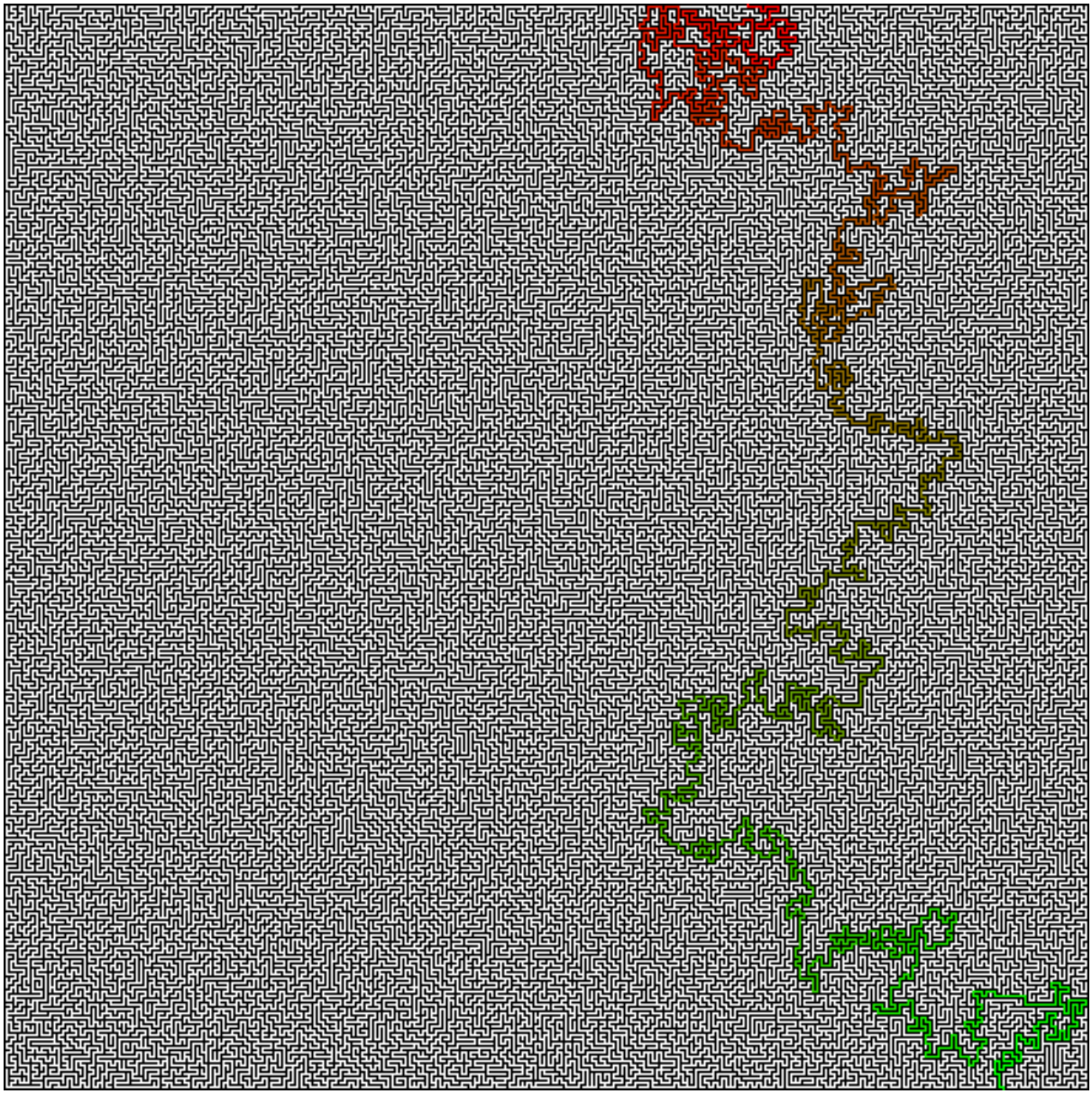
Pathfinding algoritmer — Eksamensprojekt i programmering B 2020

Jens Tinggaard (3.E)
Odense Tekniske Gymnasium

1. maj 2020



Indholdsfortegnelse

1 Indledning

1.1 Målsætning

Fra projektbeskrivelsen:
Jeg vil gerne lave et program, der er i stand til at illustrere en pathfinding algoritme. Jeg vil gerne vise fordelen ved en pathfinding algoritme, dette kunne f.eks. gøres ved at sammenligne med en bruteforce algoritme. Umiddelbart vil det tage udgangspunkt i labyrinter, hvor jeg vil vise hvordan de forskellige algoritmer virker.

1.2 Krav til programmet

- Implementering af en pathfinding algoritme
- Kunne løse en labyrint
- Sammenligning med andre metoder til at løse en labyrint
- Visuel repræsentation af løsningen

1.3 Udviddelser

- Visualisering af hastighed vs. størrelse af labyrint for forskellige metoder
- Implementering af flere algoritmer (nogle af dem fra ovenstående liste)

2 Om pathfinding algoritmer

En pathfinding algoritme er en algoritme, som bruges til at finde vej over en graf. Der findes mange forskellige algoritmer, som alle har fordele og ulemper. Et par af de mest kendte er *Dijkstra*, *depthfirst*, *breadthfirst* og *A**. Alle disse algoritmer har til fælles, at de beskriver en fremgangsmåde, til at finde den korteste vej mellem to noder på en graf.

Pathfinding er brugt til alt muligt i dag, et klassisk eksempel er navigationstjenester som Google Maps. Når brugeren har indtastet en startposition og et mål, er det nu computerens opgave at finde den korteste vej derhen. Hvis man skal fra København til Rom er der måske en idé i, at optimere algoritmen, så den ikke starter med at kigge over St. Petersborg i Rusland. Alle disse overvejelser er vigtige at gøre sig, når man skal implementere en pathfinding algoritme, da forskellige algoritmer er stærke til hver deres ting.

2.1 Grafer, noder og kanter

Når man snakker om pathfinding algoritmer, vil termerne *graf* og *node* fremkomme. Begge disse typer er abstrakte og dækker over et større område inden for datalogien. En *graf* er en struktur, som indeholder et endeligt antal hjørner, også kaldet *noder*. Alle disse noder har ofte nogle bestemte attributter eller egenskaber, afhængig af hvilken type graf, der er tale om. Derudover er disse noder forbundet gennem hvad der kaldes *kanter* en kant er i bund og grund en forbindelse mellem to noder, man sætter ofte en vægt på kanterne (medmindre alle kanter er vægtet ligeligt). Denne vægt bruges også i beskæfigelsen med grafer — den er meget essentiel i forbindelse med pathfinding algoritmer.[8]

Når man har med pathfinding algoritmer at gøre, vil alle noder have følgende attributter: **Naboer** alle noder er klar over hvilke noder de ligger op ad. **Via** alle noder er klar over hvilken node der forbinder dem til startnoden. **Pris** alle noder er klar over hvor langt de har til startnoden – målt i samlet vægt af kanter op til denne – inden denne pris er bestemt vil den være ∞. **Afstand til mål** når man bruger *A**, vil alle noder også være klar over deres afstand til målnoden, denne afstand er målt direkte, såkaldt fugleflugt, hvorimod at *pris* er målt i den samlede pris fra de foregående noder + vægten af kanten fra den foregående node (*via*) til den nuværende.

2.2 Prioritetskø

Algoritmerne *Dijkstra* og *A** gør begge brug af en prioritetskø, kaldet en *priority queue* på engelsk. En prioritetskø skal ses som en liste, som er dynamisk sorteret efter et parameter. I dette tilfælde er de sorteret efter en afstand, præcis hvilken afhænger af algoritmen — mere herom senere.

2.3 Algoritmer

Det vises kort, hvordan nogle af de mest kendte algoritmer virker.

2.3.1 Depthfirst

Depthfirst er en algoritme, som – som navnet antager – scanner dybden først. Betragt Node A, som har forbindelse til Node B og C, algoritmen tager fat i den første af disse Noder; B, som i dette tilfælde har forbindelse til D og E, igen tager algoritmen den første af disse Noder og kigger videre ud i *dybden*, til slutnoden er fundet.

2.3.2 Breadthfirst

Breadthfirst fungerer lige modsat af depthfirst, breadthfirst kigger nemlig i bredden først. Det vil altså sige at efter Noden B er undersøgt, går algoritmen videre til C i stedet for D som depthfirst. Bagefter kigger algoritmen på D og så E og derefter de Noder som er forbundet til C.

2.3.3 Dijkstra

Dijkstra er en algoritme opkaldt efter dens skaber; *Edsger W. Dijkstra.*[7] Dijkstra tager fat i problemstillingen med vægting af sider på en graf, da disse spiller en rolle i udfaldet af algoritmen. Algoritmen er garenteret, til altid at finde den korteste vej fra start til slut. Algoritmen gør brug af en Prioritetskø, som er sorteret efter afstand på alle de foregående kanter. Man tager altid fat i den lavest rangede Node i køen, hvilket vil sige at man får Noden, som er nået kortest uanset antal af foregående Noder. Man kan sige at Dijkstra er breadthfirst vægtet efter afstand i stedet for antal noder. Det betyder altså at når man har fundet en løsning, behøver man blot at markere den aktuelle kantlængde og tømme sin prioritetskø op til denne størrelse. Finder man en kortere vej, gemmes denne på samme vis. Man er på denne måde garanteret, at finde den korteste vej.

2.3.4 A*

A* er Dijkstra med en opgradering. Afstanden til målet regnes nemlig med, målt i fugleflugt. For hver Node gemmes den foregående Node, samt den samlede afstand stadig. Derudover gemmes afstanden til målet også (er alle Noder defineret i koordinater, kan Pythagoras bruges). Vægtningen af prioritetskøen foregår nu efter den samlede værdi af den tilbagelagte afstand, samt afstanden til målet. A* kræver altså lidt mere regnerkaft, men forbedrer samtidig målrettetheden af algoritmen, som forklaret i afsnittet "Om pathfinding algoritmer".

3 Installation og brug

Den letteste måde at bruge programmet på er:

```
$ git clone https://github.com/Tinggaard/pathfinding
$ cd pathfinding && virtualenv venv && . venv/bin/activate
$ pip install -e .
```

Herefter vil det være muligt at køre hovedprogrammet (forudsat at brugeren stadig er i virtualenvironment), ved blot at kalde:

```
$ pathfinding [-h] (-i INPUT | -g width height) [-v] [-s] [-f] [-o OUTPUT]
              [-a {astar,dijkstra,breadthfirst,depthfirst,rightturn}]
```

For at få en detaljeret beskrivelse af alle flagene, sættes flaget `-h` blot.

Alternativt kan programmet installeres ved

```
$ pip install -r requirements.txt
```

Herefter kaldes programmet som følger:

```
$ # cd pathfinding
$ ./main.py [-h] (-i INPUT | -g width height) [-v] [-s] [-f] [-o OUTPUT]
            [-a {astar,dijkstra,breadthfirst,depthfirst,rightturn}]
```

3.1 Eksempel

Ønsker man at finde vej gennem `mazes/perfect/1001.png`, ved hjælp af breadthfirst algoritmen, samt at gemme outputtet igen som f.eks. `out/eksempel.png`, køres programmet som følger:

```
$ pathfinding -i mazes/perfect/1001.png -a breadthfirst -o out/eksempel.png
```

4 Udarbejdning af projektet

I forbindelse med projektet har jeg lavet et repository på GitHub (`https://git.io/JfINU`). Det medfølger naturligvis en README-fil, som også er vedlagt i Appendiks. Jeg vil kort gennemgå krav samt brug af programmet her, en uddybning kan findes i README-filen.

4.1 Baggrund for projektet

Projektet startede ud med instpiration fra denne video [6], hvor Dr. Mike Pound, viser hvordan han har lavet et Python program, som kan finde vej gennem labyrinter. Faktisk har han også offentliggjort koden på GitHub [3], hvilken jeg også har ladet mig inspirere fra.

Selve pathfinding-algoritme delen af projektet er baseret på filen `pathfinding/scheme.py`, som indeholder klasserne, mens algoritmerne ligger under `pathfinding/algs/*`. Filen `pathfinding/main.py` er blot en wrapper til at loade filer ind i klasserne og eksekvere den gevne algoritme.

4.2 Første udlægning

Udover at have lavet nogle funktioner til at indlæse en labyrint enten på tekst- eller billedform, var noget af det første jeg gjorde, at lave klasser til at holde på min labyrint. Mit første rigtige udlæg til dette (a36e624), bestod af to klasser: `Maze` og `Node`. Constructoren for `Node` klassen, så sådan ud:

```
1 class Node:
2     def __init__(self, location: tuple):
3         self.location = location # (y, x)
4         self.n = None # [node_index, dist]
5         self.s = None # [node_index, dist]
6         self.e = None # [node_index, dist]
7         self.w = None # [node_index, dist]
8         self.dist_goal = np.inf
```

Klassen har altså seks attributes, `location`, et tuple indeholdende koordinaterne til objektet og `dist_goal`, som er afstanden til målet — brugt i A*, samt fire naboer; `n`, `s`, `e` og `w`, som ikke har nogen værdi ind til videre, dog siger kommentaren ud for hvordan de ser ud efter at være sat — En liste med et index på pågældende `Node`, samt afstanden til denne. Som det fremgår af linje 2, er det kun `location`, der kræves, for at lave et `Node` objekt.
Constructoren for `Maze`, så sådan ud:

```
1 class Maze:
2     def __init__(self, maze: np.ndarray):
3         self.maze = maze
4         self.x = maze.shape[1]
5         self.y = maze.shape[0]
6         self.start = (0, np.argmax(maze[0]))
7         self.end = (self.y-1, np.argmax(maze[-1]))
8         self.nodes = self.get_nodes()
```

```
9           self.node_count = len(self.nodes) # + start and end
10          self.gen_graph()
```

Klassen tager altså ét argument, til sin constructor, ud fra hvilken hele klassen dannes. Dette argument er et binært 2D numpy array[5], som udelukkende består af 1- og 0-taller, hvor 1 fortolkes som sti, mens 0 er væg. Startnodens placering SKAL være i toppen af labyrinten, mens slutnoden på lige vis SKAL være i bunden af labyrinten, ellers fungerer programmet ikke! Linje 6 og 7 finder koordinaterne for hhv. start og slutnoden, ved at kalde funktionen `np.argmax(maze[i])`, som finder indexet, på den største værdi af input arrayet.[4] På linje 8 kaldes `self.get_nodes()` metoden, som finder alle noder i `self.maze`, ved simpelthen blot at se om den pågældende pixel, for det første er hvid (1), samt den ligger i et kryds mellem andre hvide pixels. Se den fulde funktion her.

4.3 Implementering af algoritmer

Den første algoritme jeg implementerede, er faktisk ikke en af de ovennævnte, men i stedet en "højresvingsmetode", som blot drejer til højre ved hvert kryds og på den måde altid følger den sammve væg. Det er en klassisk taktik, hvis man ikke kan se hele labyrinten, f.eks. som menneske inde i en. Metoden blev implementeret under commit `ea5096e`. Det er dog ikke den algoritme jeg vil gennemgå her, men i stedet Dijkstra, som blev implementeret ved commit `cd33a78`.

Som det kort blev gennemgået i afsnittet "Dijkstra", virker Dijkstra ved brug af en prioritetskø, som holder styr på den tilbagelagte længde, for hver Node og at algoritmen altid arbejder med den Node, som har tilbagelagt kortest afstand.
Nedenstående kode, viser et kort udtræk (linje 27-58) af filen `pathfinding/algs/dijkstra.py`, som den så ud ved pågældende commit (tomme linjer og kommentarer fjernet).

```
1   while pq:
2       current = hq.heappop(pq)
3       if current.dist > goal:
4           break
5       if current == end:
6           goal = current.dist
7       for near in current.nearby:
8           if near is not None:
9               node = self.get_node(near)
10              if not visited[node.location]:
11                  visited[node.location] = True
12                  cy, cx = current.location
13                  ny, nx = node.location
14                  distance = abs(cy-ny) if cy-ny != 0 else abs(cx-ny)
15                  node.dist = current.dist + distance
16                  node.via = self.get_node_index(cy, cx)
17                  hq.heappush(pq, node)
```

Forud for denne kode, bliver der erklæret nogle variabler, som bruges i løbet af loopet. I linje 1, sættes et loop til at køre, så længe der stadig er indhold i listen `pq`, da en liste evalueres til at være *sand*, hvis den indeholder noget, ellers er den *falsk* i hvilket tilfælde loopet stopper. Variablen `current` bliver sat til det første index af prioritetskøen (modulet `heapq`[2] er importeret som `hq`), som er en instance af `Node` klassen, de bliver sorteret, da `__lt__()` metoden er erklæret for klassen, som kan sammenligne `self.dist` mellem to

`Node` objekter, tak til [1]. Line 3-4 fortæller loopet at stopper, hvis `current`s distance er lig med `goal` (initialiseret til ∞). Linje 5-6 sætter `goal` til at være lig `current`s distance, hvis den aktuelle node er identisk med slutnoden. På linje 7 initialiseres et for-loop, som itererer over alle nabonoderne for den aktuelle node. Linje 8 tjekker om naboen er en node (ikke blot en mur) og sætter så `node` til at være lig nabonoden vha. metoden `get_node()`, som returnerer en `Node`, baseret på et index i `self.nodes`. Inden while-loopet har jeg også lavet et array af samme størrelse som `self.maze`, bestående af boolske udtryk, på om det enkelte felt har været besøgt før (alle felter initialiseret til `False`). På linje 10 tjekker jeg så om det aktuelle nabofelt har været besøgt førhen, ellers sættes feltet til at være besøgt på linje 11. Afstanden mellem den aktuelle node og nabonoden findes på linje 12-14 og nabonodens afstand sættes til at være denne + den tidligere afstand til `current` på linje 15. Linje 16 sætter nabonodens `via` til at være koordinaterne for `current` og slutteligt lægges nabonoden ind i prioritetskøen på linje 17.

Dette er blot en kort gennemgang af hvordan jeg har implementeret Dijkstra, den endelige version af koden har lidt småændringer hist og her, men konceptet er det samme. Det med at have et boolsk array er dog lidt specielt og kan kun tillades, da kanternes vægt er lig med den faktiske afstand. Ellers kunne der være en smutvej mellem to noder, som blot blev sprunget over, hvorved den korteste afstand faktisk ikke blev fundet.

5 Konklusion

Litteratur

[1] Bao, Fanchen. *python - heapq with custom compare predicate - Stack Overflow*. Stack Overflow. URL: https://stackoverflow.com/a/59956131.

[2] Docs, Python. *heapq — Heapq queue algorithm — Python 3.8.2 documentation*. Python. URL: https://docs.python.org/3/library/heapq.html.

[3] mikepound. *mikepound/mazesolving: A variety of algorithms to solve mazes from an input image*. GitHub. URL: https://github.com/mikepound/mazesolving/.

[4] Numpy. *numpy.argmax — NumPy v1.18 Manual*. NumPy. URL: https://numpy.org/doc/stable/reference/generated/numpy.argmax.html.

[5] Numpy. *numpy.ndarray — NumPy v1.18 Manual*. NumPy. URL: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html.

[6] Pound, Mike. *Maze Solving - Computerphile*. YouTube. Feb. 2017. URL: https://www.youtube.com/watch?v=rop0W4QDOUI.

[7] Wikipedia. *Dijkstra's algorithm - Wikipedia*. Wikipedia. URL: https://en.wikipedia.org/wiki/Dijkstra's_algorithm.

[8] Wikipedia. *Graph (abstract data type) - Wikipedia*. Wikipedia. URL: https://en.wikipedia.org/wiki/Graph_(abstract_data_type).

Appendiks