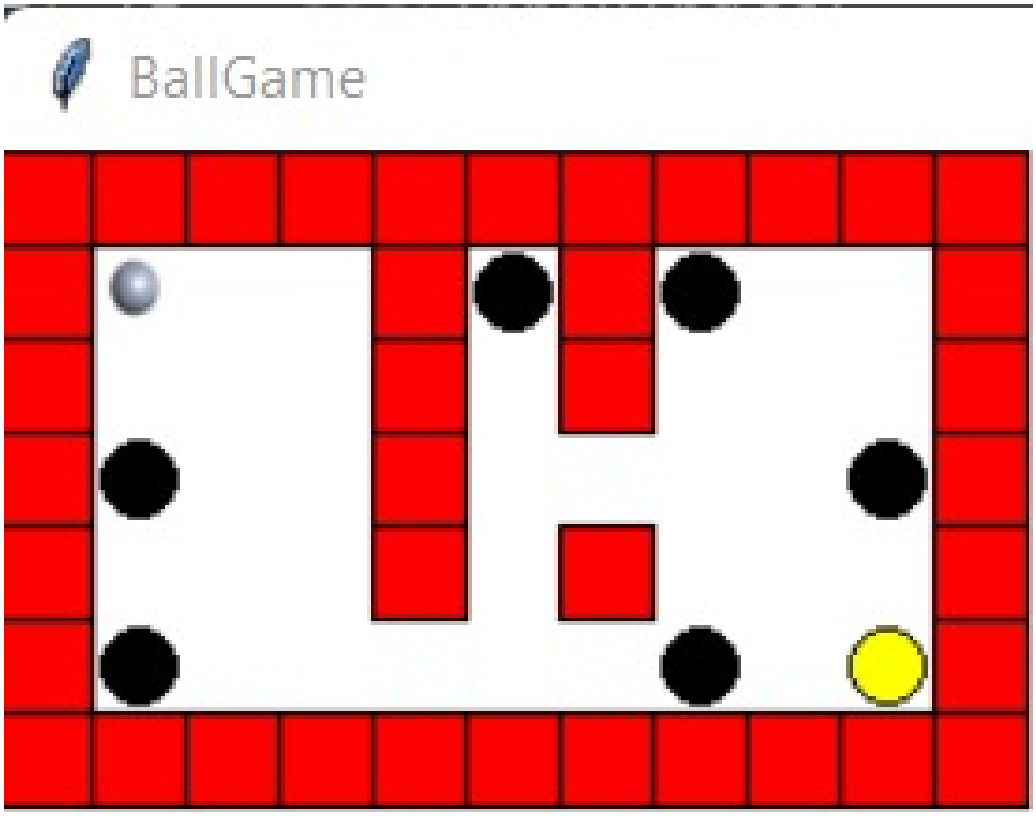

Figure 1.2: Harta "easy" a jocului

1.4 Noile Harti

Am creat acest joc astfel incat acesta sa aiba 3 niveluri de dificultate: usor, medu si greu. Utilizatorul va putea accesa fiecare dintre aceste niveluri utilizand comenzile:

Interfata a fost creata astfel incat sa fie ca mai usor de accesat de catre utilizatori, pentru a le crea o placer sporita de a se juca un joc cat mai simplist.

Pentru a arata ca toti algoritmii si toate implementarile functioneaza asa cum trebuie, am creat o serie de harti, mai usoare sau mai dificile, pentru fiecare problema.

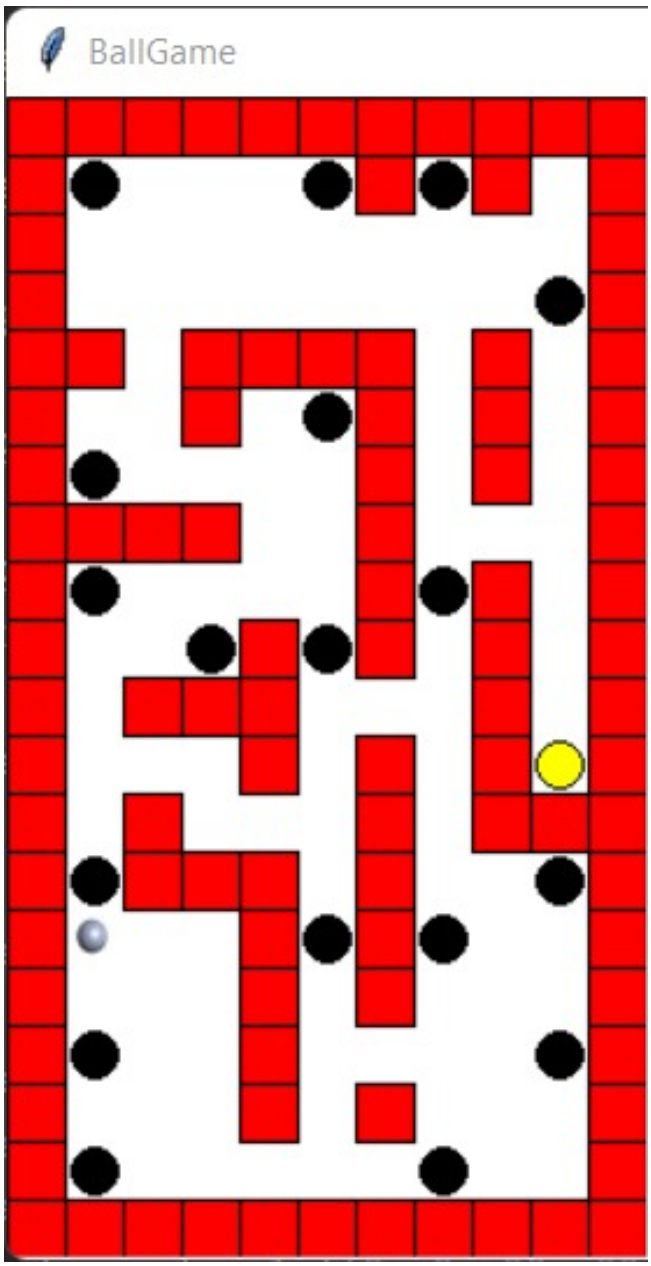

Figure 1.3: Harta ”medium” a jocului

Asa, numele fiecaror layout-uri sunt:

Harta easy este o simpla mapa Unde bila se poate plimba cu usurinta pentru a ajunge in locul de Finish.

Aceasta are doar cateva gauri care ii poate ingreuna castigul, ceea ce o face de un nivel usor la un astfel de joc.

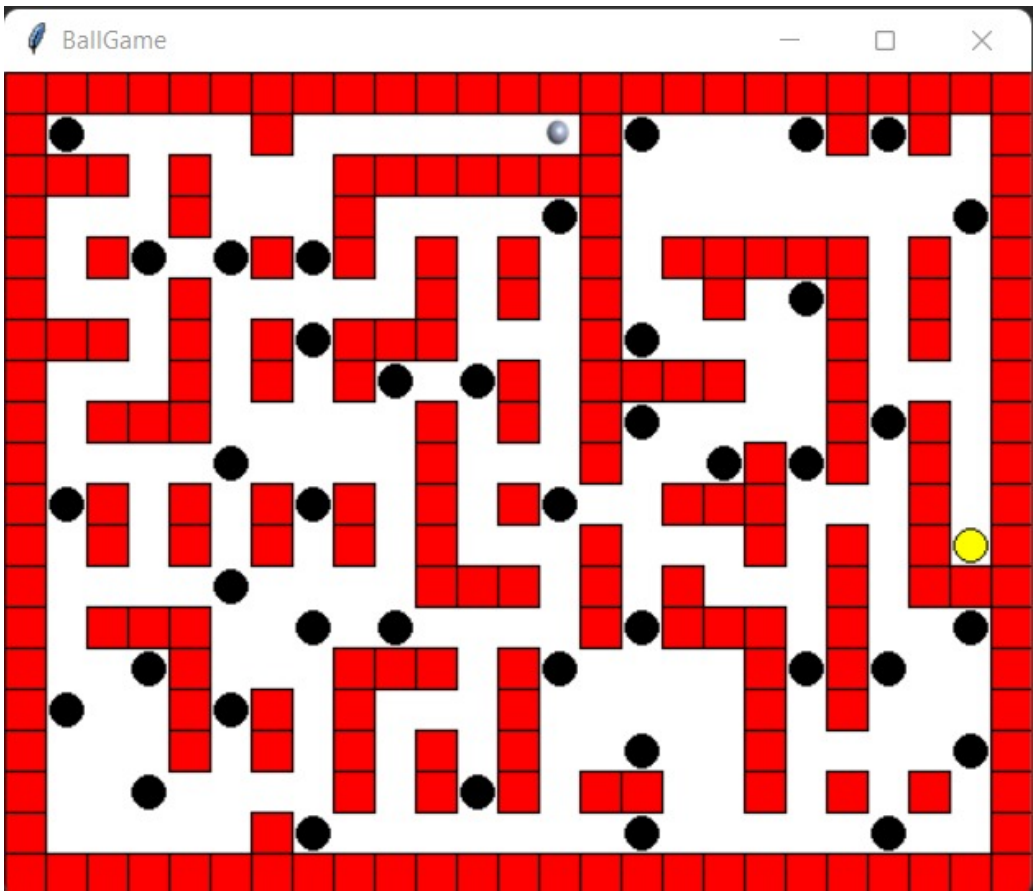

Figure 1.4: Harta ”hard” a jocului

1.5 Algoritmii

Pentru acest proiect am avut de implementat o serie de algoritmi pentru a reusi sa facem jocul Labirinth functional:

- Breath First Search (BFS)
- Depth First Search (DFS)
- Uniform Cost Search
- A* Search

Breath First Search(BFS) este un algoritm pentru căutarea unei structuri de date arborescente pentru un nod care satisface o proprietate dată. Începe de la rădăcina copacului și explorează toate nodurile la adâncimea actuală înainte de a trece la nodurile de la următorul nivel de adâncime. Memoria suplimentară, de obicei o coadă, este necesară pentru a ține evidența nodurilor copil care au fost întâlnite, dar care nu au fost încă explorate.

Această funcție împinge nodurile nevizitate în coadă.

Nodurile sunt afișate unul câte unul și se parcurg următorii pași:

1. BNodul este marcat ca vizitat.
2. Dacă este un nod de obiectiv, bucla se oprește, iar soluția se obține prin backtracking folosind părinții stocați.
3. Dacă nu este un nod obiectiv, este extins.
4. Dacă nodul succesor nu este vizitat și nu a fost extins ca copil al altui nod, apoi este împins în coadă și părintele său este stocat."'

Depth First Search (DFS) este un algoritm pentru parcurgerea sau căutarea structurilor de date arborescente sau grafice.
Algoritmul începe de la nodul rădăcină (selectând un nod arbitrar ca nod rădăcină în cazul unui grafic) și explorează cât mai departe posibil de-a lungul fiecărei ramuri înainte de a reveni.

Această funcție împinge nodurile nevizitate în stivă.

Nodurile sunt afișate unul câte unul și se parcurg următorii pași:

1. Nodul este marcat ca vizitat.
2. Dacă este un nod de obiectiv, bucla se oprește, iar soluția se obține prin backtracking folosind părinții stocați.
3. Dacă nu este un nod obiectiv, este extins.
4. Dacă nodul succesor nu este vizitat, atunci acesta este împins în stivă și părintele său este stocat.

Uniform Cost Search este o variantă a algoritmului lui Dijikstra. Aici, în loc să inserăm toate nodurile într-o coadă de prioritate, inserăm doar sursa, apoi inserăm unul câte unul când este necesar. La fiecare pas, verificăm dacă articolul este deja în coada de prioritate (folosind matricea vizitată). Dacă da, executăm tasta de scădere, altfel o introducem. Această variantă a lui Dijkstra este utilă pentru grafice infinite și pentru acele grafice care sunt prea mari pentru a fi reprezentate în memorie. Căutarea Uniform-Cost este folosită în principal în inteligența artificială. Această funcție împinge nodurile nevizitate în coada priori-

tară.

Nodurile sunt afișate unul câte unul și se parcurg următorii pași:

1. Nodul este marcat ca vizitat.
2. Dacă este un nod de obiectiv, bucla se oprește, iar soluția se obține prin backtracking folosind părinții stocați.
3. Dacă nu este un nod obiectiv, este extins.
4. Dacă nodul succesor nu este vizitat, se calculează costul acestuia.
5. Dacă costul nodului succesor a fost calculat mai devreme în timpul extinderii unui alt nod, iar dacă noul cost calculat este mai mic decât vechiul cost, atunci costul și părintele sunt actualizate, și este împins în coada de prioritate cu un cost nou ca prioritate.

A* Search este un algoritm de parcurgere a grafului și de căutare a căii, care este adesea folosit în multe domenii ale informaticii datorită completitudinii, optimității și eficienței optime. Un dezavantaj practic major este complexitatea spațiului ,deoarece stochează toate nodurile generate în memorie.
Astfel, în sistemele practice de rutare de călătorie, este în general depășit de algoritmi care pot preprocesa graficul pentru a obține performanțe mai bune, precum și abordări limitate de memorie; cu toate acestea, A* este încă cea mai bună soluție în multe cazuri.

Această funcție împinge nodurile nevizitate în coada prioritară.

Nodurile sunt afișate unul câte unul și se parcurg următorii pași:

1. Nodul este marcat ca vizitat.
2. Dacă este un nod de obiectiv, bucla se oprește, iar soluția se obține prin backtracking folosind părinții stocați.
3. Dacă nu este un nod obiectiv, este extins.
4. Dacă nodul succesor nu este vizitat, costul acestuia este calculat folosind funcția euristică.
5. Dacă costul nodului succesor a fost calculat mai devreme în timpul extinderii unui alt nod, iar dacă noul cost calculat este mai mic decât vechiul cost, atunci costul și părintele sunt actualizate, și este împins în coada de prioritate cu un cost nou ca prioritate.

1.6 Concluzii si Dezvoltari Ulterioare

In concluzie, jocul Bilei in Labirint este unul foarta usor de jucat, accesibil pentru orice utilizator dornic sa se relaxeze, jucandu-se.
Recomandam cu drag acest joc.

DezvoltariUlterioare:
crearea unei interfete mai usor de folosit ar fi ideala ca si o viitoare dezvoltare, acest lucru ar determina utilizatorul sa se joace cat mai mult.

Chapter 2

Anexa- Codul sursa

BFS:

```
def bfs(self,maze):
crt = (self.getStartState(maze), [])
frontiera = Queue()
frontiera.push(crt)
teritoriu = []
while not frontiera.isEmpty():
nodcrt = frontiera.pop()
teritoriu.append(nodcrt[0])
if self.isGoalState(maze,nodcrt[0]):
return nodcrt[1];

childNodes = []
ycrt, xcrt = nodcrt[0]
print(ycrt, xcrt)
if maze == 1:
if self.l1[ycrt - 1][xcrt] != 1:
childNodes.append(((ycrt - 1, xcrt), 'UP'))
if self.l1[ycrt][xcrt + 1] != 1:
childNodes.append(((ycrt, xcrt + 1), 'RIGHT'))
if self.l1[ycrt + 1][xcrt] != 1:
childNodes.append(((ycrt + 1, xcrt), 'DOWN'))
if self.l1[ycrt][xcrt - 1] != 1:
childNodes.append(((ycrt, xcrt - 1), 'LEFT'))
elif maze == 2:
if self.l2[ycrt - 1][xcrt] != 1:
childNodes.append(((ycrt - 1, xcrt), 'UP'))
if self.l2[ycrt][xcrt + 1] != 1:
childNodes.append(((ycrt, xcrt + 1), 'RIGHT'))
if self.l2[ycrt + 1][xcrt] != 1:
childNodes.append(((ycrt + 1, xcrt), 'DOWN'))
if self.l2[ycrt][xcrt - 1] != 1:
childNodes.append(((ycrt, xcrt - 1), 'LEFT'))
elif maze == 3:
if self.l3[ycrt - 1][xcrt] != 1:
```

```
childNodes.append(((ycrt - 1, xcrt), 'UP'))
if self.l3[ycrt][xcrt + 1] != 1:
childNodes.append(((ycrt, xcrt + 1), 'RIGHT'))
if self.l3[ycrt + 1][xcrt] != 1:
childNodes.append(((ycrt + 1, xcrt), 'DOWN'))
if self.l3[ycrt][xcrt - 1] != 1:
childNodes.append(((ycrt, xcrt - 1), 'LEFT'))

for node in childNodes:
(stare, actiune) = node
front = []
for nod in frontiera.list:
front.append(nod[0])
if stare not in teritoriu and stare not in front:
cale = nodcrt[1] + [actiune]
frontiera.push((stare, cale))
return []
```

DFS:

```
def dfs(self,maze):
crt = (self.getStartState(maze), [])
print(crt)
frontiera = Stack()
frontiera.push(crt)
teritoriu = []
while not frontiera.isEmpty():
nodcrt = frontiera.pop()
teritoriu.append(nodcrt[0])
if self.isGoalState(maze, nodcrt[0]):
return nodcrt[1]

succesori = []
ycrt,xcrt=nodcrt[0]
print(ycrt,xcrt)
if maze==1:
if self.l1[ycrt-1][xcrt] !=1:
succesori.append(((ycrt-1,xcrt),'UP'))
if self.l1[ycrt][xcrt+1]!=1:
succesori.append(((ycrt,xcrt+1),'RIGHT'))
if self.l1[ycrt + 1][xcrt] != 1:
succesori.append(((ycrt + 1, xcrt), 'DOWN'))
if self.l1[ycrt][xcrt-1]!=1:
succesori.append(((ycrt,xcrt-1),'LEFT'))
elif maze==2:
if self.l2[ycrt - 1][xcrt] != 1:
succesori.append(((ycrt - 1, xcrt), 'UP'))
if self.l2[ycrt][xcrt + 1] != 1:
```

```
succesori.append(((ycrt, xcrt + 1), 'RIGHT'))
if self.l2[ycrt + 1][xcrt] != 1:
succesori.append(((ycrt + 1, xcrt), 'DOWN'))
if self.l2[ycrt][xcrt - 1] != 1:
succesori.append(((ycrt, xcrt - 1), 'LEFT'))
elif maze==3:
if self.l3[ycrt-1][xcrt] !=1:
succesori.append(((ycrt-1,xcrt),'UP'))
if self.l3[ycrt][xcrt+1]!=1:
succesori.append(((ycrt,xcrt+1),'RIGHT'))
if self.l3[ycrt + 1][xcrt] != 1:
succesori.append(((ycrt + 1, xcrt), 'DOWN'))
if self.l3[ycrt][xcrt-1]!=1:
succesori.append(((ycrt,xcrt-1),'LEFT'))

for succesor in succesori:
(stare, actiune) = succesor
front = []
for nod in frontiera.list:
front.append(nod[0])
if stare not in teritoriu and stare not in front:
cale = nodcrt[1] + [actiune]
frontiera.push((stare, cale))
return []
```

UCS:

```
def ucs(self,maze):
nodCrt = (self.getStartState(maze), [], 0)
frontiera = PriorityQueue()
teritoriu = []
frontiera.push(nodCrt, 0)
while not frontiera.isEmpty():
nodCrt = frontiera.pop()
if self.isGoalState(maze,nodCrt[0]):
return nodCrt[1]
teritoriu.append(nodCrt[0])
succesori = []
ycrt, xcrt = nodCrt[0]
print(ycrt, xcrt, nodCrt[2])
if maze == 1:
if self.l1[ycrt - 1][xcrt] != 1:
succesori.append(((ycrt - 1, xcrt), 'UP', 1))
if self.l1[ycrt][xcrt + 1] != 1:
succesori.append(((ycrt, xcrt + 1), 'RIGHT',1))
if self.l1[ycrt + 1][xcrt] != 1:
succesori.append(((ycrt + 1, xcrt), 'DOWN',1))
if self.l1[ycrt][xcrt - 1] != 1:
```

```
succesori.append(((ycrt, xcrt - 1), 'LEFT',1))
elif maze == 2:
if self.l2[ycrt - 1][xcrt] != 1:
succesori.append(((ycrt - 1, xcrt), 'UP',1))
if self.l2[ycrt][xcrt + 1] != 1:
succesori.append(((ycrt, xcrt + 1), 'RIGHT',1))
if self.l2[ycrt + 1][xcrt] != 1:
succesori.append(((ycrt + 1, xcrt), 'DOWN',1))
if self.l2[ycrt][xcrt - 1] != 1:
succesori.append(((ycrt, xcrt - 1), 'LEFT',1))
elif maze == 3:
if self.l3[ycrt - 1][xcrt] != 1:
succesori.append(((ycrt - 1, xcrt), 'UP',1))
if self.l3[ycrt][xcrt + 1] != 1:
succesori.append(((ycrt, xcrt + 1), 'RIGHT',1))
if self.l3[ycrt + 1][xcrt] != 1:
succesori.append(((ycrt + 1, xcrt), 'DOWN',1))
if self.l3[ycrt][xcrt - 1] != 1:
succesori.append(((ycrt, xcrt - 1), 'LEFT',1))
for succesor in succesori:
(stare, mutare, cost) = succesor
if stare not in teritoriu and stare not in (nod[0] for nod in frontiera.heap):
cale = nodCrt[1] + [mutare]
g = nodCrt[2]+1
frontiera.push((stare, cale, g), g)
elif stare in (nod[0] for nod in frontiera.heap):
cale = nodCrt[1] + [mutare]
g = nodCrt[2]+1
frontiera.update((stare, cale, g), g)
return []
```

Bibliography

[1] Boris A Kordemsky. *The Moscow puzzles: 359 mathematical recreations.* Courier Corporation, 1992.