GYMNÁZIUM
JÍROVCOVA

MATURITNÍ PRÁCE

Vizualizace významných algoritmů

Alexandr Bihun

vedoucí práce: Dr. rer. nat. Michal Kočer

V Českých Budějovicích 14. února 2024 **2023/2024**

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně s vyznačením všech použitých pramenů.

V Českých Budějovicích dne .... podpis ....

Alexandr Bihun

Abstrakt

Tato maturitní práce se zaměřuje na vysvětlení chodu známých algoritmů v oblasti pathfindingu (vyhledávání cest), rovněž jako na jejich analýzu a přiblížení jejich využití v opravdovém světě. Dále bude naznačeno, jak jsem implementoval za pomoci knihovny Pygame v jazyce Python aplikaci vizualizující principy jednotlivých algoritmů.

Klíčová slova

algoritmy, analýza algoritmů, vyhledávání cest, grafy, vizualizace, python, pygame

Poděkování

Tady bude poděkování.

Obsah

Úvod

Přesto, že si to většina lidí nejspíše neuvědomuje, využívají algoritmy na denním pořádku. V této maturitní práci se proto pokusím nejprve objasnit co se za tímto pojmem vůbec skrývá a jak o algoritmech přemýšlet, rovněž nastíním některé teoretické základy tohoto odvětví počítačové vědy. Dále představím mnou vybrané algoritmy, zaměřené na hledání cest v rozlišných prostředích. Důvodem tohoto výběru je, že to byly jedny z prvních mně představených algoritmů, navíc jsou pro jejich řadu využití široce používané.

Hlavním cílem této práce je navrhnout a úspěšně vyvinout program, který bude schopen tyto vybrané algoritmy efektivně vizualizovat. Bude kladen důraz na to, aby byl program jednoduchý a uživatelsky přívětivý, zároveň však poskytoval všechny potřebné funkce. Smyslem této vizualizace bude pomoci uživateli těmto algoritmům lépe porozumět a „osahat" si je a jejich chování, což očekávám že povede k hlubšímu pochopení těchto algoritmů. Přidaným benefitem vizualizace je bezpochyby její hravá forma s interaktivita v kontrastu s běžným teoretickým přístupem k výkladu algoritmů.

Část I

Představení a analýza vybraných algoritmů

1 Algoritmus

Samotné slovo algoritmus vzniklo zkomolením jména významného perského matematika Abu Jafara Muhammada ibn Mūsā al-Chwārizmiho, který v první polovině devátého století ve svých dílech položil základy algebry a způsobů řešení lineárních a kvadratických rovnic. Po vzniku latinského překladu jeho spisu o indickém početním systému, ve kterém ukazuje, jak provádět základní početní operace, nabylo jeho jméno nového významu. Do latiny byl totiž přeložen pod titulem *Algoritmi de Numero Indorum* (česky „Algoritmi o číslech od Indů“), kde slovo Algortimi je latinizovaná forma jeho jména. Toto slovo se pak začalo používat jako označení různých matematických postupů. [20] [12] [11]

1.1 Definice algoritmu

Obecně se dá říci, že algoritmus je nějaká přesně daná posloupnost kroků, kterou lze dosáhnout kýženého výsledku. Tím pádem definici algoritmu splňují například recepty z kuchařek, návody na konstrukci nábytku, pracovní postupy a podobně. [12]

Nejčastěji se ale s algoritmy setkáváme v kontextu matematické informatiky, kde popisují početní proceduru, kterou lze řešit konkrétní úlohy. Tyto algoritmy pak musí být schopné přijmout jakýkoli vstup popisující zadaný problém a vyřešit ho, tj. vyprodukovat korektní výstup. Zároveň musí být zapsány tak, aby jim porozuměl počítač. K tomuto účelu slouží *programovací jazyky*, které se skládají ze slov s jasně danými významy. Spustitelný algoritmus přepsaný ve vhodném programovacím jazyce nazýváme *program*. [6]

1.1.1 Vlastnosti algoritmu

Podle [18] a [20] od algoritmu požadujeme (většinou)[1] tyto vlastnosti:

1. Elementárnost - algoritmus sestává z konečného počtu jednoduchých, srozumitelných kroků.

2. Konečnost - algoritmus doběhne v konečném množství kroků.

3. Korektnost - algoritmus produkuje pro každý správný vstup korektní výsledek.

4. Obecnost - algoritmus řeší všechny instance daného problému. [2]

5. Determinovanost - každý krok vykonávání algoritmu je jednoznačně určený.

1.2 Ukázky jednoduchých algoritmů

Nejstarší dochované algoritmy se datují již do Sumerské říše, odkud pochází hliněná tabulka s prvním dochovaným algoritmem na dělení, její odhadované stáří činí 4500 let. V antickém Řecku vznikaly první algoritmy pro aritmetiku, jako například Euklidův algoritmus, či Eratosthenovo síto. [3]

1.2.1 Eratosthenovo síto

Tento algoritmus pro hledání prvočísel popsal poprvé řek Nikómachos z Gerasy, připisuje ho Eratosthenovi z Kyrény. Jeho algoritmus vygeneruje všechna prvočísla menší než nějaké číslo n podle jednoduché procedury [3]. Toto číslo n, podle kterého se odvíjí průběh algoritmu, označujeme jako vstup algoritmu.
Samotné kroky algoritmu pak jsou:

1. Vytvoř posloupnost čísel od 2 do n.

2. Vyber nejmenší dosud nevybrané číslo posloupnosti a označ ho jako prvočíslo.

3. Odstraň všechny násobky právě vybraného prvočísla.

4. Vrať se na krok 2, pokud jsi naposledy nevybral číslo větší než $\sqrt{n}$.

[1] Existují algoritmy, které např. generují pouze přibližné řešení.
[2] Instance problému je jeden konkrétní vstup pro tento problém.

5. Na konci zůstanou v posloupnosti pouze prvočísla.

Tento algoritmus jsme právě popsali v prostém jazyce. Je očividně proveditelný člověkem a jeho bezchybným provedením lze dojít ke korektnímu výsledku. Mohli bychom ho stejně tak vyjádřit v *pseudokódu*, což je speciální druh jazyka, který připomíná běžné programovací jazyky. Pseudokód se však vyhýbá implementačním detailům a konkrétním standardům opravdových jazyků, zároveň je však tak přesný, aby šel s trochou snahy jednoduše převést do vhodného programovacího jazyka a jednoznačně vyjádřil myšlenku. [10]

1.2.2 Euklidův algoritmus

Euklidův algoritmus je dodnes používaný algoritmus pro nalezení největšího společného dělitele dvou přirozených čísel [10]. Jeho vyjádření v pseudokódu vypadá následovně:

Algoritmus 1: Euklides

Vstup: $x, y \in \mathbb{N}$

$a \leftarrow x, b \leftarrow y$

Dokud *můžeš* **dělej:**

  Pokud $a < b$ **pak:**

    prohoď a s b

  Pokud $b = 0$ **pak:**

    vyskoč z cyklu

  $a \leftarrow a \bmod b$ ◁ `mod značí zbytek po vydělení` a `hodnotou` b

Výstup: Největší společný dělitel $a = \gcd(x, y)$

V hlavičce je algoritmus pojmenovaný a očíslovaný v rámci celého dokumentu. Výraz $a \leftarrow x$ vyjadřuje vytvoření nové proměnné a (pokud do té doby neexistovala) a uložení hodnoty proměnné x do a. Proměnná v tomto kontextu je jako krabička, do které lze uložit informaci (jako číslo nebo slovo), a kdykoliv lze nahlédnout dovnitř a zobrazit si tuto informaci nebo ji nahradit jinou. Svislé čáry značí bloky kódu, v bloku kódu se nejčastěji vyskytuje vnitřní logika cyklu, podmínky nebo funkce. Dále cokoliv za značkou ◁ je komentář, čili text pouze pro vysvětlení samotného kódu.

Existují i jiné způsoby zápisu algoritmů jako např. grafický zápis flowchartem neboli vývojovým diagramem, či pomocí struktogramu [18]. V této práci budeme nadále používat pro popis složitějších algoritmů pouze pseudokód, pro jeho jednoduchost a zároveň přesnost.

2 Analýza algoritmů

Pro jeden problém obvykle existuje více algoritmů, které ho řeší. Abychom mohli porovnávat různé algoritmy mezi sebou, potřebujeme zavést nějaké metriky či veličiny, které nám budou popisovat jejich vlastnosti.

Pro nás nejdůležitějšími vlastnostmi algoritmu jsou jeho doba běhu a množství paměti potřebné pro jeho běh. Důvodem je, že samotná konečnost algoritmu není zárukou toho, že se po jeho spuštění dočkáme výsledku. Může se totiž stát, že instrukcí bude tak moc, že bychom se jejich zpracování a tudíž výsledku nemuseli vůbec dočkat.

Obdobně na dnešních počítačích nemáme neomezené množství výpočetní paměti, přestože trendem v této oblasti je neustálý růst, stejně jako u rychlosti výpočetních jednotek[1]. Proto musíme algoritmy optimalizovat i z tohoto hlediska. [20]

2.1 Časová a prostorová složitost

Časovou složitost algoritmu definujeme jako funkci f přiřazující každé velikosti vstupu počet elementárních instrukcí nutných pro vykonání algoritmu se vstupem této velikosti. Elementárními instrukcemi pak rozumíme aritmetické operace, porovnání apod. jednoduše to, co zvládne běžný procesor jednou nebo pár instrukcemi. Dále prohlásíme, že každá jedna instrukce trvá vždy konstantně času. Vstupů jedné velikosti bude obvykle více, proto vždy vybereme ten, který vyžaduje nejvíc instrukcí. Tím pádem bude funkce dávat počty instrukcí v nejhorším případě a ty by měly být i úměrné s dobou běhu algoritmu. [10]

Prakticky to znamená, že si můžeme napsat algoritmus v pseudokódu a spočítat kolikrát se vykoná každá instrukce pro různě velké vstupy. Obvykle bude tato funkce rostoucí a nás nejvíce zajímá, jak rychle roste vzhledem k růstu velikosti vstupu. To znamená, že nás zajímá limitní chování funkce složitosti. Proto se zavádí takzvaná *asymptotická notace*.

Prostorová složitost je zavedena obdobně, s rozdílem, že místo počtu instrukcí určuje,

[1] Fenomén, že se přibližně každé dva roky zdvojnásobí výkon nových počítačů, se někdy nazývá *Moorův zákon*.

kolik výpočetní paměti algoritmus potřebuje pro svůj běh v závislosti na velikosti vstupu. [10]

2.2 Asymptotická notace

Asymptotická notace je způsob, jak vyjádřit řád růstu funkce. Jejím úkolem je zjednodušit funkci složitosti algoritmu s ohledem na to, že s dostatečně velkými vstupy bude rychlost růstu funkce určovat jen nejvýznamnější, tj. nejrychleji rostoucí člen. Toho docílí eliminací všech méně významných členů včetně konstant. Rozlišují se tři notace: $\mathcal{O}$-notace, Ω-notace, Θ-notace. V této sekci bylo čerpáno z [4].

2.2.1 $\mathcal{O}$-notace

$\mathcal{O}$-notace udává asymptotické omezení shora. Určuje, že funkce roste maximálně stejně rychle jako určitá míra.

Formálně definujeme, že funkce $f(n)$ náleží do třídy složitosti $\mathcal{O}(g(n))$, pokud existuje konstanta $c > 0$ a n_0 takové, že pro každé $n > n_0$ platí $f(n) \leq c \cdot g(n)$.

Situaci, kdy $f(n)$ náleží do $\mathcal{O}(g(n))$ značíme $f(n) = \mathcal{O}(g(n))$.

Pokud by např. funkce $2n^2 + 100n + 3000$ charakterizovala časovou složitost nějakého algoritmu, zapíšeme skutečnost, že její řád růstu je n^2 následovně: $2n^2+100n+3000 = \mathcal{O}(n^2)$. Tvrdíme, že časová složitost takového algoritmu je $\mathcal{O}(n^2)$. Je vidět, že $\mathcal{O}$ „seškrtne“ všechny méně významné členy, rovněž jako konstanty[2] násobící všechny členy. Takto zavedená notace zjednodušuje porovnávání různých algoritmů mezi sebou.

$\mathcal{O}$-notace udává dobu běhu programu v nejhorším případě, tj. na asymptoticky nejsložitějším vstupu. Je možné, že existují i vstupy, pro které má algoritmus lepší asymptotickou časovou složitost než $\mathcal{O}(g(n))$, které vyšlo pro nejhorší případ. Přesto, jelikož $\mathcal{O}$ omezuje ze shora, nebude tvrzení, že algoritmus má v každém případě složitost $\mathcal{O}(g(n))$ chybné. Uvažme, že funkce $h(n) = n^2$ je nejen $\mathcal{O}(n^2)$, ale i $\mathcal{O}(n^3)$, obecně je $\mathcal{O}(n^c)$, pro $c \geq 2$.

Ω-notace a Θ-notace jsou zavedeny obdobně. Ω-notace udává asymptotické omezení zdola, tj. určuje funkce asymptoticky rostoucí alespoň stejně rychle jako nějaká daná míra.

Θ-notace udává nejtěsnější mez, a to oboustrannou, tj. říká, že funkce roste stejně rychle jako daná míra. Pokud platí $f(n) = \mathcal{O}(g(n))$ a $f(n) = \Omega(g(n))$, pak platí $f(n) = \Theta(g(n))$.

[2] V praxi se může velká konstanta promítnout do doby běhu programu, proto se někdy zohledňuje, obzvlášť vybíráme-li mezi dvěma algoritmy se stejnými asymptotickými složitostmi.

Ω značení pak používáme pro charakterizaci složitosti nejlepšího případu, $\Theta(g)$ se používá pro průměrný případ. Formální definice jsou uvedeny v [4].

Složitost	n = 10	n = 100	n = 1000	n = 100 000
log n	3.3 ns	6.6 ns	10 ns	16.6 ns
n	10 ns	100 ns	1 μs	100 μs
n log n	33 ns	664 ns	10 μs	1.66 ms
n^2	100 ns	10 μs	1 ms	10 s
n^3	1 μs	1 ms	1 s	11.5 dnů
2^n	1 μs	$4 \cdot 10^{13}$ let	$3 \cdot 10^{284}$let	$\approx \infty$
n!	3 ms	$3 \cdot 10^{141}$ let	$\approx \infty$	$\approx \infty$

Tabulka 2.1: Odhad doby běhu algoritmů s různými složitostmi

Běžný počítač provede okolo 10^9 operací za vteřinu. Tabulka 2.1 ukazuje některé časté složitostní funkce[3] a odhad, jak dlouho by algoritmus s uvedenou složitostí běžel na běžném počítači pro různě velké vstupy. [20]

Z tabulky 2.1 je vidět, že polynomiální nebo logaritmické složitosti nabízí „rozumný“ čas běhu vůči velikosti vstupu. Naopak algoritmy s exponenciální nebo horší složitostí jsou prakticky nepoužitelné.

[3]Pro funkce složitosti s logaritmem obvykle myslíme logaritmus se základem dva. Ten se v počítačové vědě objevuje tak často, že se u něj dvojka ani nezapisuje.

3 Algoritmy pro hledání cest

Tato kapitola se zaměří na popis a analýzu vybraných algoritmů pro hledání cest. Vyhledávání cesty je problém, který se objevuje v různých odvětvích lidské činnosti, například při plánování nejkratší nebo nejrychlejší trasy mezi dvěma městy pomocí internetových mapových aplikací nebo v GPS navigaci. Algoritmy pro hledání cest se také využívají pro vyhledávání jízdních řádů, směrování paketů v počítačových sítích, v počítačových hrách, v robotice pro plánování pohybu robotů a podobně. Tyto algoritmy budou hlavním předmětem mého vizualizačního programu. Zdrojem pro algoritmy zpracované v této kapitole jsou [4, 10, 20, 8, 7, 17, 16, 2, 15]

.

3.1 Základy teorie grafů

Algoritmy, které si představíme, jsou založeny na poznatcích matematické disciplíny *teorie grafů*. Řada matematických i praktických problémů ze skutečného světa se totiž dá převést na grafový problém. Proto si v této sekci představíme základní pojmy z teorie grafů. Čerpáno bylo ze zdrojů [10] [18] [4] [9].

- *Neorientovaný graf* G je dvojice (V, E), kde V je množina *vrcholů* grafu a E je množina *hran* grafu. Každá hrana $e \in E$ je neuspořádanou dvojicí $\{u, v\}$ pro $u, v \in V$. Značení $|V|$ určuje celkový počet vrcholů v grafu G, podobně $|E|$ pro hrany.
- *Orientovaný graf* se od neorientovaného liší tím, že hrany mají směr. Každá hrana je uspořádanou dvojicí (u, v), kde u je počáteční vrchol a v koncový vrchol.
- *Neohodnocený graf* je takový, který nemá přiřazené žádné hodnoty (váhy) jednotlivým hranám.

- *Ohodnocený graf* má přiřazené hodnoty (váhy) jednotlivým hranám, což umožňuje kvantifikovat například vzdálenost či náklad spojený s každou hranou.
- V neorientovaném grafu jsou *sousedé* vrcholu v všechny vrcholy spojené s v hranou. Pro orientovaný graf jsou *následníci* vrcholu v ty vrcholy, do kterých vede hrana z v, *předchůdci* jsou vrcholy, z kterých vede hrana do v a předchůdci a následníci dohromady jsou sousedé vrcholu v.
- *Cesta* v grafu G označuje takovou posloupnost vrcholů a hran $(v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n)$, kde v_i jsou vrcholy grafu G a e_i jsou jsou hrany grafu G. Každá hrana e_i má koncové vrcholy v_{i-1} a v_i a žádný vrchol se v posloupnosti neopakuje.
- Vrchol w je *dosažitelný* z vrcholu v, pokud existuje cesta z v do w.

Tento výčet není kompletní, ale tyto pojmy nám budou stačit k pochopení všech následujících algoritmů.

(a) (b)

Obrázek 3.1: Příklady grafů. (a) Neorientovaný graf. (b) Orientovaný graf.

Příklady nakreslení grafu vidíme na obrázku 3.1. Pomocí neorientovaného neohodnoceného grafů můžeme například reprezentovat vztahy mezi lidmi, kde vrcholy budou jednotlivé osoby a hrany povedou mezi těmi dvojicemi vrcholů, které spolu kamarádí. Nebo můžeme pomocí ohodnoceného grafu modelovat síť měst, kde hodnota hran mezi dvěma městy bude značit délku silnice mezi městy. Často se také grafy využívají pro popis *stavového prostoru* her, kde vrcholy představují stavy a hrany mezi nimi akce, kterými lze přejít z jednoho stavu do druhého.

3.1.1 Reprezentace grafu v počítači

Existuje několik způsobů, jak efektivně reprezentovat graf v počítači. Uvedu zde dvě nejběžnější metody použitelné pro jak neorientované, tak orientované grafy. Budou určené pro neohodnocené grafy, ale s mírnými úpravami se dají použít i pro ohodnocené grafy.

- *Matice sousednosti:* Očíslujeme všechny vrcholy grafu od 1 do $|V|$. Matice sousednosi A má velikost $|V| \times |V|$ a je definovaná jako $A = (a_{ij})$, kde

$$a_{ij} = \begin{cases} 1 & \text{pokud } (i,j) \in E, \\ 0 & \text{jinak.} \end{cases}$$

  Výhodou této reprezentace je, že zjistit zda jsou dva vrcholy spojené hranou zvládneme v konstantním čase $\mathcal{O}(1)$, vůbec nezáleží na velikosti grafu. Vyjmenování všech následníků vrcholu zabere $\Theta(|V|)$. Nevýhodou je, že zabírá prostor $\Theta(|V|^2)$.

- *Seznam sousedů:* Vrcholy opět očíslujeme od 1 do $|V|$. Tato reprezentace uchovává pole, které má na i-té pozici ukazatel na seznam následníků vrcholu i. Tato metoda je efektivnější pro *řídké grafy*, ve kterých $|E| \ll |V|^2$. Zabírá prostor $\Theta(|E| + |V|)$, ale ověřit existenci hrany (i, j) zabere $\mathcal{O}(|V|)$. Výhodou je, že najít všechny následníky vrcholu je lineární s jejich počtem, tedy $\mathcal{O}$(počet následníků).

3.2 Prohledávání do hloubky

Algoritmus prohledávání do hloubky (anglicky *depth-first search*, zkráceně *DFS*) je algoritmus pro procházení grafu. Jak implikuje název, DFS prochází graf vždy tak *hluboko*, jak to jde. DFS začne v počátečním vrcholu v_0 a prozkoumává vždy hrany naposledy nalezeného vrcholu v, z kterého ještě vedou neprozkoumané hrany. Jakmile narazí na takový vrchol, který nemá žádné neprozkoumané sousedy, nemůže už jít hlouběji a metodou nazývanou *backtracking* se vrátí na poslední vrchol s alespoň jedním neprozkoumaným sousedem. Tento proces se opakuje do té doby, než jsou nalezeny všechny vrcholy dosažitelné z v_0.

Pro implementaci algoritmu DFS se využívá datové struktury[1] *zásobník*, případně lze použít namísto zásobníku techniku *rekurze*, která stejně využívá systémový zásobník. Zásobník je datová struktura, která si pamatuje pořadí svých prvků, a řídí se pravidlem LIFO - Last In, First Out. To znamená, že nové prvky přidává na konec a odebírá je rovněž z konce[2]. Těmto operacím se obvykle říká PUSH a POP.

DFS se zásobníkem je popsáno pseudokódem 2. Algoritmus 2 pouze navštíví každý vrchol dosažitelný z v_0 a označí ho za navštívený, nic ale nevrací. To proto, že DFS je algoritmus

[1]Datová struktura je abstraktní způsob ukládání dat v počítači, s kterými umí provádět určité operace.

[2]Stejně, jako kdybychom chtěli přidat/odebrat náboj ze zásobníku pistole.

Algoritmus 2: Prohledávání do hloubky

Vstup: Graf $G = (V, E)$, počáteční vrchol $v_0 \in V$

Přidej v_0 do zásobníku Z a označ v_0 jako navštívený.

Dokud *zásobník Z není prázdný* **dělej:**

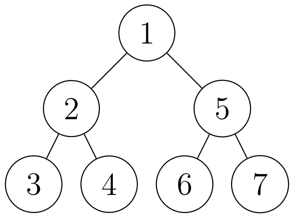
    $v \leftarrow$ Z.POP() $\triangleleft$ `Odebere ze zásobníku horní prvek a uloží ho do` v

    Pro *všechny sousedy w vrcholu v* **dělej:**

        Pokud *w není navštívený* **pak:**

            Z.PUSH(w)

            Označ w jako navštívený.



Obrázek 3.2: Graf s vrcholy označenými podle pořadí, v jakém je projde DFS

na procházení grafu. Mohli bychom ho ale lehce modifikovat tak, aby našel cestu z vrcholu v_0 do v_1. Konkrétně by si pro každý vrchol pamatoval jeho předchůdce a jakmile by našel cílový vrchol, tak by postupně vypsal cíl, předchůdce cíle, atd. Nevýhodou prohledávání do hloubky je, že pokud ho využijeme k nalezení cesty mezi dvěma vrcholy, nalezená cesta není nutně nejkratší možná, v důsledku pořadí, v jakém DFS prochází vrcholy.

Časová komplexita je $\mathcal{O}(|V| + |E|)$, protože v nejhorším případě projde celý graf. Prostorová složitost je $\Theta(|V| + |E|)$.

3.3 Prohledávání do šířky

Algoritmus prohledávání do šířky (anglicky *breadth-first search*, zkráceně *BFS*) je dalším algoritmem pro procházení grafu. BFS prochazí graf do *šířky*, tj. postupně prozkoumává všechny sousedy počátečního vrcholu v_0, pak sousedy sousedy sousedů atd.

Na rozdíl od DFS používá BFS frontu namísto zásobníku. Fronta je datová struktura, která pracuje podle pravidla FIFO (First In, First Out), což znamená, že prvek, který je v frontě nejdéle, bude odebrán jako první. Operaci přidání prvku na konec fronty se obvykle říká ENQUEUE a odebrání prvku ze začátku fronty DEQUEUE.

Algoritmu se někdy přezdívá „algoritmus vlny", protože nalezne nejdřív všechny vrcholy vzdálené od v_0 o jedna (sousedy v_0), pak ty vzdálené o dva (sousedy sousedů v_0) a tak dále, jako by se z v_0 šířila vlna po vodní hladině.

Detailní popis nalezneme v pseudokódu 3.

Algoritmus 3: Prohledávání do šířky

Vstup: Graf $G = (V, E)$, počáteční vrchol $v_0 \in V$

Přidej v_0 do fronty Q a označ v_0 jako navštívený.

Dokud *fronta Q není prázdná* **dělej:**

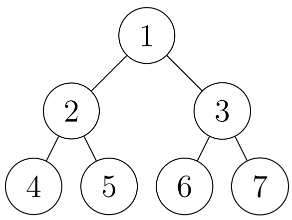
    $v \leftarrow$ Q.DEQUEUE()

    Pro *všechny sousedy w vrcholu v* **dělej:**

        Pokud *w není navštívený* **pak:**

            Q.ENQUEUE(w)

            Označ w jako navštívený.



Obrázek 3.3: Graf s vrcholy označenými podle pořadí, v jakém je projde BFS

Časová složitost BFS je $\mathcal{O}(|V| + |E|)$, což znamená, že v nejhorším případě projde celý graf. Prostorová složitost je $\Theta(|V| + |E|)$.

Výhodou BFS je to, že při hledání cesty vždy najde nejkratší[3] cestu mezi dvěma vrcholy, případně pozná, že mezi nimi neexistuje cesta. Protože nijak nezapočítává ohodnocení hran, bude i v ohodnocených grafech nacházet cestu s nejméně vrcholy. V takových grafech ale jako nejkratší cestu obvykle považujeme tu, která má nejmenší součet ohodnocení všech hran. Z tohoto důvodu nenalezne BFS optimální cestu na ohodnocených grafech.

[3] Takovou, která obsahuje nejméně vrcholů.

3.4 Uspořádané vyhledávání

Dosud představené algoritmy jsou primárně určené pro procházení grafů, přestože se dají aplikovat na hledání cesty mezi dvěma vrcholy. Naopak další algoritmy, které představím, jsou zaměřené na hledání optimálních cest, a to i v ohodnocených grafech.

Tyto algoritmy vybírají který vrchol *expandovat*[4] jako další v pořadí podle nějaké evaluační funkce $f : V \to \mathbb{R}$. Ta vrací nějakou reálnou hodnotu pro každý vrchol $v \in V$. Hromadně se označují jako algoritmy třídy uspořádaného vyhledávání (anglicky best-first search), protože pořadí, v jakém prohledávají vrcholy, je nějak uspořádané podle f.

Typicky se jako první prochází vrchol v s nejmenší hodnotou $f(v)$. K tomu se často používá prioritní fronta, která řadí své prvky vzestupně podle priority, přiřazené ke každému prvku. Většinou se implementuje pomocí datové struktury *haldy*. Více o haldách viz [10].

Poznámka: při rešerši na uspořádané vyhledávání jsem se častokrát setkával s rozlišnou terminologií. V některých zdrojích [2] se jako best first-search označuje algoritmus, který jiní [15] pojmenovávají jako greedy best-first search. Jiní [17, 19, 7] zase považují best-first search jako algoritmický princip nebo třídu algoritmů. Já jsem zvolil stejné pojmenování jako [15, 17, 7], protože mi přišlo nejvíce konzistentní a logické.

3.4.1 Uniform Cost Search

Uniform cost search, dále jen UCS, je algoritmus třídy uspořádaného vyhledávání, který najde optimální cestu mezi počátečním vrcholem v_0 a cílovým vrcholem c v grafu s kladně ohodnocenými hranami. UCS funguje podobně jako BFS, jen místo postupného prohledávání vrcholů ve stejné hloubce (se stejným minimálním počtem hran od startu) prohledává UCS ve „vrstvách“ stejné ceny.

Evaluační funkcí pro UCS je $f(v) = g(v)$, kde $g(v)$ je cena cesty mezi počátečním vrcholem s a vrcholem v. UCS používá prioritní frontu, většinou označovanou jako OPEN. Na začátku je do ní vložen pouze počáteční vrchol s prioritou 0. V každé iteraci je z OPEN odebrán vrchol s největší prioritou a expandován. Největší prioritu mají prvky s nejnižší $g(v)$. To znamená, že UCS vždy expanduje vrchol s nejnižší kumulativní cenou od počátečního vrcholu s a tudíž nalezne optimální cestu do každého vrcholu, protože jinak by už byl vrchol expandovaný po levnější cestě. Expandované vrcholy jsou přidané do seznamu CLOSED. Pro následníky

[4] Expanzí vrcholu myslíme jeho prozkoumání a přidání jeho neprozkoumaných sousedů do fronty.

expandovaného vrcholu v, kteří nejsou v CLOSED, je spočítaná jejich g hodnota pro cestu z vrcholu s přes v. V případě, že ještě nejsou v OPEN, jsou tam přidané s vypočtenou prioritou. V opačném případě už v OPEN jsou s nějakou g hodnotou, pak pouze pokud je nová g hodnota menší než předešlá g hodnota, je jejich priorita v OPEN aktualizována.

V průběhu běhu algoritmu si budeme pro každý expandovaný vrchol označovat jeho předchůdce. Díky tomu můžeme po nalezení cílového vrcholu rekonstruovat cestu vedoucí z v_0 do c.

Podbrobně popsaný je UCS v pseudokódu 4.

Algoritmus 4: Uniform cost search

Vstup: Graf $G = (V, E)$, počáteční vrchol $v_0 \in V$, hledaný vrchol $c \in V$

Výstup: Seznam *rodice*, uchovávající předchůdce každého nalezeného vrcholu, g uchovávající cenu cesty do každého nalezeného vrcholu; nebo informaci o neúspěchu

```
g(v0) ← 0                                    ◁ g(n) je cena cesty z v0 do n
Vlož v0 do OPEN
CLOSE ← ∅                                    ◁ CLOSE je prázdný seznam
rodice(v0) ← ∅
Dokud OPEN není prázdný dělej:
    u ← OPEN.extractMin()  ◁ Odebere z fronty vrchol s nejmenší hodnotou
     g a uloží ho do u
    Pokud u je c pak:
        Vrať rodice, g
    Vlož u do CLOSED
    Pro všechny následníky v vrcholu u, kteří nejsou v CLOSED dělej:
        tmpG ← g(u) + w(u, v)                ◁ w(u, v) je váha hrany (u, v)
        Pokud v není v OPEN pak:
            g(v) ← tmpG
            rodice(v) ← u                    ◁ Nastaví vrchol u jako předchůdce v
            Vlož v do OPEN
        jinak pokud v je v OPEN a tmpG je menší než g(v) pak:
            g(v) ← tmpG
            rodice(v) ← u
Vrať nenalezeno
```

Často se setkáme s podobným algoritmem, nazývaným *Dijkstrův algoritmus*. Ten se od

UCS liší minimálně, rozdíly mezi nimi a argumenty pro používaní UCS v praxi jsou popsané v [7].

3.4.2 Hladové uspořádané vyhledávání

Algoritmus hladového uspořádaného vyhledávaní (anglicky greedy best-first search) je dalším algoritmem pro hledání cesty v kladně ohodnoceném grafu. Pracuje na předpokladu, že pokud bude opakovaně *expandovat* vrchol, který je zdánlivě nejblíž k cílí, najde cestu do cíle nejrychleji. Princip tohoto algoritmu pochází z intuitivní myšlenky, že pokud budeme hledat nejkratší cestu z Prahy do Brna, nebudeme jako první zvažovat cestu procházející Plzní a podobně.

Jedná se o *informovaný* algoritmus, což znamená, že má navíc informaci o odhadu vzdálenosti každého vrcholu $v \in V$ od cílového vrcholu. Tento odhad je typicky zprostředkován *heuristickou funkcí* $h(v)$.

Heuristické funkce (heuristiky) můžou být jakékoli, pokud např. při hledání cesty mezi dvěma lokacemi budeme znát jejich souřadnice, můžeme je využít pro výpočet heuristiky.

Nejčastěji používané heuristické funkce jsou:

1. **Eukleidovská vzdálenost:** $h(v) = \sqrt{(x_c - x_v)^2 + (y_c - y_v)^2}$, kde x_c,y_c jsou souřadnice cílového vrcholu a x_v, y_v jsou souřadnice vrcholu v. Tuto heuristiku lze použít na grafy reprezentující klasické mapy.

2. **Manhattanská vzdálenost:** $h(v) = |(x_c - x_v)| + |(y_c - y_v)|$, kde x_c,y_c jsou souřadnice cílového vrcholu a x_v, y_v jsou souřadnice vrcholu v. Tato heuristika je ideální pro mapy reprezentované čtvercovou mřížkou, kde jsou povoleny pouze vertikální a horizontální pohyby.

3. **Octile heuristika:** $h(v) = \Delta x + \Delta y + (\sqrt{2} - 2) \cdot min(\Delta x, \Delta y)$, kde $\Delta x = |x_c - x_v|, \Delta y = |y_c - y_v|$ Tato heuristika je ideální pro osmisměrné čtvercové mřížky, tedy takové, kde je kromě vertikálního a horizontálního pohybu povolen i diagonální pohyb. Počítá s cenou 1 pro vertikální a horizontální pohyby a cenou $\sqrt{2}$ pro diagonální.

Heuristická funkce $h(n)$ je *přípustná*, pokud pro každý vrchol $v \in V$ platí $h(v) \leq h^*(v)$, kde $h^*(v)$ je *ideální* heuristika, tedy skutečná vzdálenost od cíle.

Pro hladové uspořádané vyhledávání se evaluační funkce rovná heuristické funkci: $f(v) = h(v)$.

Na začátku vloží do prioritní fronty počáteční vrchol s prioritou 0. Dokud není prioritní fronta prázdná, tak v každé iteraci vybere z prioritní fronty vrchol s nejmenší $f(v)$ a ten expanduje, dokud nedorazí do cíle.

Nevýhodou tohoto algoritmu je, že vrcholy prozkoumává jen podle heuristiky. Jeho efektivita záleží na přesnosti heuristické funkce, pokud by byla heuristika ideální prozkoumá jen vrcholy vedoucí do cíle a to po optimální cestě. Jeho ohodnocovací funkce nepřihlíží k vzdálenosti od počátečního vrcholu a algoritmus nijak nezohledňuje celkovou ušlou vzdálenost od počátečního vrcholu, což může vést k nalezení neoptimálních cest. Nicméně nějakou cestu najde většinou rychleji než UCS, protože prozkoumáváním vrcholů zdánlivě bližších k cíli jako první často sníží celkový počet prozkoumaných vrcholů.

Poznámka: algoritmu se říká hladový, protože jako hladové algoritmy se označují ty algoritmy, které v každém kroku volí lokální optimum s vidinou, že tyto volby povedou celkově do globálního optima, neboli k optimálnímu řešení.

3.4.3 Algoritmus A*

Algoritmus A* navrhli Peter Hart, Nils Nilsson a Bertram Raphael v roce 1968. Algoritmus A* kombinuje efektivitu hladového uspořádaného vyhledávání s optimalitou algoritmu UCS.

Protože se jedná o další algoritmus z třídy uspořádaného vyhledávání, funguje podobně jako UCS i hladové uspořádané vyhledávání. A* taktéž prochází vrcholy grafu podle hodnoty ohodnocovací funkce a vybírá vrcholy s nejnižší hodnotou $f(v)$, jediným rozdílem je ohodnocovací funkce samotná. Tento algoritmus jako svou ohodnocovací funkci $f(v)$ využívá součet heuristické funkce $h(v)$ a funkce $g(v)$, která udává délku nejkratší cesty z počátečního vrcholu do vrcholu v: $f(v) = g(v) + h(v)$. Heuristika zde figuruje jako informace navíc, kterou čistě z grafu nevyčteme, proto se i A* řadí mezi informované algoritmy.

Pokud bude $h(v)$ přípustná, pak A* vždy najde optimální cestu do cílového vrcholu. Zvolit přípustnou heuristiku bývá jednoduché, stačí aby nikdy nepřecenila skutečnou cenu cesty do cíle. Například pro hledání cesty v mapě můžeme použít vzdálenost do cíle vzdušnou čarou, neboli eukleidovskou vzdálenost. Ovšem čím menší $h(v)$ bude tím více vrcholů A* prozkoumá.

Pokud bychom zvolili takovou heuristiku, že $\forall v \in V : h(v) = 0$ tak A* degraduje do UCS, naopak pokud $\forall v \in V : g(v) = 0$ tak A* degraduje do hladového uspořádaného vyhledávání řízeného pouze heuristikou.

A* je velmi populární volbou pro implementaci pathfindingu ve hrách nebo v mapových aplikacích pro jeho optimalitu a zároveň efektivitu a rychlost výpočtu. Existují i optimalizované verze A*, které například omezují paměťové nároky.

Část II

Implementace vizualizačního programu

4 Záměr práce

Cílem druhé části této práce je vytvoření vizualizačního programu, který bude sloužit k názornému ilustrování chodu algoritmů popsaných v předchozí části. Při tvorbě programu bude kladen důraz na lehce osvojitelné ovládání, které umožní komukoliv používat program bez nutnosti školení.

Vizualizační program bude navržen tak, aby uživatelům umožnil pohodlně a jednoduše upravovat veškeré vstupní parametry vizualizace. Taková míra interaktivity dovolí uživateli zkoumat chování algoritmů na různých grafech a objevovat jejich silné a slabé stránky. Primárním záměrem programu je pak pomoci studentům nebo případným zájemcům o problematiku pathfindingu do hloubky porozumět principům za jednotlivými algoritmy pro hledání cest.

5 Plánování

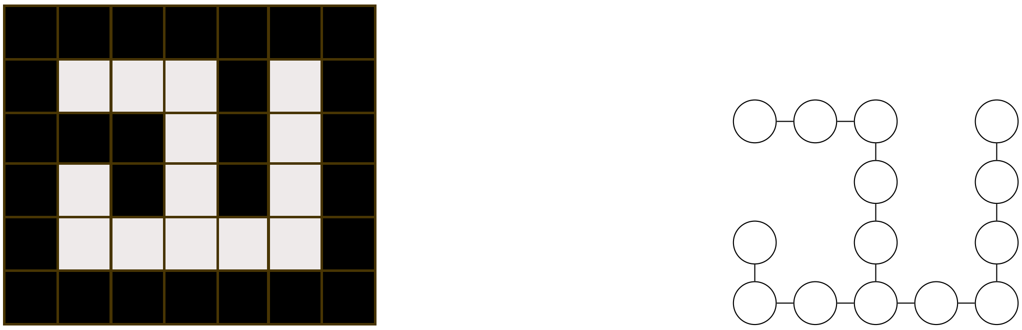
Vizualizace je proces vytváření obrazu něčeho, co není zrovna vidět nebo nějakého abstraktního konceptu. Algoritmy jsou abstraktní koncepty z definice. Co víc, i když si algoritmus úspěšně naprogramujeme, typicky nevidíme a ani nesledujeme každý jeho krok, zajímá nás jen výsledek. To je pochopitelné pokud algoritmu rozumíme a používáme ho k řešení konkrétního problému. Naopak hlavním smyslem vizualizace algoritmu je názorně ilustrovat chod algoritmu, zejména pro výukové účely.

Než jsem se pustil do samotného programování vizualizační aplikace, musel jsem si nejprve promyslet, jak si vůbec vizualizační program představuji. Jelikož je účelem programu vizualizovat grafové algoritmy, bylo jedním z nejdůležitějších rozhodnutí, jakým způsobem graficky reprezentovat graf. Nabízela se možnost grafy reprezentovat podobně jako na obrázku 3.1. Avšak takový přístup by značně omezil a ztížil uživateli návrh vlastních grafů, protože by musel každý jeden vrchol manuálně vytvořit a zdlouhavě spojovat vybrané vrcholy hranami. Domnívám se, že pro větší počet vrcholů už by se vizualizace značně znepřehlednila. Já od vizualizace požaduji i možnost zobrazovat „velké" grafy, protože na grafu s malým počtem vrcholů by vizualizace nemusely být dostatečně názorné.

Proto jsem se rozhodl reprezentovat graf pomocí mřížkové sítě. Na mřížkovou síť lze nahlížet jako speciální druh grafu, kde každé políčko sítě bude reprezentovat vrchol a mezi sousedními políčky povedou pomyslné hrany. Takový graf sám o sobě by byl velmi jednotvárný a nezajímavý, proto do sítě půjde umísťovat neprůchodné stěny, které graf zredukují o některé prvky.

Jednou možností, jak realizovat stěny, je nechat všechna políčka průchozí a umisťovat stěny mezi některé sousední políčka, což bude v grafu ekvivalentní s smazáním hran mezi odpovídajícími vrcholy. Druhou možností je určit některá políčka jako stěny a odpovídající vrcholy z grafu smazat, stejně jako všechny hrany, jichž jsou součástí.

Zvolil jsem druhou možnost, protože takovou mřížku půjde jednoduše upravovat. Ukázka takové mřížky je na obrázku 5.1. Uživatel si bude moct vybrat nástroj štětce a tím označit



(a) (b)

Obrázek 5.1: (a) Mřížka v aplikaci, černá políčka jsou stěny. (b) Odpovídající graf.

políčka, která mají figurovat jako stěny. Další nutné nástroje zahrnují nástroj pro umisťování a přemísťování startovních a cílových vrcholů.

Poslední důležitou volbou bylo které všechny algoritmy bude program schopen vizualizovat. Můj výběr se skládá z algoritmů DFS, BFS, hladového uspořádaného vyhledávání a A*. Důvodem proč jsem vynechal UCS je, že na neohodnocených grafech prohledává velmi podobně jako BFS. Moje mřížková síť odpovídá neohodnocenému grafu, protože jako sousedy vrcholu bude považovat jen ty políčka sousedící ve vertikálním a horizonatálním směru. Cena přechodu z jednoho políčka do druhého tak bude vždy stejná.

5.1 Výběr prostředků pro vizualizaci

Dalším krokem bylo zvolit si vhodné softwarové nástroje pro implementaci takového programu. Já se rozhodl pro programovací jazyk Python, protože umožňuje poměrně rychlý vývoj a mám s ním největší zkušenosti.

Samotný Python jsem potřeboval doplnit nějakou knihovnou pro práci s GUI[1] - grafickým uživatelským rozhraním. Mou zvolenou knihovnou se stala knihovna pygame. Pygame je populární knihovna primárně určena pro vývoj her. Pygame je postavena nad knihovnou Simple DirectMedia Layer (SDL), která obsluhuje nízkoúrovňové úlohy jako je renderování grafiky, zpracování zvuku a zachycování vstupu. Díky tomu se při vývoji s pygame můžeme zaměřit na implementaci logiky celé aplikace a nemusíme řešit tyto nízkoúrovňové záležitosti. Nesmírnou výhodou pygame je její jednoduchost v kombinaci s množstvím dostupných online tutoriálů, vysvětlujících jak základní principy, tak pokročilou práci s knihovnou. [5] [1]

[1]Z anglického *Graphical User Interface*.

6 Vlastní vývoj

V této kapitole představím hotový program formou vybraných útržků ze zdrojového kódu. Nebudu zde uvádět celý kód, protože je poměrně dlouhý. Celý zdrojový kód je volně dostupný na adrese `https://github.com/alexandrBihun/Maze-Visualizer`.

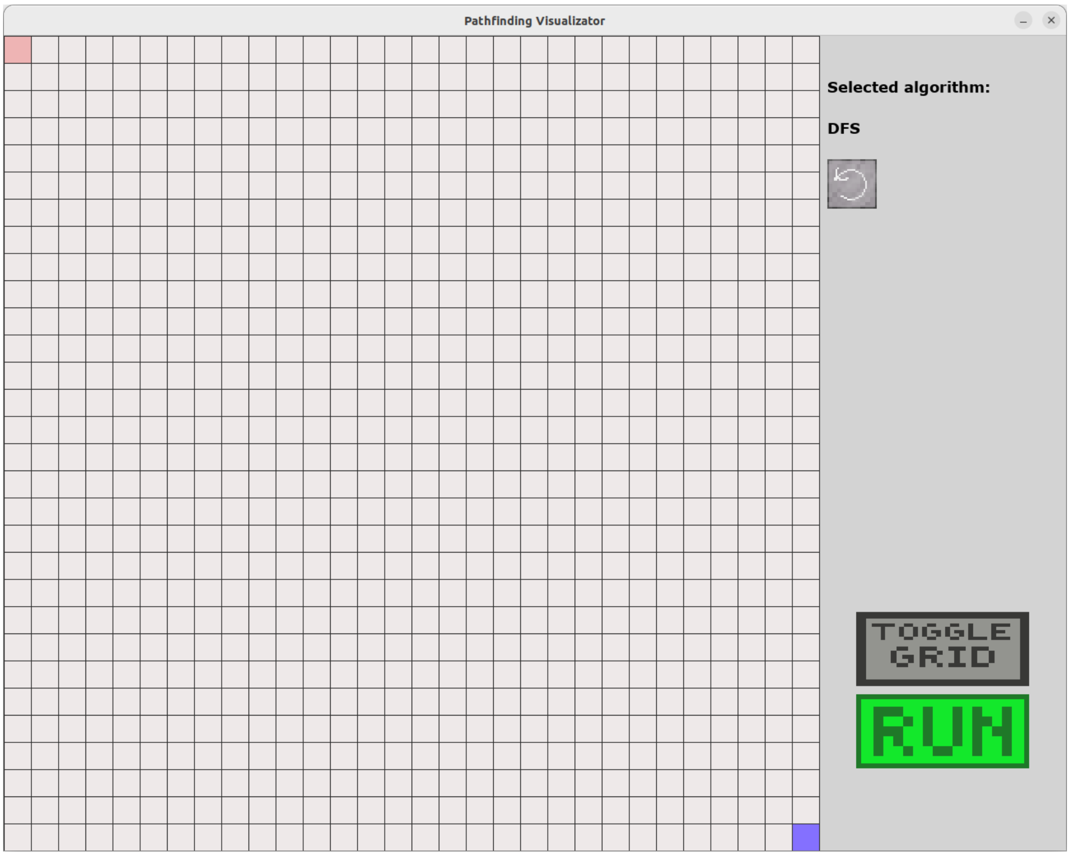
6.1 Struktura programu

Pro základní strukturu pygame aplikace mi byl inspirací [**yt_zelda**]. Rozdělil jsem podobným způsobem program na jednotlivé moduly, které spolu komunikují. Každý modul obsahuje třídy a funkce zaměřené na jeden aspekt programu. Zjednodušený diagram interakce mezi moduly vidíme na **??**

USE-CASE DIAGRAM

APLIKAČNÍ DIAGRAM nebo DIAGRAM PROPOJENÝCH TŘÍD

6.2 Problémy



Obrázek 6.1: Hlavní obrazovka

7 Uživatelská příručka

V této kapitole popíšu jak s mou aplikací pracovat a využít všechny její funkce. Rovněž zde uvedu některé vizualizace vytvořené mým programem.

7.1 Ovládání

Nejsnazší a také nejpohodlnější způsob, jak program ovládat je pomocí klávesnice:

- Tlačítky 1, 2, 3, 4 se mění vybraný algoritmus (DFS, BFS, Greedy Best First Search, A*).
- Mezerník spustí vizualizaci.
- 'g' vygeneruje bludiště.
- 's' vybere nástroj pro umisťování startu.
- 'f' vybere nástroj pro umisťování cíle.
- 'd' vybere nástroj pro kreslení a gumování stěn.
- 'c' smaže políčka vybarvená dokončenou vizualizací.
- 'r' restartuje celou aplikaci.
- Šipka nahoru zpomalí vizualizaci.
- Šipka dolů zrychlí vizualizaci.

  Kromě tlačítek klávesnice lze některé funkce ovládat i tlačítky v postranním panelu aplikace. Konkrétně tlačítko v horní části cyklí přes algoritmy, tlačítko „TOGGLE GRID“ zapíná/vypíná vykreslování mřížky a tlačítko „RUN“ spouští vizualizaci. Tyto tlačítka jsem vytvořil aby bylo možné program na základní úrovni ovládat i bez znalosti kláves

Mimo to lze v souboru `settings.py` upravovat pokročilé možnosti. Proměnná `sideLength` udává počet políček ve straně mřížky. Výchozí hodnotou pro `sideLength` je 50, program správně vykresluje políčka až pro hodnoty `sideLength` ≤ 990, což je výchozí počet pixelů ve straně mřížkového prostoru. Pro 990 tak jedno políčko zabírá jeden pixel obrazovky, nastavení vyšší hodnoty má za následek nevykreslení některých políček. Vizualizace stále bude fungovat, jen některé vrcholy grafu už nebudou vidět.

7.2 Barevné rozlišní políček

V programu se

7.3 Ukázky použití

Závěr

Tady bude závěr.

Co do budoucna: možnost ukládat a nahrávat nakreslená bludiště, přidat do vizualizace zvuk, rozšířit UI, menu Nápověda přímo v programu s informacemi o programu a algoritmech,

8 Výpisy použitých programů

Bibliografie

1. *About.* [B.r.]. Dostupné také z: `https://www.pygame.org/wiki/about`. (cit. 8. 2. 2024).
2. ADAIXO, Michaël Carlos Gonçalves. *Influence Map-Based Pathfinding Algorithms in Video Games.* 2014. Dostupné také z: `https://api.semanticscholar.org/CorpusID:86577873`. Dipl. pr. UNIVERSIDADE DA BEIRA INTERIOR.
3. CHABERT, Jean-Luc et al. *A History of Algorithms: From the Pebble to the Microchip.* Springer Berlin, Heidelberg, 1999. ISBN 9783540633693.
4. CORMEN, Thomas H.; STEIN, Clifford; RIVEST, Ronald L.; LEISERSON, Charles E. *Introduction to Algorithms.* Fourth edition. Cambridge, Massachusetts: The MIT press, 2022. ISBN 9780262046305.
5. DATASCIENTEST. *PyGame: The 2D video game creation tool in Python.* [B.r.]. Dostupné také z: `https://datascientest.com/en/pygame-the-2d-video-game-creation-tool-in-python`. (cit. 8. 2. 2024).
6. DVORSKÝ, Jiří. *Algoritmy I.* 2007. Dostupné také z: `https://fei.znoj.cz/soubory/ALGI/skripta.pdf`. Verze ze dne 28. února 2007.
7. FELNER, Ariel. *Dijkstra's Algorithm versus Uniform Cost Search or a Case Against Dijkstra's Algorithm.* 2011. Dostupné také z: `https://web.archive.org/web/20200218150951/https://www.aaai.org/ocs/index.php/SOCS/SOCS11/paper/viewFile/4017/4357`.
8. GARG, Prateek. *Depth First Search.* [B.r.]. Dostupné také z: `https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/`. (cit. 28. 1. 2024).
9. KOVÁŘ, Petr. *Úvod do Teorie grafů.* 2021. Dostupné také z: `https://homel.vsb.cz/~kov16/files/uvod_do_teorie_grafu.pdf`.

10. MAREŠ, Martin; VALLA, Tomáš. *Průvodce labyrintem algoritmů*. Druhé vydání. Praha: CZ.NIC, 2022. ISBN 978-80-88168-63-8.

11. MEHRI, Bahman. From Al-Khwarizmi to Algorithm. *Sharif University of Technology, Iran*. 2017.

12. NECKÁŘ, Jan. *Algoritmy*. © 2016. Dostupné také z: https://www.algoritmy.net/. (cit. 20. 1. 2024).

13. PATEL, Amit. Heuristics. [B.r.]. Dostupné také z: http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#heuristics-for-grid-maps. (cit 5. 2. 2024).

14. PATEL, Amit. Implementation of A*. 2020. Dostupné také z: https://www.redblobgames.com/pathfinding/a-star/implementation.html. (cit 5. 2. 2024).

15. PATEL, Amit. Introduction to the A* Algorithm. 2020. Dostupné také z: https://www.redblobgames.com/pathfinding/a-star/introduction.html. (cit 5. 2. 2024).

16. SIMIC, Milos. Uniform-Cost Search vs. Best-First Search. 2022. Dostupné také z: https://www.baeldung.com/cs/uniform-cost-search-vs-best-first-search. (cit. 5. 2. 2024).

17. UHLÍK, Jan. *Grafy a grafové algoritmy pro knihovnu algoritmů*. 2018. Dostupné také z: https://dspace.cvut.cz/bitstream/handle/10467/76776/F8-BP-2018-Uhlik-Jan-thesis.pdf. Bak. pr. České vysoké učení technické v Praze, Fakulta informačních technologi.

18. VIRIUS, Miroslav. *Základy algoritmizace*. Praha: ČVUT, 2008. ISBN 978-80-01-04003-4. Dostupné také z: http://www.jaderny-prvak.8u.cz/wp-content/uploads/2013/02/Základy-algoritmizace-skripta-21.pdf.

19. WIKI. Best-first search. [B.r.]. Dostupné také z: https://en.wikipedia.org/wiki/Best-first_search.

20. ČERNÝ, Jakub. *Základní grafové algoritmy*. MFF UK, 2010. Dostupné také z: https://docplayer.cz/4802558-Zakladni-grafove-algoritmy.html.

Seznam obrázků

Seznam tabulek

Přílohy

A Příloha s kódem