ALGORITMY NALEZENÍ NEJKRATŠÍ CESTY

Lukáš Podolák

Střední průmyslová škola elektrotechnická

V Úžlabině 320, Praha 10

„Prohlašuji, že jsem tuto práci vypracoval samostatně a použil jsem literárních pramenů a informací, které cituji a uvádím v seznamu použité literatury a zdrojů informací.“

V Praze dne ....

podpis autora (jméno a příjmení)

Anotace

Tato práce se zabývá popisem vytváření programu pro nalezení nejkratší cesty mezi dvěma body v dvourozměrném poli. V práci je popsána teoretická část fungování vyhledávacích algoritmů a následně jsou rozebrány jednotlivé kroky vývoje programu.

Annotation

This work deals with the description of creating a program to find the shortest path between two points in a two-dimensional array. The thesis describes the theoretical part of the search algorithms and then discusses the various steps of program development.

Obsah

Úvod

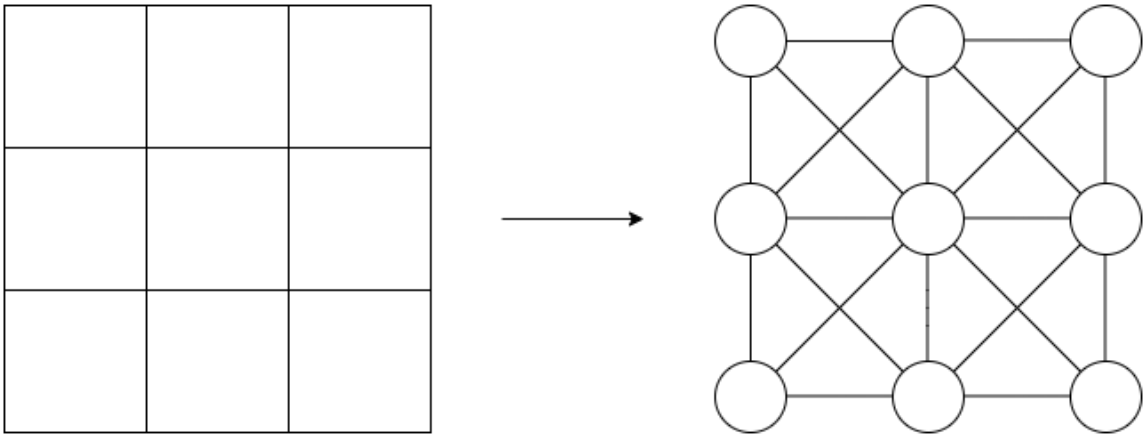
Při volbě tématu pro svou dlouhodobou maturitní práci jsem se rozhodl pro tento projekt (Algoritmus nalezení nejkratší cesty) z důvodu, že jsem již dříve viděl na internetu různé vizualizace fungování těchto algoritmů a zaujalo mě to. Chtěl jsem pochopit jak tyto algoritmy doopravdy fungují a pro svůj program jsem si vybral algoritmus A* (A Star).

1 Teoretická část

1.1 Graf

Graf je datová struktura, která se skládá z dvou objektů. Prvním je vrchol (uzel) a druhým objektem je hrana, která propojuje vrcholy grafu a má své hodnocení (např. délka hrany). Hrana je vždy jednou ze dvou následujících variant. Orientovaná hrana, která má jasně daný svůj směr, anebo neorientovaná hrana, která směr určený nemá.

Grafy se často využívají pro znázornění nějaké fyzické sítě, jako je například přenosová soustava elektrické energie. V mé práci využívám čtvercové pole rozdělené na menší čtverečky, které tvoří vrcholy grafu a můžu se zde pohybovat mezi vrcholy jak diagonálně tak i horizontálně a vertikálně.



Obrázek 1 Znázornění pole programu v grafu

1.2 Vyhledávací algoritmus A* (A Star)

A Star je počítačový algoritmus určený pro vyhledávání optimální cesty. Vyhledává je v hodnocených grafech. V mém programu optimální cestou myslím nejkratší cestu, ale lze za optimální považovat také například nejrychlejší, nebo nejlevnější cestu. K nalezení této cesty algoritmus A Star používá funkci $f(x)$, která se skládá z dalších dvou funkcí $f(x) = g(x) + h(x)$. Funkce $g(x)$ vyjadřuje vzdálenost mezi počátečním uzlem a uzlem, na kterém se právě nacházíme. Funkce $h(x)$ je takzvaná heuristická funkce, která se snaží odhadovat správný směr postupu vyhledávacího algoritmu. Docílí toho pomocí spočítání

vzdálenosti mezi koncovým uzlem a aktuálním uzlem. Pro tento výpočet lze použít takzvanou vzdálenost vzdušnou čarou jakožto nejkratší možnou vzdálenost bez ohledu na překážky.

1.2.1 Průběh algoritmu

Vytvoří se udržovaná prioritní fronta takzvaný OpenList, do kterého se ukládají nenavštívené uzly. Čím menší má uzel hodnotu $f(x)$, tak tím vyšší má prioritu v této frontě. V každém kroku algoritmu se uzel s nejvyšší prioritou (to znamená s nejnižší hodnotou $f(x)$) odebere z OpenListu a následně se spočítají hodnoty f a h pro jeho všechny sousední uzly. Tyto sousední uzly jsou přidány do OpenListu a pokud v OpenListu již jsou, protože například také sousedí s jiným uzlem a nově vypočítané hodnoty jsou nižší, tak se těmto uzlům hodnoty f a h pouze aktualizují a v OpenListu zůstávají. Aktuální uzel, který byl odebrán z OpenListu se ukládá do nové fronty, která se nazývá CloseList a tato fronta udržuje již prohledané uzly. Algoritmus nadále pokračuje v prohledávání svých sousedů, dokud cílový uzel nemá menší hodnotu f než jakýkoliv jiný uzel v OpenListu, nebo dokud není OpenList prázdný. Hodnota f koncového uzlu se stává délkou nejkratší cesty grafem. Pro následné znázornění konkrétní nejkratší cesty začneme procházet CloseList od koncového bodu a přecházíme vždy na uzel, ze kterého jsme na aktuální uzel přišli, dokud se nedostaneme na uzel, na kterém jsme začínali.

1.2.2 Složitost

Časová složitost vyhledávacího algoritmu A Star závisí na použité heuristické funkci.
V nejhorším možném případě je počet prozkoumaných uzlů exponenciální vzhledem k délce cesty, ale naopak v optimálním případě je složitost polynomiální (efektivní). Tento stav nastane, pokud heuristická funkce splňuje následující podmínku:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

Rovnice 1 Rovnice složitosti algoritmu A Star

Kde h^* je optimální heuristická funkce, neboli přesná vzdálenost ke koncovému uzlu. Dá se také říct, že chyba funkce $h(x)$ neporoste rychleji, než logaritmus optimální heuristiky.

1.3 Dijkstrův vyhledávací algoritmus

Tento algoritmus se využívá pro nalezení nejkratší cesty v grafu. Poprvé ho popsal nizozemský informatik Edsger Dijkstra v roce 1956 a publikoval o tři roky později v roce 1959. Je to jeden z nejvyužívanějších a nejefektivnějších vyhledávacích algoritmů. Využívá se

v mnoha mapových aplikacích, nebo ho například využívá síťový protokol STP (Spanning Tree Protocol) v ethernetových LAN sítích.

1.3.1 Popis algoritmu

Vstupem algoritmu je ohodnocený graf G a počáteční uzel s (Start). Dále se vytvoří množina V, což je množina všech vrcholů v grafu G a množina E, která obsahuje všechny hrany grafu G. Dijkstrův vyhledávací algoritmus pracuje na principu toho, že si pro každý vrchol v z množiny V pamatuje délku nejkratší cesty, pomocí které se k němu dá dostat. Tato hodnota se obvykle označuje $d[v]$ po spuštění programu mají všechny vrcholy v na začátku $d[v] = \infty$ (nekonečno znamená, že neznáme cestu k vrcholu). Ale počátečnímu vrcholu s se hned nastaví hodnota na $d[v] = 0$. Algoritmus si také udržuje další dvě množiny. Množinu Z, která obsahuje již navštívené vrcholy a množinu N, ve které jsou ty vrcholy, které zatím navštívené nebyly. Algoritmus pracuje v cyklu do té doby, dokud v množině N jsou nějaké hodnoty. V každém průchodu cyklu algoritmu se jeden vrchol v_{min} (má nejmenší hodnotu $d[v]$ ze všech vrcholů v množině N) přesune z množiny N do množiny Z. Pro každý vrchol u, do kterého vede hrana s délkou označenou jako $l(v_{min}, u)$, se provede tato operace. Pokud platí, že $\left(d[v_{min}] + l(v_{min}, u)\right) < d[u]$ tak se do $d[u]$ přiřadí hodnota $d[v_{min}] + l(v_{min}, u)$, pokud ne tak se neprovede nic. Když algoritmus skončí, tak je pro každý vrchol v z množiny V délka jeho nejkratší cesty k počátečnímu vrcholu s uložena v hodnotě $d[v]$.

1.3.2 Složitost

Tento algoritmus lze implementovat do programu s asymptotickou složitostí.
$O(E + V \log_{10} V)$, kde E je počet hran v grafu G.

1.4 BFS (Breadth-first search)

Český překlad názvu tohoto algoritmu je prohledávání do šířky. Tento algoritmus je nejjednodušší ze všech zmíněných a zároveň nejméně efektivní, jelikož postupně prochází všechny vrcholy v daném grafu.

1.4.1 Popis algoritmu

Prohledávání do šířky, neboli BFS, funguje na tom principu, že systematicky prohledává všechny uzly v grafu. Při prohledávání tento algoritmus nepoužívá žádnou heuristickou

funkci, jako například algoritmus A* (A Star), ale prochází jednotlivé uzly a pro každý z nich projde všechny jeho následovníky. Zároveň si ukládá předchůdce jednotlivých uzlů a z nich následně vytváří strom cest z jednotlivých uzlů do uzlu počátečního (kořene).

Tento algoritmus využívá takzvanou FIFO frontu. Tato fronta funguje tak, že první uzel, který do ní vstoupí z ní také jako první vystoupí.

Na začátku algoritmu se nastaví hodnoty pro jediný uzel ,o kterém známe všechny informace a to je počáteční uzel. Následně se spustí cyklus, který běží do té doby, dokud FIFO fronta není prázdná. Na začátku cyklu z fronty vyjmeme uzel a u něj hledáme všechny jeho následovníky (sousedy), kteří mají svůj stav nastavený jako FRESH (tento uzel zatím nebyl nalezen ani prohledán). Pokud jsou takové uzly nalezeny, tak se jim nastaví tyto hodnoty: vzdálenost od počátku o jedno vyšší než v uzlu předchozím a stav se změní na OPEN. Po nastavení všech hodnot je uložen do fronty FIFO. Když jsou prohledáni všichni následovníci uzlu, tak je uzel uzavřen a jeho stav se změní na CLOSE.

1.4.2 Složitost

Časová složitost tohoto algoritmu je $O(|V| + |E|)$, kde V se rovná množině všech vrcholů a E je množina všech hran v grafu. Prostorová náročnost se počítá jako $O(B^M)$, kde B je nejvyšší stupeň větvení stromu a M odpovídá nejvyšší délce cesty ve stromě.

1.5 Objektově orientované programování

Při programování své aplikace jsem často využíval OOP neboli Objektově orientované programování z důvodu lepší orientace v kódu a možnosti opakovaného využívání definovaných prvků.

1.5.1 Dědičnost

OOP také nabízí dědičnost, což umožňuje definovat hierarchickou strukturu jednotlivých tříd. To znamená, že to co mají nějaké třídy společné můžeme definovat u jejich předka a potomci k tomu budou mít stále přístup.

1.5.2 Zapouzdření

Zapouzdření v objektově orientovaném programování umožňuje využívání atributů private a public. Atribut private umožňuje skrýt například metodu tak, že nebude přístupná z jiné třídy než z té, ve které je definována. Atribut public naopak umožňuje tuto metodu zveřejnit

tak, že bude přístupná z jakékoliv části programu. Následně je tady ještě další varianta a to dát přístup k této metodě jen potomkům třídy, ve které je tato metoda definována. Tuto variantu lze uskutečnit pomocí atributu protected.

1.5.3 Polymorfismus

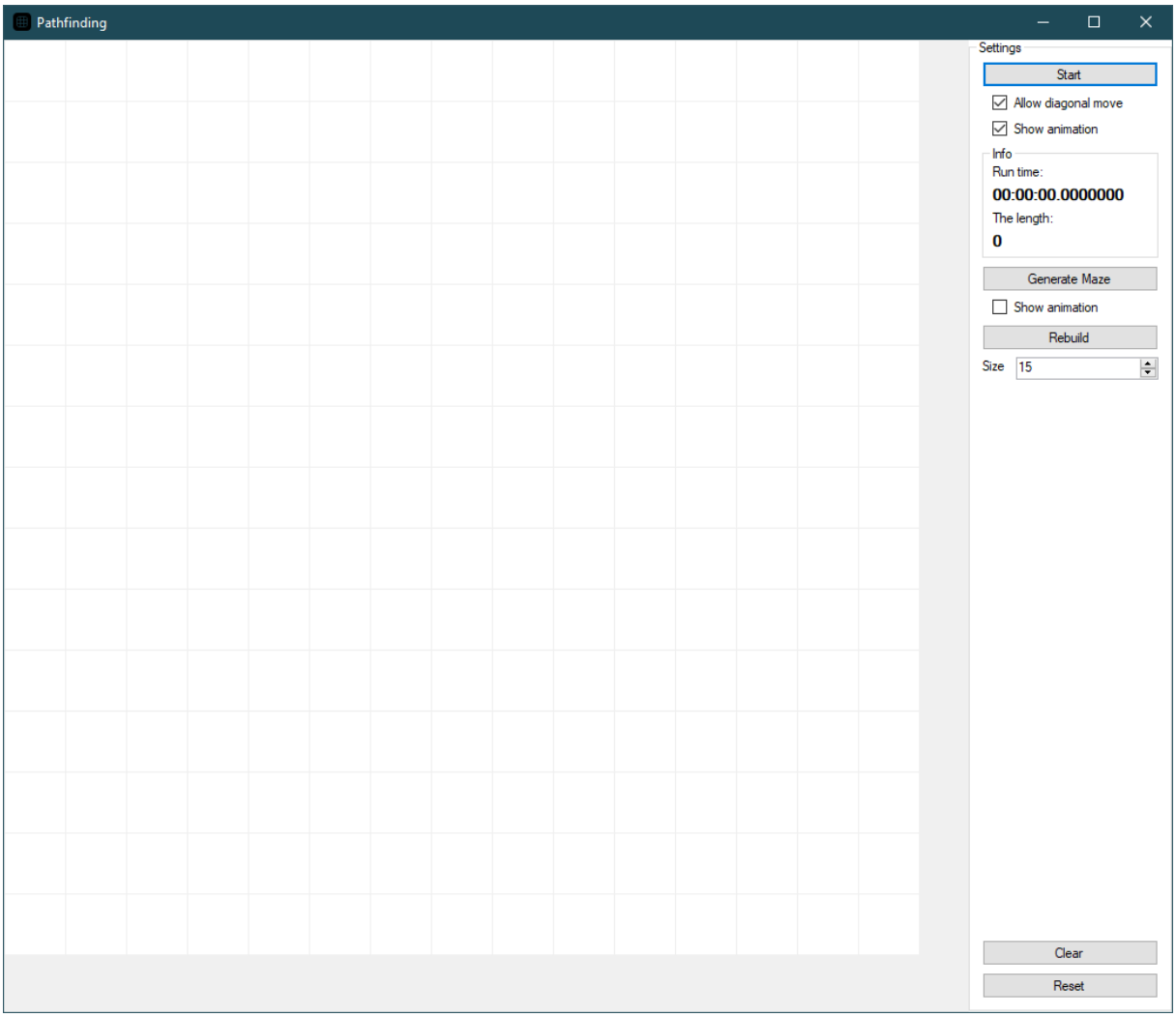
Polymorfismus nám umožňuje definovat různé chování metod se stejným názvem. Pokud je metoda v předkovi deklarována jako abstract nebo dynamic, může ji překrýt metoda v potomkovi pomocí modifikátoru override.

2 Postup vývoje programu

2.1 Návrh GUI

Po založení projektu jsem začal s vytvářením grafického uživatelského rozhraní neboli GUI. Rozhodl jsem se, že bude rozděleno na dvě hlavní části. V levé části okna se generuje pole, ve kterém se znázorňuje průběh algoritmu během vyhledávání a v druhé části, která je umístěná na pravé straně a je pojmenována Settings, jsou umístěny ovládací prvky programu. Jako první je nahoře tlačítko Start, které po zadání počátečního a koncového bodu spustí vyhledávací algoritmus. Pod tímto tlačítkem jsou dva související CheckBoxy. První je pro povolení algoritmu pohybovat se diagonálně a druhý je pro povolení zobrazování animace průběhu algoritmu. Oba tyto CheckBoxy jsou ve výchozím nastavení po spuštění programu zaškrtnuté, což znamená, že jsou aktivní. Dále zde nalezneme skupinu Info, ve které se nic nenastavuje. Jen zobrazuje informace o provedení vyhledávacího algoritmu. Jako první v této skupině totiž jsou stopky (měří čas běhu vyhledávacího algoritmu) a dále popisek s názvem The length. Ten ukazuje délku cesty, kterou vyhledávací algoritmus nalezne. Pod skupinou Info se nachází tlačítko pojmenované jako Generate Maze (Generovat bludiště), které ve vygenerovaném poli nalevo vygeneruje pseudonáhodné bludiště. Pod tímto tlačítkem se nachází související CheckBox, který povoluje zobrazení animace průběhu generování bludiště. Tento CheckBox je ve výchozím nastavení po spuštění programu nataven na hodnotu false, což znamená, že není zaškrtnutý a tím pádem se animace nezobrazí. Následuje tlačítko Rebuild, které vymaže a vygeneruje nové pole v levé části programu v závislosti na zadané velikosti v ovládacím prvku pod tímto tlačítkem. Zde jdou zadat hodnoty od 5 do 57, což vygeneruje graf o velikosti 25 až 3 249 uzlů. Poté je zde

mezera a v pravém dolním rohu programu se nachází poslední dvě tlačítka. První tlačítko má název Clear a to v poli vymaže všechny uzly, až na nakreslené nebo vygenerované bariéry. A posledním tlačítkem je tlačítko Reset, které kompletně vymaže pole nalevo.



Obrázek 2 Ukázka GUI

2.2 Programování aplikace

2.2.1 Třída Form1.cs

Začal jsem v této třídě a přidal jsem funkcionalitu všem ovládacím prvkům programu. Jako první je zde inicializace komponent a dále generování pole nalevo pomocí funkce GenerateArea s parametrem pro jeho velikost. Tento parametr je ve výchozím nastavení po spuštění programu nastaven na hodnotu 15, což znamená, že pole je velké 15 × 15 uzlů.

Dále jsem zde naprogramoval ovládací prvky pro toto pole. Pole se ovládá tak, že když kliknete na nějaký uzel poprvé, tak se označí písmenem S (Start) neboli startovní uzel. Druhým kliknutím označíte jiný uzel, který dostane písmeno E (End) jako koncový uzel.

Následně můžete pomocí stlačeného levého tlačítka myši jezdit po tomto poli a tyto uzly se zabarví do černé barvy, což značí, že to je bariéra a program tímto uzlem nemůže projít. Dále pomocí pravého tlačítka myši můžeme odebrat bariéru, nebo můžeme odebrat start a cíl a umístit je na jiné místo. Na pozadí jsou typy těchto uzlů označovány pomocí tagů: N – NULL, C – CLOSE, O – OPEN, B – BARRIER, S – START, E – END.

Poté jsou v této třídě už jen funkce k tlačítkům Start, Generate Maze, Rebuild, Clear, Reset, ve kterých je buď jen cyklus pro přepsání pole anebo volají jinou funkci.

2.2.2 Třída Points.cs

Tato třída je nastavená jako public static a udržuje hodnotu čtyř proměnných. První dvě jsou datového typu bool a je v nich uloženo, zda byl nastaven start a cíl. Další dvě proměnné jsou datové struktury Point a ty ukládají souřadnice nastaveného startu a cíle, pokud jsou nastaveny v poli. Následně zde je funkce Clear, pomocí které můžu jednoduše vymazat natavení proměnných v této třídě.

2.2.3 Třída PathNode.cs

Tato třída je nastavení jako public a je to třída pro jednotlivé uzly, které se vytvářejí při generování pole. Je zde veřejně uložená lokace tohoto uzlu a uzlu jeho rodiče pokud existuje, protože například startovní uzel svého rodiče nemá. Dále jsou zde proměnné potřebné pro běh vyhledávacího algoritmu A* (A Star) a to gCost, hCost, fCost a již zmiňovaný tag pro uložení jakého typu tento uzel je.

Následně v této třídě je uložen přetížený konstruktor. V první variantě při inicializaci této třídy zadáváme pouze lokaci (souřadnice) tohoto uzlu a v druhé variantě můžeme zadat lokaci a zároveň hodnotu G neboli gCost tohoto uzlu. Zadané parametry se uloží do již vytvořených proměnných.

Dále je zde jednoduchá funkce pro výpočet proměnné fCost, což je jen součet hodnot gCost a hCost. Tato funkce je pojmenována jako CalculateFCost. Následuje funkce Clear, díky které lze vymazat všechny nastavené hodnoty v této třídě.

2.2.4 Třída MazeGenerator.cs

Tato třída je zodpovědná za náhodné vygenerování bludiště v poli. Pro generování náhodného bludiště jsem využil algoritmus založený na principu rekurzivního backtrackingu. To znamená, že tomuto algoritmu je na začátku zadám jeden počáteční bod a program najde

všechny jeho sousedy a náhodně z nich jednoho vybere. Následně se ta samá funkce zavolá znovu a jako počáteční bod jí je nastaven soused vyhledaný aktuálním průběhem funkce. Takto jde program pořád dokola, dokud nenarazí na uzel, pro který nedokáže najít žádného souseda. V tomto případě se vrátí zpět po již vygenerované cestě k prvnímu uzlu, u kterého tohoto souseda dokáže najít. Následně generuje jinou větev této cesty. Tento proces opakuje pořád dokola, dokud graf, ve kterém bludiště generuje, úplně nezaplní.

Jako první je v této třídě inicializace privátní datové struktury Stack, ve které jsou uložené hodnoty Point. Tato datová struktura se v češtině nazývá jako zásobník, protože funguje na podobném principu. Poslední hodnotu, kterou do této struktury vložím jako první vyjmu zase ven. Používají se k tomu funkce Push pro vložení a funkce Pop pro vyjmutí. Následně je zde konstruktor této třídy a poté už následuje samotná funkce pro generování bludiště s názvem GenerateMaze a dvěma parametry první je datového typu Point a zadávají se zde souřadnice počátečního uzlu a druhý je datového typu bool a tato proměnná se využívá jen pro to, abych měl kontrolu nad tím, zda zobrazit animaci průběhu algoritmu či nikoliv. V této funkci jako první najdu sousedy a následně, pokud jsou nějací sousedé nalezeni, tak z nich náhodně jednoho vyberu, vložím jeho souřadnice do Stacku a spustím funkci GenerateMaze pro vybraného souseda. Pokud se žádného souseda najít nepodaří, tak pomocí funkce Pop vyhodím poslední hodnotu ze Stacku a následné přečtu poslední hodnotu ve Stacku a znovu pro ni spustím funkci GenerateMaze. Následuje funkce ColorIt, která zakreslí vygenerované bludiště do pole a do grafu. Poslední je funkce GetNeighbours s parametrem datového typu Point, pro který tato funkce vyhledává sousedy. Tato funkce se využívá na začátku funkce GenerateMaze a vrací list se souřadnicemi sousedních uzlů.

2.2.5 Třída AStarV2.cs

Nejdůležitější třídou v tomto programu je právě třída AStarV2, která se stará o vyhledávaní cesty v zadaném grafu podle algoritmu A* (A Star). Jako první jsou v této třídě definované dvě privátní konstanty, které určují vzdálenost mezi jednotlivými uzly. První je pro uzly propojené vertikálně nebo horizontálně, a tato hodnota je nastavená na hodnotu 10. Další konstanta je pro pohyb v diagonálním směru a její hodnota je nastavená na 14. Následně se inicializuje openList a closeList, které jsem již zmiňoval v popisu tohoto algoritmu. Do těchto dvou listů se ukládá hodnota datového typu Point. Následuje konstruktor této třídy a poté je zde už nejdůležitější funkce FindPath s jedním parametrem datového typu bool, dle kterého

program určuje, zda má či nemá zobrazovat aminaci průběhu algoritmu. V této funkci je cyklus while, který probíhá do té doby, dokud je v proměnné openList něco uloženo. Jako první v tomto cyklu zkontroluji, zda aktuální uzel má stejnou lokaci jako uzel koncový. To by totiž znamenalo, že cesta byla nalezena a zavolá se funkce pro vykreslení cesty. Po této podmínce se aktuální uzel vyjme z openListu a přidá se do closeListu a zároveň se tomuto uzlu přidá tag C (Close). Následuje cyklus foreach, který prochází všechny vyhledané sousedy aktuálního uzlu, kteří nejsou označeny jako bariéra nebo jako close. Následně se spočítá takzvaná tentativeGCost ze součtu gCost aktuálního uzlu a vypočítané gCost sousedního uzlu. Pokud je tato tentativeGCost menší nebo rovna gCost, kterou vyhledaný soused už má nastavenou, tak se tomuto sousedovi gCost aktualizuje na tentativeGCost, vypočítá se a nastaví se hCost, nastaví se aktuální uzel jako rodič sousedního uzlu a zavolá se funkce pro výpočet a nastavení fCost sousedního uzlu. Následně pokud openList neobsahuje tohoto souseda, tak se do open listu přidá a pokud tohoto souseda obsahuje a zároveň, pokud nová varianta tohoto souseda má fCost menší nebo rovnu verzi souseda, která již byla uložena v openListu, tak se tato varianta aktualizuje podle aktuální varianty. Následně, když skončí cyklus while s tím, že cíl nebyl nalezen, tak to znamená, že cesta k cíli neexistuje a program zobrazí informační okno s touto informací.

Další funkce, která je v této třídě je funkce s názvem CalculatePath, které je potřeba jako parametr zadat cílový uzel a tato funkce pomocí v uzlu uložených potomků vyhledá a zobrazí nalezenou cestu. Po této funkce je zde také funkce s názvem GetNeighbourList, která jak její název napovídá, vrací list se souřadnicemi sousedních bodů. Tato funkce také má parametr datového typu bool názvem diagonalMove, který povoluje vyhledávání sousedů také v diagonálním směru. Další velice důležitou funkcí v této třídě je privátní funkce s návratovou hodnotou datového typu int, která se jmenuje CalculateDistanceCost a má dva parametry. Oba jsou datové struktury Point pro vložení souřadnic. První se jmenuje pathNodeAPosition a druhý pathNodeBPosition. Toto je ta takzvaná heuristická funkce pro výpočet vzdálenosti mezi zadanými dvěma body. Pro výpočet využívá následující vzorec:

$$h = D \times Min\{|X_A - X_B||Y_A - Y_B|\} + S \times \big||X_A - X_B| - |Y_A - Y_B|\big|$$

Rovnice 2 Rovnice heuristické funkce

Kde funkce Min vybere ze zadaných dvou hodnot tu menší a dosadí ji do vzorce, proměnná D odpovídá na začátku třídy definované konstantě MOVE_DIAGONAL_COST a proměnná

S odpovídá konstantě MOVE_STRAIGHT_COST. V programu výpočet této hodnoty vypadá takto:

```
int xDistance = Math.Abs(pathNodeAPosition.X - pathNodeBPosition.X);
int yDistance = Math.Abs(pathNodeAPosition.Y - pathNodeBPosition.Y);
int remaining = Math.Abs(xDistance - yDistance);
return MOVE_DIAGONAL_COST * Math.Min(xDistance, yDistance) + MOVE_STRAIGHT_COST
* remaining;
```

Následuje a jako poslední v této třídě je funkce s názvem GetLowestFCostNode. Tato funkce má návratovou hodnotu typu Point za parametr chce list Pointů. V mém programu se jí předává openList. Tato funkce Jednoduše projde všechny uzly se souřadnicemi, které jsou v zadaném listu a najde v nich uzel, který má nejnižší fCost. A potom vrátí souřadnice tohoto uzlu.

3 Problémy a jejich řešení

S největším problémem, se kterým jsem se potýkal bylo, když jsem se poprvé pokoušel o implementaci vyhledávacího algoritmu A* (A Star). Problém jsem měl s tím, že jsem si generoval uzly sousedů při jejich vyhledávání. Což mi můj program ve větším poli hodně zpomalovalo. Vyřešil jsem to tím, že jsem začal třídu AStar přepisovat a předělal jsem ji tak, že se všechny uzly generují již při generování pole, do kterého můžeme zakreslovat na začátku programu. To znamená, že při spuštění vyhledávacího algoritmu už mám všechny uzly vygenerované, a když hledám sousedy, tak si pro ně jen „sáhnu“ do tohoto pole a vyhledávacímu algoritmu předám jen souřadnice těchto nalezených sousedů.

Další problém, co jsem měl bylo, že jsem z vytvořených tříd neměl přístup k ovládání jednodílných políček ve vygenerovaném poli. Potřeboval jsem to z toho důvodu, aby například třída MazeGenerator a její funkce pro generování bludiště mohli zobrazovat animaci generování, která funguje tak, že se každý další vygenerovaný krok zabarví v poli do oranžové barvy. Tento problém jsem nakonec vyřešil jednoduše tím, že konstruktoru této třídy předávám parametr typu Form1 a při inicializaci této třídy ve třídě Form1 zadávám do parametru odkaz na aktuální třídu this.

Závěr

Práce se mi povedla dokončit tak, jak jsem si představoval a obecně jsem se svým programem spokojen. Jsem rád, že jsem si toto téma vybral a dokázal zpracovat tak, že se program dá označit za funkční a využitelný. Myslím, že to pro mě byla velká zkušenost po stránce programátorské, protože jsem si musel o fungování tohoto algoritmu zjistit vše, pochopit ho a naimplementovat do svého programu tak, aby fungoval a byl efektivní. Zároveň psaní protokolu, ve kterém je vše rozebrané, je také dobrá zkušenost pro moje případné budoucí zaměstnání a jsem za ni rád. Celkově jsem se svou prací spokojen.

Seznam použité literatury a zdrojů

Graf (teorie grafů). In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-03-06]. Dostupné z: https://cs.wikipedia.org/wiki/Graf_(teorie_grafů)

MAUTNER, Pavel. *Grafové algoritmy: Programovací techniky* [online]. Plzeň, 2020 [cit. 2021-03-06]. Dostupné z: http://www.kiv.zcu.cz/~mautner/Pt/grafove_algoritmy1.pdf

A*: A star. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-03-06]. Dostupné z: https://cs.wikipedia.org/wiki/A*

Dijkstrův algoritmus. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-03-06]. Dostupné z: https://cs.wikipedia.org/wiki/Dijkstrův_algoritmus

Dijkstra's algorithm: Dijkstra's Shortest Path First algorithm, SPF algorithm. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-03-06]. Dostupné z: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

HORDĚJČUK, Vojtěch. Algoritmus Dijkstra. *Voho* [online]. [cit. 2021-03-06]. Dostupné z: http://voho.eu/wiki/algoritmus-dijkstra/

Prohledávání do šířky: Breadth-first search. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2021-03-06]. Dostupné z: https://cs.wikipedia.org/wiki/Prohledávání_do_šířky

SONG, John. A Comparsion of Pathfinding Algorithms. *YouTube* [online]. 16. 9. 2019 [cit. 2021-03-06]. Dostupné z: https://www.youtube.com/watch?v=GC-nBgi9r0U

Přílohy

Seznam obrázků

Seznam rovnic