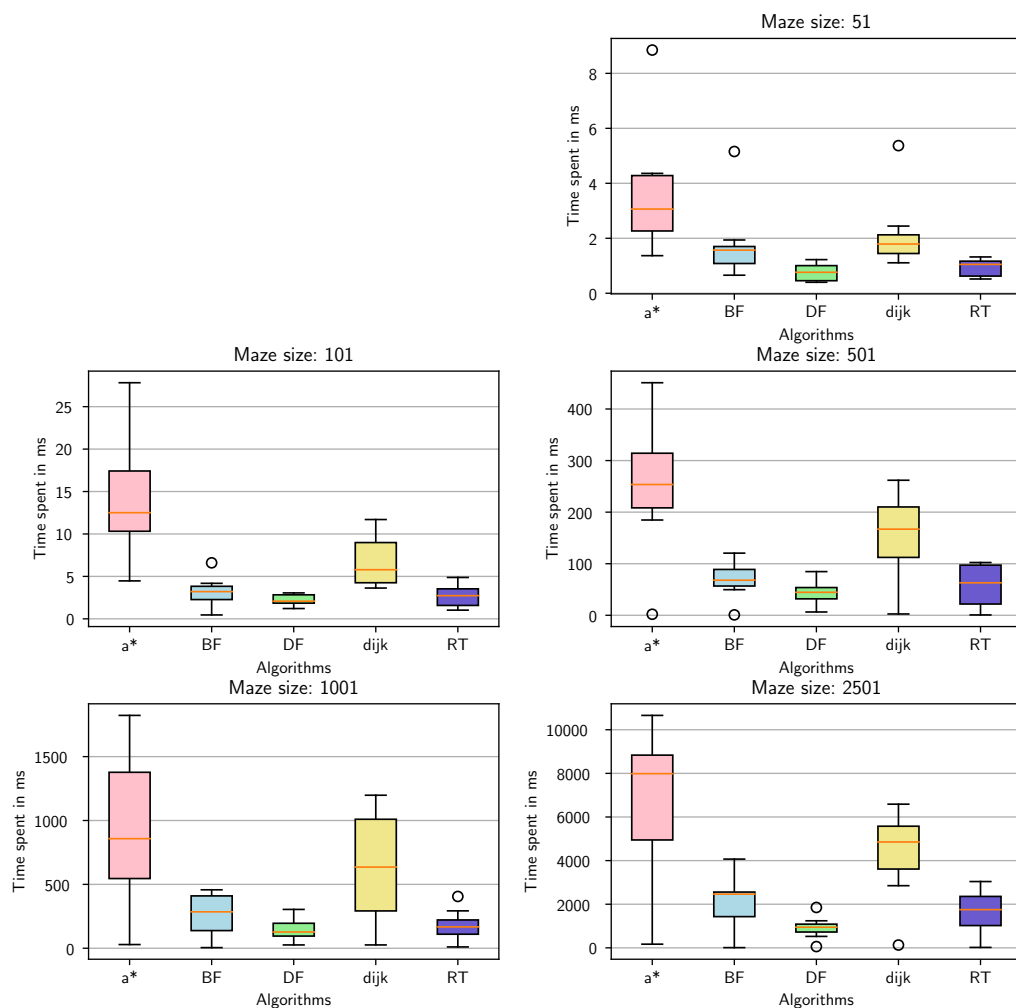


Pathfinding algoritmer — Eksamensprojekt i programmering B 2020

Jens Tinggaard (3.E)
Odense Tekniske Gymnasium

1. maj 2020



Indholdsfortegnelse

1	Indledning	2
1.1	Målsætning	2
1.2	Krav til programmet	2
1.3	Udvidelser	2
2	Om pathfinding algoritmer	3
2.1	Grafer, noder og kanter	3
2.2	Prioritetskø	4
2.3	Algoritmer	4
2.3.1	Depthfirst	4
2.3.2	Breadthfirst	4
2.3.3	Dijkstra	4
2.3.4	A*	4
3	Installation og brug	5
3.1	Eksempel	5
4	Udarbejdning af projektet	6
4.1	Baggrund for projektet	6
4.2	Første udlægning	6
4.3	Implementering af algoritmer	7
4.3.1	Optimering	8
4.4	Sammenligning	8
4.4.1	A* er langsom	8
5	Konklusion	9
	Litteratur	10
	Appendiks A Kildekode	11
A.1	pathfinding/main.py	11
A.2	pathfinding/scheme.py	16
A.3	pathfinding/algs/__init__.py	23
A.4	pathfinding/algs/astar.py	23
A.5	pathfinding/algs/breadthfirst.py	25
A.6	pathfinding/algs/depthfirst.py	26
A.7	pathfinding/algs/dijkstra.py	27
A.8	pathfinding/algs/rightturn.py	29
A.9	pathfinding/comparer.py	31

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 fordelene ved en pathfinding algoritme, dette kunne f.eks. gøres ved at sammenligne med en brute force 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 Udvidelser

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

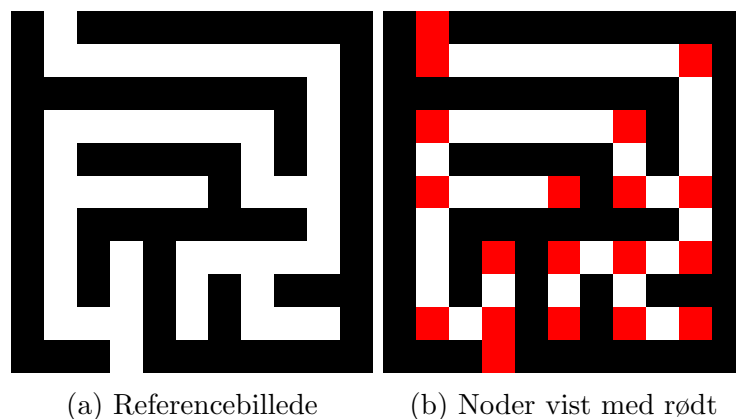
2 Om pathfinding algoritmer

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. Petersburg 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 termene *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 beskæftigelsen med grafer — den er meget essentiel i forbindelse med pathfinding algoritmer.[9]



Figur 1: 19 noder markeret med rødt, alt hvidt er kant

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 startnoder. **Pris** alle noder er klar over hvor langt de har til startnoder – 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 målnoder, 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. **ID** slutteligt er det naturligvis vigtigt at kunne identificere sine noder, jeg har markeret mine ved deres lokation ($y; x$) koordinat.

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*. [8] Dijkstra tager fat i problemstillingen med vægting af sider på en graf, da disse spiller en rolle i udfaldet af algoritmen. Algoritmen er garanteret, 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]

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 regnekraft, men forbedrer samtidig målrettetheden af algoritmen, som forklaret i afsnittet ”Om pathfinding algoritmer”.

3 Installation og brug

Forud for installation, skal der være installeret en C++ 11 compiler på systemet (som f.eks. gcc (v.4.7+)). Installationen vil fejle, hvis dette ikke er opfyldt, da pydaedalus biblioteket er en wrapper for daedalus, som er skrevet i C++.

Den letteste måde at bruge programmet på er:

```
$ git clone https://github.com/Tinggaard/pathfinding
$ git checkout assignment # repo fremstår som ved aflevering
$ 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). Strukturen for at kalde det er som følger:

```
$ pathfinding [OPTIONS] COMMAND [ARGS]...
```

Options:

--help Show this message and exit.

Commands:

generate	Generate maze from scratch Takes outputfile and maze size as...
solve	Solve maze from given inputfile, using options listed below
visualize	Visualize pathfinding algorithm as videofile on given filename

For at få en detaljeret beskrivelse af alle flagene, sættes flaget `-h` blot — dette kan sættes ved alle kommandoerne, for at få mere information om hvordan de hver især virker.

Alternativt kan programmet installeres ved

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


Herefter køres ovenstående program, ved at eksekvere `./pathfinding/main.py` filen.

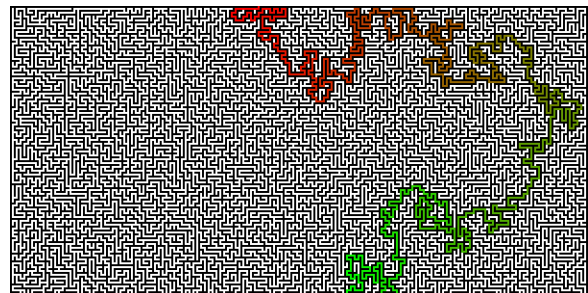
3.1 Eksempel

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 solve mazes/perfect/1001.png -a breadthfirst -o out/eksempel.png
```

Kommandoen `visualize` kan generere videofiler, som viser hvordan algoritmen har fundet frem til den givne løsning. Det fungerer helt klart bedst på billeder af størrelsen 50x50, evt. 100x100, da alternativet er at computeren kører i en evighed, pga mangel på RAM. Se et eksempel på output, i filen `out/solution.mp4`, el-

ler hent den her (dobbelklik klipsen):  Billedoutputtet kan se ud som på figur 2.



Figur 2: Eksempel output, fra programmet

4 Udarbejdning af projektet

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

4.1 Baggrund for projektet

Projektet startede ud med inspiration fra denne video [7], 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 [4], 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[6], 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.[5] 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 samme 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`[3] 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.

4.3.1 Optimering

Efter noget tid, indså jeg at mit program kørte meget langsomt og fandt hurtigt ud af at det var måden jeg havde valgt at opbygge min nodestruktur på i `Graph` klassen. Det var nemlig opdelt i en liste og alle nodes naboer var opgivet efter et index i denne liste. Koden brute-forced sig altså vej gennem alle disse noder, hver gang den skulle finde en nabo. Jeg fandt derfor på at bruge et dictionary, hvilket er en python wrapper for et hashmap, hvor opslagstiden er 1. Alt dette blev implementeret under commit 2028cb7.

4.4 Sammenligning

Jeg har skrevet et lille shell script, til at kalde alle algoritmer, på alle billeder i `mazes/`, outputtet, redirectes over i tekstfilen `timings.txt`, som filen `comparer.py` kan tage sig af. Filen skal kaldes fra det directory, hvori den ligger, da den er hardcoded til at finde en fil, en mappe oppe, ved navn `timings.txt`. Ud fra denne, bruges der regular expressions, til at finde informationer om hvert kald af programmet og ud fra dette plotte boksplot, til sammenligning af algoritmerne.

4.4.1 A* er langsom

Som man tydeligt kan se på plottet, er A* i gennemsnit den langsomste algoritme, ved alle labyrintstørrelser, derefter følger Dijkstra. Årsager til dette, formoder jeg er at er labyrinternes opbygning. De starter alle i toppen og skal alle til bunden, dog er det ikke den nogen lige vej de skal igennem. Derfor er det hurtigere at brute-force sig igennem løsningen, frem for at forsøge beregne sig frem til løsningen. Alt dette resulterer i at Dijkstra og A* kører langsommere, da de har flere beregninger med, i form af en prioritetskø, samt en afstandsberegning ved brug af A*.

Hvis man ville ændre dette udfald, kunne man omskrive programmet, så det kan starte labyrinten på et givent punkt (midt i et sted) og derfra finde sig vej ud. Men selv der, er det ikke givet at A* ville være hurtigere, grundet de mange tilfældige sving, der tages på vej ud af labyrinten.

5 Konklusion

Alt i alt, har jeg fået opfyldt de mål, jeg satte mig for i min målsætning. Dog har det vist sig at være brute-force algoritmerne, som måtte tage sejren over A^* og Dijkstra. Jeg har også formået at visualisere min løsning, både den færdige, men også en ”vejledning”, til hvordan løsningen blev fundet — videofil outputtet.

Jeg har først forklaret hvad pathfinding er og lidt om hvor det anvendes, i afsnittet ”Om pathfinding algoritmer”, udover naturligvis at have forklaret hvordan man bruger programmet, har jeg også forklaret lidt om udviklingen af programmet, i form af hvordan jeg har valgt at opbygge Dijkstra algoritmen, samt en optimering heraf. Slutteligt har jeg vist hvordan min sammenligning af algoritmerne virker, samt diskuteret hvorfor udfaldet var sådan.

Udvidelser til programmet, kunne inkludere at fjerne noder, som blot er forbundet til to eller færre andre noder og derved optimere hastigheden af samtlige algoritmer. Derudover kunne det vært sjovt at implementere en metode, til at lade startpunktet ligge et sted inde midt i labyrinten, da man måske ville se A^* og Dijkstra, være hurtigere.

Litteratur

- [1] Bao, Fanchen. *python - heapq with custom compare predicate - Stack Overflow*. Stack Overflow. URL: <https://stackoverflow.com/a/59956131>.
- [2] Cormen, Thomas H. m.fl. *Introduction to Algorithms*. Engelsk. Third edition. The MIT Press, 2009, s. 658–659. ISBN: 978-0-262-53305-8. URL: [http://ressources.unisciel.fr/algoprogram/s00aaroot/aa00module1/res/\[Cormen-AL2011\]Introduction_To_Algorithms-A3.pdf](http://ressources.unisciel.fr/algoprogram/s00aaroot/aa00module1/res/[Cormen-AL2011]Introduction_To_Algorithms-A3.pdf).
- [3] Docs, Python. *heapq — Heapq queue algorithm — Python 3.8.2 documentation*. Python. URL: <https://docs.python.org/3/library/heapq.html>.
- [4] mikepound. *mikepound/mazesolving: A variety of algorithms to solve mazes from an input image*. GitHub. URL: <https://github.com/mikepound/mazesolving/>.
- [5] Numpy. *numpy.argmax — NumPy v1.18 Manual*. NumPy. URL: <https://numpy.org/doc/stable/reference/generated/numpy.argmax.html>.
- [6] Numpy. *numpy.ndarray — NumPy v1.18 Manual*. NumPy. URL: <https://numpy.org/doc/stable/reference/generated/numpy.ndarray.html>.
- [7] Pound, Mike. *Maze Solving - Computerphile*. YouTube. Feb. 2017. URL: <https://www.youtube.com/watch?v=rop0W4QDOUI>.
- [8] Wikipedia. *Dijkstra's algorithm - Wikipedia*. Wikipedia. URL: https://en.wikipedia.org/wiki/Dijkstra's_algorithm.
- [9] Wikipedia. *Graph (abstract data type) - Wikipedia*. Wikipedia. URL: [https://en.wikipedia.org/wiki/Graph_\(abstract_data_type\)](https://en.wikipedia.org/wiki/Graph_(abstract_data_type)).

Appendiks

A Kildekode

A.1 pathfinding/main.py

```
1  #!/usr/bin/env python3
2  import sys
3  import re
4  import os.path
5  from time import time
6  # import logging
7
8  import click
9  import numpy as np
10 from PIL import Image
11 from daedalus import Maze
12
13 from .scheme import Graph
14
15
16 # vars for determining filetype
17 IMAGE = 1
18 TEXT = 2
19 VIDEO = 3
20
21 METHODS = {
22     'astar': Graph.astar,
23     'dijkstra': Graph.dijkstra,
24     'breadthfirst': Graph.breadthfirst,
25     'depthfirst': Graph.depthfirst,
26     'rightturn': Graph.rightturn
27 }
28
29
30 @click.group()
31 def cli():
32     pass
33
34
35 @cli.command()
36 @click.argument('filename', type=click.Path(exists=True))
37 @click.option('-v', '--verbose', is_flag=True, help='Ramp up verbosity level')
38 @click.option('-f', '--force', is_flag=True, help='Do not ask before overwriting
39 ↪ files')
40 @click.option('-o', '--output', type=click.STRING, help='Path to save maze to')
41 @click.option('-a', '--algorithm', type=click.Choice(['astar', 'dijkstra',
42 ↪ 'breadthfirst', 'depthfirst', 'rightturn']), default='dijkstra',
43 ↪ show_default=True, help='Pathfinding algorithm to use')
44 def solve(filename, verbose, force, output, algorithm):
45     """Solve maze from given inputfile, using options listed below"""
46     maze = load(filename) #loading
47
48     struct = Graph(maze) #generate struct
49     alg = METHODS[algorithm] # alg to use
```

```

47     start = time()
48     explored, path, nodes, length = alg(struct) #solve!
49     end = time()
50     elapsed = round((end-start)*1000, 5)
51
52     if not struct.solved:
53         click.secho('ERROR: The algorithm could not solve the maze', fg='red',
54                     ↪ err=True)
55
56     click.secho(f'SUCCESS: Solved {filename} in {elapsed} ms using {algorithm}',
57                 ↪ fg='green')
58
59     if verbose:
60         click.echo(f'Nodes explored: {explored}')
61         click.echo(f'Nodes in path: {nodes}')
62         click.echo(f'Length of path: {length}')
63
64     if output:
65         file_exists(output, force)
66         struct.save_solution(output, filetype(output))
67
68 @cli.command()
69 @click.argument('filename', type=click.Path(exists=True))
70 @click.option('-v', '--verbose', is_flag=True, help='Ramp up verbosity level')
71 @click.option('-f', '--force', is_flag=True, help='Do not ask before overwriting
72 ↪ files')
73 @click.option('-o', '--output', default='solution.mp4', type=click.STRING,
74 ↪ show_default=True, help='Path to save maze to')
75 @click.option('-a', '--algorithm', type=click.Choice(['astar', 'dijkstra',
76 ↪ 'breadthfirst', 'depthfirst', 'rightturn']), default='dijkstra',
77 ↪ show_default=True, help='Pathfinding algorithm to use')
78 def visualize(filename, verbose, force, output, algorithm):
79     """Visualize pathfinding algorithm as videofile on given filename"""
80     maze = load(filename) #loading
81
82     if not filetype(output) == VIDEO:
83         click.secho('ERROR: Outputfile not a vide extension, valid extensions are:
84 ↪ .flv, .mp4, .avi', fg='red', err=True)
85         sys.exit(1)
86
87     struct = Graph(maze) #generate struct
88     struct.visualize()
89     alg = METHODS[algorithm] # alg to use
90     start = time()
91     explored, path, nodes, length = alg(struct) #solve!
92     end = time()
93     elapsed = round((end-start)*1000, 5)
94
95     if not struct.solved:
96         click.secho('ERROR: The algorithm could not solve the maze', fg='red',
97                     ↪ err=True)
98
99     click.secho(f'SUCCESS: Solved {filename} in {elapsed} ms using {algorithm}',
100                ↪ fg='green')

```

```

96     if verbose:
97         click.echo(f'Nodes explored: {explored}')
98         click.echo(f'Nodes in path: {nodes}')
99         click.echo(f'Length of path: {length}')
100
101     if output == 'solution.mp4':
102         click.secho('INFO: No outputfile given, saving as default', fg='yellow')
103
104     file_exists(output, force)
105     struct.save_solution(output, filetype(output))
106
107
108 @cli.command()
109 @click.argument('filename', type=click.Path(allow_dash=True))
110 @click.argument('size', nargs=2, type=click.INT)
111 @click.option('-m', '--method', type=click.Choice(['braid', 'braid_tilt', 'diagonal',
112 ↪ 'perfect', 'prim', 'recursive', 'sidewinder', 'spiral']), default='perfect',
113 ↪ show_default=True, help='Generation method to use')
114 @click.option('-f', '--force', is_flag=True, help='Do not ask before overwriting
115 ↪ files')
116 def generate(filename, size, method, force):
117     """
118     Generate maze from scratch
119
120     Takes outputfile and maze size as arguments
121     """
122     file_exists(filename, force)
123     write(filename, gen_maze(size, method))
124
125
126 # Prompt user if file already exists and force is unset
127 def file_exists(filename: str, force: bool) -> None:
128     if os.path.isfile(filename) and not force:
129         click.confirm(f'The file "{filename}" already exists, overwrite?', abort=True)
130
131
132 def gen_maze(size: tuple, method: str = 'perfect') -> np.ndarray:
133     width, height = size
134
135     methods = {
136         'braid': Maze.create_braid,
137         'braid_tilt': Maze.create_braid_tilt,
138         'diagonal': Maze.create_diagonal,
139         'perfect': Maze.create_perfect,
140         'prim': Maze.create_prim,
141         'recursive': Maze.create_recursive,
142         'sidewinder': Maze.create_sidewinder,
143         'spiral': Maze.create_spiral,
144         # 'unicursal': Maze.create_unicursal,
145     }
146
147     method = methods[method]
148
149     # basic chekcs
150     # maybe use logging instead...?
151     if not height % 2:
152         click.secho('NOTE: height must be odd, automatically incremented by 1',
153 ↪ fg='yellow')

```

```

150         height += 1
151
152     if not width % 2:
153         click.secho('NOTE: width must be odd, automatically incremented by 1',
154                     ↪ fg='yellow')
155         width += 1
156
157     # genereate maze accordingly
158     maze = Maze(width, height)
159
160     # the actual generation
161     method(maze)
162
163     # invert array, as library treats 0 as path and 1 as wall
164     inv = np.array(maze, dtype=np.bool)
165     return np.logical_not(inv).astype(int)
166
167 def filetype(filename: str) -> int:
168     file, ext = os.path.splitext(filename)
169     ext = ext.lower()
170     # some image files not supported
171     if ext in ['.png', '.bmp']:
172         return IMAGE
173     elif ext in ['.txt', '.text']:
174         return TEXT
175     elif ext in ['.mp4', '.flv', '.avi']:
176         return VIDEO
177
178     elif ext in ['.jpg', '.gif', '.tiff', '.jpeg', '.svg', '.jfif']:
179         click.secho('ERROR: Imagefile must be of type ".png" og ".bmp"', fg='red',
180                     ↪ err=True)
181         sys.exit(1)
182
183     else:
184         click.secho(f'ERROR: The filetype of the file "{filename}" is not supported,
185                     ↪ please try something else. Valid extensions include .png, .bmp, .txt,
186                     ↪ .text, .mp4, .flv and .avi', fg='red', err=True)
187         sys.exit(1)
188
189 def load(filename: str) -> np.ndarray:
190     extension = filetype(filename)
191
192     if extension == IMAGE:
193         return _load_img(filename)
194     elif extension == TEXT:
195         return _load_txt(filename)
196
197     else:
198         click.secho('ERROR: Cannot load a videofile.', fg='red', err=True)
199         sys.exit(1)
200
201 # convert binary image to maze
202 # black (0) begin wall and white (255) being path
203 def _load_img(filename: str) -> np.ndarray:
204     # convert to binary array

```

```

204     return np.array(Image.open(filename).convert('1')).astype(np.uint8)
205
206
207 # convert textfile to maze
208 # Taking pound (#) as wall and space ( ) as path
209 def _load_txt(filename: str) -> np.ndarray:
210     # open file
211     with open(filename, 'r') as f:
212         # replace # with 1 and " " with 0
213         lines = [l.strip().replace("#", "0").replace(" ", "1")
214                  for l in f.readlines()]
215
216     # converting to ints and changing shape
217     maze = [[] for _ in lines]
218     for no, line in enumerate(lines):
219
220         #regex validation
221         if not re.match(r'^[01]*$', line):
222             print('ERROR: Textfile can only contain pounds and spaces ("#" and " "),
223                   ↪ failed on line {}'.format(no+1))
224             sys.exit(1)
225
226         # converting to ints
227         for num in line:
228             maze[no].append(int(num))
229
230     maze = np.array(maze)
231
232 # convenient function that reads the filetype, and the save it as an image
233 def write(filename: str, maze: np.ndarray) -> None:
234     extension = filetype(filename)
235
236     if extension == IMAGE:
237         return _write_img(filename, maze)
238     elif extension == TEXT:
239         return _write_txt(filename, maze)
240
241     else:
242         click.secho('ERROR: Trying to write imagefile in the wrong context', fg='red',
243                   ↪ err=True)
244         sys.exit(1)
245
246 # write maze to disk as image
247 def _write_img(destination: str, maze: np.ndarray) -> None:
248     Image.fromarray((maze*255).astype(np.uint8)).save(destination)
249
250
251 # write maze to disk as text file
252 def _write_txt(destination: str, maze: np.ndarray) -> None:
253     # create (and truncate) file
254     with open(destination, 'w+') as f:
255         # create normal python list, that are type independent
256         for row in maze:
257             tmp = []
258             for val in row:
259                 tmp.append(str(val))

```



```

260         f.write(''.join(tmp).replace('0', '#').replace('1', ' ') + '\n')
261
262
263 if __name__ == '__main__':
264     cli()

```

A.2 pathfinding/scheme.py

```

1  # std lib
2  import sys
3  import os.path
4  from time import time
5
6  # 3rd party
7  import click
8  from PIL import Image
9  import numpy as np
10 import matplotlib.pyplot as plt
11 from celluloid import Camera
12
13 # 1st party
14 from pathfinding import algs
15
16
17 # vars for determining filetype
18 IMAGE = 1
19 TEXT = 2
20 VIDEO = 3
21
22
23 # class containing neccesary information on every node
24 class Node:
25     def __init__(self, location: tuple):
26         self.location = location # (y, x)
27
28         # w s e n
29         # [location, dist]
30         self.nearby = [None] * 4
31
32         # location of node travelled by
33         self.via = None
34         # tracking distance travelled
35         self.dist = np.inf
36
37         # a*
38         self.dist_goal = np.inf
39
40         # set to be combined self.dist_goal and self.dist
41         self.combined = np.inf
42
43
44 # if printing the Node class, return it's location
45 def __repr__(self) -> str:
46     return 'Node{}'.format(self.location)
47
48

```

```

49     # two nodes are equal if they have the same location
50     def __eq__(self, other) -> bool:
51         return self.location == other.location
52
53
54     # heapq comparison for dijkstra and a*
55     def __lt__(self, other) -> bool:
56         return self.combined < other.combined
57
58
59     # used to creating sets for counting explored nodes
60     def __hash__(self) -> hash:
61         return hash(self.location)
62
63
64     # # iterate the neighbours
65     # def __iter__(self):
66     #     pass
67     #
68     # # self.location[index]
69     # def __getitem__(self, index):
70     #     pass
71
72
73 # class containing the whole data structure
74 class Graph:
75     def __init__(self, maze: np.ndarray):
76         self.maze = maze
77         self.animate = False
78
79         self.x = maze.shape[1]
80         self.y = maze.shape[0]
81
82         # start and end nodes for algs
83         self.first = (0, np.argmax(maze[0]))
84         self.last = (self.y-1, np.argmax(maze[-1]))
85
86         self.nodes = self.get_nodes()
87         self.node_count = len(self.nodes)
88
89         self.start = self.nodes[self.first] # Node(self.first)
90         self.end = self.nodes[self.last] # Node(self.last)
91
92         # connect nodes
93         self.gen_graph()
94
95         self.solved = False
96         self.path = None
97
98
99     # if printing the Graph class, return the np.ndarray structure
100    def __repr__(self) -> str:
101        return str(self.maze)
102
103
104    # find nodes on map
105    def get_nodes(self) -> dict:
106        # initiate dict

```

```

107     nodes = {self.first: Node(self.first), self.last: Node(self.last)}
108
109     # iterating the maze finding nodes
110     for y, line in enumerate(self.maze[1:-1], 1):
111         for x, field in enumerate(line):
112
113             # skip if wall (0 == False)
114             if not field:
115                 continue
116
117             # nodes around current
118             up = self.maze[y-1,x]
119             down = self.maze[y+1,x]
120             left = self.maze[y,x-1]
121             right = self.maze[y,x+1]
122
123             horizontal = (down and up) and not (left or right)
124             vertical = (left and right) and not (down or up)
125
126             # if field is straight, skip it
127             if vertical or horizontal:
128                 continue
129
130             # otherwise add it as a node
131             nodes[(y,x)] = Node((y,x))
132
133     return nodes
134
135
136     # used to get a node, from a specified index
137     # hashes the location, and finds the node
138     def get_node(self, location: tuple) -> Node:
139         return self.nodes[location]
140
141
142     # generate graph structure to tell nearby nodes for every node
143     def gen_graph(self) -> None:
144         # first Node
145         first = self.start
146         y, x = first.location
147         for dn in range(self.y - y):
148             tmp = y+dn+1
149             if self.maze[tmp, x+1] or self.maze[tmp, x-1] or tmp == self.y-1:
150                 # south node
151                 first.nearby[1] = ((tmp, x), dn+1)
152                 break
153
154         #last Node
155         last = self.end
156         y, x = last.location
157         for up in range(y):
158             tmp = y-up-1
159             if self.maze[tmp, x+1] or self.maze[tmp, x-1] or tmp == 0:
160                 # north node
161                 last.nearby[3] = ((tmp, x), up+1)
162                 break
163
164     # all other nodes

```

```

165     # skip the first and last node (first two declared)
166     for node in self.nodes.values():
167
168         # if the first or last node, skip
169         if node.location[0] == 0 or node.location[0] == self.y - 1:
170             continue
171
172         # current node location
173         y, x = node.location
174
175         # adjacent pixels
176         above = self.maze[y-1,x]
177         below = self.maze[y+1,x]
178         left = self.maze[y,x-1]
179         right = self.maze[y,x+1]
180
181         if above:
182             # decrement y with 1
183             for up in range(y):
184                 tmp = y-up-1
185                 # if found start node
186                 if tmp == 0:
187                     node.nearby[3] = ((tmp, x), up+1)
188                     break
189                 # if right or left are path or above is not, save length and exit
190                 if self.maze[tmp, x+1] or self.maze[tmp, x-1] \
191                     or not self.maze[tmp-1, x]:
192
193                     node.nearby[3] = ((tmp, x), up+1)
194                     break
195
196         if below:
197             # increment y with 1
198             for dn in range(self.y - y):
199                 tmp = y+dn+1
200                 # if found end node
201                 if tmp == self.y-1:
202                     node.nearby[1] = ((tmp, x), dn+1)
203                     break
204
205                 # if right or left are path or below is not, save length and exit
206                 if self.maze[tmp, x+1] or self.maze[tmp, x-1] \
207                     or not self.maze[tmp+1, x]:
208
209                     node.nearby[1] = ((tmp, x), dn+1)
210                     break
211
212         if left:
213             # decrement x with 1
214             for lt in range(x):
215                 tmp = x-lt-1
216                 # if up or down are path or left is not, save length and exit
217                 if self.maze[y+1, tmp] or self.maze[y-1, tmp] \
218                     or not self.maze[y, tmp-1]:
219
220                     node.nearby[0] = ((y, tmp), lt+1)
221                     break
222

```

```

223         if right:
224             # increment x with 1
225             for rt in range(self.x - x):
226                 tmp = x+rt+1
227                 # if up or down are path or right is not, save length and exit
228                 if self.maze[y+1, tmp] or self.maze[y-1, tmp] \
229                     or not self.maze[y, tmp+1]:
230
231                     node.nearby[2] = ((y, tmp), rt+1)
232                     break
233
234
235
236 # used to write image, used in assignment...
237 def save_nodes(self, destination: str):
238     writable = self.maze.copy()*255
239
240     # make array 3D
241     writable = writable[..., np.newaxis]
242     writable = np.concatenate((writable, writable, writable), axis=2)
243
244     for location in self.nodes.keys():
245         writable[location] = np.array([255, 0, 0])
246
247     Image.fromarray(writable).save(destination)
248
249
250
251 # decorator, to check if solved flag is set...
252 def _solved(func):
253     def checker(self, *args, **kwargs):
254         if self.solved:
255             return func(self, *args, **kwargs)
256         click.secho('ERROR: Graph not solved, cannot show solution', fg='red',
257                     ↪ err=True)
258     return checker
259
260 # convenient function that reads the filetype, and the save it as an image
261 @_solved
262 def save_solution(self, destination: str, extension: int) -> None:
263
264     if extension == IMAGE:
265         return self._save_solution_img(destination)
266
267     elif extension == TEXT:
268         self._save_solution_text(destination)
269
270     elif extension == VIDEO:
271         return self._save_solution_vid(destination)
272
273
274 # write maze solution to disk as image
275 def _save_solution_img(self, destination: str) -> None:
276     mz = self.maze.copy()*255
277
278     # make array 3D
279     mz = mz[..., np.newaxis]

```

```

280     mz = np.concatenate((mz, mz, mz), axis=2)
281
282     count = len(self.path)
283
284     for i in range(count - 1):
285         c = self.path[i].location #current
286         n = self.path[i+1].location #next
287
288         # red -> green
289         val = int((i/count)*255)
290         bgr = [255 - val, val, 0]
291
292         # y vals are the same: going horizontal
293         if c[0] == n[0]:
294             for loc in range(min(c[1], n[1]), max(c[1], n[1]) + 1):
295                 mz[c[0], loc] = bgr
296
297         # x vals are the same: going vertical
298         else:
299             for loc in range(min(c[0], n[0]), max(c[0], n[0]) + 1):
300                 mz[loc, c[1]] = bgr
301
302     Image.fromarray(mz).save(destination)
303
304
305     def _save_solution_txt(self, destination: str) -> None:
306         sol = []
307         # create normal python list, that are type independent
308         for row in self.maze:
309             tmp = []
310             for val in row:
311                 tmp.append(str(val))
312             sol.append(tmp)
313
314         # if fancy flag not set
315         if not fancy:
316             for node in self.path:
317                 y, x = node.location
318                 sol[y][x] = 'n'
319
320             # create (and truncate) file
321             with open(destination, 'w+') as f:
322                 # create normal python list, that are type independent
323                 for line in sol:
324                     f.write(''.join(line).replace('0', '#').replace('1', ' ') + '\n')
325             return
326
327     def _save_solution_vid(self, destination: str):
328         assert self.animate
329
330         for _ in range(20): # video finishes too early - cannot see solution
331             plt.imshow(self.mz)
332             self.cam.snap()
333         count = len(self.cam._photos)
334         if count > 2000:
335             click.confirm(f'Videofile will take some time to render, due to the amount
336                 ↪ of frames in solution ({count} frames in total), continue?',
337                 ↪ abort=True)

```

```

336         anim = self.cam.animate(blit=True, interval=30)
337         anim.save(destination)
338
339
340     # set the animate var to true
341     def visualize(self):
342         if self.x * self.y >= 100000:
343             click.confirm('Maze is very large, are you sure you want to animate it?',
344                 ↪ abort=True)
345
346         self.animate = True
347
348         self.mz = self.maze.copy()*255
349         # make array 3D
350         self.mz = self.mz[..., np.newaxis]
351         self.mz = np.concatenate((self.mz, self.mz, self.mz), axis=2)
352
353         # colors
354         self.EXPLORED = np.array([255, 0 , 0], dtype=np.uint8) #red
355         self.CURRENT = np.array([255, 255, 0], dtype=np.uint8) #green
356         self.PARENT = np.array([0, 255, 0], dtype=np.uint8) # green
357         self.mz[self.last] = np.array([0, 0, 255], dtype=np.uint8) #blue
358
359         self.cam = Camera(plt.figure())
360
361         self.imshow = plt.imshow(self.mz, interpolation='nearest', aspect='equal',
362             ↪ vmin=0, vmax=255, cmap="RdBu")
363         self.imshow.set_cmap('hot')
364         plt.axis('off')
365         self.cam.snap()
366
367     # decorator, to check if animate flag is set...
368     def _animate(func):
369         def checker(self, *args, **kwargs):
370             if self.animate:
371                 return func(self, *args, **kwargs)
372             return checker
373
374     @_animate
375     def frame(self, cy, cx, ny, nx):
376         miny, maxy = sorted([cy, ny])
377         minx, maxx = sorted([cx, nx])
378         self.mz[miny:maxy+1, minx:maxx+1] = self.EXPLORED
379         self.mz[ny, nx] = self.CURRENT
380         self.mz[cy, cx] = self.PARENT
381         plt.imshow(self.mz)
382         self.cam.snap()
383         self.mz[ny, nx] = self.EXPLORED
384         self.mz[cy, cx] = self.EXPLORED
385
386
387     def rightturn(self):
388         return algs.rightturn(self)
389
390
391     def breadthfirst(self):

```

```

392         return algs.breadthfirst(self)
393
394
395     def depthfirst(self):
396         return algs.depthfirst(self)
397
398
399     def dijkstra(self):
400         return algs.dijkstra(self)
401
402
403     def astar(self):
404         return algs.astar(self)

```

A.3 pathfinding/algs/__init__.py

```

1  from .rightturn import rightturn
2  from .breadthfirst import breadthfirst
3  from .depthfirst import depthfirst
4  from .dijkstra import dijkstra
5  from .astar import astar

```

A.4 pathfinding/algs/astar.py

```

1  import numpy as np
2  import heapq as hq
3
4  def astar(self):
5
6      start = self.start
7      end = self.end
8
9      #goal coordinates for calculating a*
10     gy, gx = end.location
11
12     # set initial value
13     start.dist = 0
14     start.dist_goal = np.hypot(start.location[0]-gy, start.location[1]-gx)
15     start.combined = start.dist + start.dist_goal
16
17     # bool array
18     visited = np.full((self.y, self.x), False)
19
20     # initiate vars
21     explored = 0
22     goal = np.inf
23
24     # priority queue
25     pq = [start]
26
27
28     # directions
29     # w s e n
30     # 0 1 2 3

```



```

31
32 while pq:
33     # explored nodes
34     explored += 1
35
36     current = hq.heappop(pq)
37
38     # if dist to node > to goal; break
39     if current.dist > goal:
40         break
41
42     if current == end:
43         goal = current.dist
44
45
46     for near in current.nearby:
47         # if not a wall
48         if near is not None:
49             node = self.get_node(near[0])
50             # and if not visited
51             if not visited[node.location]:
52                 visited[node.location] = True
53
54             cy, cx = current.location
55             ny, nx = node.location
56
57             # animate stuff
58             self.frame(cy, cx, ny, nx)
59
60             # calculate difference in locations (one is always 0)
61             distance = abs(cy-ny) + abs(cx-nx)
62             # set total distance and via node
63             node.dist = current.dist + distance
64             node.via = (cy, cx)
65
66             # distance to goal (Pythagoras)
67             node.dist_goal = np.hypot(ny-gy, nx-gx)
68
69             node.combined = node.dist + node.dist_goal
70
71             # push new node into heap
72             hq.heappush(pq, node)
73
74     # backtrack the path
75     path = []
76     current = end
77     while current != start:
78         path.append(current)
79         current = self.get_node(current.via)
80     # append the start node
81     path.append(self.start)
82
83     # reverse the path (from top to bottom)
84     path = path[::-1]
85
86     self.solved = True
87     self.path = path
88     # nodes explored, path, number of nodes, length of path

```

```
89     return explored, path, len(path), goal
```

A.5 pathfinding/algs/breadthfirst.py

```
1  import numpy as np
2  from _collections import deque as dq
3
4  def breadthfirst(self):
5      assert not self.solved
6
7      start = self.start
8      end = self.end
9
10     # set initial value
11     start.dist = 0
12
13     # bool array
14     visited = np.full((self.y, self.x), False)
15
16     # initiate vars
17     q = dq([start])
18     explored = 0
19     goal = np.inf
20
21     # directions
22     # w s e n
23     # 0 1 2 3
24     while q:
25         # explored nodes
26         explored += 1
27
28         # get first item
29         current = q.popleft()
30
31         # stop iteration, if at the end
32         if current == end:
33             goal = current.dist
34             break
35
36         for near in current.nearby:
37             # if not a wall
38             if near is not None:
39                 node = self.get_node(near[0])
40                 # and if not visited
41                 if not visited[node.location]:
42                     visited[node.location] = True
43
44                 cy, cx = current.location
45                 ny, nx = node.location
46
47                 # animate stuff
48                 self.frame(cy, cx, ny, nx)
49
50                 # calculate difference in locations (one is always 0)
51                 distance = abs(cy-ny) + abs(cx-nx)
52                 # set total distance and via node
```

```

53         node.dist = current.dist + distance
54
55         # append the node to the list to visit
56         q.append(node)
57         # set the via node for generating path
58         node.via = (cy, cx)
59
60
61     # backtrack the path
62     path = []
63     current = end
64     while current != start:
65         path.append(current)
66         current = self.get_node(current.via)
67
68     # append the start node
69     path.append(self.start)
70
71     # reverse the path (from top to bottom)
72     path = path[::-1]
73
74     self.solved = True
75     self.path = path
76     # nodes explored, path, number of nodes, length of path
77     return explored, path, len(path), goal

```

A.6 pathfinding/algs/depthfirst.py

```

1  from _collections import deque as dq
2  import numpy as np
3
4  def depthfirst(self):
5      assert not self.solved
6
7      start = self.start
8      end = self.end
9
10     # set initial value
11     start.dist = 0
12
13     # bool array
14     visited = np.full((self.y, self.x), False)
15
16     # initiate vars
17     q = dq([start])
18     explored = 0
19     goal = np.inf
20
21     # directions
22     # w s e n
23     # 0 1 2 3
24
25     while q:
26         # explored nodes
27         explored += 1
28

```

```

29     # get last item
30     current = q.popleft()
31
32     # stop iteration, if at the end
33     if current == end:
34         goal = current.dist
35         break
36
37     # the current node has now been visited
38     visited[current.location] = True
39
40     for near in current.nearby[::1]:
41         # if not a wall
42         if near is not None:
43             node = self.get_node(near[0])
44             # and if not visited
45             if not visited[node.location]:
46                 cy, cx = current.location
47                 ny, nx = node.location
48
49                 # animate stuff
50                 self.frame(cy, cx, ny, nx)
51
52                 # calculate difference in locations (one is always 0)
53                 distance = abs(cy-ny) + abs(cx-nx)
54                 # set total distance and via node
55                 node.dist = current.dist + distance
56
57                 # prepend the node to the list to visit
58                 q.appendleft(node)
59                 # set the via node for generating path
60                 node.via = (cy, cx)
61
62     # backtrack the path
63     path = []
64     current = end
65     while current != start:
66         path.append(current)
67         current = self.get_node(current.via)
68
69     # append the start node
70     path.append(self.start)
71
72     # reverse the path (from top to bottom)
73     path = path[::-1]
74
75     self.solved = True
76     self.path = path
77     # nodes explored, path, number of nodes, length of path
78     return explored, path, len(path), goal

```

A.7 pathfinding/algs/dijkstra.py

```

1  import numpy as np
2  import heapq as hq
3

```

```

4  def dijkstra(self):
5      assert not self.solved
6
7      start = self.start
8      end = self.end
9
10     # set initial value
11     start.dist = start.combined = 0
12
13     # bool array
14     visited = np.full((self.y, self.x), False)
15
16     # initiate vars
17     explored = 0
18     goal = np.inf
19
20     # priority queue
21     pq = [start]
22
23
24     # directions
25     # w s e n
26     # 0 1 2 3
27
28     while pq:
29         # explored nodes
30         explored += 1
31
32         current = hq.heappop(pq)
33
34         # if dist to node > to goal; break
35         if current.dist > goal:
36             break
37
38         if current == end:
39             goal = current.dist
40
41
42         for near in current.nearby:
43             # if not a wall
44             if near is not None:
45                 node = self.get_node(near[0])
46                 # and if not visited
47                 if not visited[node.location]:
48                     visited[node.location] = True
49
50                 cy, cx = current.location
51                 ny, nx = node.location
52
53                 # animate stuff
54                 self.frame(cy, cx, ny, nx)
55
56                 # calculate difference in locations (one is always 0)
57                 distance = abs(cy-ny) + abs(cx-nx)
58                 # set total distance and via node
59                 node.dist = node.combined = current.dist + distance
60                 node.via = (cy, cx)
61

```

```

62         # push new node into heap
63         hq.heappush(pq, node)
64
65     # backtrack the path
66     path = []
67     current = end
68     while current != start:
69         path.append(current)
70         current = self.get_node(current.via)
71
72     # append the start node
73     path.append(self.start)
74
75     # reverse the path (from top to bottom)
76     path = path[::-1]
77
78     self.solved = True
79     self.path = path
80     # nodes explored, path, number of nodes, length of path
81     return explored, path, len(path), goal

```

A.8 pathfinding/algs/rightturn.py

```

1  def rightturn(self):
2      assert not self.solved
3
4      start = self.start
5      end = self.end
6
7      second = self.get_node(start.nearby[1][0])
8      second.via = 0
9
10     # initiate vars
11     path = [start, second]
12     current = second
13     travelled = 0
14
15     direction = 0
16
17
18     # directions
19     # w s e n
20     # 0 1 2 3
21
22     while True:
23
24         if current == end:
25             self.solved = True
26             self.path = path
27             # nodes explored, path, number of nodes, length of path
28             return len(set(path)), path, len(path), travelled
29
30         if current == start:
31             return False, len(set(path))
32
33         # west

```

```

34     if direction % 4 == 0:
35         # if there is a node the that side
36         if current.nearby[0] is not None:
37             cy, cx = current.location
38
39             following = current.nearby[0]
40             current = self.get_node(following[0])
41             current.via = following
42
43             self.frame(cy, cx, *current.location)
44
45             travelled += following[1]
46
47             path.append(current)
48             direction -= 1
49             continue
50         direction += 1
51
52     # south
53     if direction % 4 == 1:
54         if current.nearby[1] is not None:
55             cy, cx = current.location
56
57             following = current.nearby[1]
58             current = self.get_node(following[0])
59             current.via = following
60
61             self.frame(cy, cx, *current.location)
62
63             travelled += following[1]
64
65             path.append(current)
66             direction -= 1
67             continue
68         direction += 1
69
70     # east
71     if direction % 4 == 2:
72         if current.nearby[2] is not None:
73             cy, cx = current.location
74
75             following = current.nearby[2]
76             current = self.get_node(following[0])
77             current.via = following
78
79             self.frame(cy, cx, *current.location)
80
81             travelled += following[1]
82
83             path.append(current)
84             direction -= 1
85             continue
86         direction += 1
87
88     # north
89     if direction % 4 == 3:
90         if current.nearby[3] is not None:
91             cy, cx = current.location

```

```

92         following = current.nearby[3]
93         current = self.get_node(following[0])
94         current.via = following
95
96         self.frame(cy, cx, *current.location)
97
98         travelled += following[1]
99
100         path.append(current)
101         direction -= 1
102         continue
103     direction += 1
104

```

A.9 pathfinding/comparer.py

```

1  #!/usr/bin/env python
2  import matplotlib.pyplot as plt
3  import numpy as np
4  import re
5  import os.path
6  import sys
7
8  # read contents of file
9  def from_file(filename: str) -> str:
10     if not os.path.isfile(filename):
11         raise FileNotFoundError('The requested file was not found')
12
13     with open(filename, 'r') as f:
14         content = f.readlines()
15
16     return content
17
18
19  def organize(load: str) -> dict:
20
21     aliases = {
22         'astar': 'a*',
23         'breadthfirst': 'BF',
24         'depthfirst': 'DF',
25         'dijkstra': 'dijk',
26         'rightturn': 'RT'
27     }
28     stats = {}
29
30     for size in [51, 101, 501, 1001, 2501]:
31         stats[size] = {}
32
33         for alg in ['a*', 'BF', 'DF', 'dijk', 'RT']:
34             stats[size][alg] = []
35
36
37     size_pattern = re.compile(r'(?<=/) [0-9]+')
38     time_pattern = re.compile(r'(?<= ) [0-9\.]+(=? )')
39     algorithm_pattern = re.compile(r'\w+$')
40

```



```

41     # skip first line with date information
42     for line in load[1:]:
43         l = line.strip()
44
45         # size of maze
46         size = int(size_pattern.search(l).group())
47
48         # time took to solve
49         time = float(time_pattern.search(l).group())
50
51         # using algorithm
52         alg = algorithm_pattern.search(l).group()
53
54         ag = aliases[alg]
55
56         stats[size][ag].append(time)
57
58     return stats
59
60
61 def plot_stats(stats: dict) -> None:
62     fig, axes = plt.subplots(nrows=3, ncols=2, figsize=(10, 15))
63
64     n = 1
65     for s in stats.items():
66         size, a = s
67         algs = list(a.keys())
68         observations = list(a.values())
69
70         # row and column
71         r, c = n // 2, n % 2
72         n+=1
73
74         bx = axes[r, c].boxplot(observations, labels = algs, patch_artist=True)
75
76         axes[r, c].set_title('Maze size: {}'.format(size))
77         axes[r, c].set_xlabel('Algorithms')
78         axes[r, c].set_ylabel('Time spent in ms')
79         axes[r, c].yaxis.grid(True) #grid
80
81         colors = ['pink', 'lightblue', 'lightgreen', 'khaki', 'slateblue']
82
83         for box, color in zip(bx['boxes'], colors):
84             box.set_facecolor(color)
85
86     fig.delaxes(axes[0,0])
87     plt.show()
88
89
90 def main(filename) -> None:
91     load = from_file(filename)
92     stats = organize(load)
93
94     plot_stats(stats)
95
96
97 if __name__ == '__main__':
98     main('../timings.txt')

```
