Forskerspirer 2018

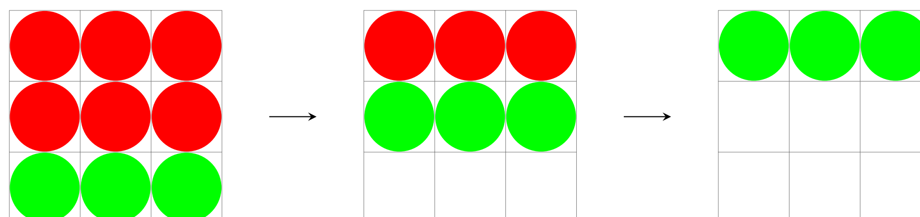
Naturvidenskab

Niels Brock

Cops and Robbers

Forfatter:
Mathias Theodor Jul Overby

Forskerkontakter:
Mikkel Abrahamsen
Søren Eilers



29. oktober 2018

Tak til

Jeg vil først og fremmest takke Mikkel Abrahamsen og Søren Eilers for at have hjulpet mig med at opstille problemformuleringen og med at realisere projektet. Den faglige sparring hjalp mit projekt meget og generelt blev mine datalogiske evner og forståelse skærpet herigennem. Derudover vil jeg også takke min koordinator for projektet, Robin Højager Olesen, som pressede mig til at gennemføre projektet lige fra starten af. Tak til alle andre, som har bidraget til projektet!

Forskerkontakter

Mikkel Abrahamsen
Adjunkt
Datalogisk Institut
Københavns Universitet

Søren Eilers
Professor
Institut for Matematisk Fag
Københavns Universitet

Mathias Theodor Jul Overby
Niels Brock 3.R

1 Indledning

I dette projekt ønsker jeg at få en bedre forståelse for, hvordan grafsøgningsalgoritmer og diskret matematik kan bruges på en praktisk problemstilling. Mange matematiske teoretiske problemstillinger kan i dag løses med computerkraft ved numerisk analyse. Efter svaret er fundet, kan der arbejdes på at få en mere teoretisk og matematisk forståelse af, hvorfor svaret er, som det er.

Grafer er en matematisk betegnelse for netværk og bruges til at modellere utallige praktiske problemstillinger, såsom vejnetværk på GPS eller tracking af hvem kender hvem på Facebook. I denne opgave laver jeg en praktisk implementering af en teoretisk problemstilling: *Cops and Robbers*. Min problemstilling undersøger, hvor mange ressourcer der er nødvendige for at gennemsøge en graf. Spørgsmål som dette, om hvordan man afsøger grafer effektivt, har stor videnskabelig bevågenhed. Der er i de seneste år blevet forsket meget i varianter af denne problemstilling med henblik på at få såvel en praktisk som teoretisk forståelse af grafer.

2 Problembeskrivelse

Problemet, jeg har sat mig for at løse, er kendt som *Cops and Robbers* og er et spil, som spilles på en $n \times n$ ikke-orienteret gitter graf med $2(n-1)n$ kanter. I et $n \times n$ gitter forstås krydspunkterne, som talpar (i, j), hvor $i, j \in \{1, ..., n\}$ og hvor (i, j) og (i', j') er forbudne hvis og kun hvis $|i - i'| + |j - j'| = 1$. Hvis eksempelvis $n = 3$, så findes der 9 krydspunkter, som ses i figur 1. I figurerne bliver et krydspunkt repræsenteret som et felt.

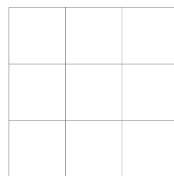


Figure 1: Eksempel på et 3×3 gitter graf

I spillet gælder det om at fange røveren, som bevæger sig rundt på grafen, med et givent antal betjente til rådighed. I den variant af spillet jeg har

valgt at fokusere på, bevæger betjente og røvere sig på samme tid i træk og derudover er røveren usynlig. Når en betjent bevæger sig fra krydspunkt a til krydspunkt b, så bliver røveren fanget, hvis han samtidig bevæger sig fra krydspunkt b til a. Røveren bliver også fanget, hvis han befinder sig på samme krydspunkt som en betjent. Det gælder om at finde en strategi for betjenten(e), som altid vil resultere i, at røveren bliver fanget, lige meget hvor røveren starter og efterfølgende rykker.

Spillet illustreres i figur 2 nedenfor. I min problemstilling kendes startpositionen af røveren ikke og dermed er alle krydspunkter, hvor der ikke er betjente, mulige placeringer for røveren. Figuren er kun til for at illustrere hvordan røverens muligheder for at være i forskellige positioner øges over tid. Det er i og for sig ligegyldigt hvor røveren er, da det gælder om at fjerne alle mulighederne, hvor røveren kunne være. Røveren er kun fanget, hvis alle muligheder er fjernet.

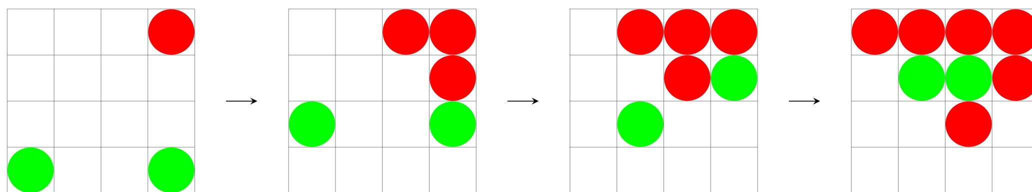


Figure 2: 4×4, 3 træk

Her spiller 2 betjente mod en røver. Farven grøn beskriver betjentene og farven rød beskriver mulige steder røveren kan befinde sig. Røveren og betjentene kan hver tur bevæge sig op, ned, til højre, til venstre eller blive, hvor de er.

Røveren starter i dette tilfælde i det øverste krydspunkt til højre og betjentene starter i de nederste krydspunkter til højre og venstre. I det første træk bevæger betjentene sig begge op og på samme tid spreder de mulige steder, hvor røveren kan befinde sig. Det vil sige, røveren kunne fra trin 1 til trin 2 rykke sig til venstre, ned eller blive. Derfor er der nu 3 krydspunkter, hvor der er mulighed for, at der er en røver. Hvis en betjent rykker sig hen på et krydspunkt, hvor der er mulighed for at røveren befinder sig, fjernes denne mulighed, som kan ses i trin 2 til 3.

Spørgsmålet, der forsøges besvaret, er, hvor mange betjente der skal til for at fange røveren for en graf af størrelse $n \times n$. Der er en kendt strategi til

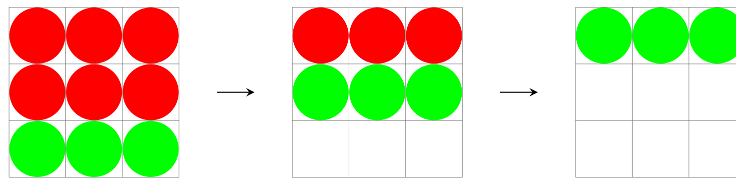


Figure 3: 3×3. Løsning med n betjente

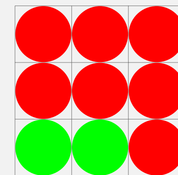
at løse problemet, når antal betjente er lig n, som illustreret i figur 3. Her rykker betjentene systematisk op og udelukker dermed de steder, hvor røveren kunne befinde sig.

Det er ligeledes blevet bevist af Brass et al. [6], at det ikke er muligt at fange røveren med mindre end eller lig med $\lfloor n/2 \rfloor$ betjente. Intervallet $(\lfloor n/2 \rfloor + 1, n)$ er dog stadigt åbent, og det er det, jeg ønsker at indsnævre. Da det er bevist med n betjente, er det interessant at undersøge om det er muligt med $n-1$ betjente. Det er hverken bevist eller modbevist, at man kan fange røveren med $n-1$ betjente.

I denne analyse anvendes computersimuleringer til at undersøge dette. I disse simuleringer bliver alle de forskellige træk betjentene kan tage gennemsøgt, for at finde ud af, om der findes en kombination af træk, som fører til, at røveren bliver fanget. Svaret kan variere for de forskellige n, og derfor foretages simulationen med forskellige n (størrelsen på grafen). Bemærk at det er ligegyldigt, hvor betjentene starter, idet de altid kan rykke hen i enhver position, da de ikke har en tidsbegrænsning.

Problemstilling

I en $n \times n$ ikke-orienteret gitter graf undersøges om der findes en strategi, som altid resulterer i at en usynlig røver bliver fanget med $n-1$ betjente til rådighed. Røverens startposition kendes ikke og røveren anses for at være fanget, hvis der ikke er flere steder, hvor røveren kan befinde sig. Dette undersøges for forskellige n. I figuren til højre er $n = 3$, betjentene er illustreret med grønt og røverens mulige positioner er røde.



3 Metode

3.1 Terminologi

Et *træk* består af, at hver betjent bevæger sig i en retning. Det kan være op, ned, til højre, til venstre eller at blive stående. Betjentene må kun bevæge sig på grafen og kan ikke bevæge sig skråt.

Ved *tilstanden af en graf* menes samtlige krydspunkters situation (befinder der sig en betjent, mulighed for røver eller ingenting). Hvis tilstanden af graf 1 er den samme som graf 2, så betyder det, at de er den samme graf. Der tages højde for rotationer og spejlinger, når der sammenlignes tilstande, som forklares i sektion 3.3.

3.2 Logikken bag algoritmen

Metoden, til besvarelse af spørgsmålet om det er muligt at fange røveren på en $n \times n$ graf med $n - 1$ betjente, er, at gennemsøge alle de tilladte træk betjentene må tage. Spørgsmålet er, om der findes en kombination af træk, som fører til at røveren altid bliver fanget. Udfordringen med dette er, at betjentene må foretage vilkårligt mange træk, da der ikke er nogen tidsbegrænsning. Der vil hermed altid være et nyt træk, som betjentene kan foretage sig og dette fortsætter i al uendelighed. Dette kan løses ved, inden hvert træk, at tjekke, om grafen allerede har været i præcis den tilstand før. For hvis grafen allerede har været i den tilstand før, så er der allerede blevet kigget på alle de efterfølgende muligheder af træk ud fra den tilstand. Der findes altså et begrænset antal muligheder for træk, der kan foretages. Flowchartet i figur 4 beskriver den overordnede logik bag ved algoritmen, som er rekursivt opbygget.

Der startes med at specificere størrelsen på grafen, n, og antal af betjente. For at besvare denne problembeskrivelse er antal af betjente altid $n - 1$. Derefter placeres alle betjentene på grafens krydspunkter. Det er underordnet, hvor betjentene starter, da alle tilstande gennemsøges. Det tjekkes om der findes et træk, ud fra den oprindelige tilstand, som fører til at grafen findes i en tilstand, som ikke er set før. Hvis der er det, så tjekkes det, om enhver position for røveren er udelukket, hvis ja, så er der fundet en løsning. Ellers tjekker programmet om der findes et træk, som fører til en tilstand,

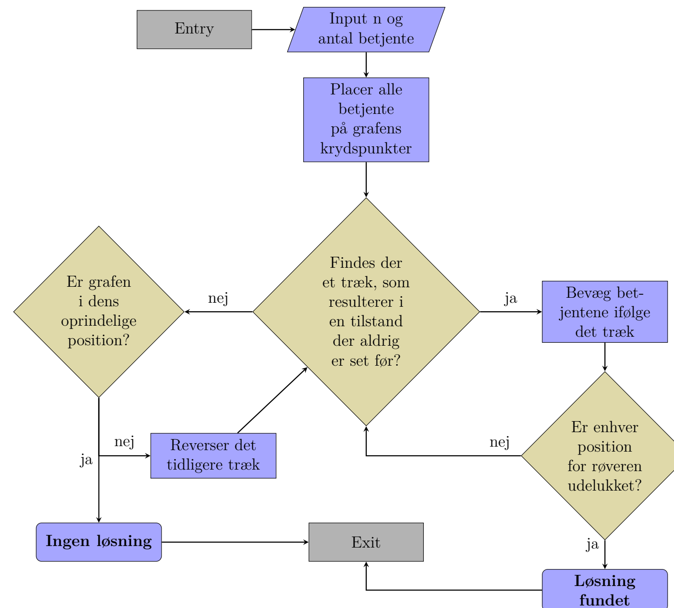


Figure 4: Flowchart over algoritmen

som ikke er set før, ud fra den nye tilstand, som lige er blevet fundet. Dette fortsætter indtil der ikke findes flere træk, som fører til nye tilstande. Derfra reverseres det seneste træk, og der tjekkes, om der findes et træk, som fører til en ny tilstand ud fra den reverserede tilstand. Dette fortsætter indtil der ikke findes flere mulige træk, som fører til nye tilstande fra den reverserede tilstand. Så reverseres det træk, som førte til den nuværende tilstand. Denne proces fortsætter indtil grafen er i dens oprindelig tilstand og der ikke findes flere træk, som fører til nye tilstande, hvilket betyder at alle træk og deres efterfølgende træk er blevet undersøgt. Dette betyder, at der ikke findes en kombination af træk, som fører til at røveren fanges. Gennemsøgningen af alle de mulige træk er illustreret i figur 5.

Hver knude repræsenterer en tilstand og den første knude er den oprindelige tilstand. $\lambda(k)$ beskriver antallet af mulige træk fra tilstand k, som resulterer i en tilstand aldrig set før ud. Figuren visualiserer algoritmen, der fortsætter indtil der ikke er flere træk, som resulterer i en tilstand aldrig set før.

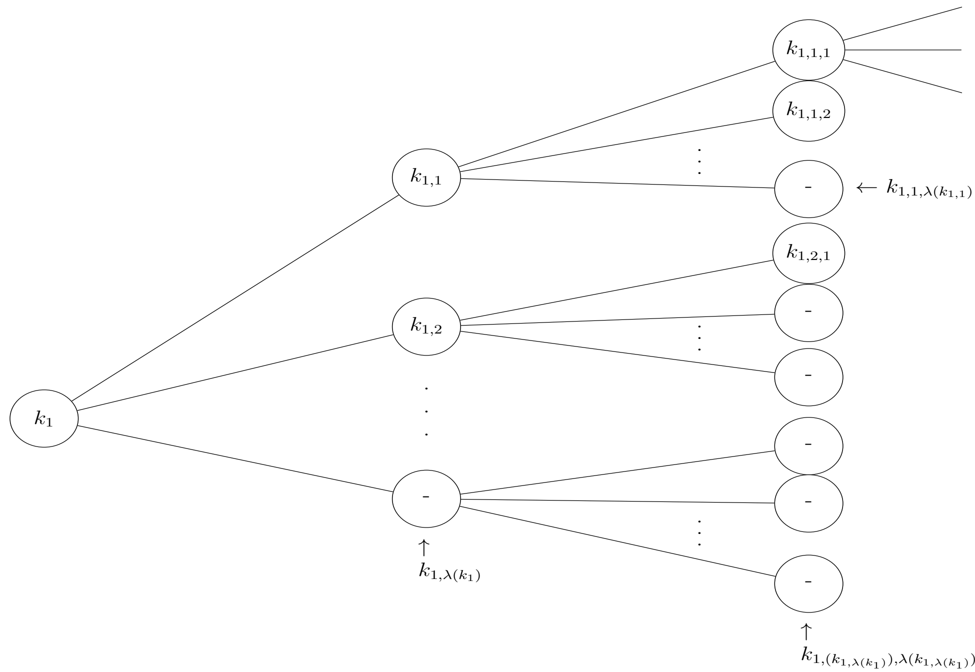


Figure 5: Træstruktur

Denne gennemgang af træk er en dybde-først søgning. På figur 5 betyder det, at der først gennemgås tilstand k_1, derefter $k_{1,1}$ og så $k_{1,1,1}$ indtil der ikke er flere tilstande, som fører til nye tilstande ud fra den nuværende tilstand. Derfra reverseres det seneste træk, hvilket er ensbetydende med, at der rykkes en knude tilbage. Hvis $k_{1,1,1}$ var den sidste knude, så ville der rykkes tilbage til $k_{1,1}$ og derefter ville $k_{1,1,2}$ være den næste knude. Processen fortsætter indtil programmet befinder sig i den oprindelige tilstand (k_1) og ikke har mulighed for at finde nye tilstande, hvilket er ensbetydende med at alle tilstande, som var mulige ud fra den oprindelige tilstand er gennemsøgt. Hvis programmet når hertil, så betyder det, at der ikke findes en kombination af træk, som fører til at røveren kan fanges med det givne antal betjente.

3.3 Tilstand af grafen

Den valgte metode kræver, at alle tidligere graftilstande gemmes for at kunne sammenligne senere grafer med de tidligere graftilstande. For at spare på brug af hukommelse gemmes grafer ikke som multidimensionale arrays, men som to tal, der beskriver henholdsvis positionen af betjentene og de steder, hvor røveren kan befinde sig. Dette kræver, at der findes et unikt tal for hver tænkelig position for betjentene og tilsvarende for mulighederne for røveren.

Dette løses ved at give hvert krydspunkt et forskelligt primtal. Betjentenes position identificeres nu ved produktet af de primtal, der associeres med de krydspunkter, hvor betjentene står. Haves et id kan placeringen af betjentene identificeres ved primfaktoropløsning af id'et. På tilsvarende måde kan et unikt id for røverens muligheder beregnes. Computeren er altså nu i stand til at repræsentere enhver graf med to tal. Hermed spares en masse hukommelse ift. at gemme de tidligere undersøgte løsninger for mindre n. Dog har det nogle begrænsninger for større n, som bliver præsenteret senere.

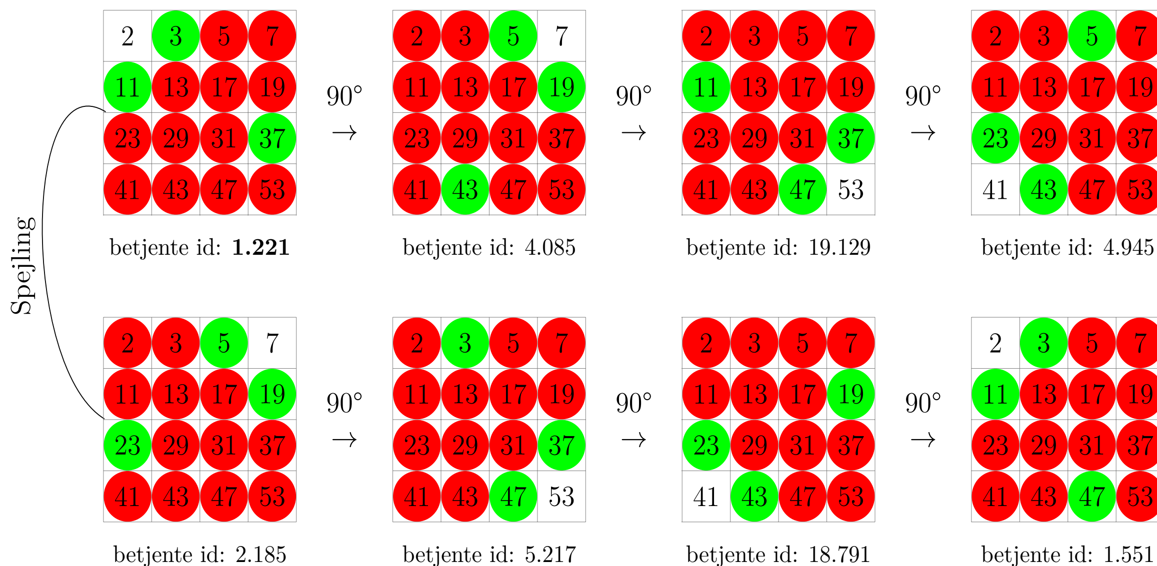


Figure 6: Repræsentation af grafer ved hjælp af primtal

Derudover roteres og spejles grafen på 7 forskellige måder for at finde ud af, om grafen har været i den samme tilstand bare fra en anden vinkel. Figur 6 viser den samme graf, bare roteret og spejlet. En graf er den samme selvom den roteres eksempelvis 90°. Hver af disse 8 grafer har muligvis forskellige ids selvom de egentlig svarer til den samme graf. Som tidligere nævnt, beregnes et id ved at tage produktet af alle de primtal, som associeres ved krydspunktet for enten betjente eller røvere. Når der beregnes id for en graf roteres og spejles den på de forskellige måder, og der beregnes betjentenes id for hver af disse grafer. Derfra tages den mindste af disse ids og det tilsvarende røver id beregnes for den rotation med lavest betjente id. Denne proces er illustreret i figur 6. I dette tilfælde er den graf uden rotation den med lavest betjente id. Betjentenes id er i dette tilfælde 1.221 ($= 3 \times 11 \times 37$) og røverens id er $5 \times 7 \times 13 \times 17 \times 19 \times 23 \times 29 \times 31 \times 41 \times 43 \times 47 \times 53$. Hver gang der findes en ny tilstand foretages denne proces for at tjekke om grafen er blevet set før. Hvis en af de her grafer i eksemplet bliver fundet senere, så foretager programmet samme proces og får samme ids og ved, at grafen er set før.

4 Projektets udførsel

4.1 Indledende resultater

Efter at have kodet algoritmen i C++ er den kommet frem til de nedenstående resultater med følgende tider (se koden til algoritmen i Annex). Disse resultater viser, at for $n = 2, 3, 4$ er der behov for at n betjente for at fange røveren.

n	Min. antal nødvendige betjente	Execution time
2	2	<1sek
3	3	<1sek
4	4	ca. 1.5sek

Tabel 1: Resultater

4.2 Videre forskning

Algoritmens beregninger viser, at der er behov for n betjente når $n = 2, 3, 4$. Det er stadig usikkert, hvor mange der skal til når $n \geq 5$. De indledende resultater tyder derfor på, at der er behov for at n betjente til at kunne fange røveren på grafen for alle n. Det er dog ikke ensbetydende med, at dette er tilfældet. Det er eksempelvis kun nødvendigt at bruge 8 betjente til at fange røveren i en $3 \times 3 \times 3$ kube, hvilket blev bevist af Berger et al. [5].

Grunden til, at det ikke var muligt at teste algoritmen når $n = 5$, er, at de ids programmet brugte til at gemme tilstande af grafen for røverens mulige positioner hurtigt blev for store når $n \geq 5$. Disse enormt store tal kan ikke umiddelbart gemmes i C++. Der findes dog måder at løse dette problem på, og det vil jeg i første omgang fokusere på. Derudover skal algoritmen også optimeres for at kunne fungere mere effektivt, bl.a. er der potentiale til at forbedre på den måde tilstande gemmes ift. rotationer og spejlinger. Når algoritmen har fået resultater for nogle større n, kan der derefter undersøges, hvorfor de resultater er som de er, på et mere teoretisk og matematisk plan.

4.3 Budget

Jeg vil undersøge, om det er muligt at låne en supercomputer med mine forskerkontakter Mikkel Abrahamsen og Søren Eilers for at undersøge større n. Når der kodes til supercomputere kræves det, at programmet er opbygget på en specifik måde og dette kan tage lidt tid at implementere. Hvis jeg får nok resultater, vil jeg skrive en artikel og sende den til en konference i algoritmik eller matematik, som f.eks. ”European Symposium on Algorithm” i München september 2019. At tage med til konferencen kan også hjælpe mig med at komme videre i projektet ved at få en generelt bedre forståelse for algoritmers opbygning og forbedring af min egen. Forskerspireprisen ville kunne finansiere dette.

Omkostning	**Pris**
Deltagergebyr (studerende 200 EUR)	1.500 kr.
Fly til München	1.500 kr.
Hotel	4.000 kr.
Mad og drikke	1.000 kr.
Ialt	**8.000 kr.**

Tabel 2: Budget

5 Afrunding

Gennemførelsen af projektet har givet mig indsigt i problemstillingen *Cops and Robbers* og en generelt bedre forståelse af grafer. Mine matematiske og datalogiske færdigheder er blevet forbedret ved at arbejde på dette projekt. En del af en problemstillingen er blevet løst, og dette åbner op for muligheden for at arbejde videre med projekt og eventuelt skrive en artikel. Processen har været en spændende introduktion til forskerverdenen og har bekræftet mig i mit valg om at fortsætte i denne retning.

Litteraturliste

[1] The complexity of searching a graph, N. Megiddo, S. L. Hakimi, M. R. Garey, D. S. Johnson, C. H. Papadimitriou, Journal of Association for Computing Machinery, Vol. 35, No. 1, January 1988, 18-44

[2] Reversibility properties of the fire-fighting problem in graphs, Rolf Klein, Elsevier B. V 2017

[3] Offline variants of the "lion and man" problem, Adrian Dumitrescu, Ichiro Suzuki, Pawel Zyliski, Theoretical Computer Science 399 2008, 220-235

[4] Vertex-to-vertex pursuit in a graph, Richard Nowakowski, Peter Winkler, North Holland 1981

[5] How many lions are needed to clear a grid?, Floarian Berger, Alexander Gilbers, Ansgar Grüne, Rolf Klein, Algorithms 2009

[6] Escaping offline seachers and isoperimetric theorems, Peter Brass, Kyue D. Kim, Hyeon-Suk Na, Chan-Su Shin, Elsevier B. V 2008

[7] Computation Geometry Colum 63, Rolf Klein, Elmar Langetepe, Association for Computer Machinery 2016

[8] Introduction to algorithms, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Third Edition, The MIT Press 2009

Annex

Se koden online her: `https://github.com/mathiaso3110/CopsAndRobbers`

Cops-and-robbers.cpp

```
#include <iostream>
#include <typeinfo>
#include <string>
#include <chrono>
#define N 3
#define amountOfCops N
#include "objects.h"
#include "move.h"
#include "id.h"
#include "previous_ids.h"
using namespace std;

// Initializing the arrays
Fire_obj fire[N][N];
Cop cops[amountOfCops];
// Cop cops[amountOfCops];
vector<Position> previous_positions;
// Initial cops is used each time find_new_position is called
    recursivly (with new initial position)
Initial_cop_pos initial_cops[amountOfCops];
int primes[N][N];
unsigned long long cop_id; unsigned long long fire_id;
bool solutionFound = false;
void find_new_position(int i);

void print(Fire_obj fire[N][N], Cop cops[amountOfCops]) {
  bool cop_pos = false;
  for (int y=N-1;y>=0;y--) {
    for (int x=0;x<N;x++) {
      for (int l=0;l<amountOfCops;l++) {
        if (cops[l].x==x&&cops[l].y==y) {
          cout << "x" << " ";
          cop_pos = true;
          break;
```

```
            }
        }
        if (!cop_pos) {cout << fire[x][y].state() << " ";}
        cop_pos=false;
    }
    cout << endl;
  }
  cout << endl;
}

void move_and_check() {
  move(fire, cops, initial_cops);
  if (!isFire(fire)) {solutionFound=true; print(fire, cops);}
  else {
    get_id(cops, fire, primes, &cop_id, &fire_id);

    if (!isSeen(previous_positions, cop_id, fire_id)) {
      find_new_position(0);
    }
  }
  revert_fire(fire);

}

void find_new_position(int i) {
  // Sets previous_cops
  if (!i) {
    copy_to(cops, initial_cops);
  }
  // NORTH
  if (!solutionFound) {
    if (cops[i].y+1<N) {
      cops[i].y += 1;
      if (i+1==amountOfCops) {
        move_and_check();
      }
      else {find_new_position(i+1);}
      cops[i].y -= 1;
    }
  }
```

```
// South
if (!solutionFound) {
   if (cops[i].y-1>=0) {
      cops[i].y -= 1;
      if (i+1==amountOfCops) {
         move_and_check();
      }
      else {find_new_position(i+1);}
      cops[i].y += 1;
   }
}
// EAST
if (!solutionFound) {
   if (cops[i].x+1<N) {
      cops[i].x += 1;
      if (i+1==amountOfCops) {
         move_and_check();
      }
      else {find_new_position(i+1);}
      cops[i].x -= 1;
   }
}
// WEST
if (!solutionFound) {
   if (cops[i].x-1>=0) {
      cops[i].x -= 1;
      if (i+1==amountOfCops) {
         move_and_check();
      }
      else {find_new_position(i+1);}
      cops[i].x += 1;
   }
}
// STAY
if (!solutionFound) {
   if (i+1==amountOfCops) {
      move_and_check();
   }
   else {find_new_position(i+1);}
}
```

```
    if (!i) {
      for (int k=0;k<amountOfCops;k++)
          {initial_cops[k].revert_coor();}
    }

    if (solutionFound && !i) {print(fire, cops);}
}

int main() {
    auto start = std::chrono::high_resolution_clock::now();
    set_cops_coordinates(cops);
    copy_to(cops, initial_cops);
    fill_initial_fire_array(fire, cops);
    fill_primes(primes);
    get_id(cops, fire, primes, &cop_id, &fire_id);
    isSeen(previous_positions, cop_id, fire_id);
    find_new_position(0);
    if (!solutionFound) {cout << "No solution found!" << endl;} else
        {cout << "Solution found!" << endl;}
    auto finish = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = finish - start;
    std::cout << "Elapsed time: " << elapsed.count() << " s\n";
    return 0;
}
```

objects.h

```
#ifndef OBJECTS_H
#define OBJECTS_H
#include <stack>
#include <vector>
using namespace std;

class Fire_obj {
    stack<bool> stack;
    bool is_new;
public:
```

```
    void set_state(bool i) {stack.push(i);}
    void revert_state() {stack.pop();}
    bool state() {return stack.top();}
    void set_is_new(bool i) {is_new=i;}
    bool get_is_new() {return is_new;}
};

class Cop {
public:
    int x, y;
};

class Initial_cop_pos {
    stack<int> x;
    stack<int> y;
public:
    void append_coor(int _x, int _y) {x.push(_x);y.push(_y);}
    int get_x() {return x.top();}
    int get_y() {return y.top();}
    void revert_coor() {x.pop(); y.pop();}
};

class Position {
public:
    int cop_id;
    vector<int> fire_ids;
    Position(int cop_id_num) : cop_id(cop_id_num) { }
    void append_fire_id(int x) {fire_ids.push_back(x);}
};
#endif
```

move.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <iostream>
#include "objects.h"
```

```
void estinguish_fire(Fire_obj fire[N][N], Cop cops[amountOfCops]) {
  // Sets the places with cops to false
  for (int i=0; i<amountOfCops; i++) {
    fire[cops[i].x][cops[i].y].revert_state();
    fire[cops[i].x][cops[i].y].set_state(false);
  }
}

void fill_initial_fire_array(Fire_obj fire[N][N], Cop
    cops[amountOfCops]) {
  // Sets first item in fire array to true
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      fire[i][j].set_state(true);
    }
  }
  estinguish_fire(fire, cops);
}

void set_cops_coordinates(Cop cops[amountOfCops]) {
  for (int i=0; i<amountOfCops; i++) {
    cops[i].x = i;
    cops[i].y = 0;
  }
}

bool illegal_move(Cop cops[amountOfCops], Initial_cop_pos
    initial_cops[amountOfCops], int i, int j, int fire_i, int
    fire_j) {
  for (int h=0;h<amountOfCops;h++) {
    if (initial_cops[h].get_x()==i
      && initial_cops[h].get_y()==j
      && cops[h].x==fire_i
      && cops[h].y==fire_j)
      {return true;}
  }
  return false;
}

void copy_to(Cop cops[amountOfCops], Initial_cop_pos
```

```
    initial_cops[amountOfCops]) {
   for (int i=0;i<amountOfCops;i++) {
      initial_cops[i].append_coor(cops[i].x, cops[i].y);
   }
}


void revert_fire(Fire_obj fire[N][N]) {
   for (int i=0;i<N;i++) {
      for (int j=0;j<N;j++) {
         fire[i][j].revert_state();
      }
   }
}

void move(Fire_obj fire[N][N], Cop cops[amountOfCops],
    Initial_cop_pos initial_cops[amountOfCops]) {

   for (int i=0;i<N;i++) {
      for (int j=0;j<N;j++) {
         if (fire[i][j].state()) {fire[i][j].set_state(true);}
         else if ( (i+1<N && fire[i+1][j].state() &&
             !fire[i+1][j].get_is_new() && !illegal_move(cops,
             initial_cops, i, j, i+1, j))
               || (i-1>=0 && fire[i-1][j].state() &&
                   !fire[i-1][j].get_is_new() && !illegal_move(cops,
                   initial_cops, i, j, i-1, j))
               || (j+1<N && fire[i][j+1].state() &&
                   !fire[i][j+1].get_is_new() && !illegal_move(cops,
                   initial_cops, i, j, i, j+1))
               || (j-1>=0 && fire[i][j-1].state() &&
                   !fire[i][j-1].get_is_new() && !illegal_move(cops,
                   initial_cops, i, j, i, j-1))) {
            fire[i][j].set_state(true);
            fire[i][j].set_is_new(true);
         } else {fire[i][j].set_state(false);}
      }
   }

   for (int i=0;i<N;i++) {for (int j=0;j<N;j++)
```

```
      {fire[i][j].set_is_new(false);}}

   estinguish_fire(fire, cops);
}
#endif
```

id.h

```
#ifndef ID
#define ID
#include <iostream>
#include <cmath>
#include "objects.h"
using namespace std;

bool isFire(Fire_obj fire[N][N]) {
   for (int i=0;i<N;i++) {
      for (int j=0;j<N;j++) {
         if (fire[i][j].state()) {return true;}
      }
   }
   return false;
}

void fill_primes(int primes[N][N]) {
   int i = 0; int j = 0; int l = 2;
   bool isPrime;
   while (i<N && j<N) {
      isPrime = true;
      for (int k=2;k<=((int) sqrt(l))+1;k++) {
         if (l%k==0 && !(l==k)) {isPrime=false; break;}
      }
      if (isPrime) {
         primes[i][j] = l;
         if (j==N-1) {i++;j=0;}
         else {j++;}
      }
      l++;
   }
```

```
}

void rotate90(int* x, int* y) {
  // Counter-clockwise
  float _x = *x; float _y = *y;

  float middle = float (N-1)/2;
  _x -= middle; _y -= middle;

  // y' = y*cos(a) + x*sin(a), x' = - y*sin(a) + x*cos(a)
  float old_x = _x;
  _x = -_y;
  _y = old_x;

  _x += middle;
  _y += middle;

  *x = _x;
  *y = _y;
}

void mirror(int* x) {
  *x = N - 1 - *x;
}

void get_id(Cop cops[amountOfCops], Fire_obj fire[N][N], int
    primes[N][N], unsigned long long* cop_id, unsigned long long*
    fire_id) {
  // COPS
  unsigned long long id_num = 1;
  unsigned long long temp_id_num = 1;
  int rotations = 0;
  bool mirrorFire = false;

  // Initial position of the cops
  for (int i=0;i<amountOfCops;i++) {id_num *=
     primes[cops[i].x][cops[i].y];}

  // Rotations
  for (int i=0, j=0;j<(amountOfCops)*3;j++) {
```

```
    i = j % (amountOfCops);
    rotate90(&cops[i].x, &cops[i].y);
    temp_id_num *= primes[cops[i].x][cops[i].y];
    // If there are no more cops
    if (i == amountOfCops-1) {
      if (temp_id_num < id_num) {id_num =
          temp_id_num;rotations=ceil(j/((int)
          amountOfCops))+1;mirrorFire=false;}
      temp_id_num = 1;
    }
}

// Turn it back to initial position
for (int i=0;i<amountOfCops;i++) {
    rotate90(&cops[i].x, &cops[i].y);
}

// Mirroring
for (int i=0;i<amountOfCops;i++) {
    mirror(&cops[i].x);
    temp_id_num *= primes[cops[i].x][cops[i].y];
}
temp_id_num = 1;

// Rotations (after mirroring)
for (int i=0, j=0;j<(amountOfCops)*3;j++) {
    i = j % (amountOfCops);
    rotate90(&cops[i].x, &cops[i].y);
    temp_id_num *= primes[cops[i].x][cops[i].y];
    // If there are no more cops, check if the prime is less than
        the current and reset temp id num
    if (i == amountOfCops-1) {
      if (temp_id_num < id_num) {id_num =
          temp_id_num;rotations=ceil(j/((int)
          amountOfCops))+1;mirrorFire=true;}
      temp_id_num = 1;
    }
}

// Turn it back to initial position
```

```
    for (int i=0;i<amountOfCops;i++) {
      rotate90(&cops[i].x, &cops[i].y);
    }
    for (int i=0;i<amountOfCops;i++) {mirror(&cops[i].x);}
    *cop_id = id_num;

    // FIRE
    id_num = 1;
    for (int i=0;i<N;i++) {
      for (int j=0;j<N;j++) {
        // Rotate i and j
        int _i = i; int _j = j;
        if (mirrorFire) {mirror(&_i);}
        for (int l=0;l<rotations;l++) {rotate90(&_i, &_j);}
        if (fire[i][j].state()) {
          id_num *= primes[(int) _i][(int)_j];
        }
      }
    }
    *fire_id = id_num;
}
#endif
```

previous_ids.h

```
#ifndef PRE_ID
#define PRE_ID
#include <iostream>
#include "objects.h"
using namespace std;

bool isSeen(vector<Position>& previous_positions, int cop_id, int
     fire_id) {
    for (auto position = previous_positions.begin(); position !=
        previous_positions.end(); position++) {
      if (position->cop_id == cop_id) {
        for (auto pre_fire_id = position->fire_ids.begin();
             pre_fire_id != position->fire_ids.end(); pre_fire_id++)
             {
```

```
            if (*pre_fire_id == fire_id) {
               return true;
            }
         }
         position->append_fire_id(fire_id);
         return false;
      }
   }
   Position a(cop_id);
   a.append_fire_id(fire_id);
   previous_positions.push_back(a);
   return false;
}
#endif
```